

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Runtime Code Generation for Sparse Small Matrix Multiplies

Final Project Report

Author:
Chenyu Zhang

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Wayne Luk

Submitted in partial fulfilment of the requirements for the MSc degree in Computing
Science of Imperial College London

September 2021

Abstract

General Matrix Multiplication (GEMM) is the core of many numerical solvers in science and engineering domain. This thesis is motivated by the use of GEMM in PyFR - a Python open-source library for solving Computational Fluid Dynamics (CFD) problems using the Flux Reconstruction (FR) method. In PyFR, the GEMM is block-by-panel variant. The block matrix is small and potentially sparse, and remains constant throughout the entire simulation. Standard Basic Linear Algebra Subprograms (BLAS) libraries are the most common choices for solving GEMM. However, as these libraries are optimised for large and dense matrices, using them for PyFR results in sub-optimal performance.

PyFR uses two libraries to accelerate the computation - GiMMiK and LIBXSMM, which are both Just-In-Time compilers capable of generating optimised problem-specific GEMM kernels. This thesis focuses on LIBXSMM, which is an Intel open-source library providing specialised dense and sparse matrix operation routines for x86 platforms with SSE, AVX, AVX2 and AVX-512 extensions. This thesis targets AVX-512. LIBXSMM also supports accelerating primitive deep neuron network routines such as small convolutions.

The sparse matrices involved in the PyFR's GEMM contain repeated elements. LIBXSMM provides a specialised small and sparse GEMM routine which stores these distinct non-zero elements of the sparse matrix in vector registers. For AVX-512, this routine can accommodate maximally 240 double-precision or 480 single-precision distinct non-zero elements in the sparse matrix. A fallback dense routine will be employed if the matrix contains too many unique elements.

In this thesis we presented two improved small and sparse GEMM implementations based on the LIBXSMM method. The first implementation significantly improves the domain of applicability as it allows unlimited number of distinct non-zeros in the sparse matrices by runtime broadcasting them at runtime from the main memory. By experiments, we measured our kernel is averagely 9% slower than the reference LIBXSMM routines evaluated using the common 170 PyFR operator matrices on an Intel Xeon Platinum 8124M machine. Our second implementation was derived from the first one. It caches some matrices strides in the vector registers to avoid redundant memory access. Our evaluation shows the second implementation is between 75.6% and 236.9% performant comparing to the reference LIBXSMM routines. For the dense PyFR operator matrices, our method shows significant performance improvement of over 30% consistently.

During the evaluation process, we observed using too many registers for accumulating the intermediate values can affect the performance negatively. We conducted a systematic micro-architectural analysis and identified the issue was caused by two instructions for memory reads and writes.

Acknowledgments

I would like to thank my supervisor Professor Paul Kelly for the guidance and support, and especially for the enthusiasm on this project. I would also like to thank Freddie Witherden of Texas A&M University and previous year student Mehedi Paribartan for their valuable feedback and advice.

Finally, I would like to thank my family back in China, and also the “family” of my wonderful housemates in Manchester. Without their support and love, this journey cannot be as enjoyable as it is right now.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Objectives | 2 |
| 1.2 | Contributions | 2 |
| 2 | Background | 4 |
| 2.1 | General Matrix Multiplication | 4 |
| 2.2 | Small and Sparse GEMM | 5 |
| 2.3 | LIBXSMM | 6 |
| 2.3.1 | The Library | 6 |
| 2.3.2 | Original LIBXSMM Small and Sparse GEMM Routine | 7 |
| 2.3.3 | Paribartan’s Improvements to LIBXSMM Small and Sparse GEMM Routine | 7 |
| 2.3.4 | Hybrid Small and Sparse GEMM Routine | 11 |
| 2.4 | PyFR | 11 |
| 3 | Related Work | 16 |
| 3.1 | GiMMiK | 16 |
| 3.2 | BLASFEO | 18 |
| 3.3 | Measurement Bias is Significant, Commonplace and Unavoidable | 21 |
| 4 | Evaluation Methodology | 24 |
| 4.1 | Evaluation Platforms | 24 |
| 4.2 | Test Matrices | 26 |
| 4.3 | Benchmark Process | 27 |
| 4.4 | Performance Metric | 28 |
| 4.5 | Limitations and Threat to Validity | 29 |
| 5 | Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory | 33 |
| 5.1 | Broadcast with VBROADCASTSS/D | 33 |
| 5.2 | Kernel Design | 34 |
| 5.3 | Evaluation | 35 |
| 5.4 | Supports for AVX2 | 51 |

| | | |
|----------|---|------------|
| 6 | Multiple Vector Registers for Accumulating C Strides | 52 |
| 6.1 | N Blocking | 52 |
| 6.2 | M Blocking | 53 |
| 6.3 | Kernel Design | 54 |
| 6.4 | Evaluation - N Blocking | 54 |
| 6.5 | Evaluation - M Blocking | 68 |
| 6.6 | Evaluation - Both N and M Blocking | 80 |
| 7 | Small and Sparse GEMM Kernel with Caching B Strides in Vector Registers | 83 |
| 7.1 | Runtime Broadcasting with FMA instruction | 83 |
| 7.2 | Kernel Design | 84 |
| 7.3 | Performance Prediction | 86 |
| 7.4 | Evaluation - Single Accumulation | 87 |
| 7.5 | Evaluation - N Blocking | 88 |
| 7.6 | Evaluation - M Blocking | 89 |
| 7.7 | Evaluation - Both N and M Blocking | 101 |
| 8 | Possible Reasons for Why Too Large M Blocking Factors Decrease Kernel Performance | 104 |
| 8.1 | Saturation of Write Buffer by Multiple Non-Temporal Stores | 104 |
| 8.2 | CPU Stalls Due to Loading B Strides | 106 |
| 8.3 | Sanity Check - Are There Any More Factors? | 110 |
| 9 | Conclusion and Future Work | 111 |
| 9.1 | Summary | 111 |
| 9.2 | Future Work | 112 |
| | Bibliography | 120 |
| A | Characteristics of PyFR operator matrices | 121 |
| B | Kernel Examples | 125 |
| B.1 | Reference LIBXSMM kernels | 126 |
| B.2 | GiMMiK Kernel | 136 |
| B.3 | Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory | 137 |
| B.4 | Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory and Caching B Strides in Vector Registers | 145 |
| C | Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory | 150 |
| C.1 | FyFR Operator Matrices | 150 |
| C.2 | Synthetic Matrices | 168 |

| | | |
|----------|---|------------|
| D | Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory and Multiple Accumulations | 178 |
| D.1 | N Blocking | 178 |
| D.2 | M Blocking | 206 |
| E | Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory and Caching B Strides in Vector Register | 234 |
| E.1 | Single Accumulation | 234 |
| E.2 | N Blocking | 254 |
| E.3 | M Blocking | 273 |

Chapter 1

Introduction

Matrix Multiplication (MM) is the core of many numerical solvers in the science and engineering domain. For problems involving solving Partial Differential Equations (PDEs), the resulting matrices are generally small (size less than an order of tens) and sparse. Despite the popular demand for small and sparse MM, the standard GEMM (GEneral Matrix Multiplication) routines provided by many BLAS (Basic Linear Algebra Subprograms) libraries do not support this in the most efficient manner. These libraries generally exploit the techniques of tiling and cache blocking, which are tuned for large matrix sizes, resulting in sub-optimal performance for the small and sparse MM [1, 2].

Our project is motivated by the use of small and sparse GEMM in PyFR [3, 4], which is an open-source Python framework for solving advection-diffusion problems using the Flux Reconstruction (FR) method [5]. The main application of PyFR is Computational Fluid Dynamics (CFD), in which the governing equations of compressible flows, namely Euler and Navier–Stokes equations, are solved. For PyFR, one of the core computations is small, sparse and block-by-panel GEMM. Because the block matrix remains constant throughout the entire simulation, a problem-specific GEMM kernel can be generated Just-In-Time (JIT) for each constant block matrix at the start of the simulation. The specialised kernels are reused throughout the simulation so the kernel generation time can be amortised.

Currently, PyFR uses two libraries for generating the bespoke GEMM kernels - an Intel open-source library LIBXSMM [1, 6] targeting x86 CPUs, and GiMMiK [7] targeting both GPU and CPU platforms. Our project aims to improve the LIBXSMM small and sparse GEMM implementation with a focus on x86 CPU platforms supporting the AVX-512 extension.

1.1 Objectives

One major limitation of the current LIBXSMM small and sparse GEMM implementation is that it does not allow a large number of distinct non-zero constants present in the sparse matrix. For matrices with too many constants, a fallback dense kernel is generated which yields sub-optimal performance for sparse problems. The reason for this is that the LIBXSMM sparse kernel pre-loads these distinct constants into the vector registers in either a broadcasted or a packed form, so these constants can be accessed quickly for GEMM computations. For the packed form, which allows more distinct constants in the sparse matrix, each AVX-512 `zmm` register (512 bits wide) can store 8 double-precision or 16 single-precision floating-point numbers. As the LIBXSMM sparse kernel uses maximally 30 `zmm` registers for storing the constants, the sparse kernel can only accommodate maximally 240 double-precision or 480 single-precision distinct non-zero constants present in the sparse matrix.

This idea of storing packed constants in the vector registers was investigated by previous year student Paribartan [8] in 2020. During his research, he reported that the LIBXSMM sparse routine is generally less performant than the LIBXSMM dense routine for moderately dense matrices with densities higher than 0.5. He concluded this is because the dense kernel is highly optimised even for moderately dense problems. It would be beneficial if we could improve the sparse kernel with techniques from the dense routine so that it can achieve better performance for these moderately dense matrices.

This brings us to the two objectives of our project:

1. Improve the LIBXSMM small and sparse kernel so that it can accommodate an unlimited number of distinct non-zeros present in the sparse matrix.
2. Improve the LIBXSMM small and sparse kernel so that it can achieve higher performance (floating point operations per second).

1.2 Contributions

This thesis makes the following contributions:

1. **An improved an automatic benchmark suite for small and sparse GEMM routines.** We developed an automatic benchmark suite for small and sparse GEMM routines based on the work from the previous year student Paribartan [8]. Our enhanced performance evaluation has improved ability of re-evaluation and better reflects the practical significance. The test suite is based on two sets of matrices - a complete set of all the 170 PyFR operator matrices and a set of 100 synthetic matrices aiming to provide a broader coverage

and a finer resolution of the matrices characteristics space. This is presented in Chapter 4.

- 2. An improved LIBXSMM’s sparse GEMM kernel which allows unlimited number of distinct non-zeros in the sparse matrix.** We improved LIBXSMM’s small and sparse GEMM routine with the technique to runtime broadcast matrix elements from memory. Comparing to the reference LIBXSMM routine, which allows maximally 240 double-precision or 480 single-precision constants present in the sparse matrix for AVX-512 (and much fewer for AVX2), our improved routine allows an unlimited number of constants in the sparse matrix for both AVX-512 and AVX2. This is presented in Chapter 5.
- 3. Exploration of multiple accumulations techniques.** We explored techniques to accumulate multiple intermediate GEMM results simultaneously in the free registers to exploit Instruction-Level Parallelism. We evaluated two techniques - N blocking and M blocking, and determined the optimum blocking factors experimentally. We reported N blocking can provide a maximum speedup of 30% for matrices with densities larger than 0.1. M blocking can provide a maximum speedup of 60% for matrices with densities larger than 0.4. This is presented in Chapter 6.
- 4. A high-performance LIBXSMM’s sparse GEMM kernel for AVX-512.** We further improved our routine with techniques to cache and reuse matrix strides in the free vector registers, and runtime broadcast matrix elements using FMA instructions. This implementation is only supported by AVX-512. With the optimum N blocking and M blocking setting, comparing with reference LIBXSMM, our implementation shows a significant performance superiority up to 136% when executing on the dense PyFR operator matrices. This is presented in Chapter 7.
- 5. A systematic micro-architectural analysis of performance bottleneck.** During our evaluation process, we observed that the kernels with large M blocking factors show unexpectedly low performance. We successfully identified the problem was caused by two instructions for memory writes and reads. We conducted a systematic micro-architectural analysis and suggested possible underlying causes for this. This is presented in Chapter 8.

Chapter 2

Background

In this chapter, we will provide the context of our project. At the start we will introduce what is GEMM with a simple vectorised implementation. This is followed by two simple but effective optimisation techniques for small and sparse GEMM. In Section 2.3, we will cover the Intel open-source library LIBXSMM with a focus on its small and sparse GEMM implementation. This section will also include improvements made by Paribartan [8] in 2020. In the end, we will provide an overview of PyFR, which greatly benefits from the LIBXSMM small and sparse GEMM routine.

2.1 General Matrix Multiplication

GEMM (GEneral Matrix Multiplication) denotes the following calculation:

$$\mathbf{C}_{m \times n} \leftarrow \alpha \mathbf{A}_{m \times k} \times \mathbf{B}_{k \times n} + \beta \mathbf{C}_{m \times n}, \quad (2.1)$$

where α and β are constants, and \mathbf{A} , \mathbf{B} , \mathbf{C} are matrices of shape $m \times k$, $k \times n$ and $m \times n$ respectively.

For CPUs supporting vector or SIMD (Single Instruction Multiple Data) instruction set extensions, GEMM routines can be vectorised for higher execution speed. As illustrated by Figure 2.1, a basic vectorised routine for GEMM can be executed as the following steps:

1. Load and scale ($\times\beta$) the first stride of \mathbf{C} into a vector register vc . The stride has the same size as the vector registers.
2. Load the first stride of \mathbf{B} into a vector register vb .

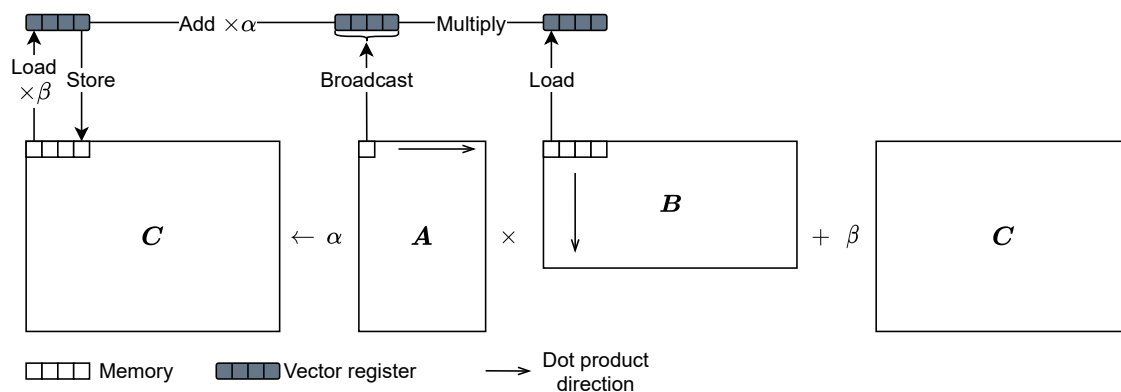


Figure 2.1: A simple vectorised GEMM routine.

3. Load and broadcast the first element of \mathbf{A} to a vector register va , such that each element in va is the first element of \mathbf{A} .
4. Element-wise multiply the vectors in va and vb . Scale ($\times\alpha$) and accumulate the result into vc .
5. Repeat steps 2 to 4 with loading the \mathbf{A} element of the next column and the \mathbf{B} stride of the next row.
6. After exhausting the entire row of \mathbf{A} (the column of \mathbf{B}), repeat from step 1 with loading the \mathbf{C} stride and the \mathbf{A} element of the next row.
7. Once the entire column of \mathbf{C} is exhausted, repeated from step 1 with loading the \mathbf{C} stride and the \mathbf{B} stride of the next stridden column.

This simple example algorithm has a time complexity of $O(m \times n \times k)$ arising from the three nested loops at steps 5, 6 and 7. It is worth noting that, matrix multiplication can be visualised as computing multiple dot products between \mathbf{A} rows and \mathbf{B} columns, as illustrated by the arrows in Figure 2.1.

For the cases when α and β are either 0 or 1, step 4 can be executed by a single FMA (Fused Multiply-Add) instruction of an x86 CPU supporting AVX extensions or above. Step 2 (explicit instruction for loading \mathbf{B} strides) can then be omitted as FMA accepts memory operands.

2.2 Small and Sparse GEMM

Small and sparse GEMM can be considered as a special case of GEMM, such that the matrix dimensions (m, n, k) are small and either \mathbf{A} or \mathbf{B} is of sparse nature. For the case of sparse \mathbf{A} , the simple GEMM routine proposed in Section 2.1 can be optimised by the following two techniques:

1. Sparsity elimination: for elements of zero in \mathbf{A} , the broadcasting and the subsequent FMA can be omitted.
2. Loop unrolling: benefited from a smaller problem size, the loops can be fully unrolled, eliminating branching cost.

2.3 LIBXSMM

2.3.1 The Library

In 2016, Heinecke et al. from Intel developed LIBXSMM (Small Matrix Multiplication LIBRARY for X86 architectures), which delivers specialised GEMM routines tailored for small and dense matrices [1, 6]. The library originally targeted x86 CPUs with AVX2 and AVX-512 extensions. It exploits an optimised tiling scheme for small GEMM which outperforms the generic BLAS routines by 10 times. For LIBXSMM, the problem is considered small if $\sqrt[3]{m \times n \times k} \leq 80$ (see Equation 2.1 for m , n and k). This threshold of 80 was later extended to 128 [9]. For problems with matrix size larger than this, a fallback GEMM routine from generic BLAS library is generated.

LIBXSMM embeds a well-tuned Just-In-Time (JIT) low-level x86 code generator for the backend. When a GEMM routine is called, the library first examines the input matrices and the code generator then dispatches the encoded/binary kernel directly into main memory. A function pointer to this kernel is returned to the caller for execution. Because this process does not require any external compiler or assembler, LIBXSMM can generate binary kernels extremely fast. The code generation process is illustrated in Figure 2.2.

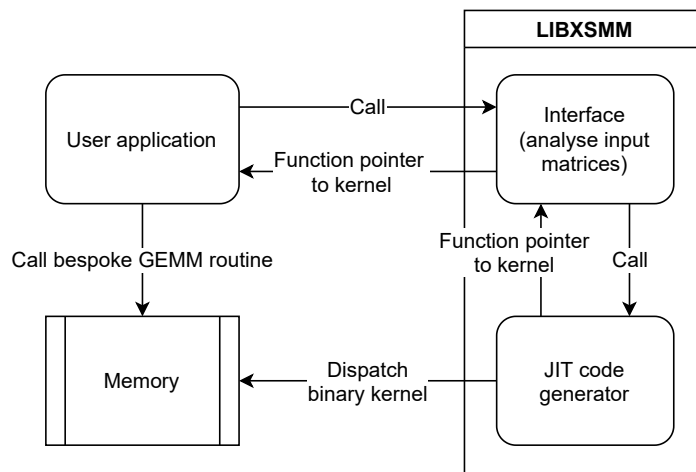


Figure 2.2: Diagrammatic representation of a LIBXSMM calling procedure. Reproduced from [8]. (JIT: Just-In-Time)

In the subsequent releases of LIBXSMM [9], its functionality was extended to the

domains of small and sparse GEMM, and small convolutions for Convolutional Neural Networks (CNN). It now also supports other SIMD architectures including SSE, AVX, AVX2, AVX-512 and AMX. The supported datatypes are DP (Double-Position) FP (Floating-Point) format, SP (Single-Position) FP format, bfloat16 (Brain Floating Point) format, 16-bit integer and 8-bit integer.

2.3.2 Original LIBXSMM Small and Sparse GEMM Routine

In addition to the two optimisation methods outlined in Section 2.2 (1. sparsity elimination, and 2. loop unrolling), the small and sparse GEMM routine of LIBXSMM exploits a technique of pre-broadcasting the absolute values of the distinct non-zero elements in \mathbf{A} to the vector registers. The GEMM is computed using either FMA or FNMA (Fused Negative Multiply-Add) instructions depending on the sign of the \mathbf{A} constants. Taking an example of a DP routine running on the AVX-512 architecture, which has 32 512-bit `zmm` vector registers, the kernel can pre-broadcast up to 31 distinct non-zero elements from \mathbf{A} to the vector registers. One additional register has to hold the accumulated \mathbf{C} stride. If the number of unique non-zeros in \mathbf{A} is less than or equal to 31, this method eliminates any additional instructions to load and broadcast the \mathbf{A} elements. Figure 2.3 illustrates this idea.

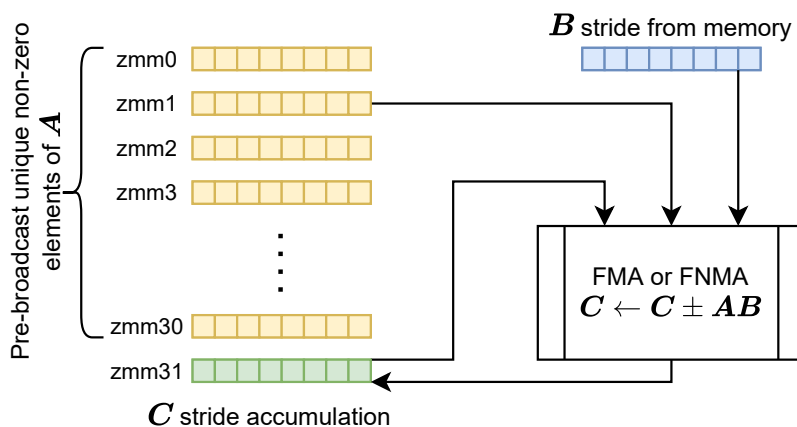


Figure 2.3: Diagrammatic representation of the default small and sparse GEMM routine of LIBXSMM. Reproduced from [8]. (FMA: fused multiply-add)

2.3.3 Paribartan’s Improvements to LIBXSMM Small and Sparse GEMM Routine

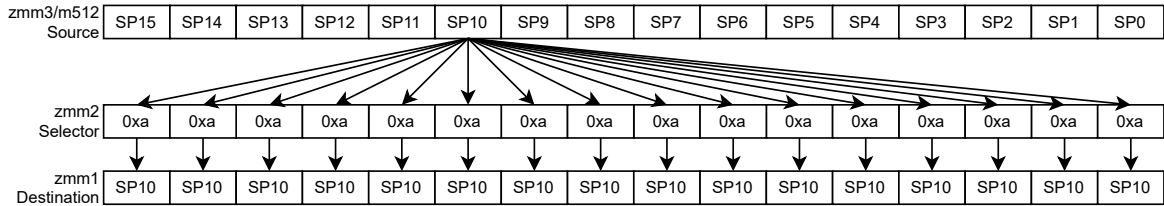
For AVX-512, the original LIBXSMM small and sparse routine only supports up to 31 unique non-zeros in \mathbf{A} , limited by the number of vector registers. It falls back to the default small and dense routine otherwise. In 2020, Paribartan [8] exploited the

idea of register packing and further extended this threshold to 240 and 480 unique non-zeros, for DP and SP respectively.

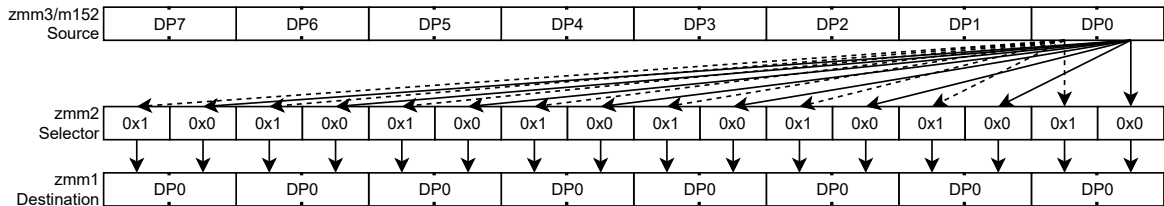
Register Packing

Register packing refers to that, instead of pre-broadcasting the non-zero elements of \mathbf{A} into the vector registers, the elements are stored in a packed form in the vector registers and are only broadcasted at runtime just before the FMAs. This allows compact packing so that each AVX-512 vector register can hold up to 8 DP or 16 SP numbers.

Paribartan exploited the instruction `VPERMD` (Permute Packed Doublewords Elements) which allows single-instruction runtime broadcasting. `VPERMD` takes three operands: a source which is either a memory location or a vector register, a selector vector register, and a destination vector register. Despite that `VPERMD` permutes doublewords, it also allows broadcasting DP numbers which are quadwords. Figure 2.4 shows the details and the required patterns of the selector register for broadcasting both DP and SP numbers.



(a) Broadcasting SP10 (single-precision floating-point number)



(b) Broadcasting DP0 (double precision floating point number)

Figure 2.4: Use of `VPERMD` for runtime broadcasting.

For DP numbers, this requires additional 8 selectors to be stored by the vector registers. As one vector register needs to hold the broadcasted value and one to hold the accumulated \mathbf{C} stride, there are 22 registers left for storing the \mathbf{A} constants, which can hold a maximum of 176 DP numbers. For SP numbers, 16 registers are to hold the selectors so there are 14 registers left for storing 224 SP numbers. By experiments, Paribartan showed that the runtime broadcasting introduces no performance penalty. For both cases, these are big improvements from the original capacity of 31. Figure 2.5 shows this idea with DP.

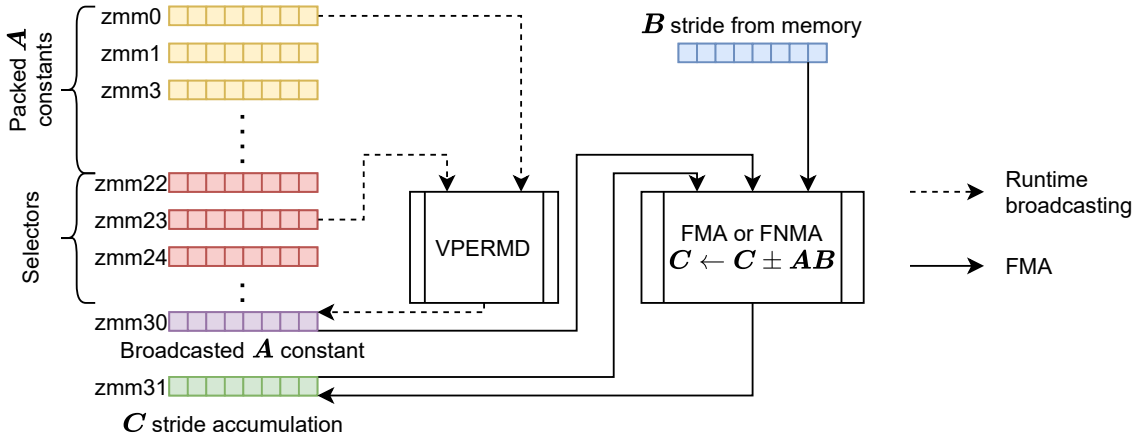


Figure 2.5: Diagrammatic representation of the small and sparse GEMM routine using register packing.

Register Packing with Selector Operands Stored in L1 Cache

To pack more distinct non-zero elements of \mathbf{A} into the registers, Paribartan experimented with storing the selector operands in memory. Because these selectors are read regularly, they are likely to present in the L1 cache and supports quick access.

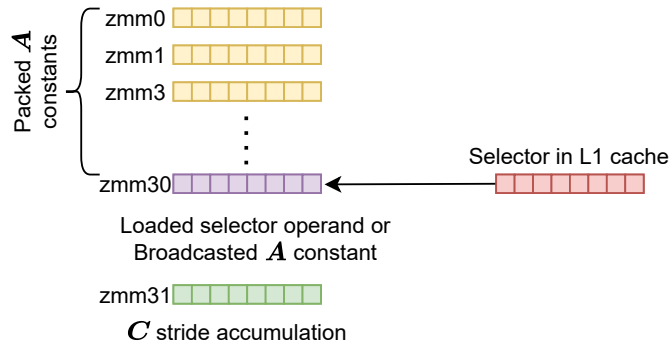
For Paribartan’s design shown in Figure 2.6, one register stores both the loaded selector operand and the broadcasted \mathbf{A} constant, and one for storing the accumulated \mathbf{C} , which leaves 30 spare registers for packing 240 DP or 480 SP.

Paribartan observed that, despite of having one additional 64-byte cache access per FMA, runtime broadcasting with the selector operands storing in the L1 cache performs no slower than the original register packing method on sparse \mathbf{A} s. Interestingly, Paribartan observed that this method with selectors storing in L1 cache could outperform the original one by up to $1.1\times$ for a few test cases. They suggested that the Skylake-SP microarchitecture could execute this kernel better by out-of-order (OOO) execution and instruction-level parallelism (ILP).

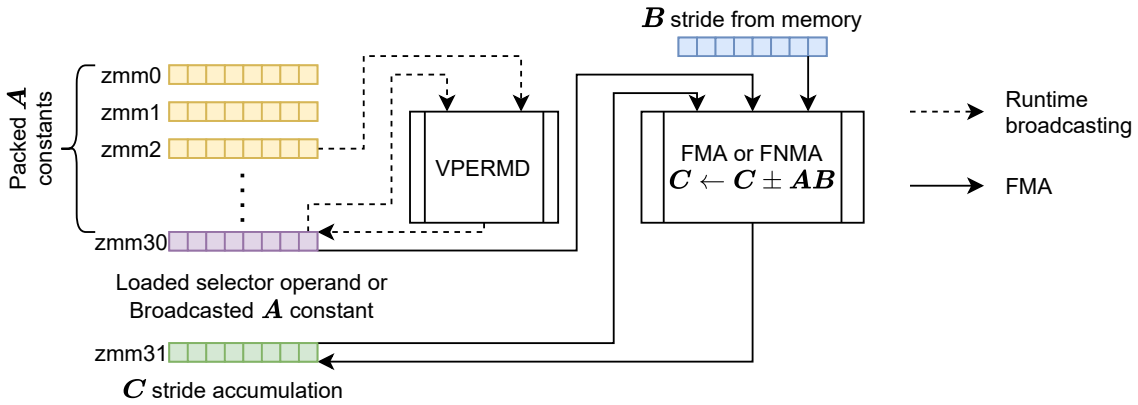
Multiple Accumulations

When the operator matrices \mathbf{A} are small, register packing leaves unused vector registers. To better utilise the free registers for these cases, Paribartan investigated two additional methods using multiple vector registers for the accumulation of the \mathbf{C} strides. These are N blocking and M blocking.

N blocking N blocking refers to that for each iteration over an \mathbf{A} constant, n_B FMA operations are performed with n_B strides of \mathbf{B} (n_B stands for N blocking factor). An example with $n_B = 2$ is illustrated in Figure 2.7. Each n_B consumes one additional



(a) Loading selector operand from L1 cache.



(b) Runtime broadcasting and FMA.

Figure 2.6: Diagrammatic representation of the small and sparse GEMM routine using register packing with selector operands from L1 cache.

vector register for accumulating the C stride. The threshold of number of unique non-zeros reduces to 232 DP or 464 SP for $n_B = 2$, 224 DP or 448 SP for $n_B = 3$, etc.

Paribartan investigated the performance of N blocking for $N = 2, 3$ experimentally [8]. They observed that for very small and sparse matrices, N blocking could provide up to $2\times$ increased performance. However, for large or dense matrices, N blocking reduced the performance which was possibly due to cache spill as more (n_B) columns of B strides were accessed regularly.

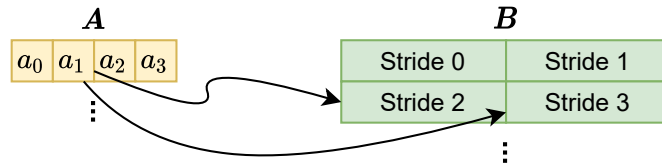


Figure 2.7: Diagrammatic representation of N blocking for $n_B = 2$. Reproduced from [8].

M blocking For M blocking, each stride of \mathbf{B} is used for m_B FMAs with different \mathbf{A} constants, as illustrated in Figure 2.8 (m_B stands for M blocking factor). Each m_B requires two more vector registers. One for \mathbf{C} stride accumulation, and one for storing the loaded selector operand and the broadcasted \mathbf{A} constant. With M blocking, the threshold of number of non-zeros in \mathbf{A} is 224 DP or 448 SP for $m_B = 2$, 208 DP or 416 SP for $m_B = 3$, etc.

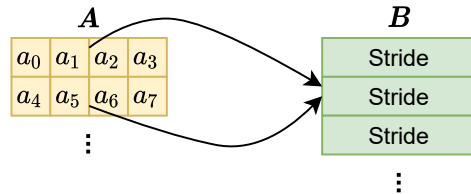


Figure 2.8: Diagrammatic representation of M blocking for $m_B = 2$. Reproduced from [8].

For the evaluations, M blocking provided either similar or reduced performance compared to the base case. Although Paribartan suggested this was likely due to L1 cache spill, we believe that it was caused by the saturation of L1 cache bandwidth as each runtime broadcasting requires a 64-byte load from L1 cache and M blocking introduces more broadcasting for each \mathbf{B} stride.

2.3.4 Hybrid Small and Sparse GEMM Routine

In a recent commit [10], LIBXSMM merges the aforementioned small and sparse GEMM implementations into a single hybrid routine, as shown by Figure 2.9. When the hybrid routine is called, it tries to generate a sparse kernel without using N blocking and a wide-sparse kernel which exploits N blocking. It also tries to generate a fallback dense kernel in case neither the sparse kernels are generated. If multiple kernels are available for the user input \mathbf{A} , LIBXSMM runs a simple benchmark and output the fastest kernel.

2.4 PyFR

Overview

As briefly described in Chapter 1, PyFR is an open-source Python library for solving advection-diffusion problems using the Flux Reconstruction (FR) method [3–5]. Currently the main application of PyFR is Computational Fluid Dynamics (CFD) in which the governing Euler and Navier-Stokes equations, are solved. Comparing to conventional CFD programs which commonly solve 1st or 2nd order Reynolds-Averaged

Navier–Stokes (RANS) equations, FR offers solutions with higher-order accuracy while still maintains the flexibility and stability of low-order methods by using unstructured grids. PyFR supports four element types: hexahedral and tetrahedral elements for 3D, quadrilateral and triangular elements for 2D. [3]

Implementation and Operator Matrices

As shown by Figure 2.10, majority of the PyFR computations are GEMMs while the rests are element-wise matrix operations [3]. For the GEMM operations (see Equation 2.1), the \mathbf{A} matrices, also known as operator matrices, are small block matrices, i.e. $m \approx k$, and remain constant throughout the entire simulation. The \mathbf{B} matrices which store the fluid states are short and wide, commonly containing 10000-100000 columns, and the contents change between the iterations. Because the constant \mathbf{A} matrices are known at the start of each simulation (also because \mathbf{A} s are commonly sparse, as we will see later), this allows PyFR to generate matrix-specific high-performant GEMM kernel at the start of each simulation. Because the kernels are reused throughout the entire simulation, the generation time is negligible.

The characteristics of the constant operator matrices depend on the following factors:

- The mesh shape: quadrilateral, hexahedron, tetrahedron or triangle.
- The quadrature method for numerical integration: Gauss-Legendre, Gauss-Legendre-Lobatto, Shunn-Ham or Williams-Shunn.
- The order of the numerical integration scheme: from the 1st to the 6th order.
- The iteration step the operator matrix is involved: m0, m3, m6, m132 or m460 (see Figure 2.10).

Appendix A lists the characteristics of the entire 170 PyFR operator matrices. These matrices are available from [6]. It is worth noting that, not only most of the operator matrices are sparse, all the operator matrices contains repeated values. This makes the LIBXSMM small and sparse GEMM routine previously discussed in Section 2.3 extremely suitable for PyFR.

Summary

In this chapter, we introduced what is GEMM and provided a simple vectorised routine. We presented the software we are aiming to improve - LIBXSMM, with a focus on its small and sparse GEMM routine. We presented the excellent work from the previous year student Paribartan [8], which significantly improved the applicability of

LIBXSMM's small and sparse implementation by his register packing technique. In the end, we provided an overview of PyFR and explained why it would benefit greatly from LIBXSMM's small and sparse routine.

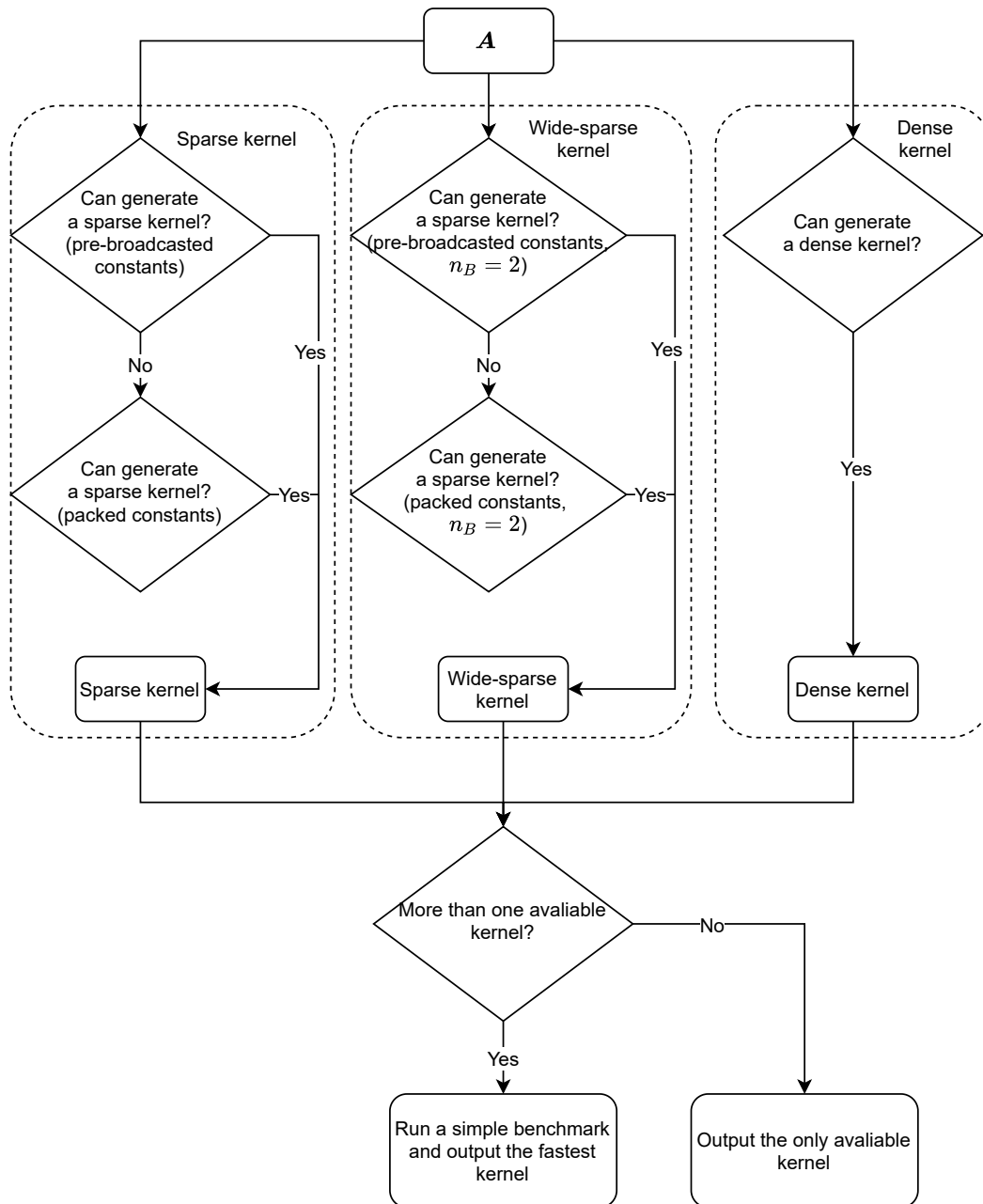


Figure 2.9: LIBXSMM hybrid small and sparse GEMM implementation.

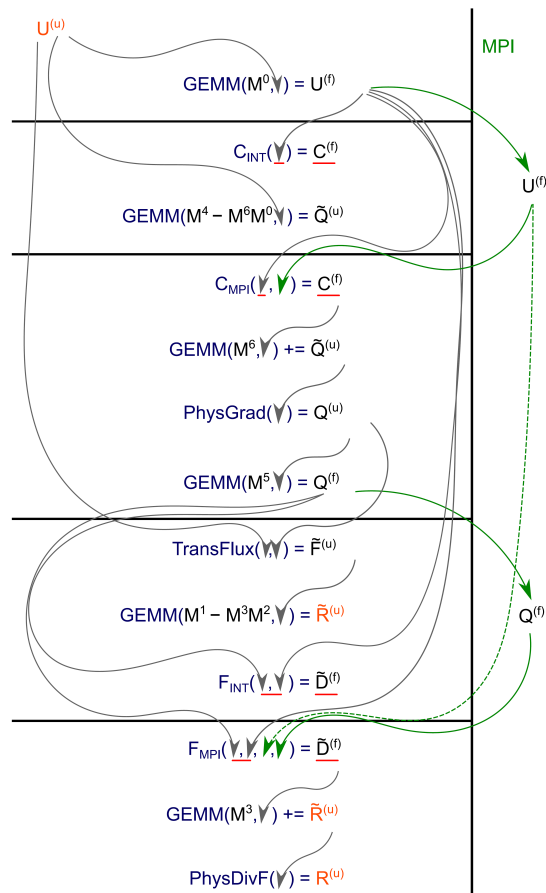


Figure 2.10: The use of GEMM and point-wise kernels in PyFR. [3]

Chapter 3

Related Work

In this section we will present two other libraries matrix operations relevant to our thesis - GiMMiK and BLASFEO. We will present the significance of measurement error and what are the best practises for avoiding them.

3.1 GiMMiK

GiMMiK is another JIT small and sparse GEMM kernel generator developed by Wozniak et al. in 2016 [7]. It originally targeted GPU platforms through CUDA and OpenCL. The hardware support was later widened to include CPU through C/OpenMP at version 2.0 [11]. As with our thesis, GiMMiK was motivated directly by the GEMMs in PyFR. Right now, it is PyFR’s default GEMM accelerator.

Unlike LIBXSMM, which directly dispatches the encoded kernel into main memory, GiMMiK generates high-level codes, e.g., *.cu and *.c, and requires a separate compilation process for generating the executables. Therefore, unlike LIBXSMM, GiMMiK depends on the compiler for code vectorisation. In 2020, Paribartan [8] observed that Intel C/C++ Compiler (ICC) was unable to fully vectorise the kernel for very big operator matrices. One example GiMMiK kernel can be found in Appendix B.2.

GiMMiK implements GEMM by a for loop which iterates through every column of \mathbf{C} . Within each iteration, there is a sub-kernel for matrix-vector multiplication, as GEMM can be decomposed into n $\underline{c}_{m \times 1} \leftarrow \mathbf{A}_{m \times k} \underline{b}_{k \times 1} + \underline{c}_{m \times 1}$ problems, where $\underline{c}_{m \times 1}$ and $\underline{b}_{m \times 1}$ are column vectors in \mathbf{C} and \mathbf{B} respectively at the same position.

For the matrix-vector multiplication sub-kernel, GiMMiK exploits the following techniques:

1. Value embedding: As shown by the kernel example in Appendix B.2, the \mathbf{A} values are embedded inside the C code as numerical values. By experiments, Wozniak

et al. [7] observed that this technique improves the performance significantly because the compiler can have explicit knowledge about the access pattern of the \mathbf{A} elements. For GPU platforms, this also allows the kernel to utilise the fast constant cache [7].

2. Sparsity elimination: As previously mentioned in Section 2.2, sparsity elimination can significantly reduce the floating-point operations (FLOPs) by removing the unnecessary computations with the zeros. Wozniak et al. [7] observed that this can significantly improve the performance. Because of lower arithmetic intensities (will be later explained in Chapter 4), Wozniak et al. [7] showed that the sparse kernel was commonly limited by memory bandwidth but not the CPU computational ability. They suggested streaming \mathbf{A} in the CSR format is not favourable because it requires additional memory bandwidth for loading the column and the row indices [7].
3. Loop unrolling: As described in Section 2.2, this is a straightforward optimisation for reducing the branching cost.
4. No cleanup code: Because generic GEMM routines usually exploit a fixed tiling scheme designed for reducing the cache and TLB miss rates, they required additional cleanup routines for dealing with the edge cases - when the matrix is smaller than the tiling size. The edge cases are commonly handled by one of the two methods - introduce additional kernel for each edge case, such as in [2], or use the same kernel with masking or zero-padding so it can be applied to non-standard shapes, as in [12]. These methods will either significantly increase the code complexity or reduce the kernel performance for these edge cases. Wozniak et al. [7] suggested that, one of the reasons for generic GEMM routine gives sub-optimal performance for small matrices is that these matrices are smaller than the fixed tiling size so they are treated as the edge cases. Using a flexible JIT approach with loop unrolling, GiMMiK avoids this problem completely.

In addition to these optimisation techniques, Wozniak et al. [7] also experimented with basic common subexpression elimination techniques. However, this was not exploited for GiMMiK because it did not improve the performance significantly.

Wozniak et al. [7] evaluated their routine using the PyFR operator matrices on two GPU platforms - a consumer class GPU GTX 780Ti and a professional GPU Tesla K40s. They reported that, for matrices with 1% density, GiMMiK outperformed cuBLAS (a BLAS implementation based on NVIDIA CUDA) by $9.98\times$ and $63.30\times$ for double-precision GEMM, on Tesla K40s and GTX 780Ti respectively. Significant improvement for single-precision GEMM was also observed. For PyFR operator matrices, GiMMiK provided significant speed up for the sparse quadrilateral and hexahedral element matrices, and for the small and dense triangular element matrices. For the large and dense tetrahedral element matrices, cuBLAS showed better performance. For example PyFR runs, GiMMiK was able to provide speed up of 1.70 and 2.19 for double-precision and single-precision simulations respectively [7].

Takeaway Points

The design of GiMMiK is highly relevant to our thesis as the small and sparse GEMM routines share similar optimisation techniques: value embedding, sparsity elimination and loop unrolling. The only difference is that GiMMiK does not account for the repeated elements in the sparse matrices. GiMMiK has a higher portability than LIBXSMM, as it generates kernels in high-level language. However, this is at the expense of relying on the compiler for generating high-performant vectorised codes. The evaluation results indicate the flexible JIT approach is very suitable for our targeting application and can outperform generic GEMM routines significantly.

3.2 BLASFEO

BLASFEO stands for Basic Linear Algebra Subroutines for Embedded Optimisation which is a specialised BLAS- and LAPACK- like library optimised for real-time optimisation tasks common for embedded software applications, developed by Frison et al. [12] in 2017. Level 3 BLAS and LAPACK routines are the core computations for 2nd order optimisation problems. Within the context of embedded optimisations, these matrices are generally small and dense. BLASFEO was developed aiming to provide optimised BLAS and LAPACK routines for small and dense problems [12].

As opposed to LIBXSMM and GiMMiK which are JIT kernel generators, BLASFEO kernels were implemented in the form of a stand-alone library. BLASFEO consists of three implementations: 1. a high-performance implementation BLASFEO HP, 2. a portable reference implementation BLASFEO RF, and 3. a wrapper to standard BLAS and LAPACK library BLASFEO WR. BLASFEO HP aims to deliver the best performance possible so the core kernels are coded in assembly. By the time of this thesis, BLASFEO HP support x86 processors with AVX-512, AVX2, AVX, and SSE, and ARM Cortex-A processors with NEONv2 and NEON. BLASFEO RF is a portable version of BLASFEO HP. It is coded in C and preserves some optimisation technique of BLASFEO HP which can be implemented using the high-level language. BLASFEO WR is a wrapper to standard BLAS and LAPACK library. It aims to provide high-performant routines for large matrices while still maintains the BLASFEO interface [12].

BLASFEO HP employs the following techniques:

1. Register blocking (tiling): Register blocking is a common optimisation method for matrix operations. LIBXSMM exploits register blocking through N and M blocking (see Section 2.3). As we will further discussed in Chapter 6, matrix operations commonly involve repeated loads and saves of some intermediate values. This introduces data dependency between instructions, leading to performance bound by instruction latency. This is sometimes known as C-slowness. With reg-

ister blocking, the instructions can be reordered so data hazard stalls are avoided. Additionally, register blocking allows the CPU to temporally cache some matrix elements in the register so they can be reused to reduce redundant memory traffic. In Chapter 7 we will show how this can be achieved by M blocking. Frison et al. [12] also considered cache blocking, which is a tiling technique used by GotoBLAS [2] to ensure the active matrix chunks fit in the cache or TLB. Cache blocking was not exploit as the matrices are already small for BLASFEO [12].

2. Vectorisation: Many processors support Single Instruction, Multiple Data (SIMD) instructions, which are commonly deeply pipelined for an increased throughput. It is beneficial for matrix operation routines to employ the SIMD instructions, a.k.a. be vectorised, for a better performance.
3. “Panel-major” matrix: Standard BLAS and LAPACK libraries usually exploit packing - the matrix elements are reordered such that the elements will be access contiguously by the kernels. Including the packing operation would also be beneficial for BLASFEO, however, Frison et al. [12] suggested the performance improvement is insignificant for small matrices. This is because the packing process has a quadratic time complexity relative to the matrix dimensions, while the matrix multiplication has a cubic complexity. For small matrices, the additional packing cost cannot be well amortised by the matrix multiplications. Instead of doing implicit packing, BLASFEO has an option to accepted packed matrices from the users. BLASFEO calls this storing format as “panel-major” format. With this option, users can explicitly pack and unpack the matrices before and after a series of matrix manipulations. Between these manipulations, the matrices are stored in the panel-major format so the cost of packing and unpacking can be amortised. Figure 3.1 shows how panel-major format works for an example GEMM operation.

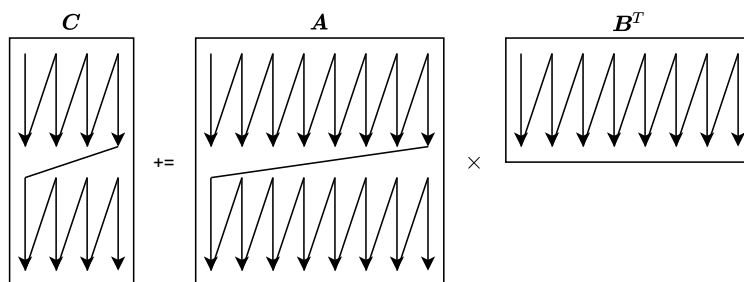


Figure 3.1: The panel-major memory layout for an example GEMM operation. Within each panel, elements are ordered in column-major format. (The GEMM is “NT” variant, which is the optimal one for column-major layout.) [12]

4. Assembly subroutines: As implied by the name, BLASFEO are implemented as several subroutines which coordinate with each other. These subroutines are coded in assembly so they can have flexible and efficient calling conventions such that the arguments can be passed by the vector registers. The calling conventions of high-level languages, such as C, have high overhead as these values have to be written to memory before calling, and read from memory after calling. [12]

- Cleanup codes: Unlike LIBXSMM and GiMMiK, BLASFEO does not use JIT approaches so cleanup codes for corner cases are required. BLASFEO deals this with three different variants of kernels. The first one is the nominal variant which has a smaller blocking size. The nominal kernels provide uncompromising performance but it is not flexible. Because complexities, nominal kernels are only included for a few corner cases. The second type is variable-size variant, which is adapted nominal variant with masking. The last one is generalised variant, which is adapted variable-size variant allowing matrix elements to be non-aligned. [12]

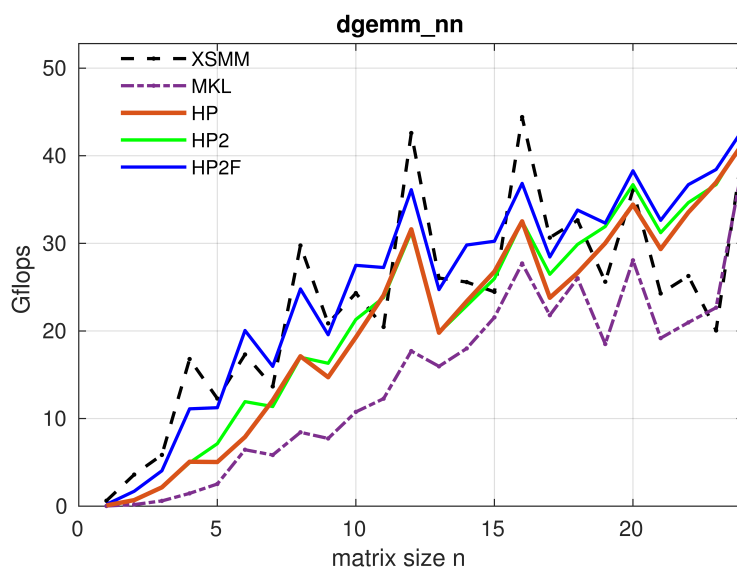


Figure 3.2: BLASFEO HP vs LIBXSMM. Evaluated on Intel Haswell processor. HP2 stands for an improved BLASFEO HP implementation which has more nominal variants dealing with the edge cases. HP2F stands for a flexible, JIT-like BLASFEO HP implementation which fully unroll the loops for maximum flexibility against edge cases. [12]

By experiments, Frison et al. [12] reported BLASFEO HP can achieve significant better performance than standard BLAS and LAPACK libraries - 20%-30% better performance for BLAS routines and 200%-300% better performance for LAPACK routines. As shown in Figure 3.2, Frison et al. [12] also evaluated the performance of BLASFEO HP GEMM kernel against the LIBXSMM’s one, together with adapted fully-unrolled BLASFEO kernels using JIT-like approach. For small matrices, BLASFEO HP was less performant than LIBXSMM while the fully-unrolled BLASFEO kernel showed comparable performance to LIBXSMM. Because of this, Frison et al. [12] suggested the flexible JIT approach employed by LIBXSMM is inherently better than the library approach of BLASFEO for small matrix operations by eliminating the corner cases.

Frison et al. [12] also evaluated the BLASFEO RF implementation. Unlike the HP one, BLASFEO RF was unable to compete with standard BLAS and LAPACK libraries for most of the routines. This indicates the compilers are unable to generate highly optimised vectorised codes, so performance critical kernels should be coded in assembly for maximum execution speed.

Takeaway Points

BLASFEO shares similarities with LIBXSMM as they both target optimised small matrix operations. BLASFEO aims to provide BLAS- and LAPACK- like routines while LIBXSMM only focus on GEMM. The experiments conducted by Frison et al. [12] strengthen the idea that LIBXSMM's JIT approach is more suitable for small GEMM as it eliminates the edge cases. The kernels should be generated in binary format for best performance.

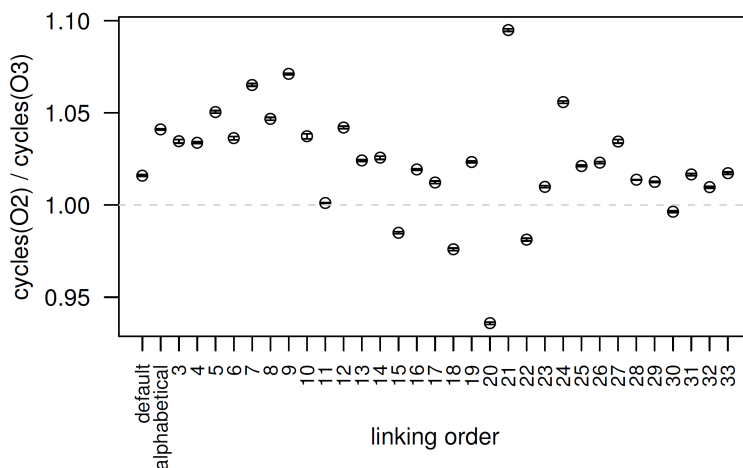
The idea of panel-major matrix format is maybe irrelevant to LIBXSMM, but we suggest it could be highly beneficial for PyFR. We suggest PyFR could store the state matrix B s and C s in this format for the entire simulation process for better execution speed, and only unpack the matrices when the final results are required. This would increase cache and TLB usage efficiency significantly as the memory accesses are contiguous.

3.3 Measurement Bias is Significant, Commonplace and Unavoidable

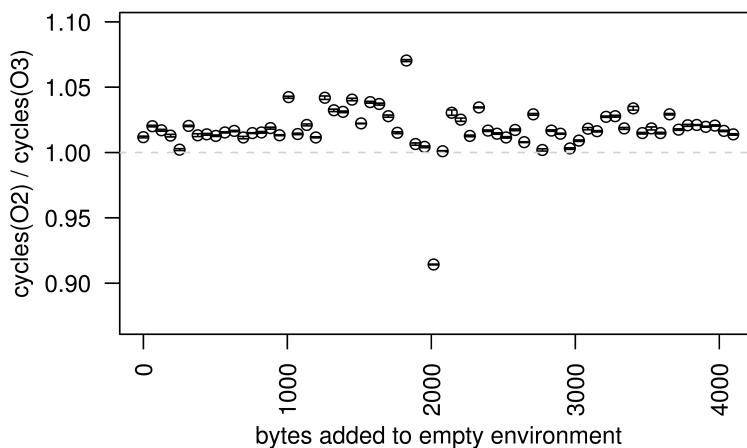
The performance of computer software is known to be sensitive to memory access layout. In 2009 Mytkowicz et al. [13] investigated if innocuous characteristics of benchmark experiment could introduce measurement bias. By experiments, Mytkowicz et al. [13] investigate the effects of two aspects - the size of UNIX environment variables and the linking order. As UNIX environments are loaded before loading the program, changing the environment size will alter the location of call stack. Changing the linking order would affect the memory arrangement of the data and code segments.

Mytkowicz et al. [13] set up an experiment which measured if changing the aforementioned aspects would affect the measured speedup provided by GCC optimisation option `O3` relative to the option `O2`. The experiment covered a big range of benchmark software - all the CINT and CFP tests of SPEC CPU2006 benchmark suite. The experiment was conducted on three different hardware - Intel Core 2 and Pentium 4, and a m5 O3CPU simulator.

By experiments, Mytkowicz et al. [13] observed that the measurement error induced by these innocuous aspects of the experiment setup is significant. Figure 3.3 shows the speedup measurement of Perlbench benchmark obtained from the Intel Core 2 machine. Both of the settings can introduce fluctuated results in the range between 0.90 and 1.10. These measurement biases are also commonplace - they were observed for all the benchmark software and on all the testing platforms [13]. These biases are also reproducible - they cannot be eliminated by repeating the measurements under the same setup.



(a) Varying linking order.



(b) Varying UNIX environment size.

Figure 3.3: The measured speedup of GCC optimisation option O3 on Perlbench benchmark with varying innocuous aspects of the experiment setup. [13]

From the field of social and natural science, Mytkowicz et al. [13] collected three methods for avoiding falling into measurement bias. The first idea is to have a diverse and comprehensive suite of benchmark software. Therefore, any bias arising from a single benchmark can be factored out. However, Mytkowicz et al. [13] showed that even their collection of all the CFP and CINT tests in SPEC CPU2006 was unable to factor out all the bias. The second method is to repeat the experiments with randomised experiment setup. Any conclusion should be drawn using statistical procedures such as t test. The final method is using casual analysis for testing any hypothesis. If we are testing a hypothesis that X causes Y. Casual analysis requires to test a derived system which does not have X but contains every other aspect of the original system. If this test does not show Y behaviour, we can be more confident that X causes Y. [13]

Takeaway Points

Mytkowicz et al. [13] showed that measurement bias is significant and commonplace, and more importantly, they are unavoidable. Any insignificant performance changes are potentially be caused by measurement bias. For our project, if the benchmark suite is not carefully designed, validity of this thesis cannot be assured. Our experiment should target a comprehensive and diverse set of matrices, and should be evaluated on different hardware. We should aim to validate our hypothesis systematically using casual analysis.

Summary

In this chapter we presented GiMMiK and BLASFEO. GiMMiK is another JIT bespoke GEMM kernel generator targeting small matrix multiplies. BLASFEO is a stand-alone library provides optimised BLAS- and LAPACK- like routines for small and dense matrices. The evaluations of GiMMiK and BLASFEO show that the JIT approach is inherently more suitable for small matrix operations comparing to the library approach of BLASFEO and other standard BLAS libraries. BLASFEO employs several optimisation techniques which we can exploit for LIBXSMM. We also presented that measurement bias is significant and unavoidable. Care should be taken when designing our experiment to avoid the bias.

Chapter 4

Evaluation Methodology

As previously described in Section 3.3, software performance measurements are extremely sensitive to the testing environment. An ill-designed benchmark suite would introduce bias to the measurement data, so it might not reflect the actual performance seen by the users. More importantly, this would lead to faulty data been analysed in the later phase of this project, resulting in wrong optimisation decisions.

In this chapter, we present the design and the reasoning behind our comprehensive test suite, which primarily focus on PyFR applications while still provides insights into other general use cases.

4.1 Evaluation Platforms

Skylake Machine

For this project, the evaluation platforms were virtual machine instances hosted on Amazon Web Service (AWS) Elastic Compute Cloud (EC2). All test machines were running on Ubuntu 20.04 LTS. Our primary test machine was *c5n.xlarge* instance which provides exclusive access to 2 physical cores (4 logical cores because of hyper-threading) of a dual-socket Intel Xeon Platinum 8124M system, running at 3.0 GHz base frequency. The CPU, which is based on Skylake-SP microarchitecture (SP stands for Scalable Performance), has 32 KB L1 instruction cache and 32 KB L1 data cache per physical core. Both the L1 caches are 8-way set associative and the cache line sizes are 64-byte. The L1 data cache supports a maximum bandwidth of 128-byte load and 64-byte store per cycle. Skylake-SP also provides a 16-way set associative 1 MB L2 cache per core, which is shared by both instruction and data. The 11-way set associative L3 cache has a size of 1.375 MB per core totalling to 24.75 MB, and is shared by all the cores on a single socket.

`c5n.xlarge` supports AVX-512 extension and features 2 FMA units per core, each of which can perform 16 double-precision floating-point operations (FLOPs) per cycle. As `c5n.xlarge` instance provides no means of controlling CPU clock frequency, we measured the CPU clock and observed it always stabilised at exactly the 3.0 GHz base frequency when executing the matrix multiplication kernels (see Section 4.3). This results in a theoretical maximum of 96 GFLOP/s per physical core. The practically achievable performance was measured using Intel LINPACK benchmark provided with the Intel Math Kernel Library (MKL) [14]. A single-thread LINPACK benchmark reported 85.1125 GFLOP/s as the double-precision peak performance.

`c5n.xlarge` provides access to 10.5 GB DDR4 random-access memory (RAM) running at 2666 MHz. The memory bandwidth for a single core was determined using the STREAM Triad benchmark [15] which measures the performance of a simple FMA kernel ($a[i] = b[i] + q*c[i]$). The Triad benchmark was compiled using Intel C/C++ Compiler (ICC) with `-O3 -xCORE-AVX512 -qopenmp -DSTREAM_ARRAY_SIZE=80000000 -DNTIMES=20` flags and executed with `OMP_NUM_THREADS=1` for a single-thread benchmark. The Triad benchmark reported a peak memory bandwidth of 12.93656 GB/s.

Cascade Lake Machine

To avoid experimental bias favouring Skylake microarchitecture, all the benchmarks were also executed on EC2 `m5c.xlarge` instance which runs on 2 physical cores of a dual-socket Intel Xeon Platinum 8259CL system based on Cascade Lake-SP. The processor has a base frequency of 2.5 GHz. Comparing to `c5n.xlarge`, `m5n.xlarge` has exactly the same cache configuration, except a larger L3 cache of 33.0 MB due to a higher core count.

Similar to `c5n.xlarge`, `m5n.xlarge` supports AVX-512 and has 2 FMA units per core. We measured that matrix multiplication kernels were executed at exactly the processor base frequency. The theoretical maximum performance of `m5n.xlarge` is 80 GFLOP/s. The practically achievable performance was evaluated using single-thread LINPACK benchmark, which was 70.5427 GFLOP/s.

`m5n.xlarge` provides 16 GB DDR4 RAM running at 2666 MHz. The single-core STREAM Triad benchmark reported a peak memory bandwidth of 12.73666 GB/s.

Metal Skylake Machine

During this project, sometimes we were in need to profile the microarchitecture events, for instance, cache miss and page fault rates. Both `c5n.xlarge` and `m5n.xlarge` did not support this as they are virtual machines. For this reason, we configured another EC2 `c5n.metal` instance which allows complete access to the entire physical node (dual-socket Intel Xeon Platinum 8124M) and hardware event counters. We used Linux

`perf` for simple and Intel VTune Profiler for more complicated profiling jobs.

Software Environment

For the evaluation process, all test machines were running on Ubuntu 20.04 LTS. The reference LIBXSMM library is the latest version on 17/07/2021, with the hash: 13550e3d68a7df0d5415c51eb8b7cd3194008219. The LIBXSMM was compiled using GNU GCC, but the compiler choice should not affect our results. During the experiments, we noticed that for some big \mathbf{A} matrices, a LIBXSMM sparse kernel cannot be generated because of hitting the LIBXSMM maximum kernel binary size limit. Therefore we increased the kernel size limit to 2 MB when evaluating our custom routines. The kernel size limit was not increased for the reference LIBXSMM as we would like to reflect how the reference library performs in its default setting. Because of this, the readers are reminded that for some matrices, a direct comparison between the reference and our custom routines are irrelevant, due to the difference in kernel size limits.

4.2 Test Matrices

PyFR Operator Matrices

Two different sets of matrices were used for evaluating the matrix multiplication kernels. The first one being the complete set of the operator matrices used in PyFR.

As described in Section 2.4, there are in total 170 operator matrices. The characteristics of these matrices only depend on the element shape, quadrature type, the order of integration scheme, the iteration step that the operator matrix is involved in, and remain constant from run to run. These matrices were available from the code samples of the LIBXSMM repository [6]. We have checked and confirmed these matrices were the same as the operator matrices used in the example runs provided in PyFR repository [4]. The detailed characteristics of the PyFR operator matrices are included in Appendix A.

Synthetic Matrices

Although the PyFR matrices set could provide accurate insight into how the kernels perform in PyFR, it does not have enough breadth to cover more general use cases. It also lacks the potential to reveal subtle relations between kernel performance and matrix characteristics because two operator matrices are usually different in more than one property. Following this reasoning, we adapted and included a synthetic set of

operator matrices from Paribartan [8], aiming to explore the matrix characteristic in a more controlled manner.

The synthetic set is generated randomly and uniformly from three base matrices, which are 128×128 in shape, have a density of 0.05, contain 16, 64 and 256 unique absolute non-zeros (U), respectively. Each synthetic matrix is different from the base ones in only one property. The matrices properties are as follows:

- Varying number of rows $R = 32, 64, 128, 256, 512, 1024$, for each U
- Varying number of columns $C = 32, 64, 128, 256, 512, 1024$, for each U
- Varying density $\rho = 0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0$, for $U = 16, 64$
- Varying density $\rho = 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0$, for $U = 256$
- Varying number of unique absolute non-zeros $U = 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256$

There were in total 100 operator matrices in the synthetic set.

4.3 Benchmark Process

In PyFR, the GEMM kernels are used with $\alpha = 1$ and either $\beta = 0$ or $\beta = 1$. For LIBXSMM kernels, β only determines if the vector registers are initialised as zeros or strides from \mathbf{C} . The kernel performance is dominated by the numerous FMA instructions. Therefore, we only tested the case with $\alpha = 1$, $\beta = 0$ and expect kernels with $\beta = 1$ to have similar performance. Only double-precision kernels were tested for similar reasoning.

For this project, the performance was expressed in units of FLOP/s, which was calculated from the time taken for executing the kernels. To reduce the measurement error, we measured the duration for running each individual kernel repeatedly for 60 times and the average was calculated. We used LIBXSMM service functions `libxsmm_timer_duration` and `libxsmm_timer_ncycles` to measure the kernel duration and number of CPU cycles, from which we also obtained the averaged CPU clock speed.

Our test setup does encourage cache hits as the matrices are likely to present in the cache after the initial iteration. This reflects realistic utilisation as for PyFR the GEMMs use the results directly from the previous step making the cache possibly hot (see Figure 2.10). This also reduces stress on the memory system so that potential

CPU bottlenecks can be revealed. To utilise most of the CPU resources and to avoid context switching, the benchmark process was prioritised by using `nice -n -20`.

In PyFR, matrix \mathbf{B} stores the fluid states and typically contains 10000-100000 columns [3]. To exploit multiprocessing of machines running PyFR, \mathbf{B} is divided into chunks of 48 columns, which are taken by identical and independent GEMM kernels for parallel execution [8]. For our benchmark, each kernel was tested with randomly generated \mathbf{B} s with the number of columns of 9600, 33600, 57600, 81600 and 105600. As with PyFR, the \mathbf{B} s were fed into the kernels in chunks of 48 columns. For each kernel, an averaged performance was calculated using weighted arithmetic mean (W.A.M.) weighted with execution time. Multi-threading was not exploited and the process was enforced to run in a single logical core by using `taskset -c 0`.

The operating system could also exert randomness and inconsistency to the experiment results. One reason for this is that the memory page mapping is different from run to run. Considering this, the experiment was repeated 10 times using shell-level loops and the averaged performance was reported as the final result.

4.4 Performance Metric

Pseudo-FLOP/s

FLOP/s is the commonest performance matrix for reporting GEMM kernel performance in the literature [2, 9, 12]. However, it does not fit the context of this project as most of the test matrices are sparse and the bespoke kernel contains no unnecessary FMA with zeros. Therefore, we adapted the idea of pseudo-FLOP/s from [8] which only accounts for the FLOPs arising from non-zero elements.

For sparse matrix multiplication, the number of pseudo-FLOPs can be calculated from problem dimensions (m , n , k) and matrix density (ρ):

$$pseudo-FLOPs = 2 \times \rho \times m \times n \times k. \quad (4.1)$$

Here we consider one FMA on a single number contributes two FLOPs - one from multiplication and one from addition. Pseudo-FLOP/s is then calculated as:

$$pseudo-FLOP/s = \frac{pseudo-FLOPs}{execution\ time}. \quad (4.2)$$

For the rest of this thesis, Pseudo-FLOP/s is the default metric for reporting kernel performance.

Arithmetic Intensity

Another important metric used in this project is arithmetic intensity, which is calculated as:

$$\text{arithmetic intensity} = \frac{\text{pseudo-FLOPs}}{\text{theoretical minimum amount of RAM access}}. \quad (4.3)$$

Unlike pseudo-FLOP/s, arithmetic intensity is a property of the matrix multiplication problem and it is independent of the kernel. Arithmetic intensity is powerful in determining the theoretically achievable performance for a problem and in indicating if a kernel speed is limited by the RAM bandwidth or the CPU computing power.

Similar to pseudo-FLOP/s, the formula to calculate the amount of memory access for our small and sparse matrix multiplication problems was modified to account for the sparsity of matrix \mathbf{A} . The theoretical minimum amount of RAM access was computed according to these rules:

- There is no memory read for loading \mathbf{A} , as it is already encoded in the binary matrix multiplication kernel.
- Memory read for loading \mathbf{B} elements at the i th row is counted if and only if the i th column of \mathbf{A} is non-empty.
- Memory write for storing \mathbf{C} elements at the i th row is counted if and only if the i th row of \mathbf{A} is non-empty.
- There is no memory read for loading \mathbf{C} , as $\beta = 0$ and the instructions for storing \mathbf{C} use a non-temporal hint.

4.5 Limitations and Threat to Validity

Single-Threaded Benchmark

One major limitation of our evaluation is that the kernels were run single-threaded. For this project, this is desired as we would like to focus on investigating and improving individual kernel performance. However, for realistic scientific computational workloads, the kernels are commonly run on multiple threads to exploit the amount of multiprocessing available on modern high-performance computers.

In a multi-thread setting, multiple kernels are sharing the L3 cache and memory bandwidth meaning each kernel is more likely to be limited by memory bottleneck.

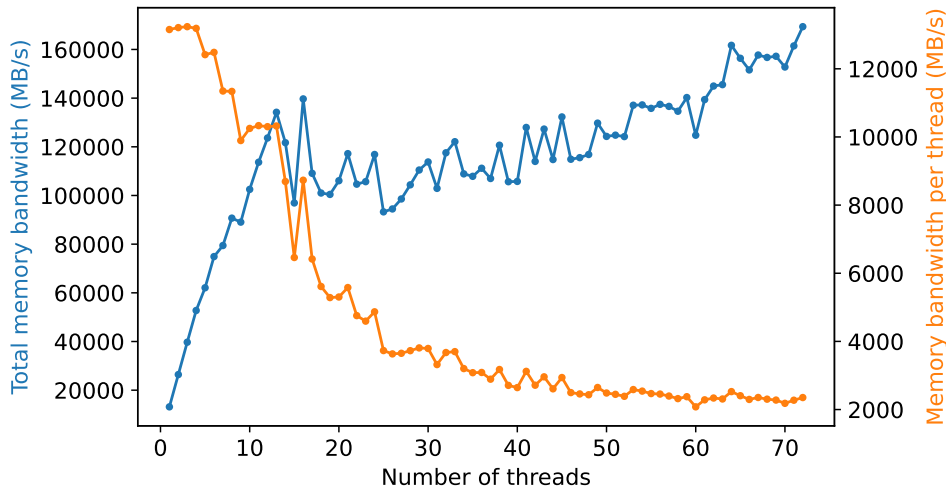


Figure 4.1: Total and per-thread memory bandwidth available across varying numbers of threads. Results measured by STREAM Triad benchmark on the m5c.metal machine.

Figure 4.1 shows the total and per-thread available bandwidth with different total numbers of threads, measured on c5n.metal. For running 72 threads, the per-thread bandwidth is 17.8% comparing to only running 1 thread. For this reason, the readers are reminded that the measured performance in this report could be higher than the per-thread performance in multi-thread settings.

It is also worth noting that, for our evaluation, we might have observed a higher clock rate than what the users might see when running the GEMM kernels using multiple cores. This is because if all the cores are heavily utilised, the CPU might lower the clock speed to control its operating temperature.

Sharing the Physical Node with Other Users

The test suite was hosted remotely on EC2 virtual machines, implying all the benchmark processes were subject to virtualisation overhead. More importantly, the test suite was sharing the computer resources, for example, memory bandwidth and L3 cache, with other virtual machines hosting on the same physical node.

To further investigate the effects of this, we experimented on benchmarking the reference LIBXSMM small and sparse kernel on three different c5n.xlarge instances. As shown in Figure 4.2, the results from different instances show a similar trend, but for some tests, there were up to 15% difference between the runs.

Because of this, we avoided directly comparing the raw kernel performance between different virtual machine instances. The experimental data was first compared to the reference LIBXSMM performance measured immediately before the benchmark process on the same instance and the ratio between these was used for comparing

across different virtual machines.

Limited Test Platform Choice

The last limitation comes with the fact that we were only evaluating the kernels on a narrow selection of micro-architectures, which are Skylake-SP and Cascade Lake-SP. The main reason behind this is the newness of AVX-512 extension so it is only available on high-end Intel processors running on Skylake or newer architectures. This presents little problem to this project, as we were targeting kernels using AVX-512 instructions.

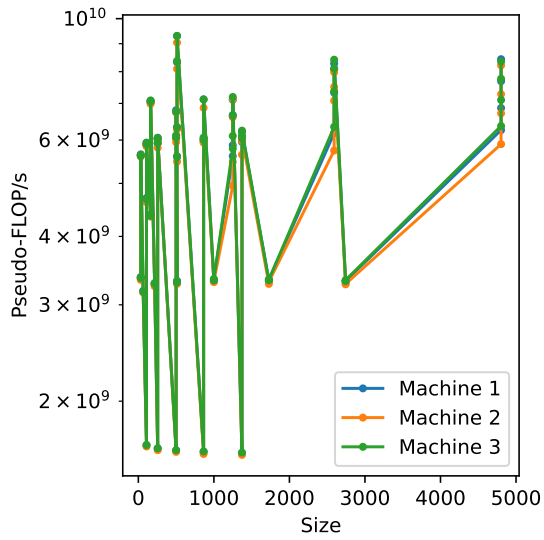
However, as we will later discuss in Chapter 5, some of our methods can be also applied to AVX2, which is very widely supported by x86 processors from both Intel and AMD. We were not able to evaluate the AVX2 kernels. In Section 9 later we will further discuss the benefits of implementing and evaluating support to AVX2 as an extension to this project.

Summary

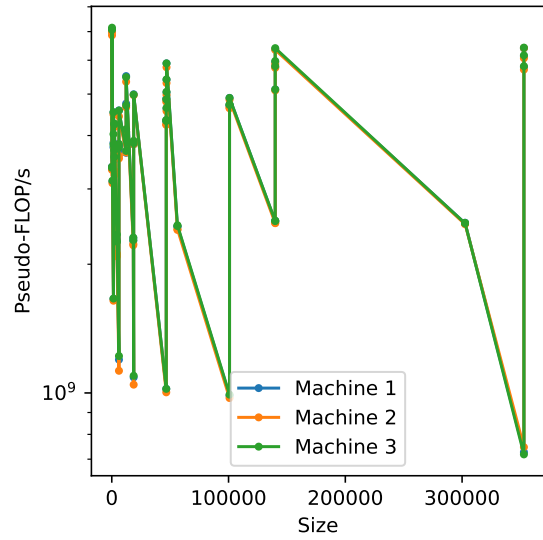
In this section, we discussed and presented our comprehensive benchmark suite. The test machines were hosted remotely on AWS EC2. To avoid bias towards a single microarchitecture, we chose c5n.xlarge (Intel Skylake-SP) and m5n.xlarge (Intel Cascade Lake-SP) instances as the test machines. A c5n.metal instance was also configured for hardware event profiling.

There were two sets of test \mathbf{A} matrices. The first set contains the complete 170 PyFR operator matrices and the second set consists of 100 synthetic matrices aiming to provide a finer resolution in the matrix characteristic space. Only the case of double-precision GEMM, $\alpha = 1$ and $\beta = 0$ was tested in this project. To reduce the measurement error, we measured the duration to run the kernel 60 times repeatedly. Each kernel was tested with 5 randomly generated \mathbf{B} matrices with different widths. We used weighted arithmetic means for calculating the average performance. We later presented two important metrics for reporting performance, namely pseudo-FLOP/s and arithmetic intensity.

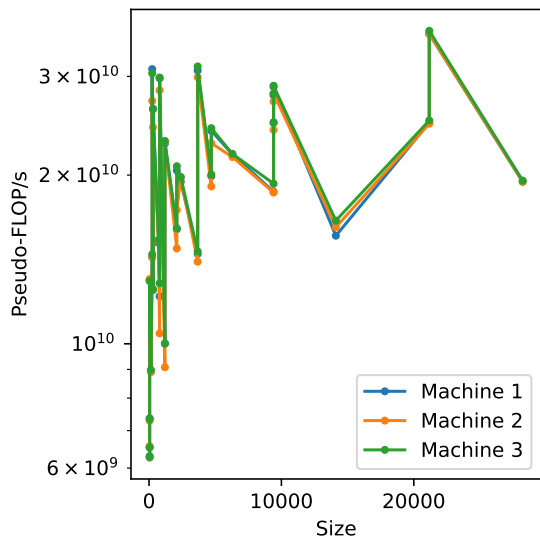
Our evaluation has three limitations. We run the kernels single-threaded so the results are not representative of a multi-thread setting. Additionally, our test programs were sharing the physical node with other virtual machine users, resulting in the inconsistency of the measured performance by up to 15%. Lastly, our benchmark suite cannot evaluate AVX2 kernels, leaving the benefits of these kernels unexplored.



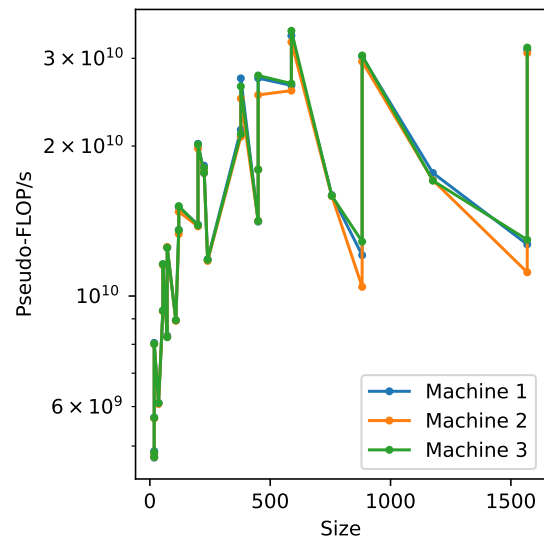
(a) \mathbf{A} for quadrilateral element.



(b) \mathbf{A} for hexahedral element.



(c) \mathbf{A} for tetrahedral element.



(d) \mathbf{A} for triangular element.

Figure 4.2: The performance of reference LIBXSMM small and sparse kernel against \mathbf{A} size. Results were obtained from 3 different EC2 c5n.xlarge instances.

Chapter 5

Small and Sparse GEMM Kernel with Runtime Broadcasting Packed \mathbf{A} Constants from Memory

In Chapter 2 we introduced the original LIBXSMM small and sparse GEMM routine designed for x86 processors supporting AVX-512. The kernel pre-broadcasts \mathbf{A} constants to vector registers so can accommodate a maximum of 31 distinct absolute non-zero \mathbf{A} values. In 2020, Paribartan [8] improved the LIBXSMM routine by packing the constants in vector registers, which are only broadcasted at runtime before each FMAs using `VPERMD`. This allows the kernel to handle up to 240 double-precision or 480 single-precision \mathbf{A} constants.

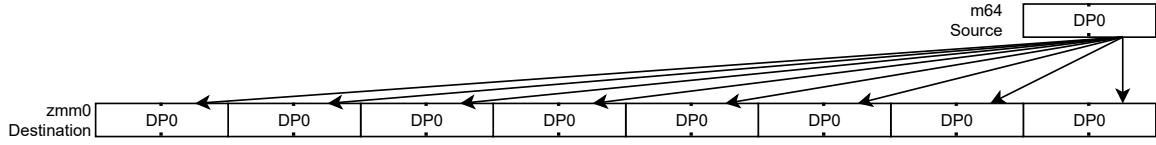
It is beneficial to further optimise the routine such that it can be applied to \mathbf{A} containing more constants. In this chapter, we present the design and evaluation of our routine which supports an unlimited number of distinct absolute values in \mathbf{A} . Instead of storing the \mathbf{A} constants in the vector registers, our kernel packs them in the main memory and uses `VBROADCASTSD` or `VBROADCASTSS` for runtime broadcasting.

Although this method does not show superior performance comparing to reference LIBXSMM, it is a fundamental building block for our high performance implementations which we will present in the Chapter 6 and 7 later.

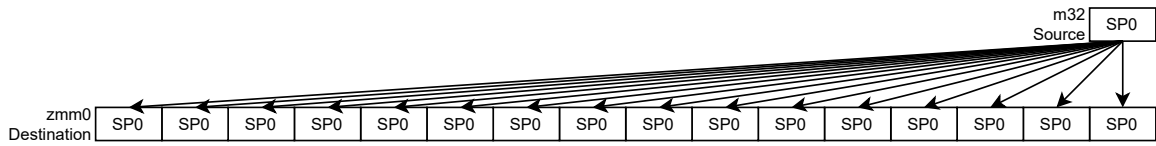
5.1 Broadcast with `VBROADCASTSS/D`

AVX, AVX2 and AVX-512 support single-instruction load and broadcast with `VBROADCASTSD` and `VBROADCASTSS` for double-precision and single-precision floating-point numbers respectively [16]. `VBROADCASTSS/D` takes two operands: a source operand which is either a memory location or a vector register, and a destination vector register

operand. For our kernel, we exploit the setting when the source operand is a memory location. As shown by Figure 5.1, this loads and broadcasts the data at the memory location to all locations in the destination vector register.



(a) VROADCASTSD for double-precision numbers.



(b) VROADCASTSS for single-precision numbers.

Figure 5.1: Runtime broadcast with VROADCASTSD and VROADCASTSS.

As tabulated in [17], our use of VROADCASTSS/D has a latency of 4 cycles. The maximum throughput is 2 instructions per cycle.

Broadcast Elements from Vector Registers with VROADCASTSS/D

The source operand of VROADCASTSS/D can also be a vector register. For this case, the lowest element in the source register is broadcasted to the destination register, as illustrated in Figure 5.2. Although this setting is not applied to our kernel, it could be beneficial for Paribartan’s register packing technique [8] by simplifying the operations for broadcasting the first packed number in the registers.

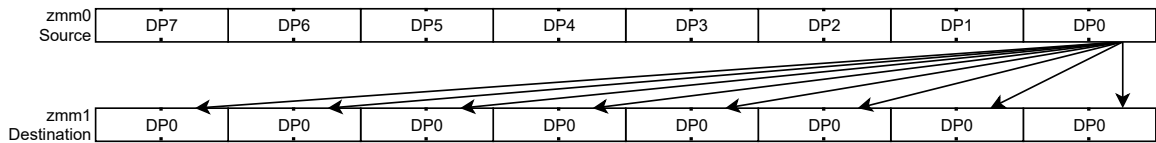


Figure 5.2: VROADCASTSD with vector register as the source operand.

5.2 Kernel Design

Figure 5.3 illustrates how we utilise VROADCASTSS/D in our matrix multiplication kernels. Comparing to the LIBXSMM routines which either pre-broadcasts or stores packed \mathbf{A} constants in the vector registers, our routine pre-packs \mathbf{A} constants in memory. These constants are loaded and broadcasted with VROADCASTSS/D at runtime

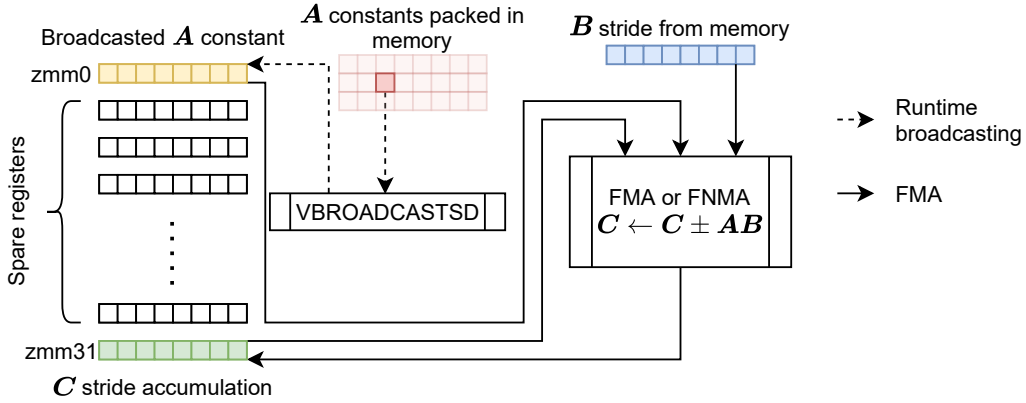


Figure 5.3: Diagrammatic representation of the small and sparse double-precision GEMM routine using `VBROADCASTSD` runtime broadcasting with \mathbf{A} constants stored in memory.

for FMA. The packing operation, which has a time complexity of $O(\rho \times m \times k)$, is performed when generating the kernel, so that the cost can be amortised when the kernel is executed repeatedly.

Because \mathbf{A} is small and sparse, the packed \mathbf{A} constants only occupy a small segment of memory (less than 192 bytes for quadrilateral and hexahedral PyFR operator matrices). Together with the fact that these constants are accessed regularly, we expect them to present in the L1 cache allowing fast runtime broadcasting. As shown in Figure 5.3, our routine requires two vector registers, leaving 30 vector registers unused. `zmm0` stores the broadcasted \mathbf{A} constant and `zmm31` accumulates the \mathbf{C} stride. As the number of \mathbf{A} constants is no longer limited by the number and size of vector registers, the kernel allows an unlimited number of distinct absolute non-zero values in \mathbf{A} .

For our routine, each FMA operation requires two memory reads - one for loading 4- or 8-byte \mathbf{A} constant and one for loading 64-byte \mathbf{B} stride. For Paribartan’s register packing method with the selector operands (64-byte) stored in memory [8], each FMA operation requires two 64-byte reads. Therefore we expect our kernel to induce less stress on L1 bandwidth so should perform not slower than Paribartan’s method.

An example kernel is included in Appendix B.3.

5.3 Evaluation

In this section we present the benchmark results of our GEMM routine, comparing to reference LIBXSMM. A detailed description of our evaluation method can be found in Chapter 4. All kernels were tested on two different sets of \mathbf{A} matrices - a set of PyFR operator matrices and a set of synthetic matrices.

As described in Section 2.3, when called, the reference LIBXSMM always tries to generate three kernels - a default sparse kernel, a wider sparse kernel (N blocking of 2) and a dense kernel. If multiple kernels are generated, it tests which kernel performs the fastest and returns the best kernel. For the case when only one kernel is generated, it returns that only kernel. The plots in the report reflect this feature of the reference LIBXSMM library by marking different types of kernels with different markers.

PyFR Operator Matrices

In this section, we only present the roofline and performance data plotting against \mathbf{A} density (ρ) and the number of \mathbf{A} constants (U) measured on the c5n.xlarge machine. Results from m5n.xlarge are very similar so there are not included in this section. For performance data plotting against the number of \mathbf{A} columns, the number of \mathbf{A} rows and \mathbf{A} size, and the results from m5n.xlarge, please refer to Appendix C.

Quadrilateral Element Matrices

Figure 5.4a shows the performance of the reference LIBXSMM routine and our method plotting against the number of distinct absolute values in \mathbf{A} . As indicated by the markers, the reference LIBXSMM kernels are wide-sparse for the majority of the cases. No dense kernel was generated due to the sparsity of the quadrilateral element matrices, as indicated by Figure 5.4b. One important feature to notice is that all the reference LIBXSMM kernels use the routine which pre-broadcasts \mathbf{A} constants into the vector registers. These are supposedly the fastest sparse routines in LIBXSMM as they have the least amount of memory access.

Comparing with the reference kernels, our method shows comparable performance for many matrices. As shown by the roofline plot Figure 5.4c, these matrices result in very small arithmetic intensity. For these problems, the performance is largely limited by memory bandwidth. For some of the matrices with more than 10 unique constants, our method performs worse than the reference library by around 15%. The extreme case is when $U = 18$, in which our method is 31% slower than the reference one. We suggest two possible reasons for this. Firstly, our method requires one more 8-byte memory read per FMA comparing to the reference method. This exerts pressure on the memory system and resulted in a performance decrease. Secondly, our method uses only one register for accumulating the \mathbf{C} strides after each FMA. As the accumulated values are used for later FMAs as a source operand, this creates data dependency between the FMA instructions. For AVX-512, although FMA instructions can achieve a maximum Instructions Per Cycle (IPC) of 2 [17], the throughput is possibly limited by the 4-cycle latency due to pipeline stall because of the data hazard. The processor possible exploits techniques such as operand forwarding to mitigate the effect of data hazard, but for deeply pipelined Single Instruction Multiple Data (SIMD) instructions such as FMA, operand forwarding might provide only limited improvement. As shown

in Figure 5.4c, the matrices showing a slow down on our method have relatively large arithmetic intensity. This indicates that for these problems, the performance is more likely to be limited by the CPU, such as instruction latency, rather than memory bandwidth.

For some matrices, our method performs faster than the reference routine by up to 10%. There is no obvious reason for this. We suggest this could be because computer performance is extremely, randomly sensitive to memory access pattern [13] and our method indeed introduces a different memory access pattern. Because the speedups are rather insignificant, we did not further investigate this. On average, our method is 97.6% performant comparing to reference LIBXSMM for quadrilateral PyFR matrices.

In the roofline plot shown in Figure 5.4c, some data points are exceeding the memory roofline. These points are produced by small \mathbf{A} s. We suggest the reasoning behind this is that some of \mathbf{B} remain in the cache during benchmark iterations. This was confirmed by including a cache flush function into the benchmark suite, which declares and assigns random values to a dummy array with a size larger than the L3 cache. With this setup, almost all the data points are either on or below the memory roofline.

Hexahedral Element Matrices

Figure 5.5a shows the performance of our method comparing to reference LIBXSMM when executing the PyFR hexahedral element matrices. Comparing to quadrilateral element matrices, hexahedral ones are generally larger in shape and have lower density. Similar to the quadrilateral ones, all hexahedral element matrices have less than 31 unique absolute non-zero constants so that the fastest LIBXSMM sparse routine which pre-broadcasts the constants was used for all the matrices. The wide-sparse kernel is generated for most of the cases. One exception of this is the 6th order Gauss-Legendre m460 matrix. Although only having a 2% density, the matrix is too big in shape (1029×343) making the fully unrolled sparse kernel hit the kernel size limitation. A fallback dense kernel is generated which is 85% slower than our sparse method with no kernel size limitation.

Similar to the quadrilateral element matrices, our method shows alike performance comparing to the reference routines for small arithmetic intensity. For matrices with arithmetic intensity larger than 2^{-1} , our method tends to perform no faster than the reference routines. For some of the matrices, our method is around 12% slower. The extreme case is $U = 11$, when our method performs 19% slower. Same as the quadrilateral element matrices, we suggested the slowdown is due to additional loads for runtime broadcasting the \mathbf{A} constants and pipeline stalls because of the data dependency between the FMA instructions. Excluding the 6th order Gauss-Legendre m460 matrix, our method is 92.3% performant comparing to the reference library for hexahedral PyFR matrices.

Figure 5.5c shows the roofline plot. Comparing to the quadrilateral element matrices,

all the data points are further away from the memory roofline. We suggested this is because the hexahedral element matrices are larger in size. This results in larger \mathbf{B} s therefore lower spatial locality for loading the \mathbf{B} elements. This would increase cache miss rates, decreasing the kernel performance.

Tetrahedral Element Matrices

Figure 5.6a shows the performance of our kernel comparing to the reference routines for FyFR tetrahedral matrices. Unlike the sparse hexahedral and quadrilateral element matrices, the tetrahedral ones are dense. As shown in Figure 5.6b, the minimum density is 50%, while most of the matrices have a density between 80% and 100%. For tetrahedral element matrices, most of the reference LIBXSMM kernels are the dense one. There are two reasons for this. Firstly, neither the sparse nor wide-sparse routine can accommodate more than 240 \mathbf{A} constants in the vector registers. Secondly, for the cases when a sparse kernel can be generated, the dense kernel could perform faster as it is highly optimised for dense problems. However, it is interesting to note that, for some very dense matrices, the wide-sparse kernel is generated instead of the dense one, for example, the three wide-sparse kernels generated at $\rho = 1$ shown in Figure 5.6b. These kernels belong to problems with relatively small arithmetic intensities, so the performance is memory bandwidth bound. For this case, the bottleneck is not the FMA instruction latency so the dense routine cannot improve the performance with multiple accumulations. The sparse kernel also fully unrolls the loops, eliminating any penalty of branching. In addition, the sparse kernel avoids any memory traffic for loading \mathbf{A} constants, which is favoured in this memory bandwidth bound case.

Comparing with the reference routines, our method is slower for the majority of the data points, especially for the problems with large arithmetic intensity. The extreme case is $U = 1760$, when our kernel is 55% slower than the dense kernel. After examining the dense kernel, we suggested three reasons for its superior performance. Firstly the dense kernel always tries to use as many vector registers available for accumulating the \mathbf{C} strides. This hides the FMA instruction latency and is crucial for dense matrices as the performance is not limited by memory bandwidth. Secondly, the dense kernel keeps some \mathbf{B} strides in vector registers so that they can be reused for multiple FMAs without accessing the main memory. In fact, the dense kernel also uses the FMA instruction differently. For AVX-512, the FMA instruction can runtime broadcast one quadword (DP number) or doubleword (FP number) memory operand right before the FMA computation [16]. Therefore, runtime broadcasting can be achieved by AVX-512 FMA instructions, freeing the registers for storing broadcasted \mathbf{A} elements. In Chapter 7 we explained how we integrated this method with our routine. A reference dense kernel example can be found in Appendix B.1. Lastly, the dense kernel exploits a tiling scheme in all m , n and k directions to increase the temporal locality for accessing the matrices. Our method can be greatly benefited from these techniques for relatively dense problems. On average, our routine is 41.2% slower than reference LIBXSMM for tetrahedral PyFR matrices.

Interestingly, although the dense kernel is highly optimised, it can only reach 37% of the performance potential of the test machine as shown in the roofline plot (Figure 5.6c). This implies room for further optimisation improving the dense routine. Similar to the quadrilateral element matrices, there are data points above the memory bandwidth roofline. We confirmed this is due to small matrices remain in the cache during the benchmark iterations by repeating the experiment with the cache flushing function.

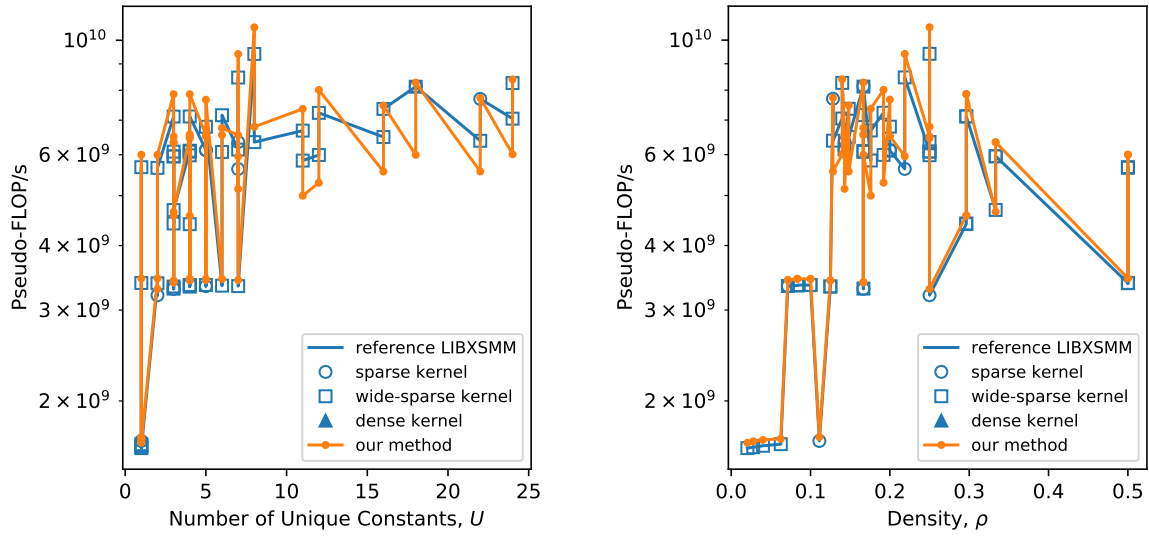
Triangular Element Matrices

Figure 5.7a shows the performance of our method comparing to reference LIBXSMM. Comparing to dense tetrahedral FyFR operator matrices, the triangular ones have a similar density, as shown in Figure 5.7b. However, the matrices are generally small in size and have a smaller number of unique constants, resulting in lower arithmetic intensities. Because of this, comparing to tetrahedral element matrices, more sparse kernels can be generated, even for fully dense matrices. Our reasoning for this is similar to that for the tetrahedral element matrices. As these dense matrices have relatively small arithmetic intensity (see Figure 5.7c), the performance is limited by memory bandwidth. Therefore, the dense kernel’s techniques such as multiple accumulations cannot improve the performance. This finding reinforces the idea that both our and LIBXSMM’s sparse routines are superior for problems with small arithmetic intensity, whatever the matrix density is.

Similar to the quadrilateral element matrices, our routine performs better than the reference sparse/wide-sparse routine for some matrices by around 10%. As shown in Figure 5.7c, this performance superiority mostly happens for arithmetic intensities between 2^0 and 2^1 . The reasoning for this is unclear but we suggest it could be due to performance sensitivity to memory access patterns.

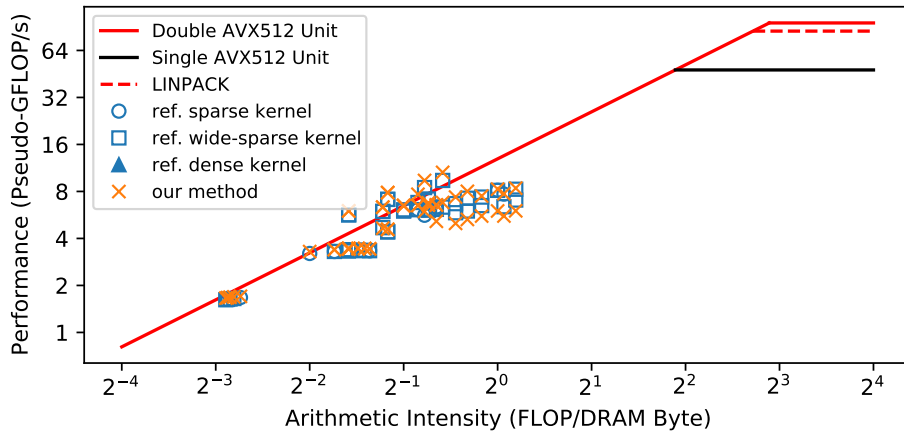
For matrices with relatively large arithmetic intensities, the dense routine is more performant than our method. Similar to tetrahedral element matrices, the dense kernels exploit multiple \mathbf{C} stride accumulations and reusing \mathbf{B} strides stored in vector registers, resulting in better performance when it is not limited by memory bandwidth roofline.

This concludes our results and evaluations for our benchmark using the PyFR matrix set.



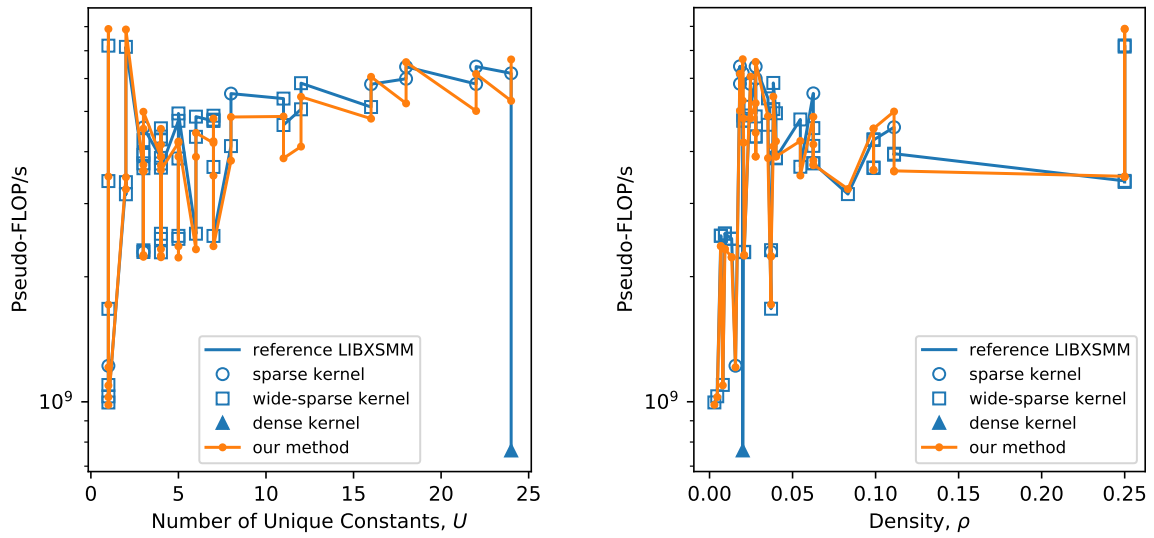
(a) Performance against number of unique A constants.

(b) Performance against A density.



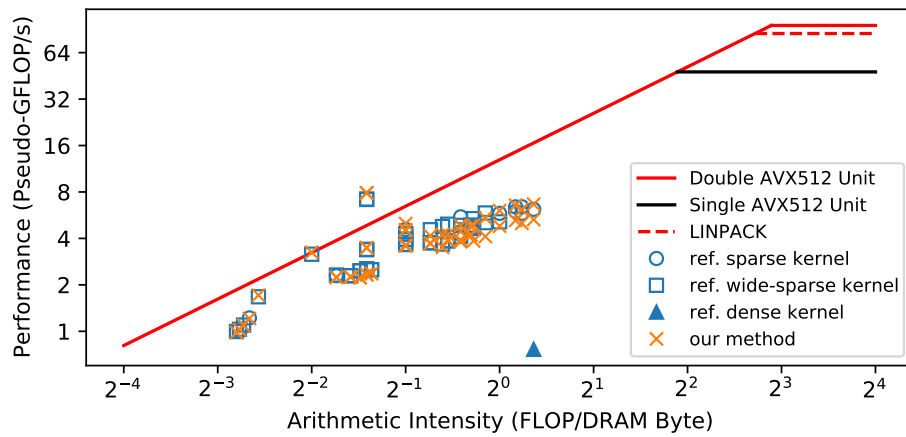
(c) Roofline plot.

Figure 5.4: Runtime broadcasting with loading A from memory vs. reference LIBXSMM implementations, for PyFR quadrilateral element operator matrices.



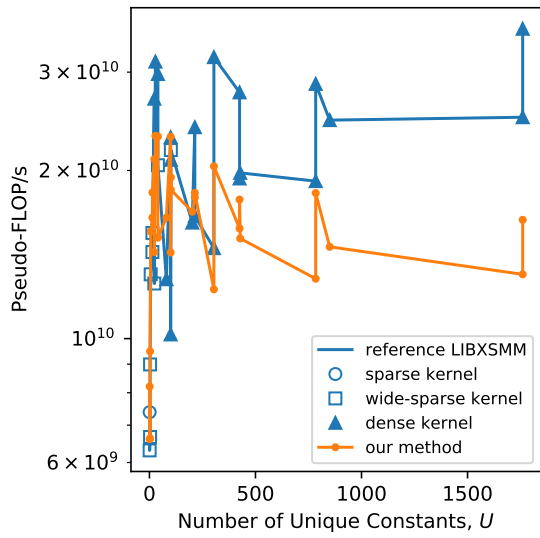
(a) Performance against number of unique A constants.

(b) Performance against A density.

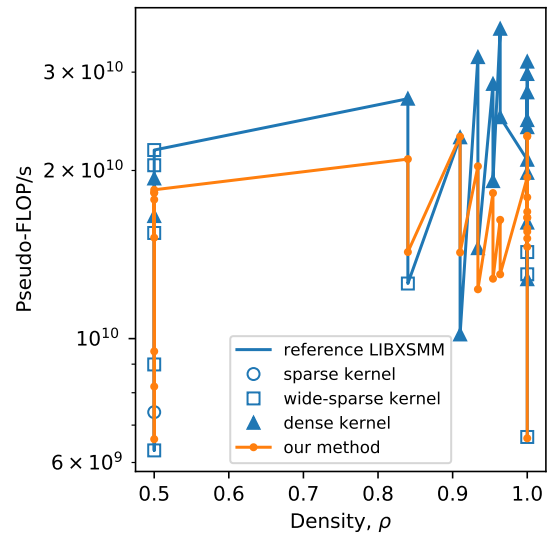


(c) Roofline plot.

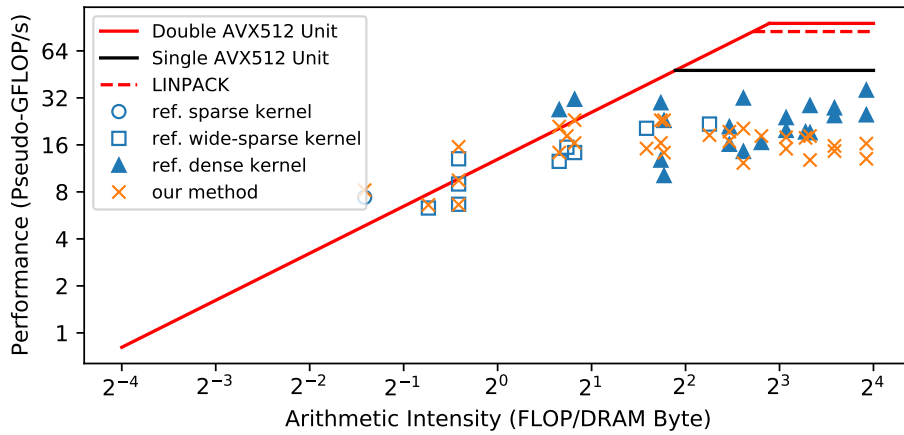
Figure 5.5: Runtime broadcasting with loading A from memory vs. reference LIBXSMM implementations, for PyFR hexahedral element operator matrices.



(a) Performance against number of unique \mathbf{A} constants.

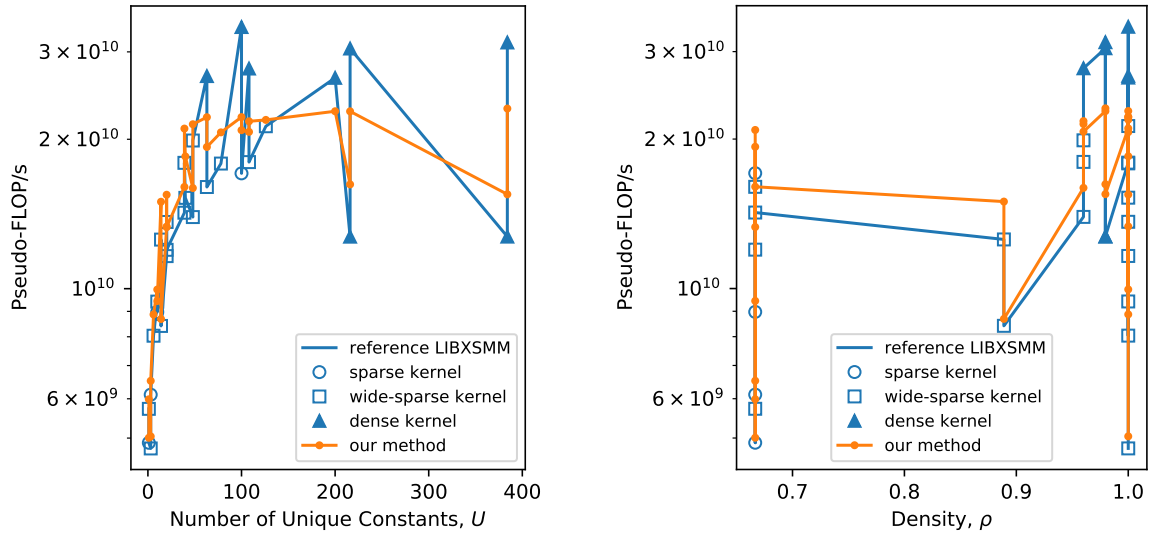


(b) Performance against \mathbf{A} density.



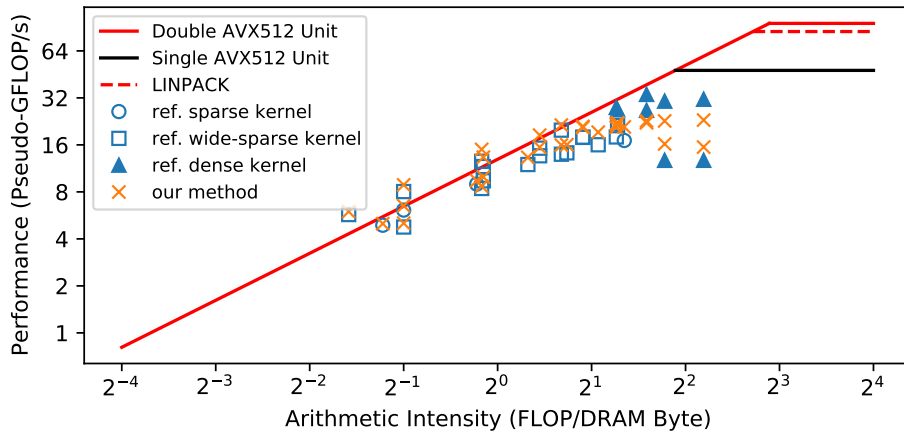
(c) Roofline plot.

Figure 5.6: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR tetrahedral element operator matrices.



(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.



(c) Roofline plot.

Figure 5.7: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR triangular element operator matrices.

Synthetic Matrices

This section presents a comparison between our routine and reference LIBXSMM evaluating using the synthetic matrix set. The synthetic matrices are generated from three base matrices, containing 16, 64, 256 distinct absolute non-zero values respectively. Comparing to the PyFR set, the synthetic set explores the \mathbf{A} characteristic space in a more controlled manner, as each matrix only differs from the base one by one property. The varying properties are: number of rows (R), number of columns (C), density (ρ) and number of unique absolute non-zero constants (U). A complete description of the synthetic set can be found in Section 4.2.

Similar to the evaluation on the PyFR set, results from m5c.xlarge (Cascade Lake-SP) match the results from c5n.xlarge (Skylake-SP). Therefore, for this section only the c5n.xlarge results are presented and discussed. For complete experiment results including the m5c.xlarge ones, please refer to Appendix C.

Varying Number of \mathbf{A} Columns

Figure 5.8a, 5.8c and 5.8e show the performance of our method comparing with reference LIBXSMM, evaluating using the synthetic matrices with 16, 64 and 256 \mathbf{A} constants (U), respectively. Figure 5.8b, 5.8d and 5.8f present these results in roofline plots. For both $U = 16$ and $U = 64$, the performance of both our method and reference routine increases with more number of columns, until 512 columns. The performance gain is due to an increase in arithmetic intensity, which gradually lifts the memory bandwidth bottleneck. The Skylake processor we were testing on has 32 KB of L1 data cache per core, which can accommodate 4096 double-precision floating-point numbers (assuming idealised fully-associate cache). This is equivalent to 512 rows of \mathbf{B} strides, each containing 8 numbers. Therefore, for \mathbf{A} with more than 512 columns, the \mathbf{B} column stride cannot be fully accommodated by the L1 cache, increasing the L1 cache miss rate. This explains why the performance decreases for $C > 512$. As shown in Figure 5.8a, our method shows a larger performance decrease comparing to the reference method. We suggest this is because our method requires more memory traffic for loading and broadcasting the \mathbf{A} constants, so it is more prone to cache miss penalty. For $U = 16, 64$ and $2^5 < C < 2^8$, the reference sparse and wide-sparse routine show 6% - 12% better performance than our method. This is because the reference routines do not require loading \mathbf{A} constant from memory and the wide-sparse kernel has 2 accumulation registers for hiding FMA instruction latency.

For $U = 64$ and $C = 2^{10}$, the reference method generates a dense kernel despite the matrix is sparse. This is because both the sparse and wide-sparse kernels are hitting the size limitation. Similarly, a sparse kernel is generated for $U = 16$ and $C = 2^{10}$, as the wide-sparse kernel is too large.

For $U = 256$, reference LIBXSMM can only generate dense kernels as the sparse

routines only allow a maximum of 240 \mathbf{A} constants. For these cases, our method is 200% more performant than the dense routine, as the dense method spends most of its FMA instructions on zeros.

Varying Number of \mathbf{A} Rows

Figure 5.9a, 5.9c and 5.9e show the kernel performance with varying numbers of \mathbf{A} rows (R). Figure 5.9b, 5.9d and 5.9f present the same data but using roofline plots. For all U , our method shows an increasing performance with increasing R . This is because of the increase in arithmetic intensities as shown in the roofline plots. For $U = 16, 64$, the reference method initially shows 10% less performance than our method for $R \leq 2^6$. For larger U , the reference method performs better than our method by up to 28% as ours is likely limited by FMA instruction latency. For very large R ($R = 2^{10}$ for $U = 16$, $R = 2^9, 2^{10}$ for $U = 64$), the wide-sparse kernel cannot be generated because of hitting the kernel size limitation. For these cases, alternative sparse or dense kernels are generated, resulting in lower performance.

For $U = 256$, no reference sparse kernel can be generated. Our method outperforms the dense kernel by around 180%.

Varying \mathbf{A} Density

Figure 5.10a, 5.10c and 5.10e show the performance of our method comparing to the reference LIBXSMM library. Figure 5.10b, 5.10d and 5.10f present the same results in roofline plots. For $U = 16$ and $U = 64$, our method shows an increasing performance with increasing ρ , and converges to 1.3×10^{10} pseudo-FLOP/s at $\rho = 0.5$, limiting by FMA instruction latency. The reference method generates wide-sparse kernels for very small ρ . For $\rho \leq 0.05$, these sparse kernels have a similar performance to our method, as they are bottlenecked by memory bandwidth. For larger ρ , the wide-sparse kernels outperform our method by up to 30% as they hide instruction latency by multiple accumulations. For ρ at around 0.25, the wide-sparse kernel can no longer be generated because of hitting the kernel size limitation. Alternative sparse kernels are generated, which show similar performance to our method. For $\rho \geq 0.4$, neither the sparse nor wide-sparse kernel is generated because of kernel size limitation. The fallback dense kernels are generated, showing an increasing performance with increasing ρ and increasing arithmetic intensity. For large ρ , the dense kernel is up to 135% faster than our method. Similar to earlier results, this is because the dense method utilises multiple accumulations, an optimised tiling scheme, and stores \mathbf{B} strides in vector registers for faster repeated access.

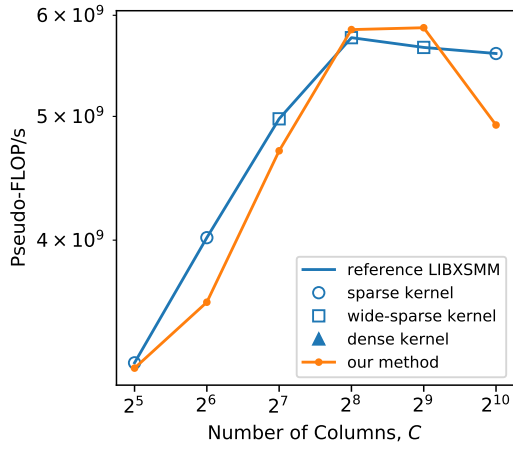
For $U = 256$, the reference library cannot generate any sparse or wide-sparse kernel for this many \mathbf{A} constants, so alternative dense kernels are generated. The dense kernels are less performant than our method for $\rho \leq 0.4$ as most of their FMA operations are

computed with zeros. For larger ρ , the dense kernel outperforms our method as it is more optimised for dense matrices.

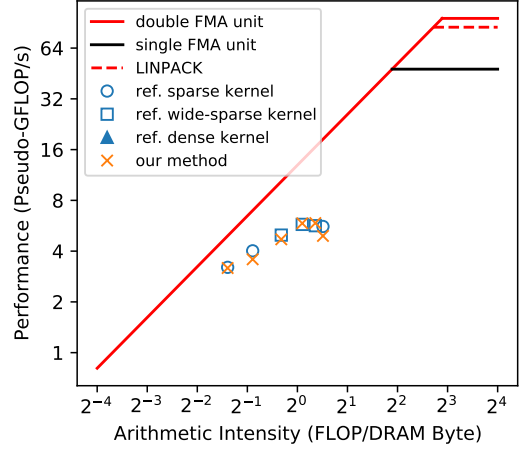
Varying the Number of Unique Absolute Non-zero Values in \mathbf{A}

Figure 5.11 shows the performance of our method and the reference LIBXSMM library with varying U . For $U \leq 224$, both the methods show a steady performance, unaffected by U . The reference wide-sparse kernels perform better than our method by $\sim 8\%$ because of multiple accumulations. For $U = 240$, the sparse kernel is generated because of the U limitation, which has a similar performance to our method. For $U = 256$, neither the reference sparse methods works. A dense kernel is generated which is 68% slower than our method.

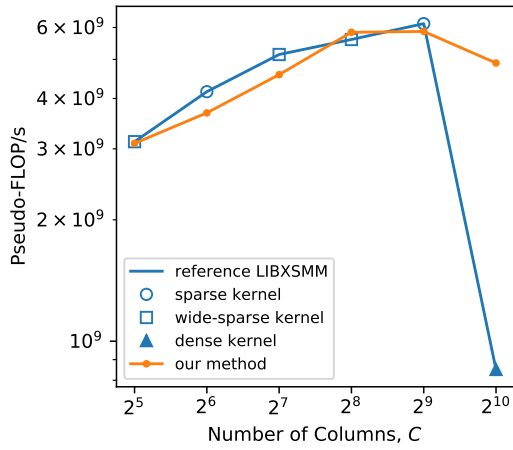
This concludes our evaluation results based on the synthetic \mathbf{A} set.



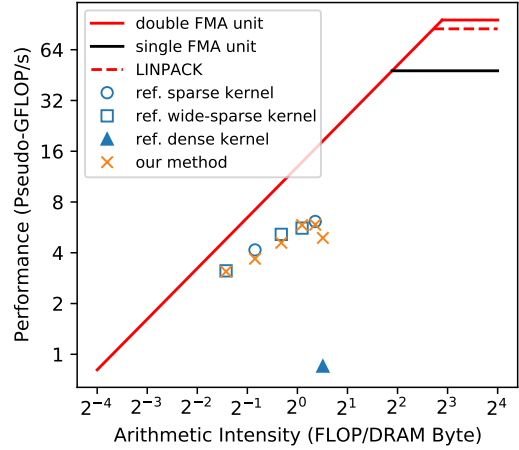
(a) Performance vs. number of columns, $U = 16$.



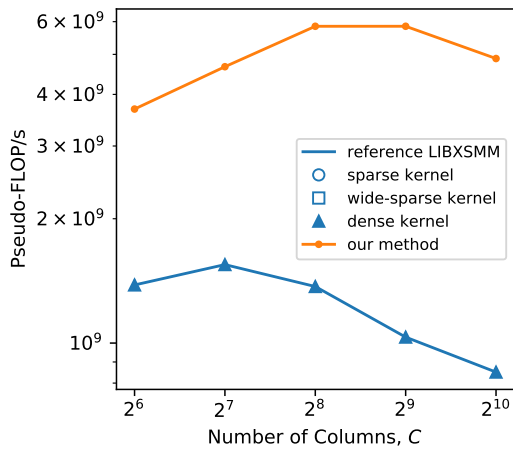
(b) Roofline plot, $U = 16$.



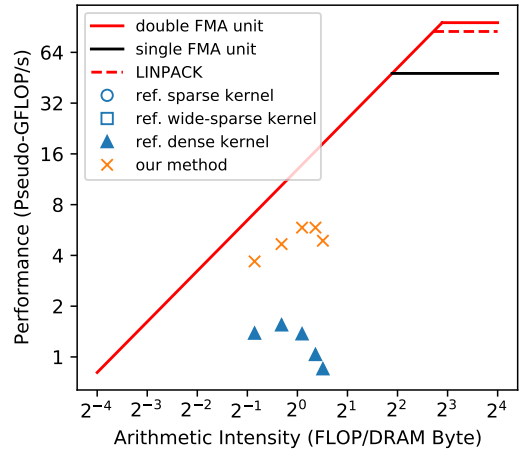
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

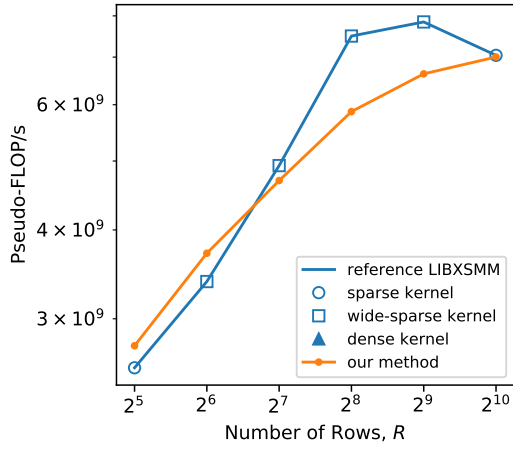


(e) Performance vs. number of columns, $U = 256$.

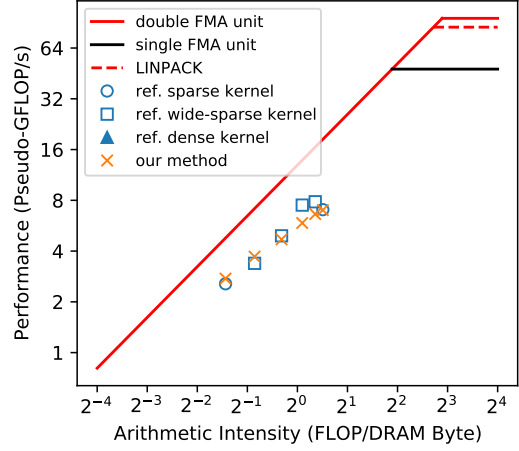


(f) Roofline plot, $U = 256$.

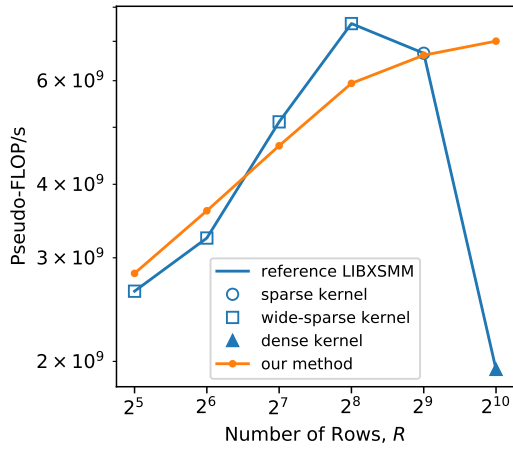
Figure 5.8: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns.



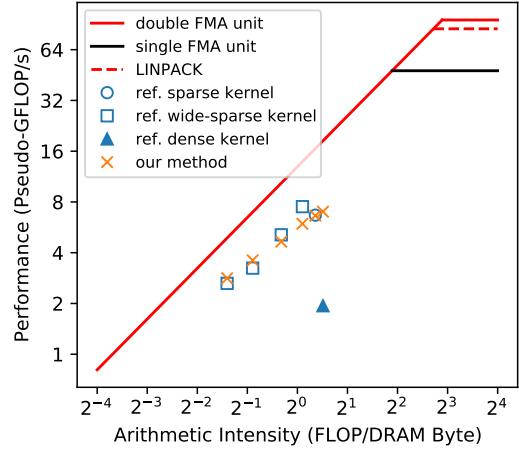
(a) Performance vs. number of rows, $U = 16$.



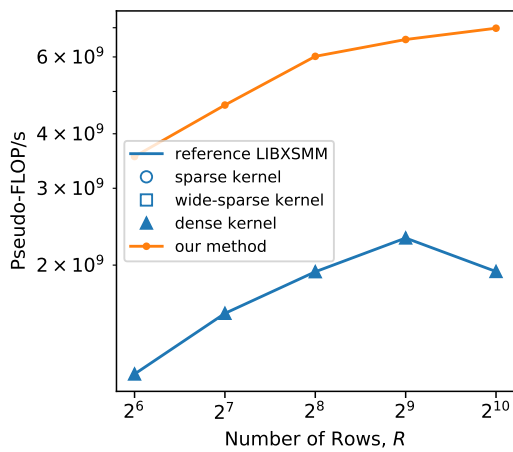
(b) Roofline plot, $U = 16$.



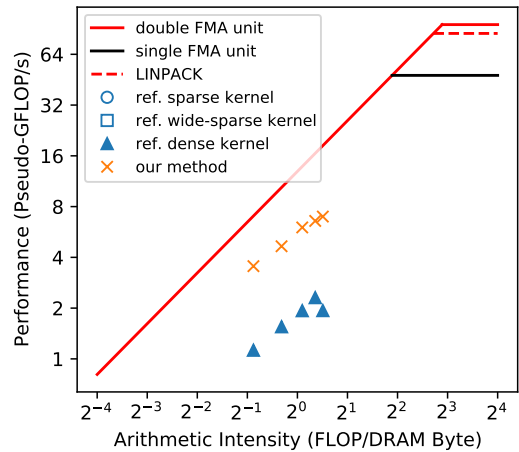
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

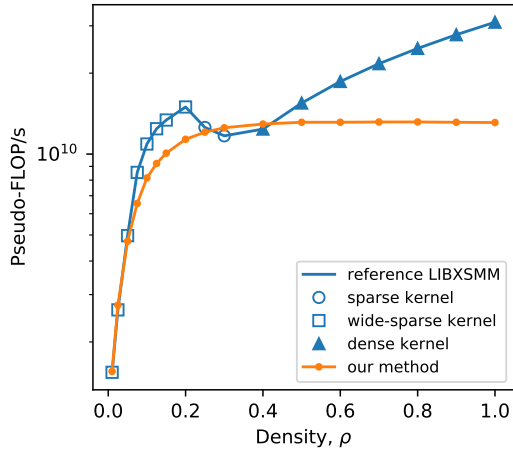


(e) Performance vs. number of rows, $U = 256$.

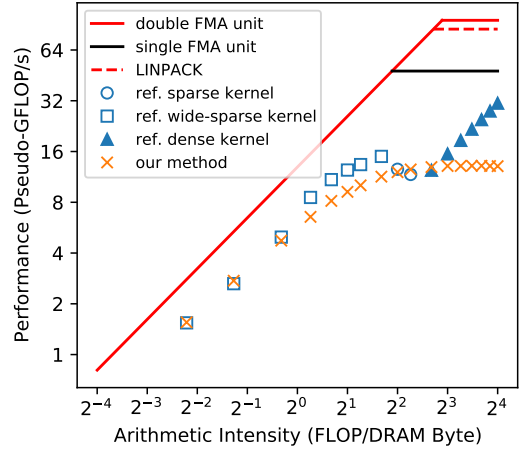


(f) Roofline plot, $U = 256$.

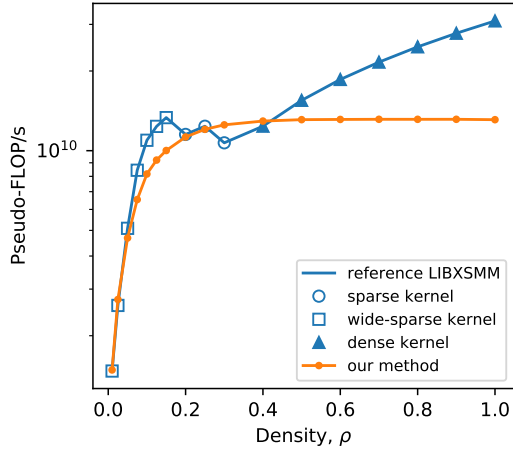
Figure 5.9: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows.



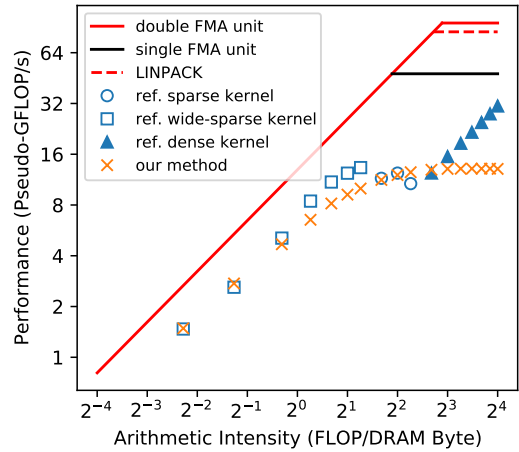
(a) Performance vs. density, $U = 16$.



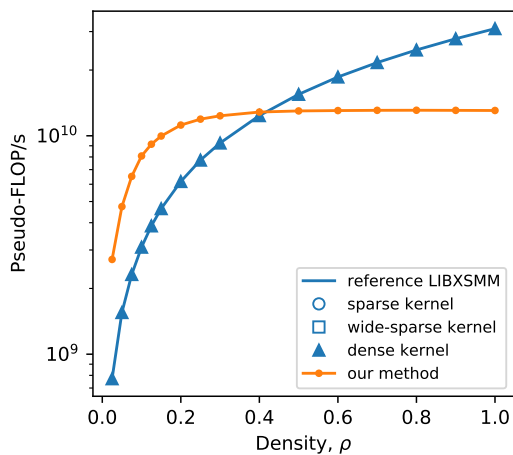
(b) Roofline plot, $U = 16$.



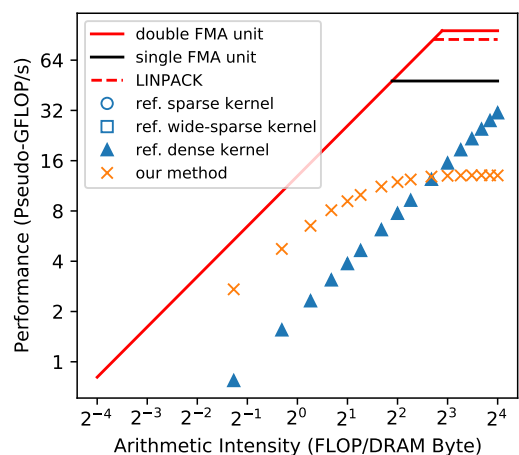
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

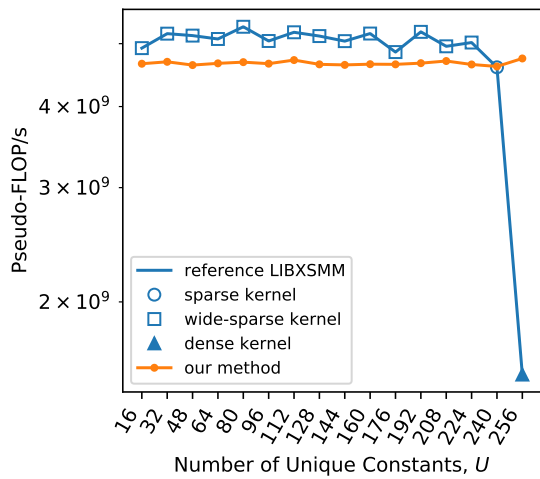


(e) Performance vs. density, $U = 256$.

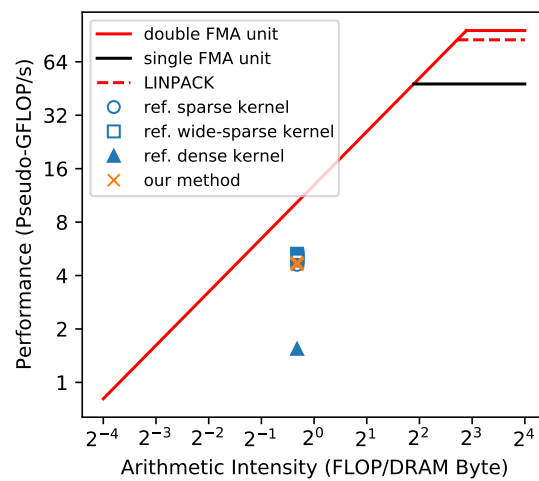


(f) Roofline plot, $U = 256$.

Figure 5.10: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density.



(a) Performance vs. number of unique constants.



(b) Roofline plot.

Figure 5.11: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of unique absolute non-zero constants in \mathbf{A} .

5.4 Supports for AVX2

As discussed in Section 5.1 earlier, instruction `VROADCASTSS/D` is available on AVX2. Comparing to the newer AVX-512, AVX2 is widely supported by both server and consumer x86 processors from both Intel and AMD. At the time of this report, LIBXSMM’s sparse GEMM routine can only accommodate a maximum of 15 distinct absolute non-zero values in \mathbf{A} for AVX2 processors. This is because AVX2 only provides 16 256-bit vector registers. With Paribartan’s register packing method [8], this threshold can be increased to 56 double-precision or 112 single-precision numbers. However, this is still significantly lower than what can be achieved on AVX-512.

Therefore it is very beneficial to develop a sparse GEMM routine for AVX2 allowing unlimited constants present in \mathbf{A} . Our method introduced in this section should be readily available for AVX2. Unfortunately, as this project focuses primarily on AVX-512, we were unable to evaluate our method on an AVX2 platform.

Summary

In this chapter we presented our GEMM routine which runtime broadcasts \mathbf{A} constants from memory using `VROADCASTSS/D`. Our method allows an unlimited number of distinct absolute values in \mathbf{A} as the constants are not stored in vector registers. Our method was evaluated using our benchmark suite on two matrices sets - PyFR operator matrices set and synthetic matrices set. Despite our method allows an unlimited number of \mathbf{A} constants, it performs 10% - 30% slower than the reference LIBXSMM’s wide-sparse kernel for certain sparse matrices. For dense matrices, our kernel is 55% less performant than the LIBXSMM’s dense routine. We suggested three reasons for this: 1. Our method uses a single vector register for accumulating \mathbf{C} strides, so the kernel is easily bottlenecked by FMA instruction latency. 2. Our method does not store \mathbf{B} strides in registers for later reuses. 3. Our method does not exploit any tiling scheme, resulting in poor temporal locality. In the later chapters, we will present how we optimise our kernel in these three aspects. In the end, we briefly described our method is very beneficial for AVX2 users. Unfortunately, because of test suite limitations, kernel performance on AVX2 platform was not measured.

Chapter 6

Multiple Vector Registers for Accumulating \mathbf{C} Strides

In Chapter 5 we presented our GEMM routine which allows an unlimited number of \mathbf{A} constants. Comparing with the performance of LIBXSMM’s wide-sparse and dense routines, our kernel is slower as it uses only one vector register for accumulating the \mathbf{C} strides. This creates strong data dependency as each FMA instruction has to wait for the result from the previous FMA before execution, leading to pipeline stall. One way of improving this is to have multiple registers accumulating different \mathbf{C} strides. As data dependencies only exist between FMA instructions using the same accumulation register, multiple FMA instructions using different accumulation registers can run independently exploiting Instruction-Level Parallelism.

In 2020, Paribartan [8] experimented with two techniques for achieving multiple accumulations, namely *N blocking* and *M blocking*. In this chapter, we present our efforts in improving our routine with these techniques.

6.1 N Blocking

Both our and LIBXSMM’s sparse kernels are developed from a simple vectorised GEMM routine which involves computing the dot products between \mathbf{A} rows and \mathbf{B} stride columns and storing the results to the corresponding \mathbf{C} locations (see Figure 2.1). The dot product is computed using a sequence of FMA instructions which frequently load and store intermediate values to an accumulation register. As discovered in Chapter 5, this creates data dependency between each FMA instruction leading to performance bottleneck due to instruction latency.

As experimented by Paribartan in 2020 [8], one method to overcome this is to compute multiple dot products between the same \mathbf{A} row and different \mathbf{B} stride columns simultaneously. As shown in Figure 6.1, this requires multiple vector registers for

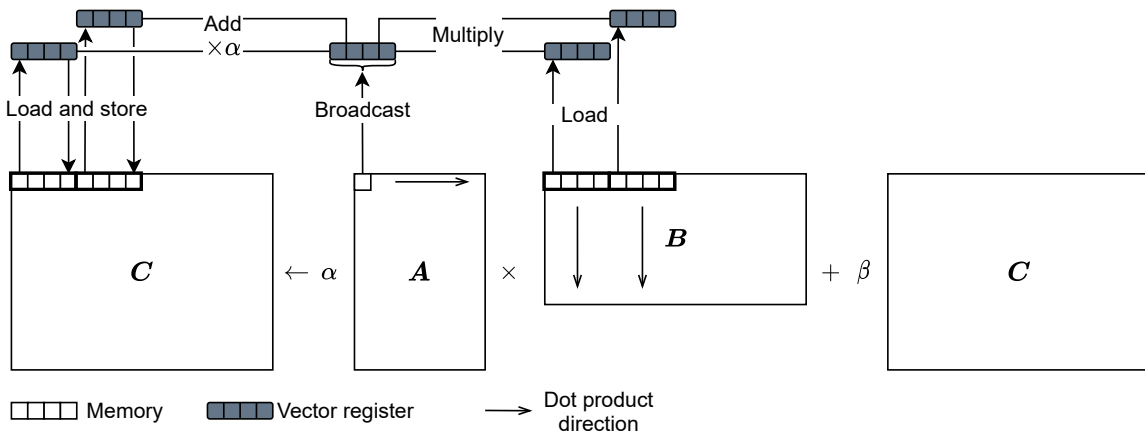


Figure 6.1: Diagrammatic representation of a simple vectorised GEMM routine with N blocking factor of 2.

accumulating C strides in the n direction, so this technique is called N blocking. N blocking increases the temporal locality of accessing A elements. There is no data dependency between FMA instructions issued for different dot products so multiple FMA can be executed independently exploiting Instruction-Level Parallelism (ILP) thus reducing pipeline stall. For our routine discussed in Chapter 5, each N blocking factor requires one more register for accumulating C strides. As B strides are loaded directly from memory, N blocking does not require more register for B strides. For the remainder of this report, we will use n_B standing for N blocking factor.

6.2 M Blocking

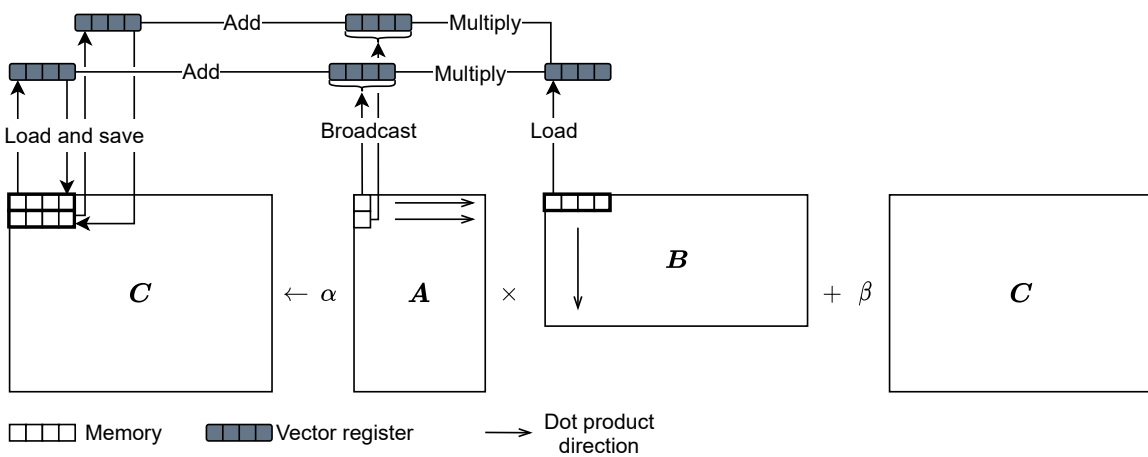


Figure 6.2: Diagrammatic representation of a simple vectorised GEMM routine with M blocking factor of 2.

Another technique for achieving multiple accumulations is M blocking, which involves accumulating multiple C strides in the m direction, as shown in Figure 6.2. Each stride

column of \mathbf{B} is used for computing multiple dot products with different \mathbf{A} rows. M blocking increases the temporal locality for accessing \mathbf{B} strides. For our method shown in Chapter 5, each M blocking factor requires one more register for accumulating \mathbf{C} stride and one more register for holding broadcasted \mathbf{A} elements. Unlike N blocking, for the context of sparse GEMM, M blocking does not always guarantee the generation of independent FMA instructions. Imaging the following \mathbf{A} matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Despite having a maximum M blocking factor of 3, the initial 2 FMA instructions have data dependency as they are issued for the same \mathbf{A} row.

For this report, we will use m_B standing for M blocking factor.

6.3 Kernel Design

Our sparse kernel presented in Chapter 5 can be easily adapted with N and M blocking techniques. For our design, we use the first (starting from `zmm0`) m_B vector registers for storing broadcasted \mathbf{A} elements and the last $n_B \times m_B$ (starting backwards from `zmm31`) registers for accumulating \mathbf{C} strides. As AVX-512 supports 32 vector registers, so n_B and m_B must obey:

$$m_B + n_B \times m_B \leq 32. \tag{6.1}$$

For $n_B = 1$, the maximum m_B is 16. For $n_B = 2$, the maximum m_B is 10. For $n_B = 3$, the maximum m_B is 8. We did not consider the case for $n_B > 3$ in this project. This is because each n_B increases the kernel width by 8. For $n_B > 3$, the GEMM kernel is wider than 24, which is incompatible with the current PyFR setting, which computes GEMM in \mathbf{B} tiles of 48 columns. Appendix B.3 shows some example kernels with different n_B and m_B .

6.4 Evaluation - N Blocking

In this section we present our evaluation of the N blocking technique using our test suite (see Chapter 4), for $n_B = 1, 2, 3$. Similar to the previous tests, we increased the maximum kernel code size limitation for our method to 2 MB.

PyFR Operator Matrices

In this section, we only present the roofline and performance data plotting against \mathbf{A} density (ρ) and the number of \mathbf{A} constants (U) measured on c5n.xlarge machine. Results from m5n.xlarge are very similar so there are not included in this section. For performance data plotting against the number of \mathbf{A} columns, the number of \mathbf{A} rows and \mathbf{A} size, and the results from m5n.xlarge, please refer to Appendix D.

Quadrilateral and Hexahedral Element Matrices

Figure 6.4 and 6.5 show the effect of N blocking for PyFR quadrilateral and hexahedral element matrices. These matrices are very sparse and contain less than 26 distinct absolute non-zero constants. As shown by Figure 6.4a and 6.5a, the performance is almost exactly the same for small U , irrelevant to n_B . Indicating by the roofline plots (Figure 6.4c and 6.5c), these kernels are heavily bottlenecked by memory bandwidth, so that N blocking should not affect performance. For $U \geq 10$, as the performance starts to be limited by FMA instruction latency, $n_B = 3$ kernels are slightly faster than that with $n_B = 2$ and $n_B = 1$. However, $n_B = 3$ does not provide a significant improvement comparing to the reference kernels.

It is interesting to note that for some matrices, $n_B = 2$ kernels perform marginally worse than both the $n_B = 3$ and $n_B = 1$ kernels. We suggest this relates to performance sensitivity to memory access patterns. In average, for the quadrilateral matrices, our methods with $n_B = 1$, $n_B = 2$ and $n_B = 3$ are 98.6%, 99.3% and 100.3% performant comparing to the reference library, respectively. For the hexahedral element matrices¹, our method is 99.0%, 95.7% and 100.2% performant comparing to the reference one, for $n_B = 1$, $n_B = 2$ and $n_B = 3$ respectively.

Tetrahedral and Triangular Element Matrices

Figure 6.6 and 6.7 shows the performance of our method with N blocking for tetrahedral and triangular PyFR matrices. As shown in Figure 6.6b and 6.7b, these matrices are much denser compared to the quadrilateral and hexahedral ones. Indicating by the roofline plots (Figure 6.6c and 6.7c), N blocking does not improve the performance for problems with small arithmetic intensity, as these kernels are heavily limited by memory bandwidth. For problems with large arithmetic intensity, N blocking shows significant improvement to kernel performance, as it issues independent FMA instructions which can execute independently, hiding the instruction latency. The maximum performance increase is 36% comparing to that without N blocking.

However, despite this improvement, our kernels with N blocking are still $\sim 30\%$ slower

¹We exclude the 6th order Gauss-Legendre m460 matrix as a sparse kernel cannot be generated as the kernel code size exceeds LIBXSMM's default setting.

than the reference dense routine. There are two reasons for this. Firstly, the dense routine achieves a higher degree of multiple accumulations utilising both N and M blocking. Therefore, it exploits ILP better as more independent FMA instructions can be executed in the pipeline. Secondly, our method requires a 64-byte memory read for each FMA for loading the \mathbf{B} stride. The architecture we were testing on, Skylake-SP, only supports maximum 128-byte L1 cache read per cycle. Together with the fact that our routine also requires to load and broadcast \mathbf{A} constants from memory, the L1 cache bandwidth can be easily saturated with multiple accumulations. The reference dense routine bypasses this problem by temporally storing the active \mathbf{B} strides in vector registers and repeatedly use them for multiple FMA instructions by M blocking. It is beneficial if we can integrate this feature into our routine.

It is interesting to note that, for tetrahedral matrices, the performance gain by N blocking gradually decreases as the arithmetic intensity become very large. In fact, $n_B = 2$ sometimes performs better than $n_B = 3$. This is because although N blocking increase the temporal locality for referencing \mathbf{A} elements, it decreases that for referencing \mathbf{B} strides as multiple \mathbf{B} stride columns are accessed simultaneously. This is more noticeable for large \mathbf{A} s with large arithmetic intensities, as the L1 data cache cannot fully accommodate multiple \mathbf{B} stride columns, resulting in an increasing cache miss rate. This suggests M blocking which increases the temporal locality of referencing \mathbf{B} strides could be very beneficial.

PyFR Operator Matrices - Summary

Figure 6.3 summarises the average performance of our kernel with different N blocking factors relative to reference LIBXSMM. The 6th order Gauss-Legendre m460 matrix was excluded for calculations as reference sparse kernel cannot be generated due to the kernel size limitation. As shown by the figure, N blocking can provide $\sim 15\%$ performance improvements for tetrahedral and triangular element matrices. Despite the performance increase, our routine is still slower than the reference dense routine for tetrahedral matrices.

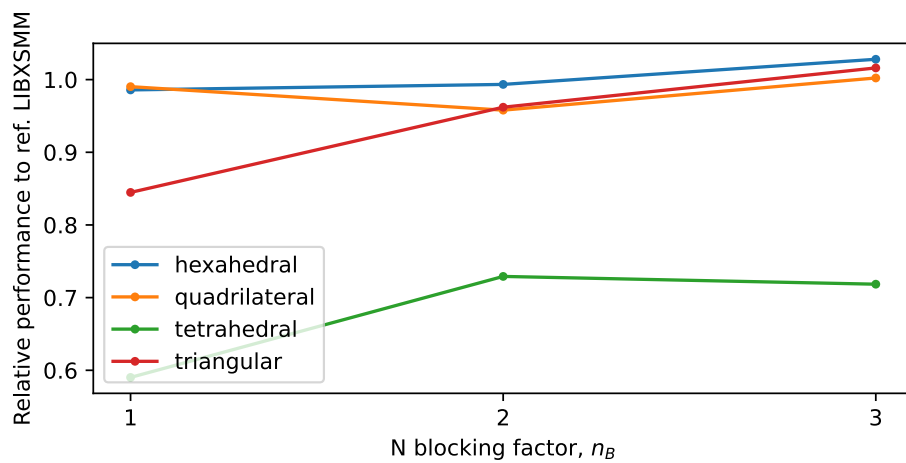
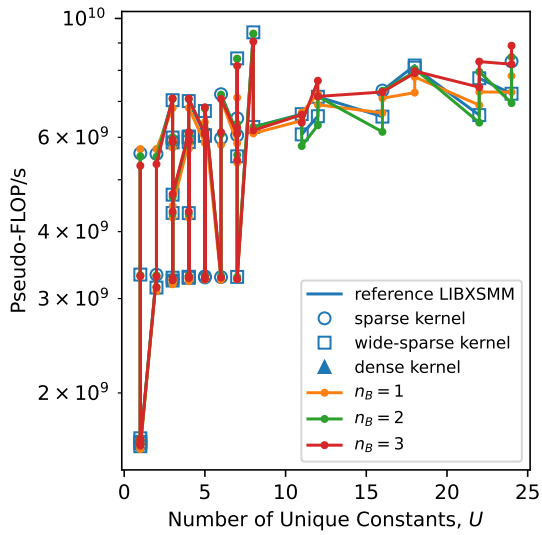
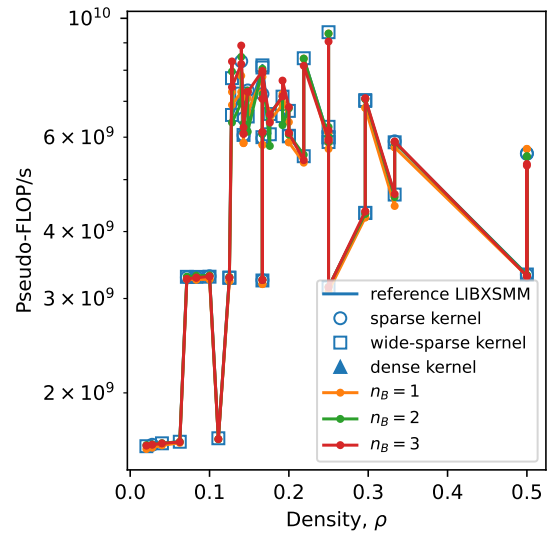


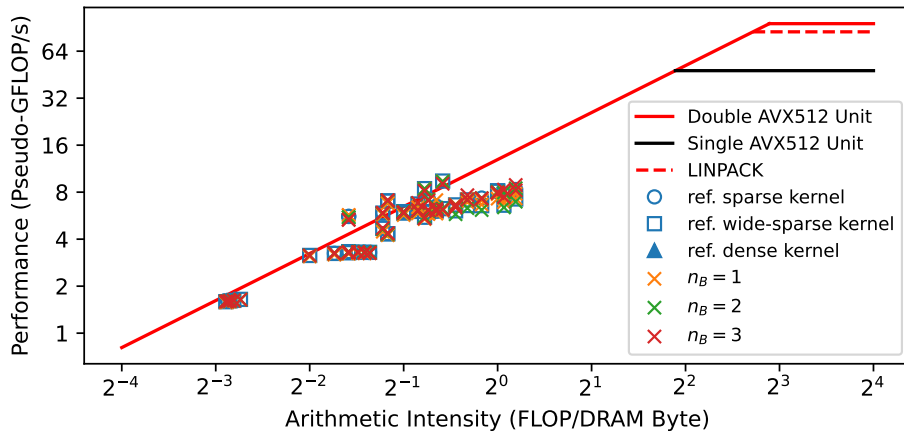
Figure 6.3: Average performance of our routines using different degrees of N blocking evaluating using PyFR matrices, relative to reference LIBXSMM.



(a) Performance against number of unique \mathbf{A} constants.

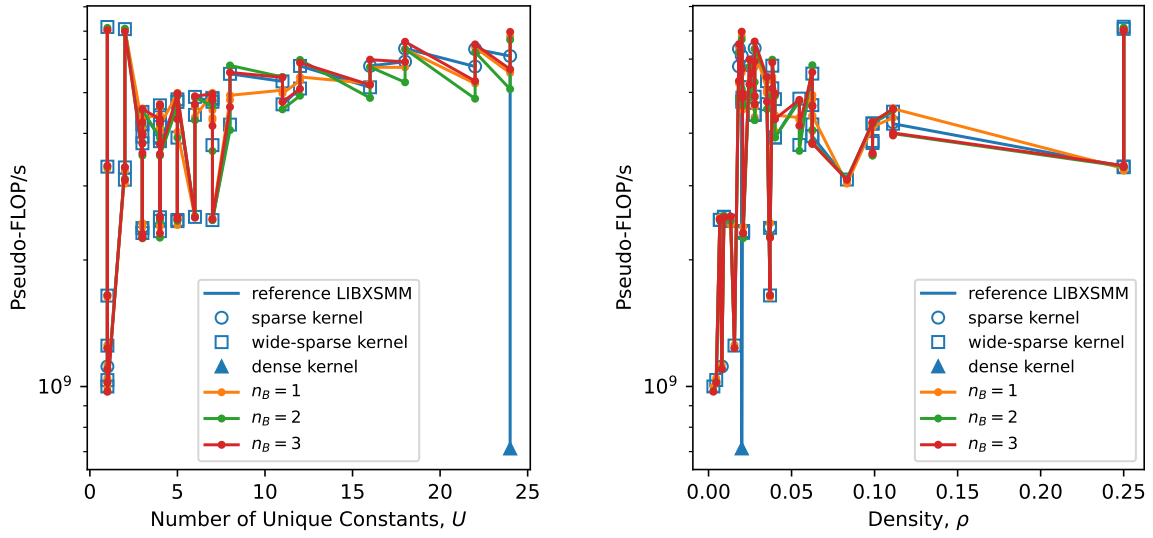


(b) Performance against \mathbf{A} density.



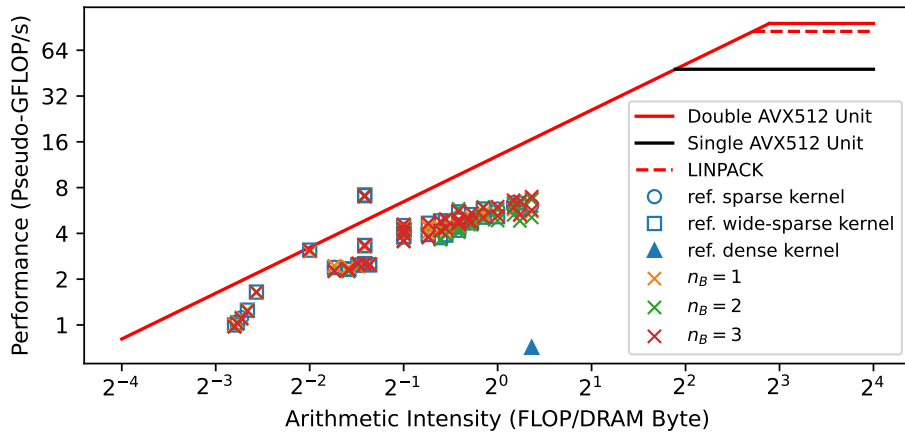
(c) Roofline plot.

Figure 6.4: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices.



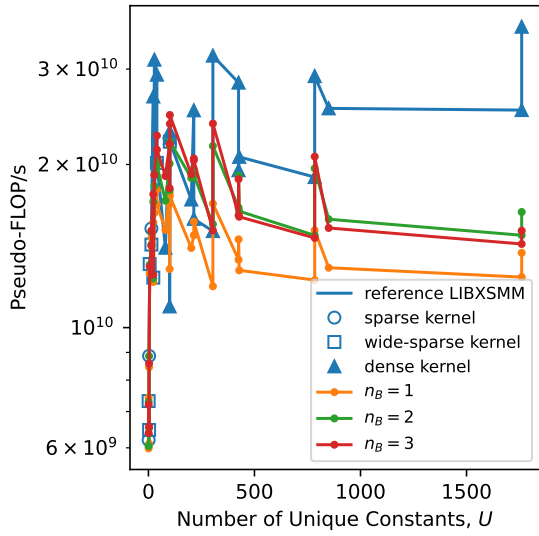
(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.

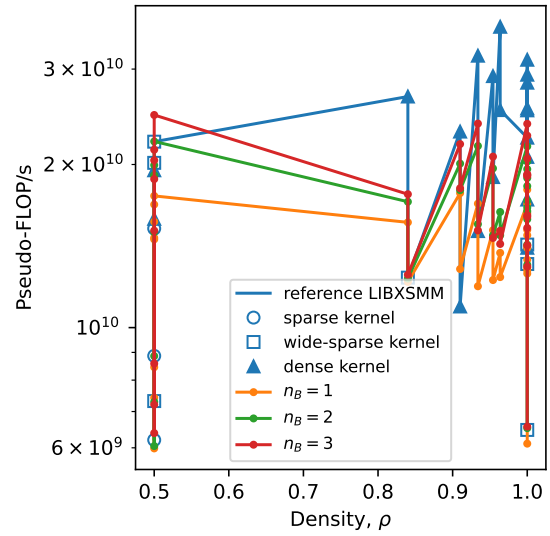


(c) Roofline plot.

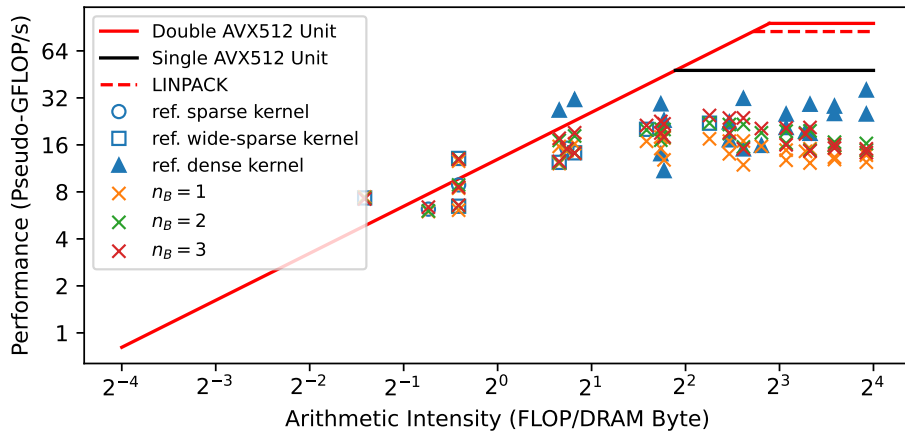
Figure 6.5: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices.



(a) Performance against number of unique \mathbf{A} constants.

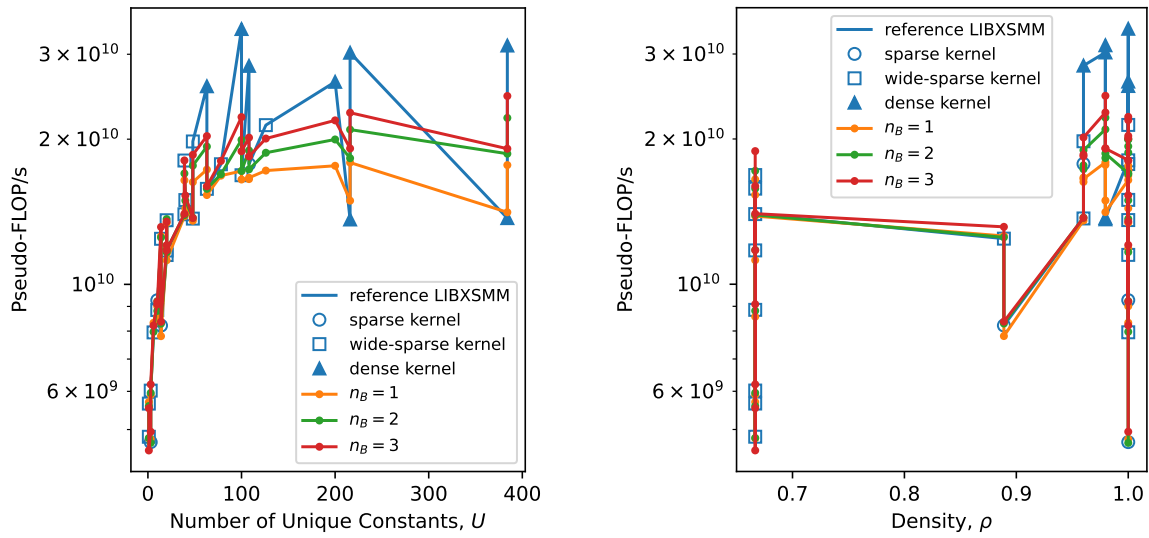


(b) Performance against \mathbf{A} density.



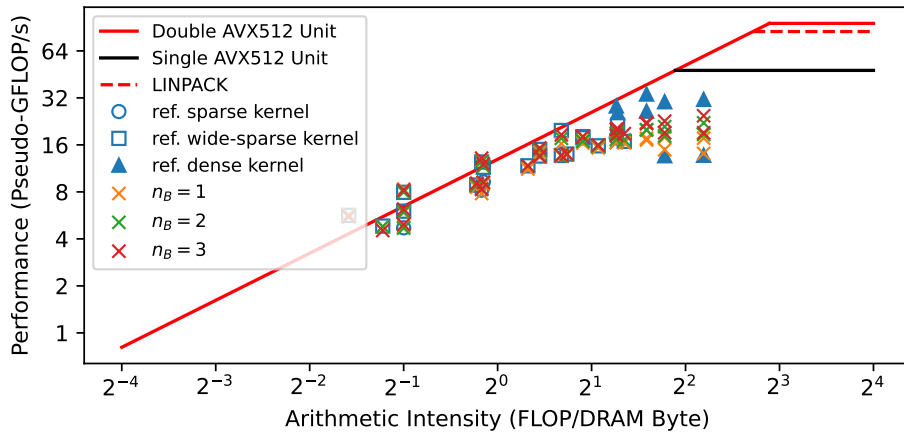
(c) Roofline plot.

Figure 6.6: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices.



(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.



(c) Roofline plot.

Figure 6.7: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices.

Synthetic Matrices

In this section we present the performance of our kernels with different N blocking factors, evaluated using the synthetic test set. Similar to evaluation on PyFR matrices, the results from c5n.xlarge and m5n.xlarge are very similar so we only present the c5n.xlarge here. The m5n.xlarge results are included in Appendix D.

Varying Number of A Columns

Figure 6.8 shows the performance of our method with different N blocking factors evaluated with the synthetic matrices with varying numbers of columns. For $U = 16, 64$ and $C = 2^5$, all the routines perform the same irreverent to n_B . This is because the performance is heavily limited by memory bandwidth. For $2^5 < C \leq 2^8$, N blocking wins, as it can harness more ILP and hide FMA instruction latency with multiple accumulations. However, for $C > 2^8$, N blocking routines perform worse than that without N blocking. As discussed earlier, this is because N blocking kernels are prone to an earlier and more severe cache spill, as N blocking requires simultaneous reads of multiple B stride columns. Therefore, N blocking should be used for only $C \leq 2^8$. This observation is consistent for all of the U s.

Varying Number of A Rows

Figure 6.9 shows the effects of N blocking on our kernel evaluated using the synthetic matrices with varying numbers of rows. For all of the U s, the kernels without N blocking outperform the ones with N blocking by $\sim 10\%$ for $R \leq 2^6$. The reasoning for this is unclear, but because the speed up is rather insignificant, it is not investigated during this project. For $R \geq 2^8$, routines with N blocking perform better than the ones without by up to 27%, as N blocking helps hiding the instruction latency. Our data shows $n_B = 2$ and $n_B = 3$ have a very similar performance. $n_B = 2$ shows a marginally better performance for $2^8 \leq R \leq 2^9$ while $n_B = 3$ is slightly better for $R = 2^7$. In summary, our results support the use of N blocking for $R \geq 2^7$.

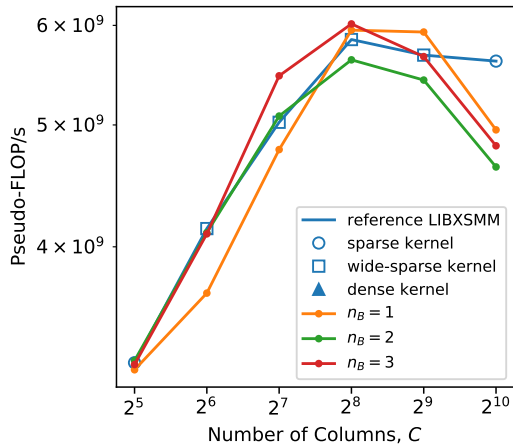
Varying A Density

Figure 6.10 shows the performance of our routines, with and without N blocking, evaluated using the synthetic matrices with varying densities. For all the U s, N blocking does not improve the performance for $\rho \leq 0.1$. This is because the arithmetic intensity is very small for these matrices so that the performance is majority memory bandwidth bound. For larger ρ , N blocking shows a consistent speedup comparing to our routine without N blocking. We observed a maximum speed up of 28%. Comparing between different N blocking factors, $n_B = 2$ shows a slightly better performance comparing to

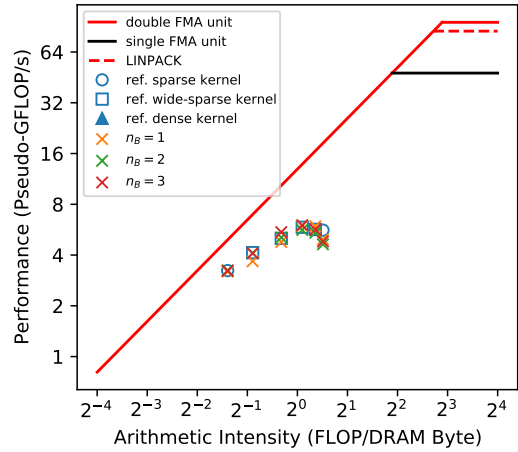
$n_B = 3$. The reasoning for this is unclear. It could be due to that $n_B = 2$ can already saturate L1 cache read bandwidth so larger n_B won't further improve the performance. Despite N blocking provides a big performance improvement, the reference dense routine outperforms our methods for $\rho \geq 0.6$. As discussed earlier, this is because the dense routine utilises way more registers for multiple accumulations and it stores and reuses some \mathbf{B} strides in registers to avoid repeated \mathbf{B} stride loads.

Varying the Number of Unique Absolute Non-zero Values in \mathbf{A}

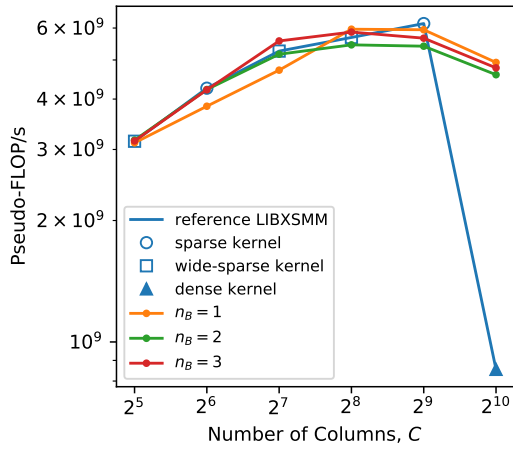
Figure 6.11 shows the performance of our routines with and without N blocking evaluated using the synthetic matrices with varying numbers of unique constants. Consistently, kernels with $n = 3$ are faster than the $n = 2$ ones by around 10%. The $n = 2$ kernels are faster than the ones without N blocking by around 10%. For these matrices, N blocking helps the performance by hiding FMA instruction latency. Comparing to the reference method, N blocking does not affect how many constants our kernel can accommodate. In summary, these results support using N blocking with our kernel for all of the U s.



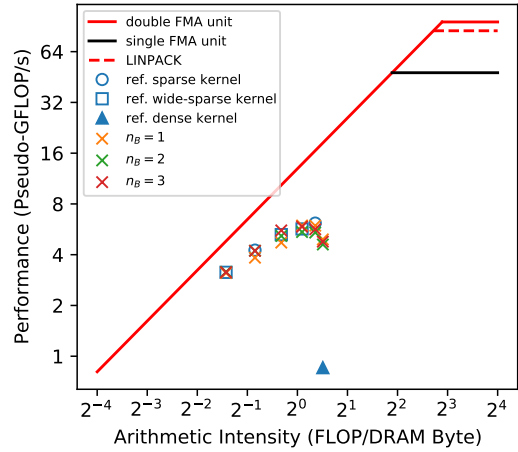
(a) Performance vs. number of columns, $U = 16$.



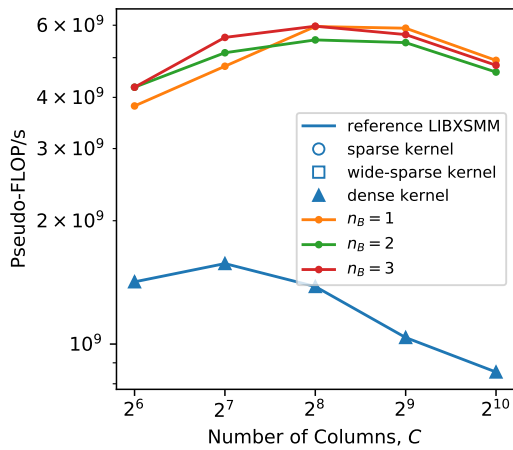
(b) Roofline plot, $U = 16$.



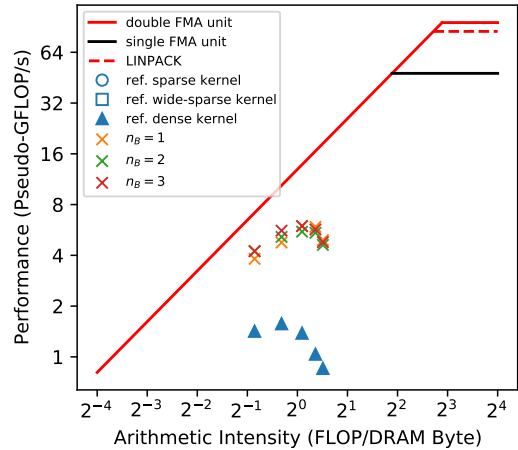
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

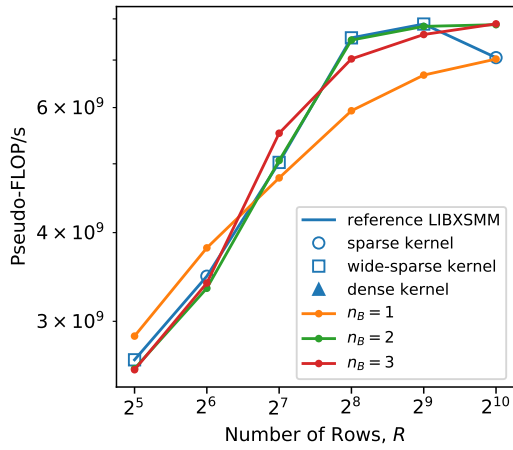


(e) Performance vs. number of columns, $U = 256$.

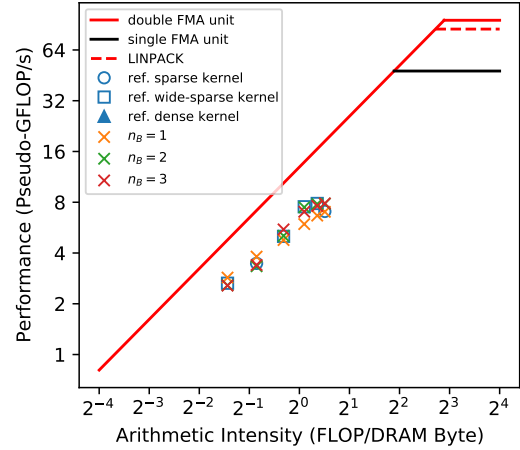


(f) Roofline plot, $U = 256$.

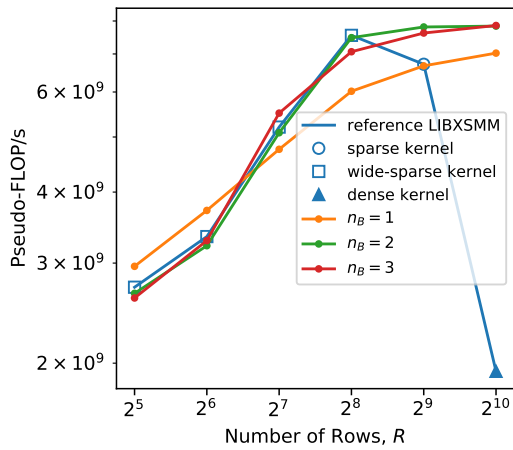
Figure 6.8: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns.



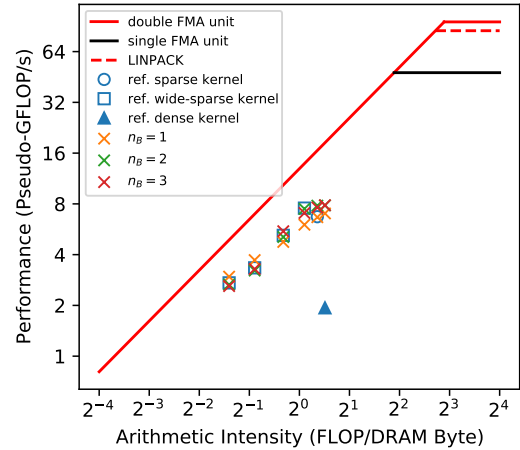
(a) Performance vs. number of rows, $U = 16$.



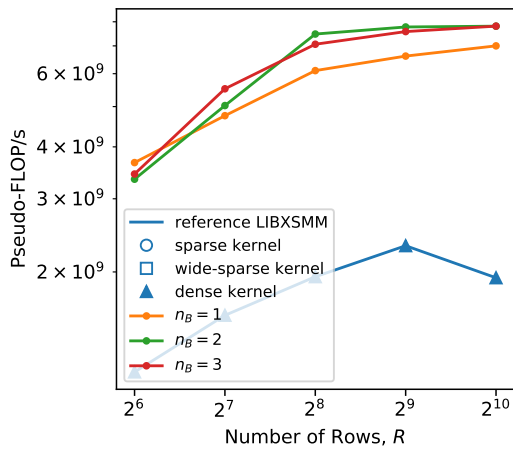
(b) Roofline plot, $U = 16$.



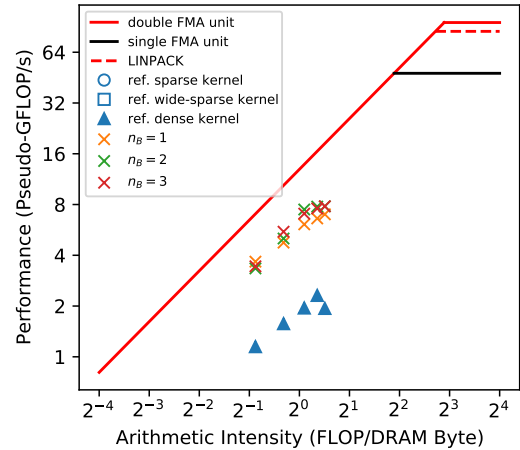
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

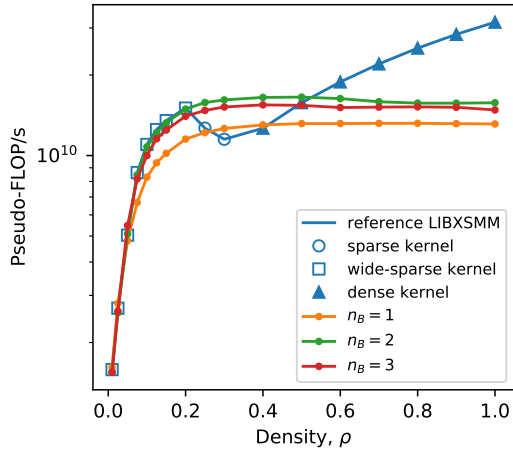


(e) Performance vs. number of rows, $U = 256$.

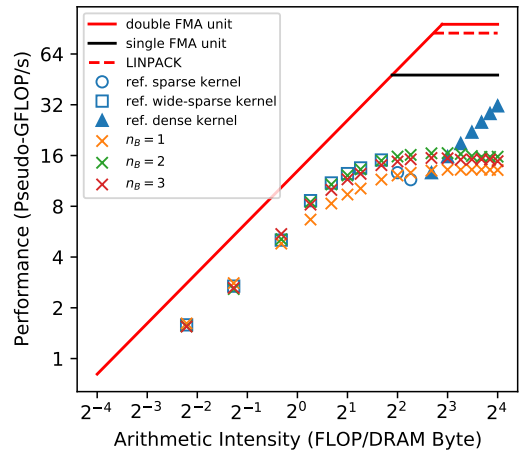


(f) Roofline plot, $U = 256$.

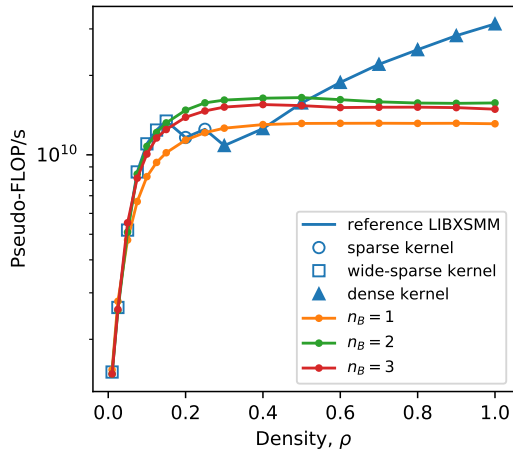
Figure 6.9: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows.



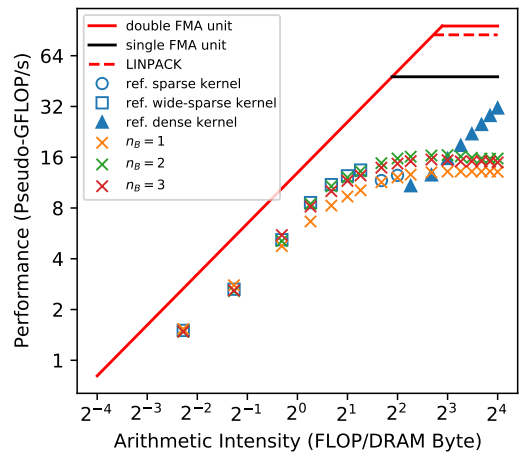
(a) Performance vs. density, $U = 16$.



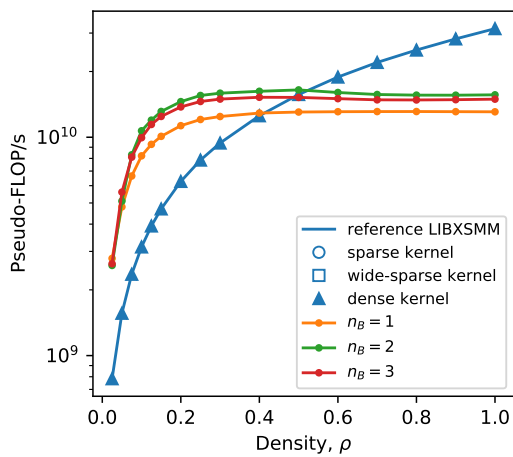
(b) Roofline plot, $U = 16$.



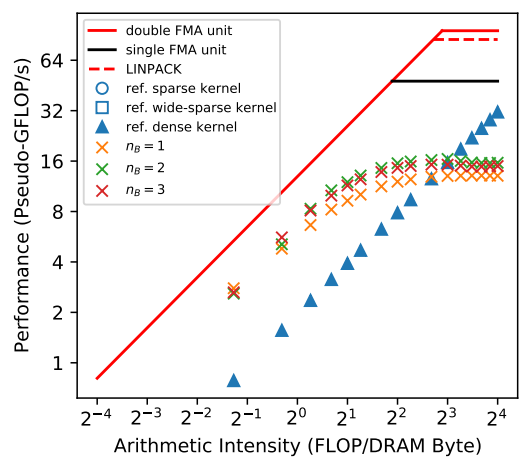
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

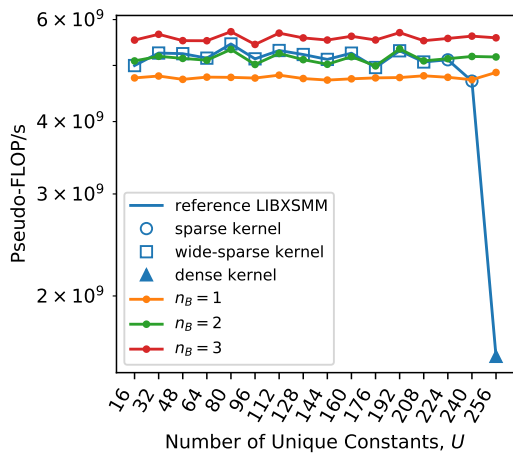


(e) Performance vs. density, $U = 256$.

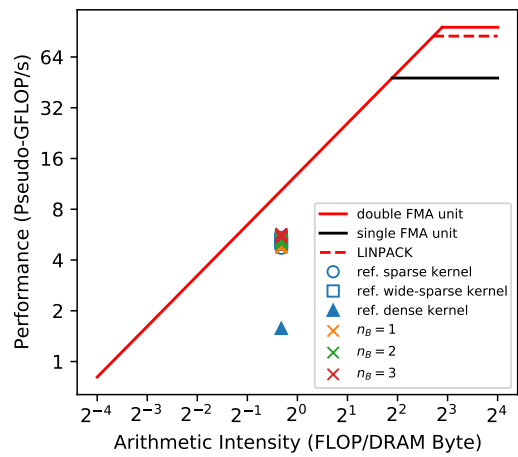


(f) Roofline plot, $U = 256$.

Figure 6.10: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density.



(a) Performance vs. number of unique absolute non-zero values.



(b) Roofline plot, $U = 16$.

Figure 6.11: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} .

6.5 Evaluation - M Blocking

In this section we present our evaluation of the M blocking technique using our test suite (see Chapter 4), for $m_B = 1, 3, 6, 12, 16$.

PyFR Operator Matrices

In this section, we only present the roofline and performance data plotting against \mathbf{A} density (ρ) and the number of \mathbf{A} constants (U) measured on c5n.xlarge machine. Results from m5n.xlarge are very similar so there are not included in this section. For performance data plotting against the number of \mathbf{A} columns, the number of \mathbf{A} rows and \mathbf{A} size, and the results from m5n.xlarge, please refer to Appendix D.

Quadrilateral and Hexahedral Element Matrices

Figure 6.13 and 6.14 present the benchmark results for our kernel using M blocking evaluated by the quadrilateral and hexahedral PyFR matrices. Unlike N blocking, M blocking does not improve the performance for these matrices significantly. This is because these matrices are very sparse (see Figure 6.13b and 6.14b), so M blocking cannot generate independent FMA instructions as effective as N blocking. In previous Section 6.2, we have explained why this is the case.

Tetrahedral and Triangular Element Matrices

Figure 6.15 and 6.16 show the performance of our routine with and without M blocking evaluated using the PyFR tetrahedral and triangular element matrices. Comparing to the sparse quadrilateral and hexahedral matrices, M blocking provides a big improvement for these dense matrices, as for dense matrices performance is more limited by instruction latency and M blocking is more efficient in issuing independent FMAs. We observed a maximum speed up of 62%. This improvement is also higher than what can be achieved by N blocking, as M blocking also increase the temporal locality of reading \mathbf{B} strides, which contributes most to the memory traffic. However, despite such an improvement, our routine with M blocking is still slower than the reference dense routine. Similar to our arguments for N blocking, the reference dense routine is superior as it caches some \mathbf{B} strides in the vector register to avoid redundant memory traffic. In Chapter 7 later we present how we further improve our routine using this method.

It is worth noting that, although for the tetrahedral matrices, larger m_B almost always show higher performance, this is not the case for triangular ones. For triangular element matrices, $m_B = 3$ and $m_B = 6$ generally show a better performance than

$m_B = 12$ and $m_B = 16$. This indicates some factors slowing down the kernel for large m_B . In the later phase of this project, when we were evaluating a more performant kernel (see Chapter 7), we also observed this behaviour. For now, we will leave the discussion of this behaviour to that later section, as it is more beneficial to address this issue for a more performant method.

PyFR Operator Matrices - Summary

Figure 6.12 summarises the average performance of our M blocking kernels comparing to reference LIBXSMM. Similar to earlier sections, we excluded the 6th order Gauss-Legendre m460 matrix for calculating the average. As shown in the figure, M blocking improves the performance significantly for dense tetrahedral and triangular matrices. For triangular matrices, $m_B = 3$ perform the best comparing to higher m_B . Despite the performance increase, our routine is still slower than the reference dense routine for tetrahedral matrices.

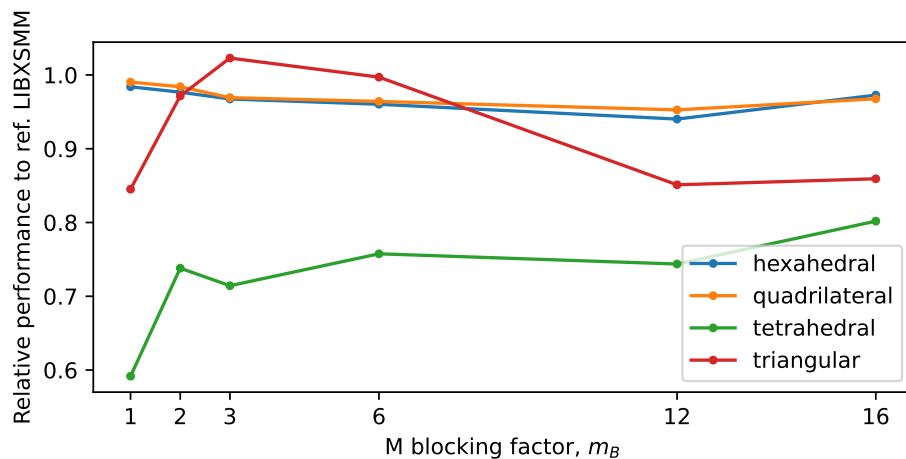
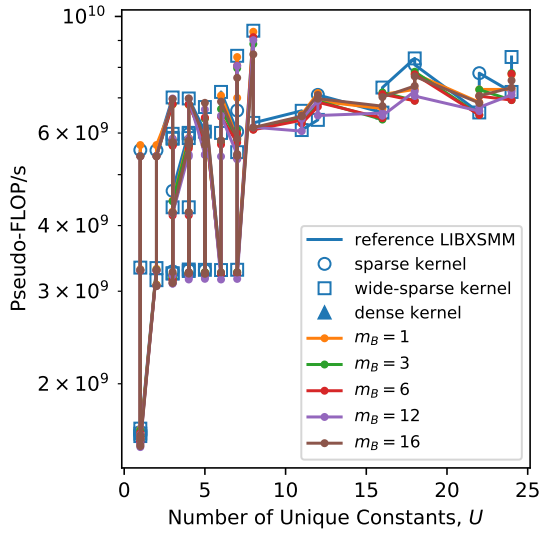
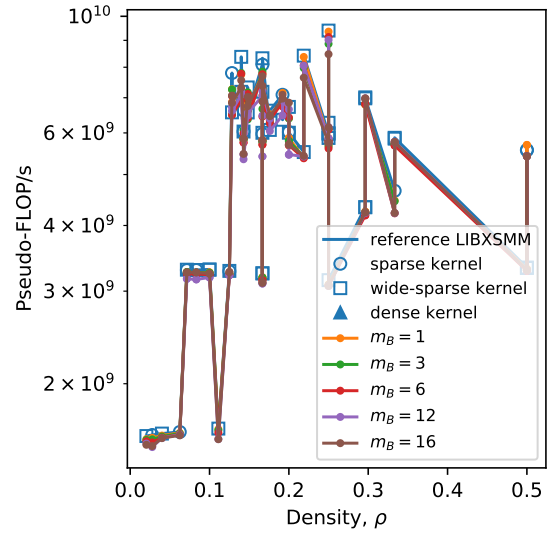


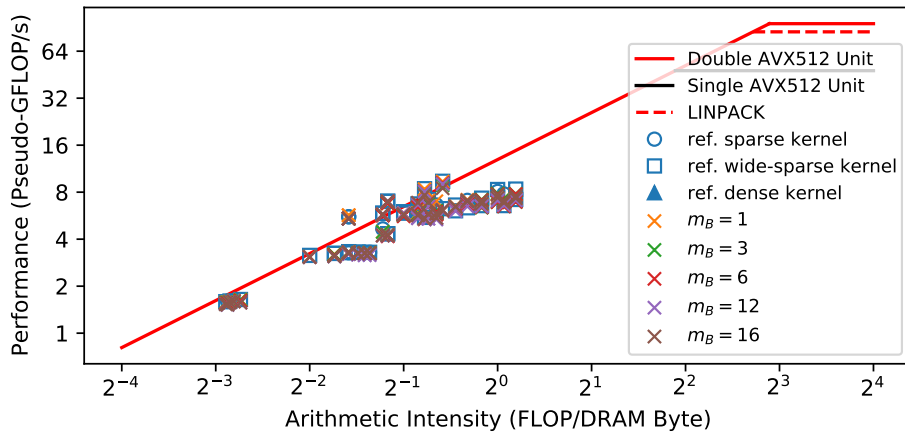
Figure 6.12: Average performance of our routines using different degrees of M blocking evaluating using PyFR matrices, relative to reference LIBXSMM.



(a) Performance against number of unique \mathbf{A} constants.

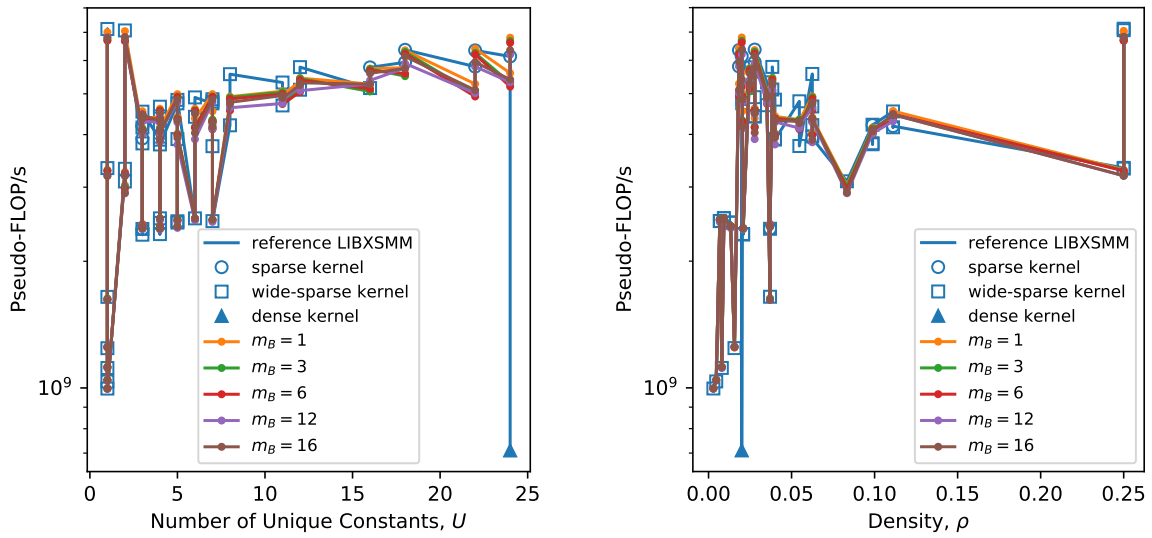


(b) Performance against \mathbf{A} density.



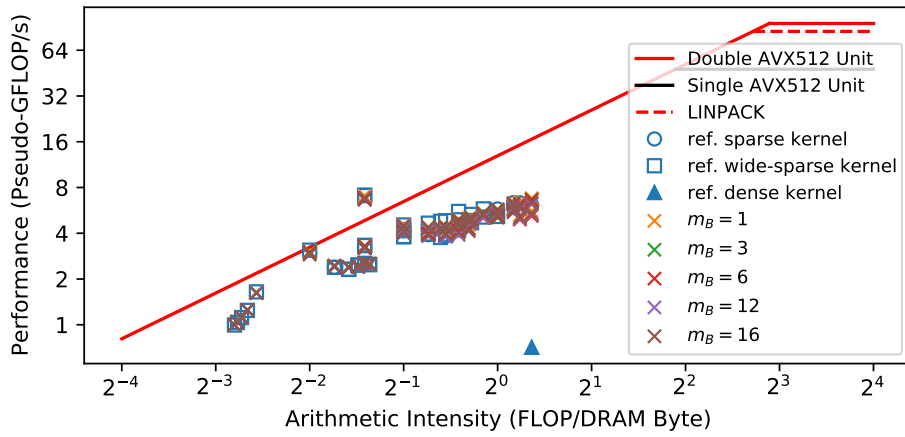
(c) Roofline plot.

Figure 6.13: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices.



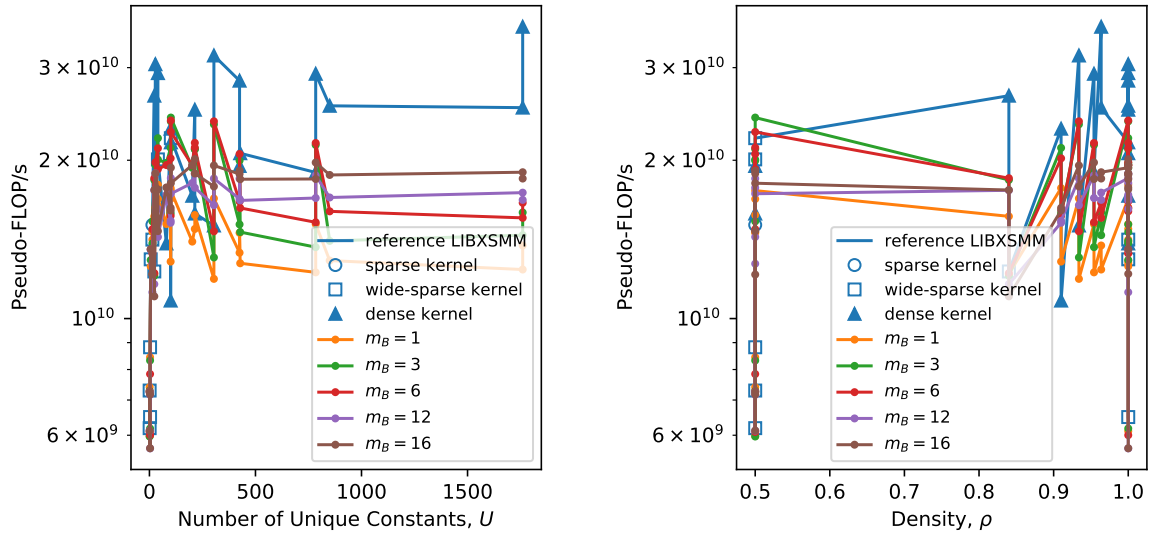
(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.



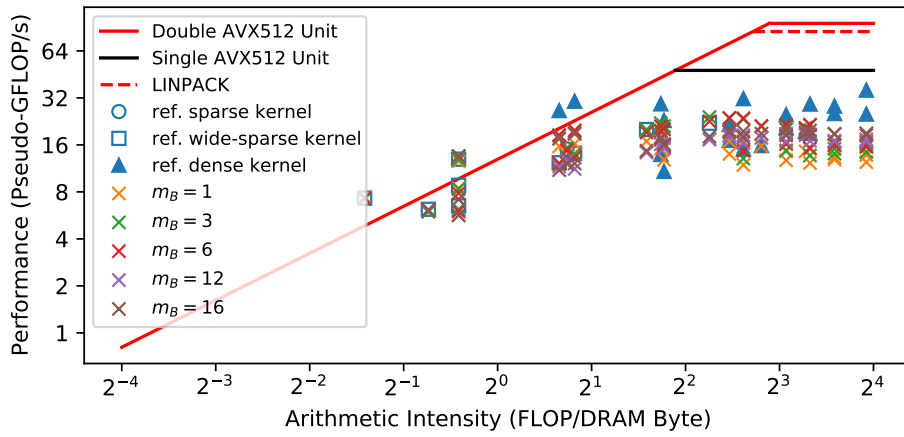
(c) Roofline plot.

Figure 6.14: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices.



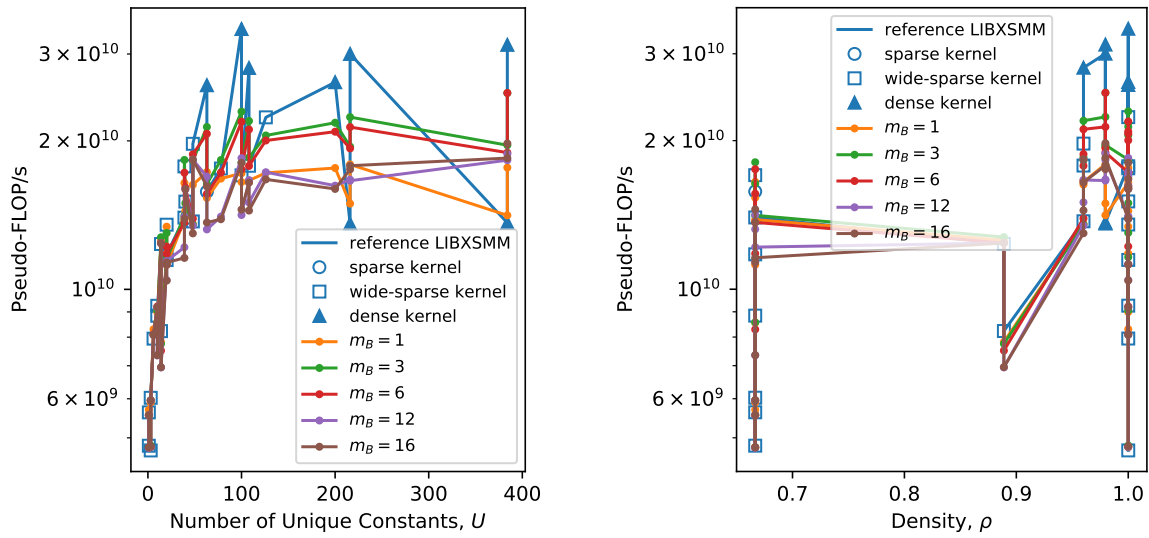
(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.



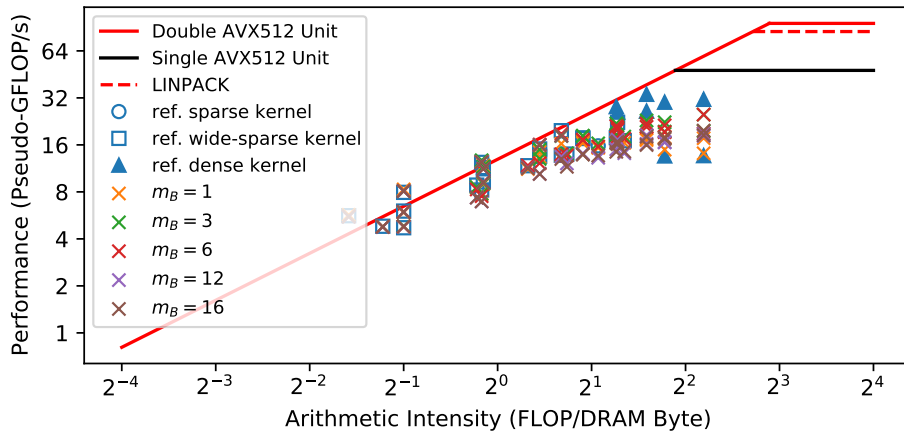
(c) Roofline plot.

Figure 6.15: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices.



(a) Performance against number of unique \mathbf{A} constants.

(b) Performance against \mathbf{A} density.



(c) Roofline plot.

Figure 6.16: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices.

Synthetic Matrices

In this section we present the performance of our kernels with different M blocking factors, evaluated using the synthetic test set. Similar to evaluation on PyFR matrices, the results from `c5n.xlarge` and `m5n.xlarge` are very similar so we only present the `c5n.xlarge` here. The `m5n.xlarge` results are included in Appendix D.

Varying Number of A Columns

Figure 6.17 presents the benchmark results of our kernels using M blocking techniques evaluated using the synthetic matrices set with varying numbers of columns. For all the Us , M blocking does not provide any speed up for these matrices. We suggest the reason for this is that, as the synthetic matrices are very sparse ($\rho = 0.05$), M blocking is inefficient in issuing independent FMA instructions for these matrices. Therefore, instruction latency cannot be hidden. In fact, M blocking introduces a slight performance decrease for these matrices. We suggest this is because M blocking decreases the temporal locality in accessing A elements.

Varying Number of A Rows

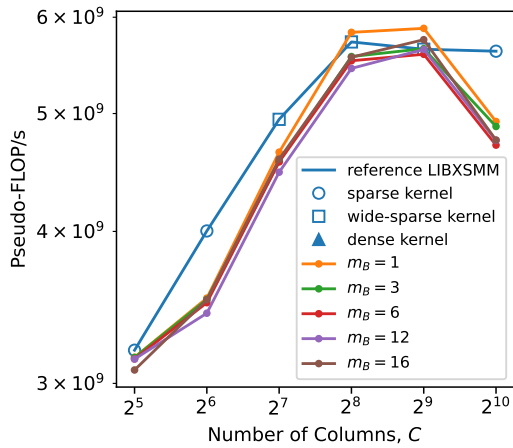
Figure 6.18 shows the performance of our M blocking kernels evaluated using the synthetic matrices with varying numbers of rows. Similar to our previous findings, M blocking does not improve the performance as it cannot effectively issue independent FMAs due to matrices sparsity. It introduces a slight performance decrease because it decreases the temporal locality of accessing A elements. It is worth noting that, we observed a larger performance decrease for increasing R . This is because that as the matrices grow taller, they require more A element accesses which magnifies the performance decrease due to lower A referencing locality.

Varying A density

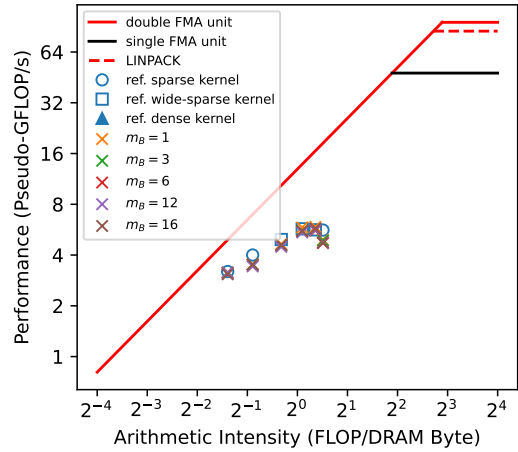
Figure 6.19 present the benchmark result for our M blocking kernels evaluated using the synthetic matrices with varying densities. For all of the Us , M blocking provides no change or a slight decrease to the kernel performance in the range of $0.0 < \rho < 0.4$. For this ρ range, N blocking is more suitable as it guarantees the generation of multiple independent FMA instructions. For $\rho \geq 0.4$, M blocking provides positive improvement to kernel performance. In fact, M blocking is superior to N blocking for this ρ range as it also increases B access temporal locality. At $\rho = 1.0$, $m_B = 16$ kernel is significantly faster than the one without M blocking by 48%.

Varying the Number of Unique Absolute Non-zero Values in A

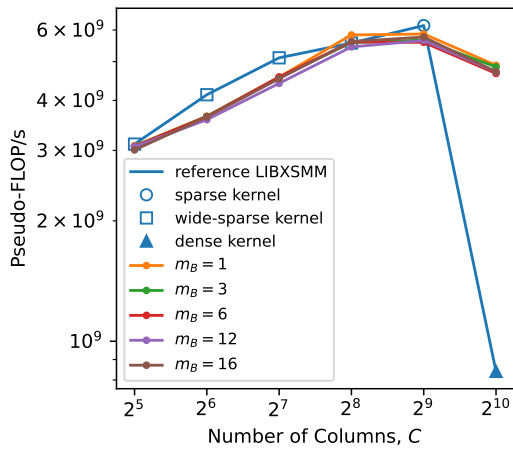
Figure 6.20 shows the performance of our M blocking kernels evaluated using the synthetic matrices containing varying numbers of unique absolute values. As shown in the figure, for the entire range of U , our routines show similar performance, irrelevant to the M blocking factor. Comparing to the reference wide-sparse kernel, our M blocking kernels are slower, as M blocking cannot effectively issue independent FMA instructions for very sparse matrices.



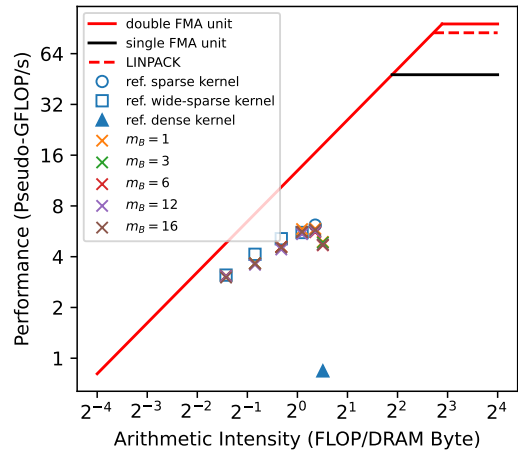
(a) Performance vs. number of columns, $U = 16$.



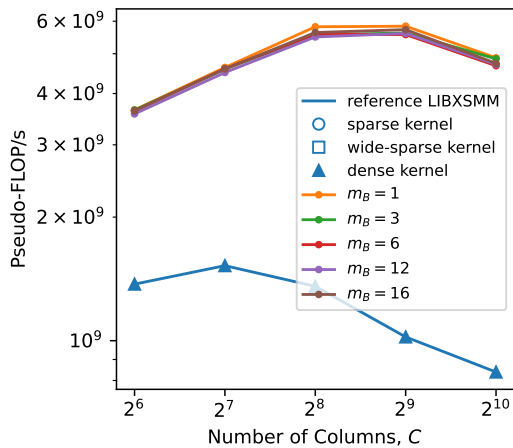
(b) Roofline plot, $U = 16$.



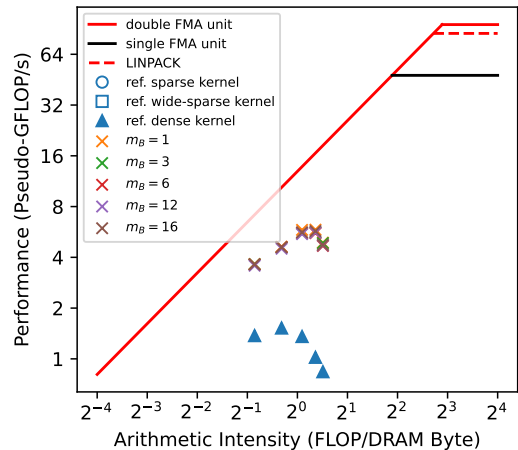
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

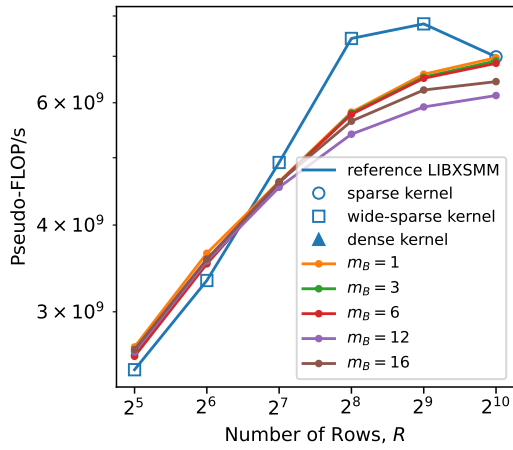


(e) Performance vs. number of columns, $U = 256$.

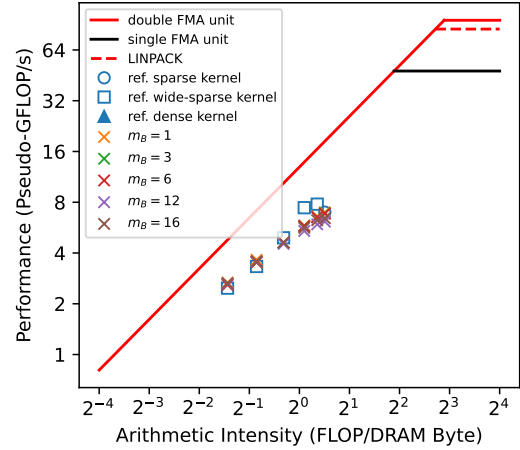


(f) Roofline plot, $U = 256$.

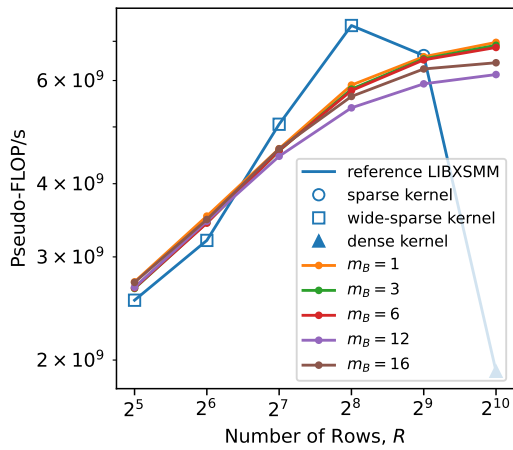
Figure 6.17: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns.



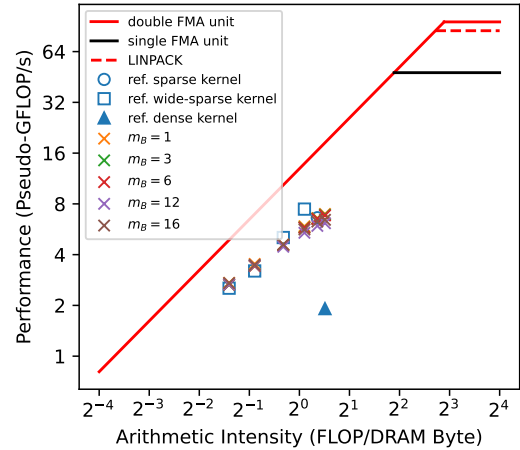
(a) Performance vs. number of rows, $U = 16$.



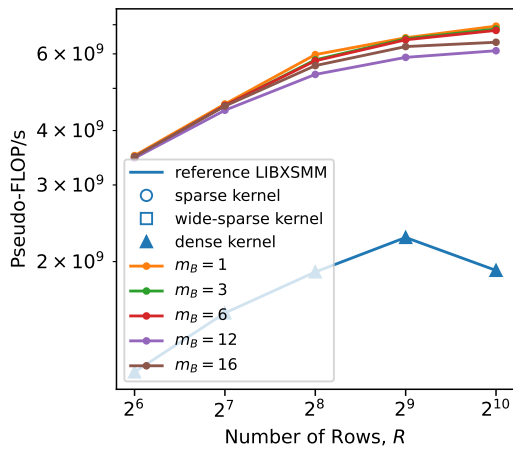
(b) Roofline plot, $U = 16$.



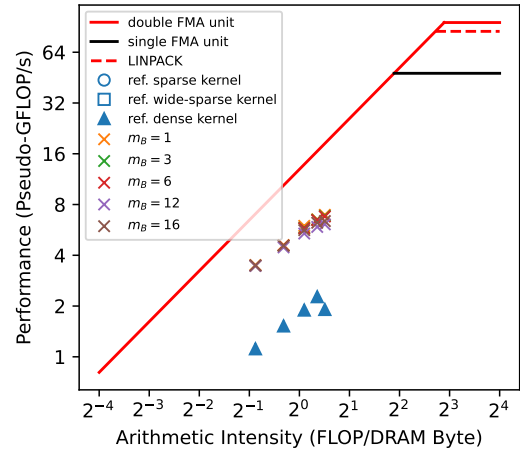
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

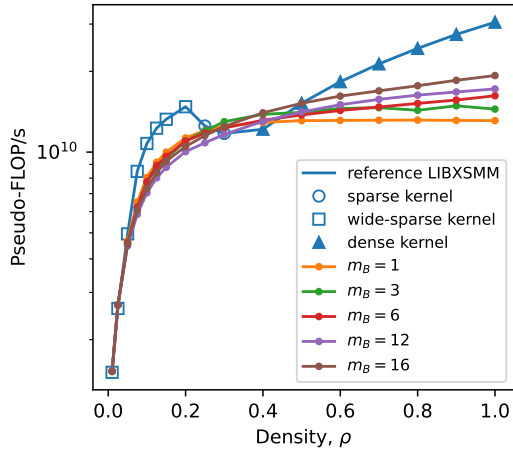


(e) Performance vs. number of rows, $U = 256$.

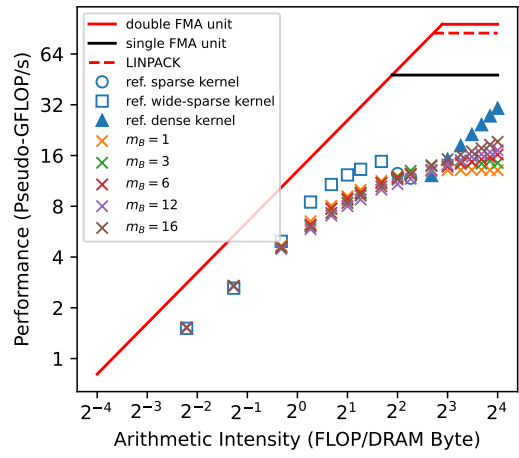


(f) Roofline plot, $U = 256$.

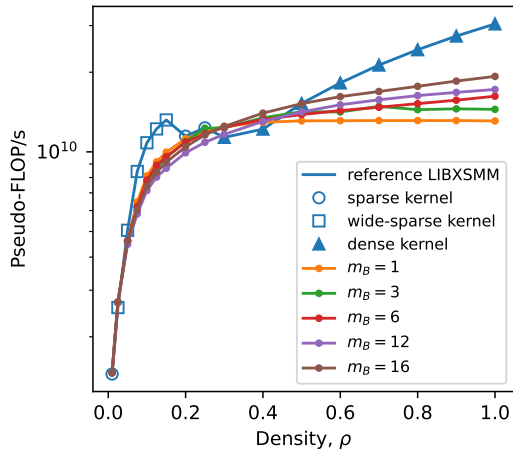
Figure 6.18: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} row.



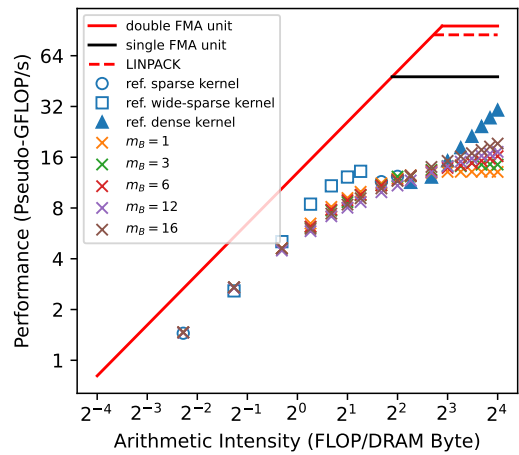
(a) Performance vs. density, $U = 16$.



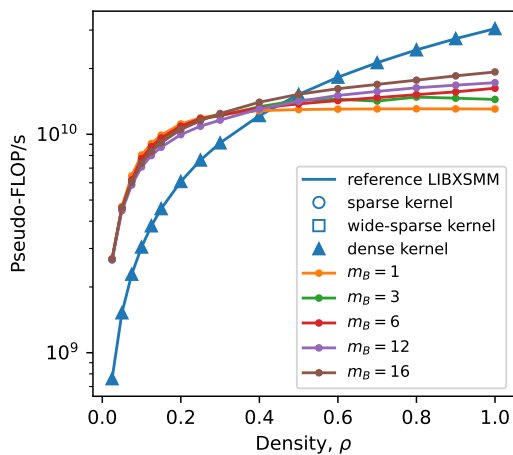
(b) Roofline plot, $U = 16$.



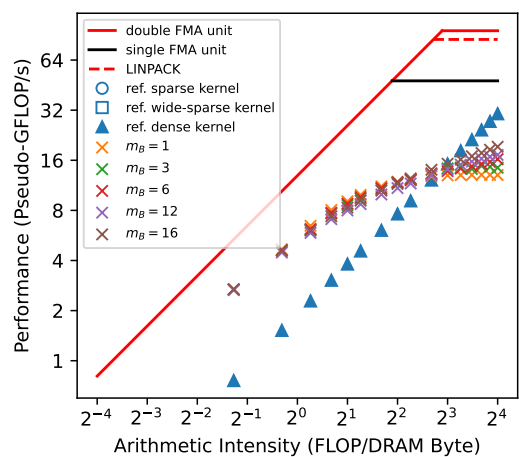
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

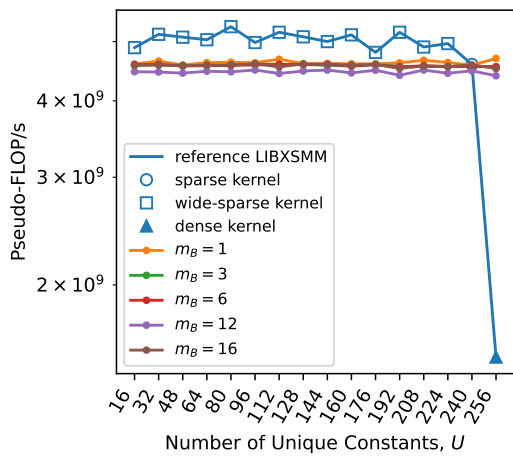


(e) Performance vs. density, $U = 256$.

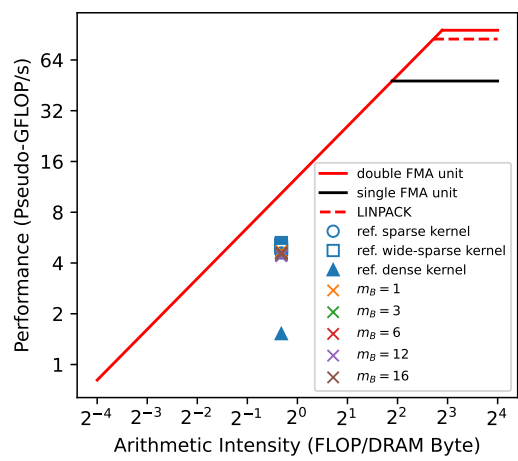


(f) Roofline plot, $U = 256$.

Figure 6.19: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with \mathbf{A} density.



(a) Performance vs. number of absolute non-zero values.



(b) Roofline plot.

Figure 6.20: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} .

6.6 Evaluation - Both N and M Blocking

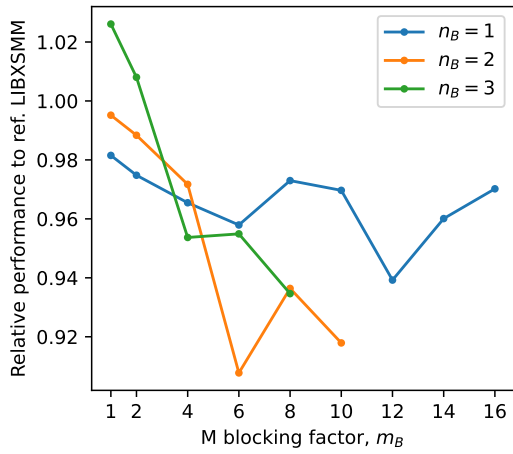
In the previous sections, we evaluated both N blocking and M blocking individually. Both these methods improve the kernel performance. N blocking provides a moderate speed up (up to $\sim 30\%$) for \mathbf{A} with density higher than 0.1. M blocking is better suitable for denser matrices. It can provide a significant speedup (up to $\sim 60\%$) for \mathbf{A} with density higher than 0.4. In this section we present our evaluation of our GEMM routine which utilise both N and M blocking techniques simultaneously.

Figure 6.21 show the relative performance of our routines with different N and M blocking factors comparing to reference LIBXSMM evaluated using PyFR matrices set on the c5n.xlarge machine. Similar to previous sections, we excluded the 6th order Gauss-Legendre m460 matrix as a reference sparse implementation cannot be generated due to the kernel size limitation. As shown in Figure 6.21a and 6.21b, M blocking slows down the kernels for all tested n_B s for the quadrilateral and hexahedral element matrices. M blocking cannot help these matrices because it is inefficient in generating independent FMA for very sparse matrices. Figure 6.21c and 6.21d show that M blocking can provided significant improvement for the dense tetrahedral and triangular matrices as they are more limited by instruction latency. For triangular matrices, some n_B and m_B settings can even provide better performance than the reference LIBXSMM implementation. As shown in 6.21d, once passing the optimum m_B setting, further m_B would only decrease the performance. In Chapter 8 later we will discuss potential causes of this in detail. Figure 6.21e shows the average performance evaluated on the entire PyFR matrices set. As shown in the figure, $n_B = 3, m_B = 2$ is the optimum setting for the PyFR matrices. $n_B = 2, m_B = 4$ also shows decent performance, which is only $\sim 2\%$ slower than the optimum one.

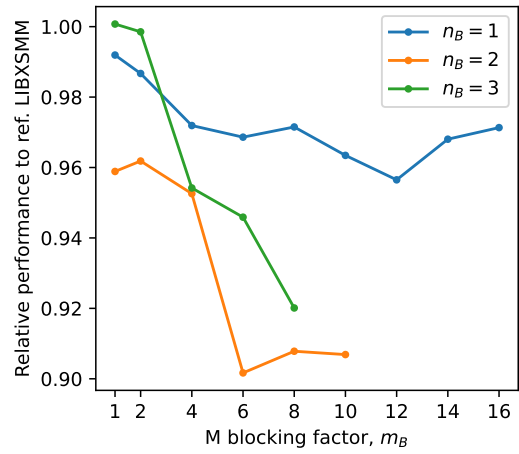
Summary

In this chapter we presented the design and evaluation of our sparse GEMM implementation with N blocking and M blocking techniques allowing multiple \mathbf{C} stride accumulations. N blocking provides a moderate speedup of around 30% for \mathbf{A} s with densities larger than 0.1. M blocking is more suitable for denser matrices as it could provide up to 60% speedup for \mathbf{A} s with densities higher than 0.4. Despite the significant performance increase, our routine is still generally slower than the LIBXSMM dense implementation. One reason behind this is that the reference dense method caches some \mathbf{B} strides in the vector registers to avoid redundant loads. In the Chapter 7 later we will improve our implementation with this technique. We also evaluated the performance of our implementation with different N and M blocking factors. Among all the tested combinations, N blocking factor of 3 together with M blocking factor of 2 provides the best average performance for all PyFR matrices. We observed that for the triangular matrices, once the M blocking factor passes the optimum one, further M blocking decreases the performance. In Chapter 8 we will discuss the potential cause

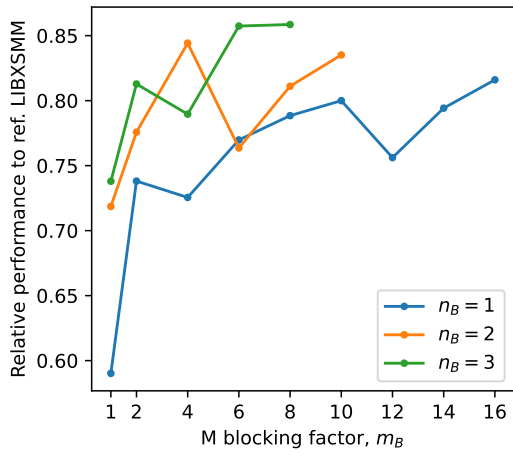
of this in detail.



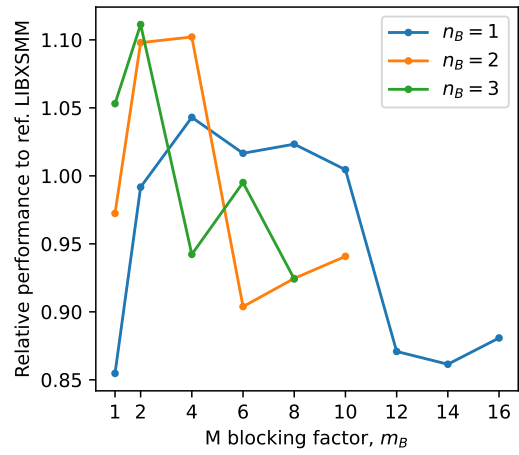
(a) Quadrilateral matrices.



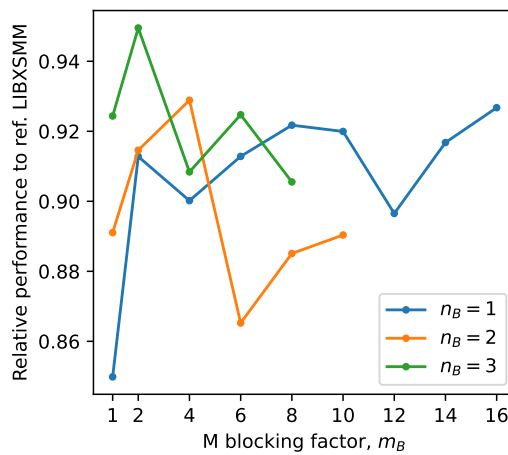
(b) Hexahedral matrices.



(c) Tetrahedral matrices.



(d) Triangular matrices.



(e) Entire PyFR matrices.

Figure 6.21: Performance of our routines with different N and M blocking factors evaluated using PyFR matrices set, relative to reference LIBXSMM.

Chapter 7

Small and Sparse GEMM Kernel with Caching \mathbf{B} Strides in Vector Registers

In Chapter 6 we presented our sparse GEMM implementations with multiple accumulations allowing us to hide some of the FMA instruction latency. Despite a significant performance increase, our routine is generally slower than the LIBXSMM dense implementation for relative dense PyFR matrices. One reason for this is that the reference dense routine caches some \mathbf{B} strides in the vector register to avoid later repeated \mathbf{B} access.

In this chapter, we will present and evaluate our efforts in improving our implementation with caching \mathbf{B} strides in the vector registers.

7.1 Runtime Broadcasting with FMA instruction

In our previous sparse GEMM implementations, we use `VBROADCASTSS/D` instruction to runtime broadcast each \mathbf{A} element before FMAs. In fact, for AVX-512, the FMA instructions can be used to broadcast single-precision or double-precision numbers from memory on-the-fly [16], as illustrated in Figure 7.1. The reason why runtime broadcasting using FMA was not employed is that the x86 FMA instructions accept at most one memory operand, which was used to pass in \mathbf{B} stride location in our previous implementations.

For our current implementation, in which we intend to cache some \mathbf{B} strides in the vector registers, \mathbf{A} elements can be runtime broadcasted by AVX-512 FMA instructions. This also frees the vector registers previously required for temporally holding the broadcasted \mathbf{A} elements, allowing a higher M blocking factor to be achieved.

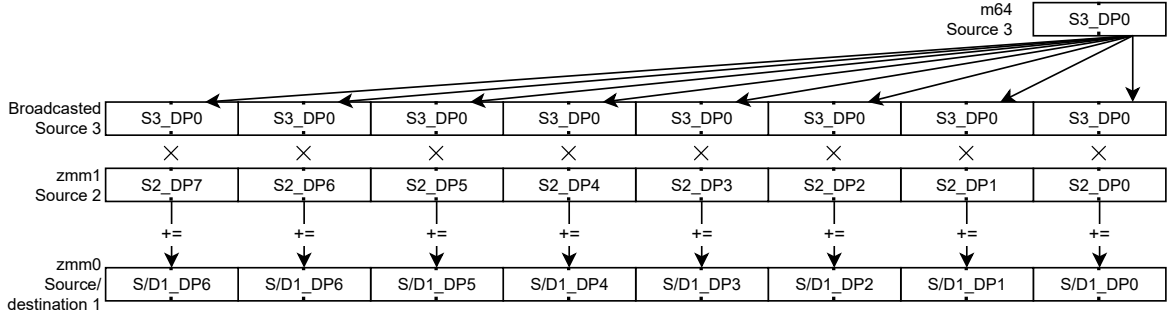


Figure 7.1: Runtime broadcasting and fused multiply-add using VFMADD231PD.

7.2 Kernel Design

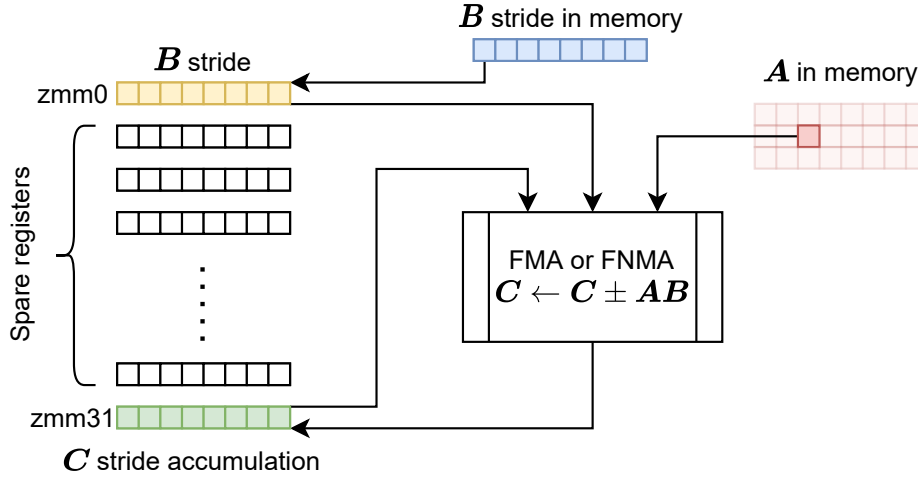


Figure 7.2: Diagrammatic representation of the small and sparse double-precision GEMM routine with caching B stride in vector registers and using AVX-512 FMA instruction for runtime broadcasting. No multiple accumulations were employed for this simple illustration.

Figure 7.2 illustrates our GEMM implementation which caches B strides in vector registers and uses FMA instruction to broadcast A elements in runtime. For simplicity, our example uses only one register for accumulating C strides but multiple accumulations can be readily adapted using the 30 spare registers. For N blocking, each n_B requires one register to cache the B strides and m_B registers for C accumulation. For M blocking, each m_B requires n_B registers for accumulating the C strides. For AVX-512, which contains 32 vector registers, n_B and m_B must obey

$$n_B + n_B \times m_B \leq 32. \quad (7.1)$$

For $m_B = 1$, i.e., no M blocking, the current implementation has exactly the same memory access pattern as our previous implementations - each FMA requires 1 8- or

4-byte read for loading \mathbf{A} element, and 1 64-byte read for loading \mathbf{B} stride. However, for implementations with M blocking, the current routine shows great advantages as it only loads a \mathbf{B} stride when it is first encountered in the current M block column. For executing this example matrix with $m_B = 3$:

$$\mathbf{A} = \begin{bmatrix} \underline{1} & 0 & 0 \\ 2 & 0 & \underline{2} \\ 3 & \underline{3} & \underline{3} \end{bmatrix},$$

\mathbf{B} strides are only loaded when the kernel is executing the underlined elements. Therefore, pairing with M blocking, caching \mathbf{B} strides in vector registers eliminates redundant \mathbf{B} loads.

Hiding Memory Latency with Pre-Loading \mathbf{B} Strides

Algorithm 1 Simplified representation of a section of our sparse GEMM routine which pre-loads \mathbf{B} strides to hide memory latency.

- 1: load the first \mathbf{B} stride to `zmm0`
 - 2: zero `zmm31` for \mathbf{C} accumulation
 - 3: load the next \mathbf{B} stride to `zmm1`
 - 4: compute FMA using \mathbf{B} stride from `zmm0` and accumulate the result to `zmm31`
 - 5: load the next \mathbf{B} stride to `zmm0`
 - 6: compute FMA using \mathbf{B} stride from `zmm1` and accumulate the result to `zmm31`
 - 7: load the next \mathbf{B} stride to `zmm1`
 - 8: ...
 - 9: load the last \mathbf{B} stride to `zmm1`
 - 10: compute FMA using \mathbf{B} stride from `zmm0` and accumulate the result to `zmm31`
 - 11: compute FMA using \mathbf{B} stride from `zmm1` and accumulate the result to `zmm31`
 - 12: store the \mathbf{C} stride accumulated at `zmm31` to memory
-

When evaluating this implementation, we observed that the performance is mostly limited by memory latency for loading \mathbf{B} strides. VTune profiling results show that for some matrices, our kernel spends more than 70% of the time waiting for \mathbf{B} strides from memory. The LIBXSMM dense implementation tries to overcome this issue by pre-loading \mathbf{B} strides several instructions before they are computed by FMA. The reference LIBXSMM sparse and wide-sparse routines use pre-fetch instructions to overcome this. For our implementation, we decided to use the pre-loading technique, as there are spare registers to cache extra \mathbf{B} strides. Algorithm 1 illustrates how we implemented pre-loading.

Depending on the matrices, our evaluation results show that the pre-loading technique can improve the performance by 2%-10%. Despite this improvement, our kernel is still mostly bound by memory latency. The reason for this is that memory latency

is typically longer than 10 nanoseconds, which is equivalent to more than 30 CPU cycles for the c5n.xlarge machine. Our pre-loading technique shown in Algorithm 1 only shifts the loading instructions one step forwards which is not sufficient to hide the entire memory latency. For this project, we were not able to implement a more aggressive pre-loading technique that can hide more memory latency.

Although Algorithm 1 only shows the kernel example using single accumulation, our implementation can be easily adapted with N blocking and M blocking. Because of pre-loading, each N blocking factor requires 2 vector registers for storing the \mathbf{B} strides. For Skylake-SP, n_B and m_B must obey:

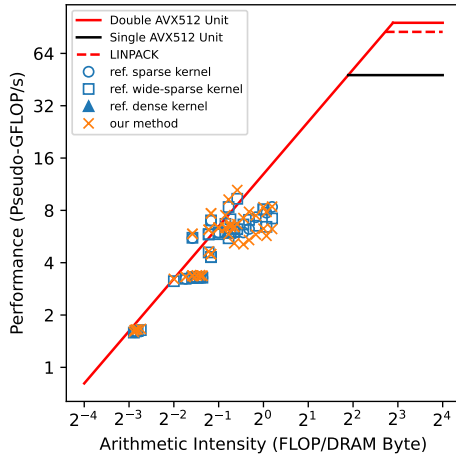
$$2n_B + n_B \times m_B \leq 32. \quad (7.2)$$

Therefore, for $n_B = 1$, maximum $m_B = 30$. For $n_B = 2$, maximum $m_B = 14$. For $n_B = 3$, maximum $m_B = 8$. Appendix B.4 includes some kernel examples with different N and M blocking factors generated by our current implementation.

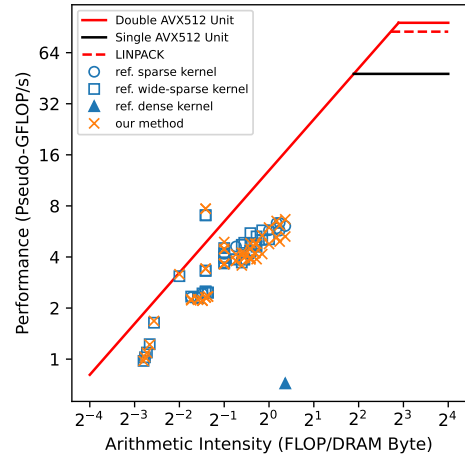
7.3 Performance Prediction

With our implementation been introduced, we predict our routine should perform as follows:

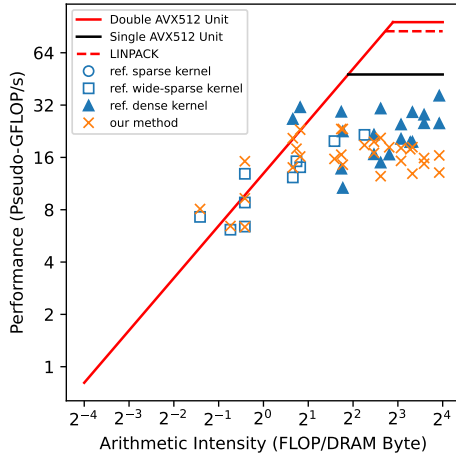
- Expectation 1. Without M blocking, our implementation should perform similarly to the reference sparse, the reference wide-sparse and our previous implementations discussed in Chapter 5 and 6 because of similar memory access patterns.
- Expectation 2. Without M blocking, our implementation should perform worse than the reference dense routine for relative dense \mathbf{A} as the cached \mathbf{B} strides are not reused in the absence of M blocking.
- Expectation 3. With M blocking, our implementation should perform better than the reference sparse, the reference wide-sparse and our previous implementations discussed in Chapter 5 and 6 as it caches and reuses \mathbf{B} strides to avoid repeated \mathbf{B} access.
- Expectation 4. Our implementation should show higher performance for larger M blocking factors, as larger m_B can issue more independent FMA instructions.



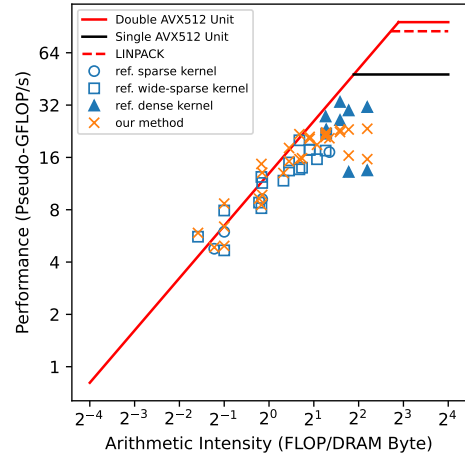
(a) Quadrilateral element matrices.



(b) Hexahedral element matrices.



(c) Tetrahedral element matrices.



(d) Triangular element matrices.

Figure 7.3: Roofline plots of runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations. Benchmark run on c5n.xlarge machine.

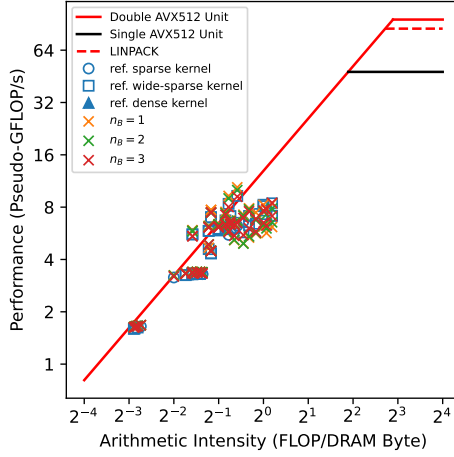
7.4 Evaluation - Single Accumulation

Figure 7.3 presents our evaluation results tested using the PyFR matrices on c5n.xlarge in the form of roofline plots. Comparing with the reference sparse, the reference wide-sparse and our previous implementation presented in Chapter 5, our implementation shows similar performance, satisfying Expectation 1. As shown in Figure 7.3c and 7.3d, for the dense tetrahedral and triangular element matrices, our implementation is slower than the LIBXSMM dense implementation, which satisfies Expectation 2. The benchmark on m5n.xlarge also supports these expectations.

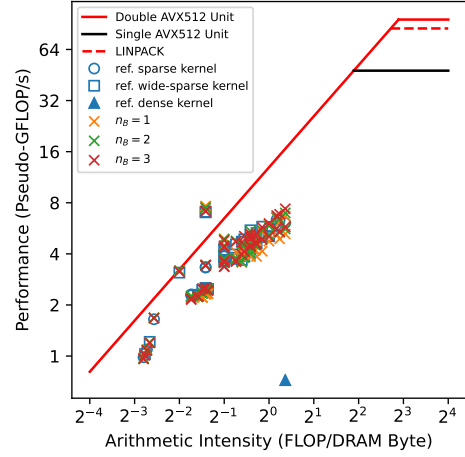
Because the results of our implementation evaluated on the synthetic set are mostly the same as what we have presented in Chapter C, they are not repeatedly discussed here. For the complete and detailed benchmark results evaluated on both c5n.xlarge

and m5n.xlarge machines, please refer to Appendix E.

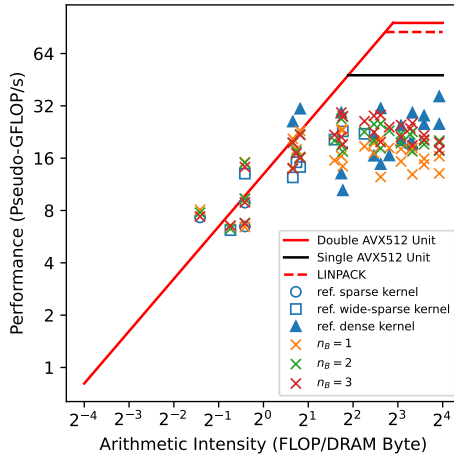
7.5 Evaluation - N Blocking



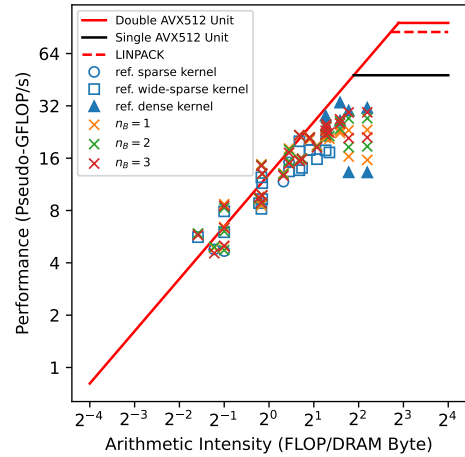
(a) Quadrilateral element matrices.



(b) Hexahedral element matrices.



(c) Tetrahedral element matrices.



(d) Triangular element matrices.

Figure 7.4: Roofline plots of runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations for varying N blocking factors. Benchmark run on c5n.xlarge machine.

Figure 7.4 presents the benchmark results of our implementation with only N blocking, evaluated using the PyFR matrices set on c5n.xlarge machine. Similar to Section 7.4 previously, our implementation performs comparably to the reference sparse, reference wide-sparse and our previous implementation presented in Chapter 6 because they have similar memory patterns. This satisfies Expectation 1. As shown in Figure 7.4c and 7.4d, although N blocking improves the kernel performance considerably, our implementation cannot outperform the reference dense routine because the latter one utilises a higher degree of multiple accumulations and reuses cached \mathbf{B} strides through

M blocking. This satisfies Expectation 2. The benchmark ran on m5n.xlarge also supports these expectations.

For the same reasoning as previously stated in Section 7.4, we will not discuss the benchmark results evaluated on the synthetic set here. Appendix E presents the complete test results on both c5n.xlarge and m5n.xlarge machines.

7.6 Evaluation - M Blocking

In this section, we present our evaluation of our implementation with only M blocking using our test suite (see Chapter 4), for $m_B = 1, 6, 12, 20, 30$.

PyFR Operator Matrices

In this section, we only present the roofline and performance data plotting against \mathbf{A} density (ρ) and the number of \mathbf{A} constants (U) measured on c5n.xlarge machine. Results from m5n.xlarge are very similar so there are not included in this section. For performance data plotting against the number of \mathbf{A} columns, the number of \mathbf{A} rows and \mathbf{A} size, and the results from m5n.xlarge, please refer to Appendix E.

Quadrilateral and hexahedral element matrices

Figure 7.6 and 7.7 show the benchmark results of our implementation paired with M blocking evaluated using the PyFR quadrilateral and hexahedral element matrices respectively. Similar to our previous implementation presented in Chapter 6, M blocking alone does not improve the kernel performance. This is because both the quadrilateral and hexahedral element matrices are very sparse (see Figure 7.6b and 7.7b) so that M blocking is not efficient in generating independent FMA instructions. The reason for this has been discussed in Section 6.2. In fact for some very sparse hexahedral element matrices, the reference wide-sparse routine outperforms our M blocking implementations because N blocking is superior in generating independent FMA instructions for these matrices. Therefore, our observation shows that M blocking is not superior for sparse matrices, which is against Expectation 3 and Expectation 4.

Tetrahedral and triangular element matrices

Figure 7.8 and 7.9 show the benchmark results of our implementation paired with M blocking evaluated using the PyFR tetrahedral and triangular element matrices. As shown in 7.8b and 7.9b, these matrices are much denser compared to the quadrilateral and hexahedral ones so that the M blocking kernels show significantly better

performance. For the cases in which the reference sparse or wide-sparse kernels are generated, our M blocking implementations show better performance than the reference ones by up to 30%. This observation satisfies Expectation 3. Additionally, our M blocking kernel can significantly outperform the reference dense kernels for the big and dense matrices. We observed a maximum performance increase of 78% and 168%, comparing to the reference dense routine and our routine without M blocking, respectively. We suggest three possible reasons why our implementation can outperform the reference dense routine even for 100% dense matrices. Firstly, our routine fully unrolls the loops, eliminating any branching overhead. Secondly, our routine packs \mathbf{A} constants in a compacted form which increases the spatial locality. Lastly, as we will shortly discuss, the dense routine uses the maximally available number of registers for multiple accumulations. In fact, having too many accumulation registers might impact the performance negatively.

Counter-intuitively, bigger m_B does not always bring better performance. For both the tetrahedral and triangular element matrices, $m_B = 6$ shows a significantly better performance than $m_B = 20$ and $m_B = 30$. In fact, for a few triangular matrices, $m_B = 30$ has worse performance than the kernel without using M blocking at all ($m_B = 1$). Our observation indicates too large M blocking factors can impact the performance negatively, which does not support Expectation 4. It is worth noting that for many tetrahedral and triangular matrices, the $m_B = 30$ and the reference dense routines show very similar performances. Both of these methods use the maximally available number of registers for accumulating the \mathbf{C} strides. Therefore, we suggest it might be possible to increase the dense kernel performance by decreasing the degree of multiple accumulations.

PyFR Operator Matrices - Summary

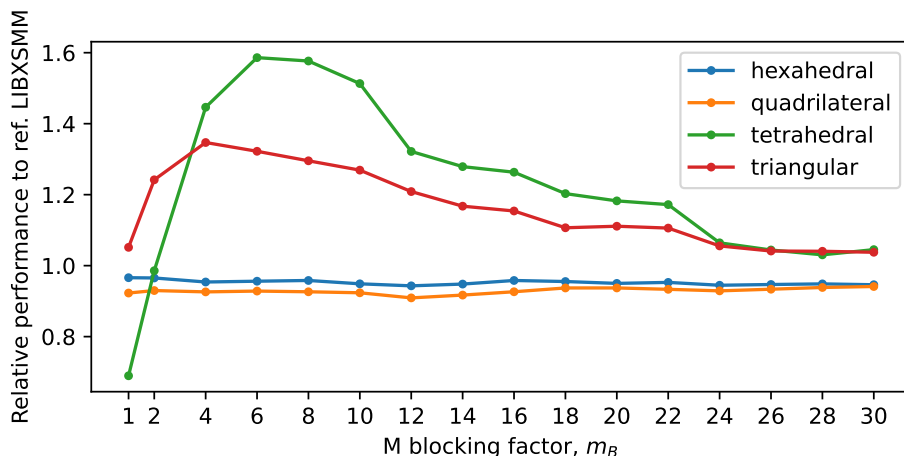
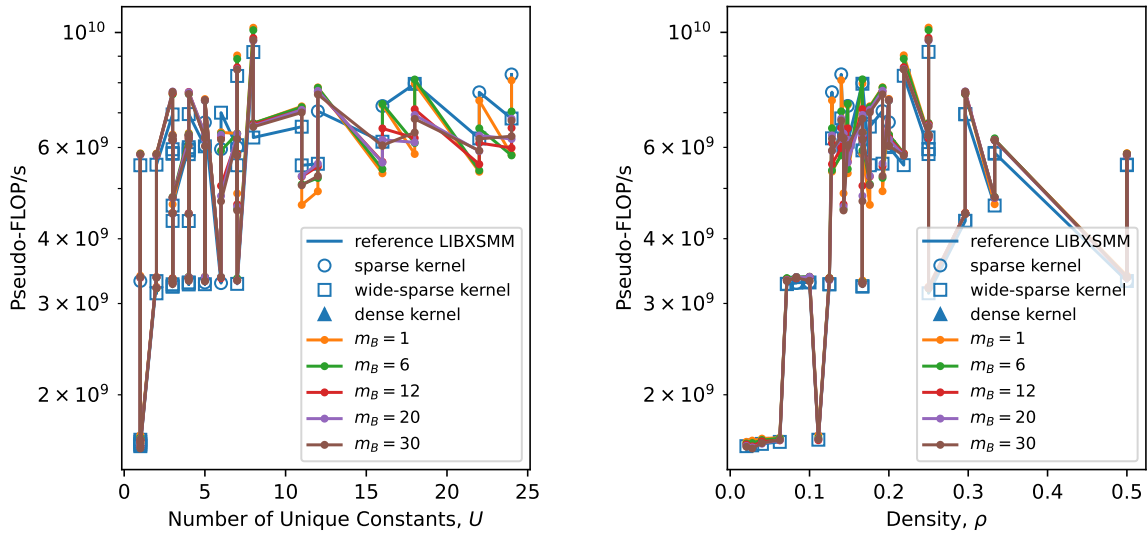


Figure 7.5: Average performance of our routines using different degrees of M blocking evaluating using PyFR matrices, relative to reference LIBXSMM.

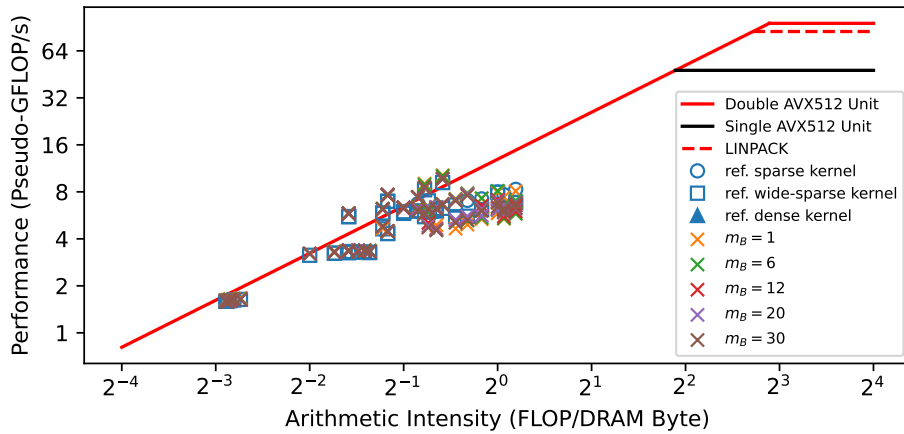
Figure 7.5 shows the average performance of our M blocking implementation with dif-

ferent M blocking factors, relative to reference LIBXSMM. We excluded the 6th order hexahedral Gauss-Legendre m460 matrix as a reference sparse kernel cannot be generated for it. As previously discussed, M blocking does not improve performance for the sparse quadrilateral and hexahedral element matrices. For tetrahedral and triangular element matrices, M blocking can increase the performance significantly, but not for very large m_B . Again, this does not support Expectation 4. For tetrahedral element matrices, the optimum m_B is 6, which outperforms reference LIBXSMM by 59% on average. For triangular element matrices, the optimum m_B is 4, which outperforms reference LIBXSMM by 35% on average.



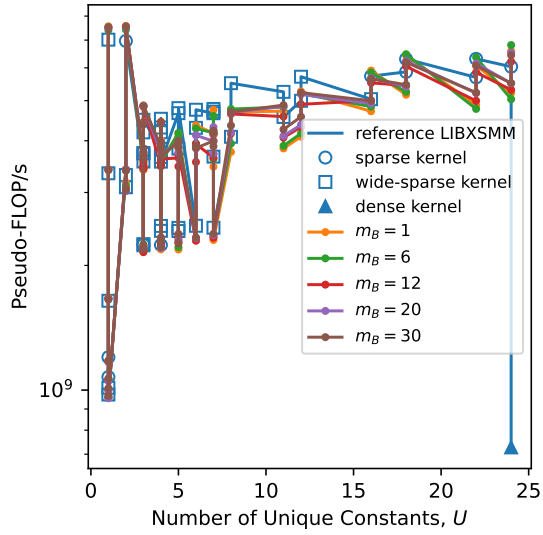
(a) Performance against number of unique A constants.

(b) Performance against A density.

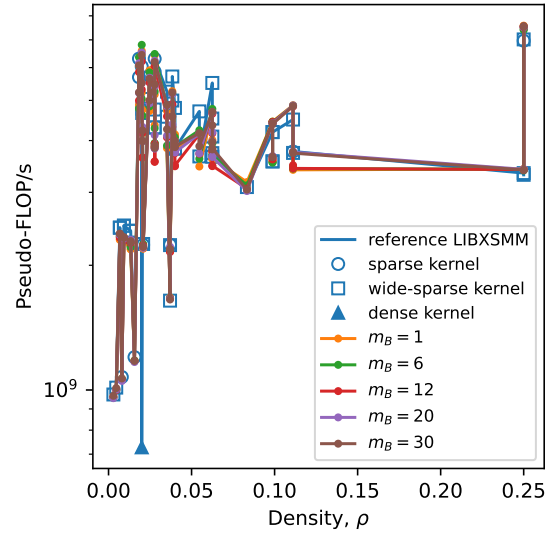


(c) Roofline plot.

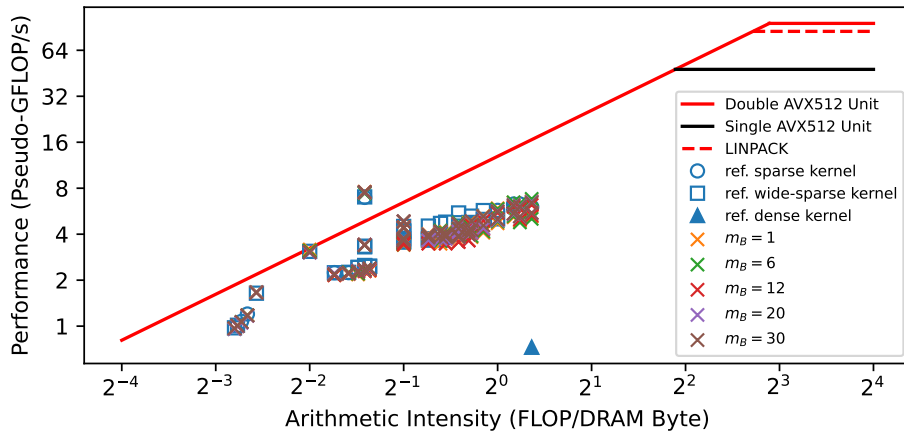
Figure 7.6: Runtime broadcasting with loading A from memory, caching B in vector registers and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral element operator matrices.



(a) Performance against number of unique A constants.

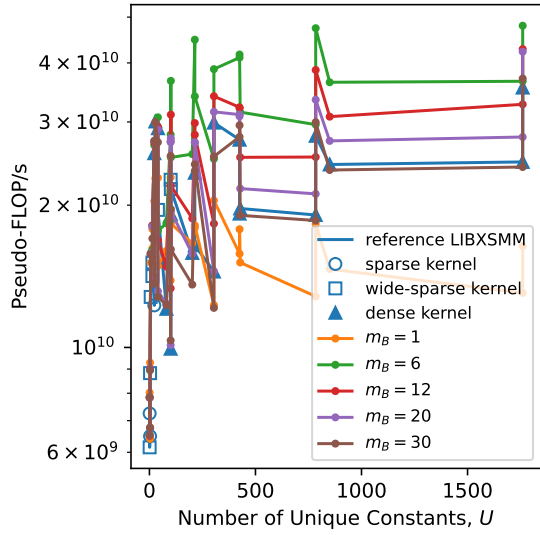


(b) Performance against A density.

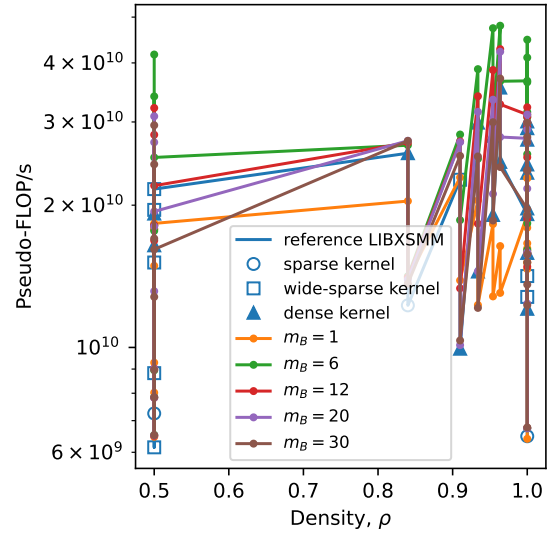


(c) Roofline plot.

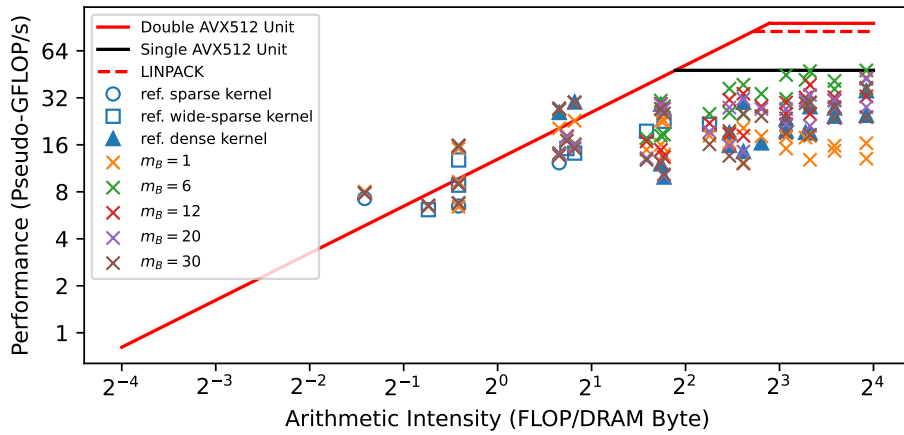
Figure 7.7: Runtime broadcasting with loading A from memory, caching B in vector registers and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral element operator matrices.



(a) Performance against number of unique A constants.

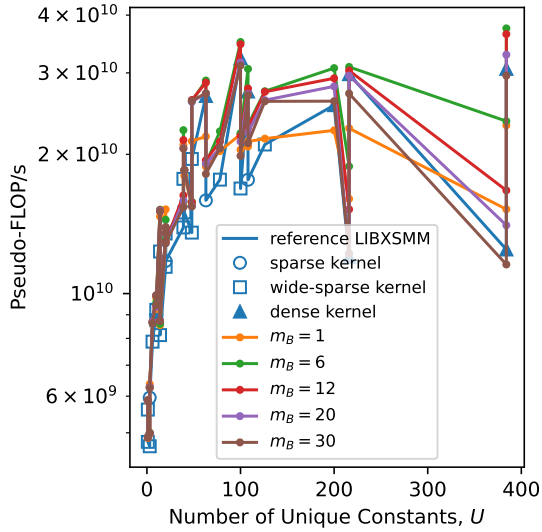


(b) Performance against A density.

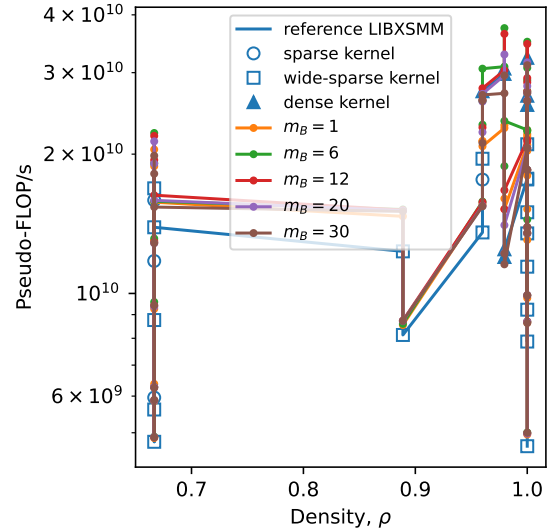


(c) Roofline plot.

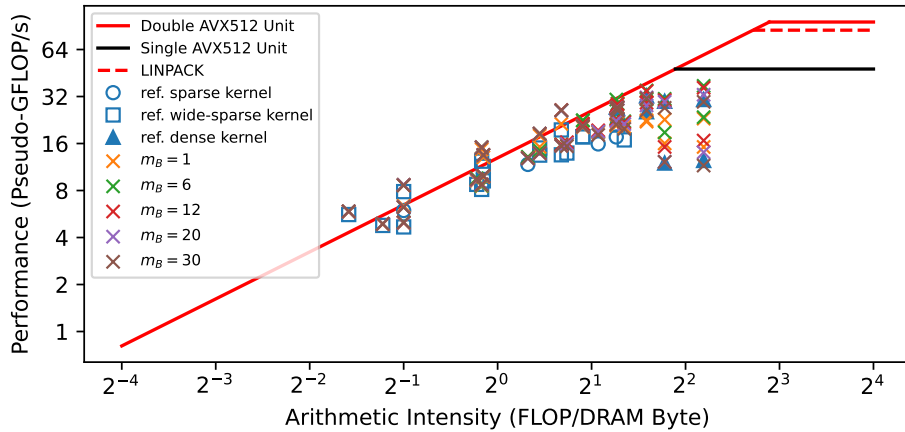
Figure 7.8: Runtime broadcasting with loading A from memory, caching B in vector registers and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral element operator matrices.



(a) Performance against number of unique A constants.



(b) Performance against A density.



(c) Roofline plot.

Figure 7.9: Runtime broadcasting with loading A from memory, caching B in vector registers and N blocking vs. reference LIBXSMM implementations, for PyFR triangular element operator matrices.

Synthetic Matrices

In this section, we present the performance of our kernels with different M blocking factors, evaluated using the synthetic test set. Similar to evaluation on PyFR matrices, the results from `c5n.xlarge` and `m5n.xlarge` are very similar so we only present the `c5n.xlarge` here. The `m5n.xlarge` results are included in Appendix E.

Varying Number of A Columns

Figure 7.10 shows the benchmark results of our M blocking implementation evaluated using the synthetic matrices with varying numbers of rows. For $U = 16$ and $U = 64$, our routine is generally slower than the reference dense and wide-sparse routines, which is against Expectation 3. This is because the synthetic matrices are very sparse ($\rho = 0.05$), so M blocking is less efficient in generating independent FMA instructions. Comparing among different M blocking factors, it is not the case that kernels with larger m_B will always outperform the ones with small m_B , which is against Expectation 4. For $U = 16$ and $2^6 \leq C \leq 2^8$, $m_B = 1$ kernel is more performant than all the M blocking kernels.

Varying Number of A Rows

Figure 7.11 present the performance of our M blocking kernels evaluated using the synthetic matrices with varying numbers of rows. Similar to our previous discussion, the reference sparse and wide-sparse routines generally outperform our M blocking kernels, except for the very large R for which the reference sparse kernel cannot be generated because of the kernel size limitation. This is because N blocking is superior to M blocking in generating independent FMA instructions for sparse matrices. Therefore Expectation 3 is not supported here. Comparing among different M blocking implementations, there is no significant performance difference for our routines with different m_B , except $m_B = 12$ is $\sim 7\%$ less performant than the other m_B s. Because this performance decrease is insignificant and can easily be caused by hardware sensitivity to memory access patterns, it was not further investigated for our project.

Varying A density

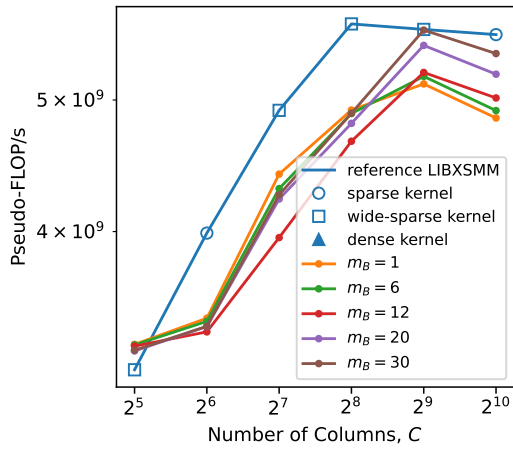
Figure 7.12 shows the benchmark results of our M blocking kernels evaluated using the synthetic set with varying densities. Comparing among our M blocking implementations, M blocking does not provide any significant performance improvement for all of the U s and $\rho < 0.2$. This is because 1. performance of these matrices is mostly limited by memory bandwidth but not instruction latency, 2. M blocking is not efficient for generating independent FMAs for sparse matrices. For $\rho \geq 0.2$, M blocking can pro-

vide significant performance improvements. For completely dense matrices, $m_B = 6$ kernel is 180% more performant than the one without using M blocking. However, it is not the case that higher m_B is always better. For all of the U s, at very high ρ , $m_B = 6$ kernels generally significantly outperform the ones with $m_B = 30$. This does not support Expectation 4.

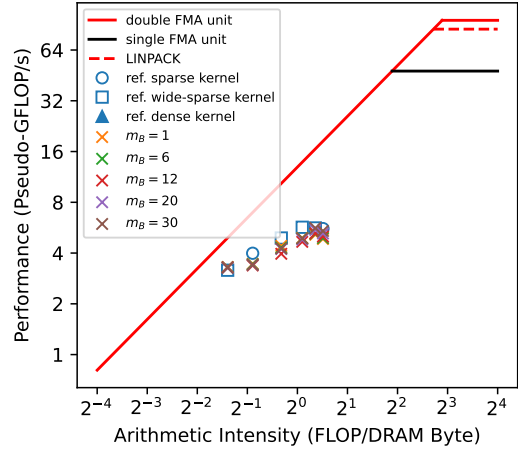
Comparing our M blocking implementations with the reference LIBXSMM routines, our kernels are slower than the wide-sparse kernel for $U = 16, 64$ and $\rho < 0.2$. This is because N blocking is more efficient in generating independent FMA for sparse matrices. Therefore Expectation 3 is not correct for sparse matrices. For $\rho \geq 0.2$ our M blocking kernels outperform both the reference sparse and the reference dense routines. The reasons for the superiority have been previously discussed when analysing the results of PyFR matrices, so they are not repeated here.

Varying the Number of Unique Absolute Non-zero Values in \mathbf{A}

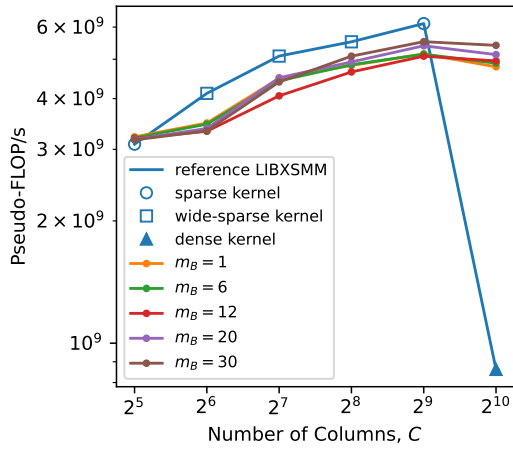
Figure 7.13 shows the benchmark results of our M blocking kernels evaluated using the synthetic matrices set with varying numbers of absolute non-zero values in \mathbf{A} . Our M blocking kernels are around 14% slower than the reference wide-sparse routine because N blocking is more efficient in generating independent FMAs for sparse matrices. This does not support Expectation 3. Comparing among different M blocking factors, the M blocking kernels show very similar performance irrelevant to m_B , except for $m_B = 12$, which is around 7% slower. The slow down for $m_B = 12$ is not significant and can be caused by hardware sensitivity to memory access patterns so it is not further investigated in this project.



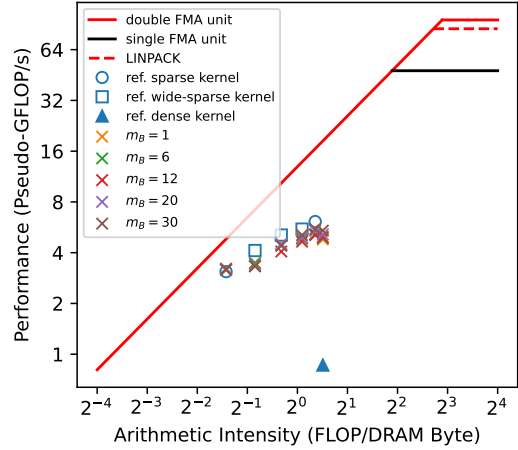
(a) Performance vs. number of columns, $U = 16$.



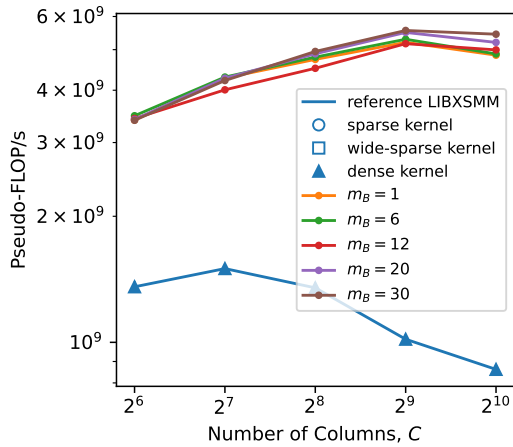
(b) Roofline plot, $U = 16$.



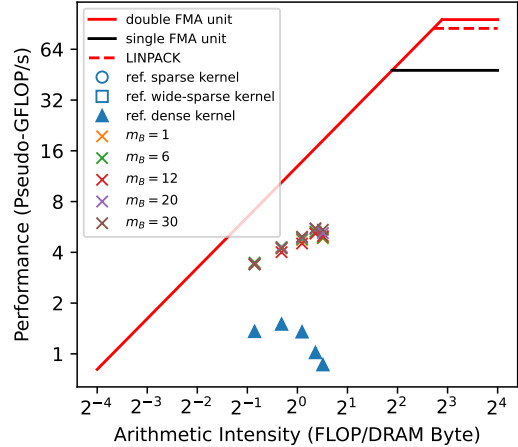
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

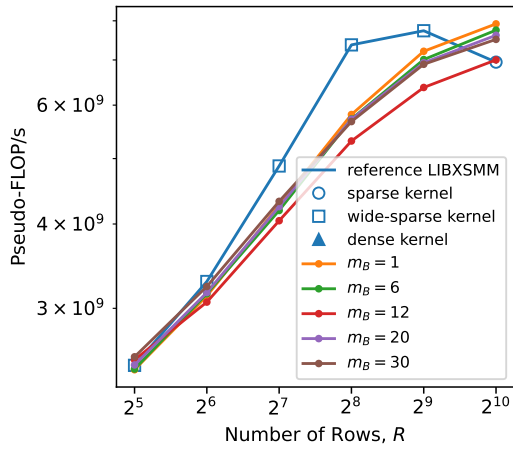


(e) Performance vs. number of columns, $U = 256$.

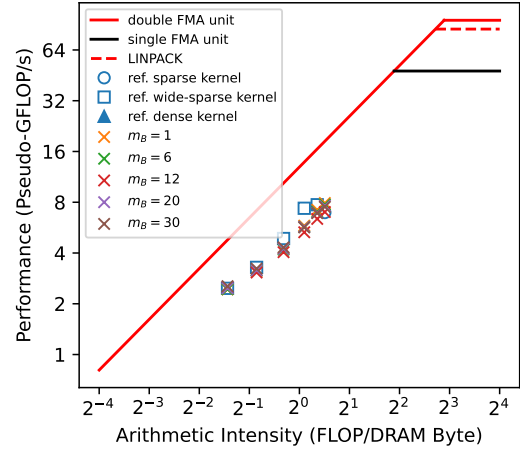


(f) Roofline plot, $U = 256$.

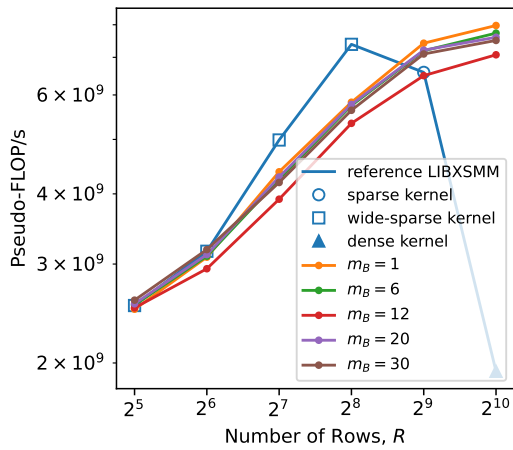
Figure 7.10: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns.



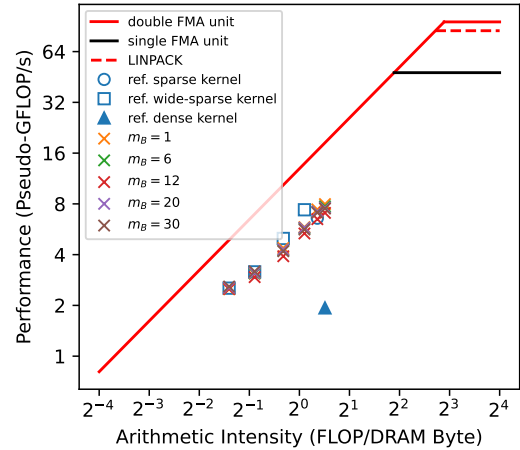
(a) Performance vs. number of rows, $U = 16$.



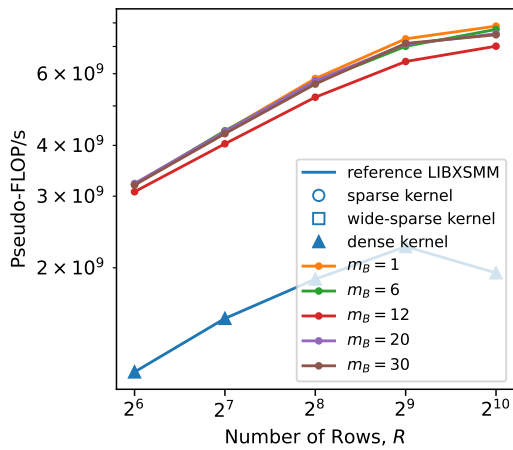
(b) Roofline plot, $U = 16$.



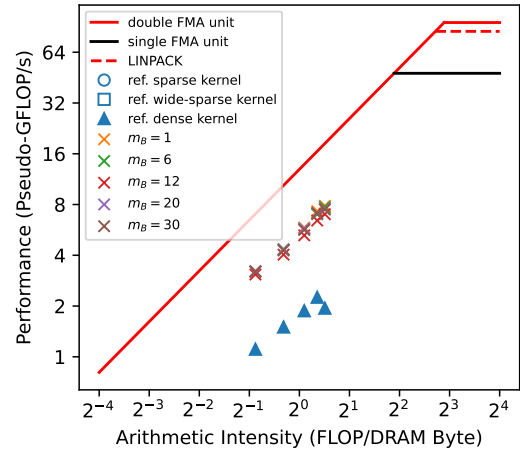
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

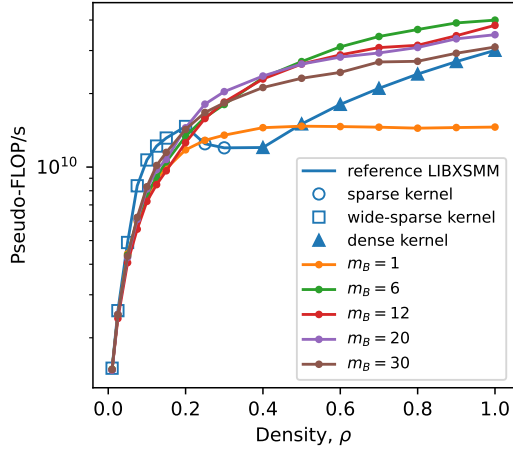


(e) Performance vs. number of rows, $U = 256$.

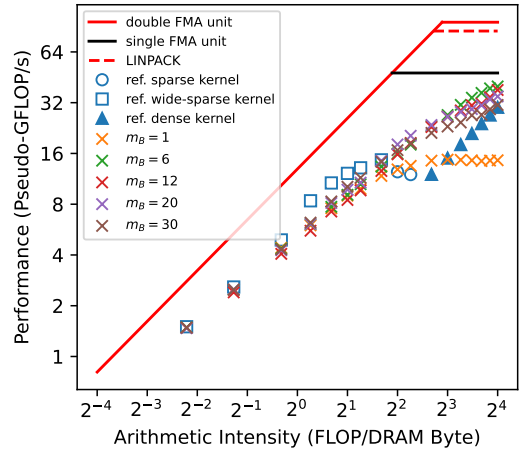


(f) Roofline plot, $U = 256$.

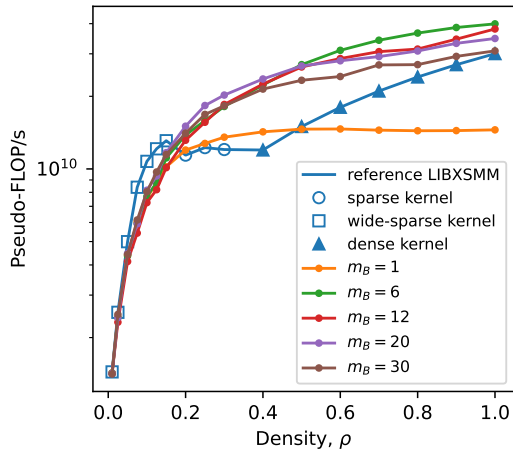
Figure 7.11: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows.



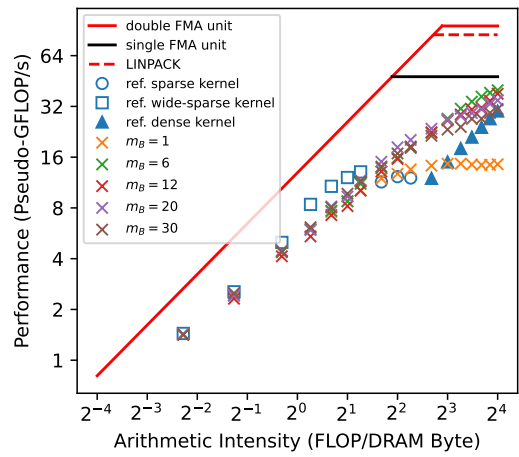
(a) Performance vs. density, $U = 16$.



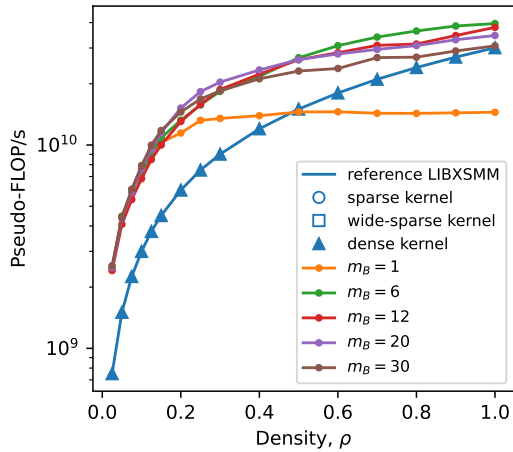
(b) Roofline plot, $U = 16$.



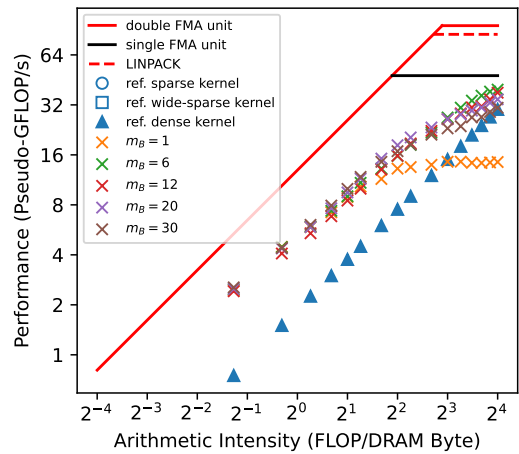
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

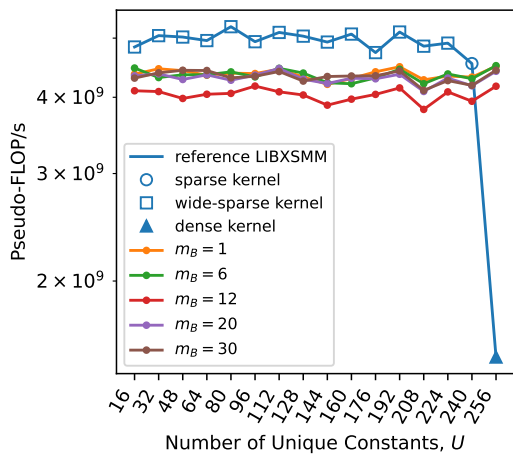


(e) Performance vs. density, $U = 256$.

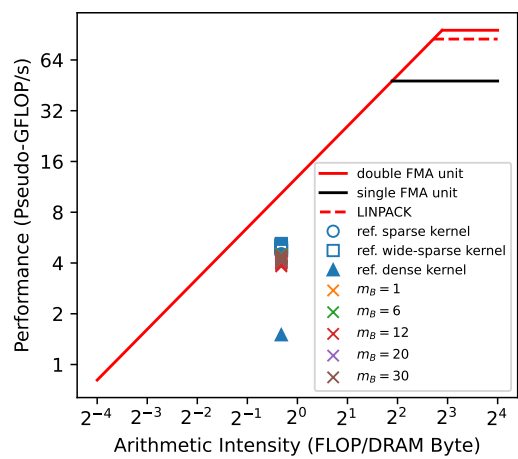


(f) Roofline plot, $U = 256$.

Figure 7.12: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with \mathbf{A} density.



(a) Performance vs. number of absolute non-zero values.



(b) Roofline plot.

Figure 7.13: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} .

7.7 Evaluation - Both N and M Blocking

Figure 7.14 shows the average performance of our implementation with different settings of n_B and m_B evaluated using the PyFR matrices set, relative to the performance of reference LIBXSMM. As shown in Figure 7.14a and 7.14b, for quadrilateral and hexahedral element matrices, the performance is mostly determined by N blocking factor and M blocking does not affect the performance significantly. Again, this is because M blocking is ineffective in generating independent FMAs for these sparse matrices. Because of this, the cached \mathbf{B} strides are not frequently reused so caching \mathbf{B} strides does not improve the performance. For quadrilateral and hexahedral element matrices, the best setting is $n_B = 3$ with small m_B , which has a similar average performance to reference LIBXSMM.

Figure 7.14c and 7.14d show the average performance evaluated using the tetrahedral and triangular element matrices. For these dense matrices, M blocking with caching \mathbf{B} strides can significantly improve the performance. The best setting is $n_B = 2$, $m_B = 4$ for both tetrahedral and triangular element matrices which can outperform reference LIBXSMM by 63% and 38% respectively. For these matrices, we observe a performance decline for very large m_B which is against Expectation 4. The exact reason for this performance decline is unsure at this stage but we will discuss some potential causes in Chapter 8 later.

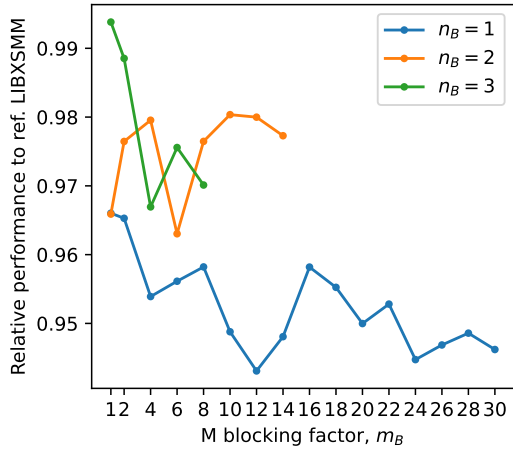
Figure 7.14e shows the average performance for the entire PyFR matrices set. $n_B = 2$, $m_B = 4$ is averagely the optimum setting for all PyFR operator matrices. Coincidentally this setting matches the minimum number of accumulation registers required to fully utilise the FMA units, as for Skylake-SP, FMA has an IPC of 2 and a latency of 4 cycles. With this setting, our implementation is 11% faster than reference LIBXSMM. It is worth noting that this 11% performance increase is significantly lower than the 63% increase for tetrahedral matrices. This is because the average performance is calculated using weight arithmetic mean weighted with execution time. As shown in Appendix A, there are twice more quadrilateral and hexahedral element matrices than the tetrahedral and triangular ones, so the performances of the former matrices dominate the average values.

Summary

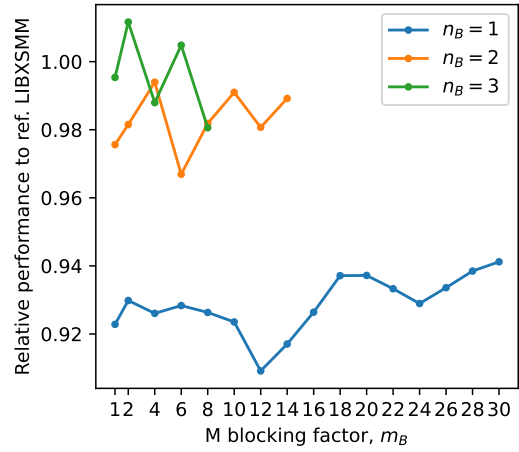
In this chapter, we presented our improved small and sparse GEMM kernel for AVX-512 which caches \mathbf{B} strides in vector registers. Because the \mathbf{B} strides are now passed to FMA instructions as register operands, we use AVX-512 FMA instructions for runtime broadcasting \mathbf{A} elements. During the initial design period, we noticed that the performance is mostly limited by memory access latency so we implemented a technique that pre-loads \mathbf{B} strides a few instructions in advance to hide memory latency. Our method can be readily adapted with N and M blocking, allowing multiple accu-

mulations to hide instruction latency. More importantly, M blocking allows reusing cached \mathbf{B} strides, eliminating repeated \mathbf{B} loads.

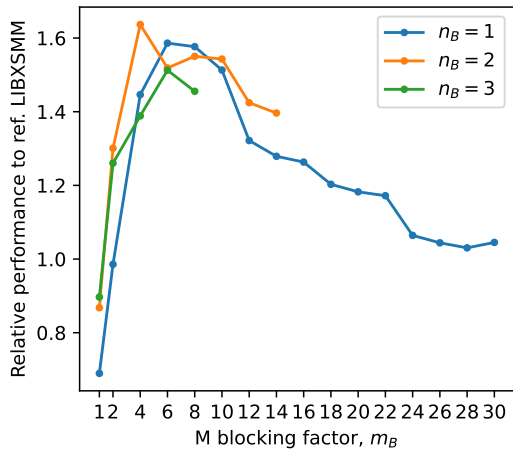
We evaluated our kernel using the benchmark set presented in Chapter 4. Without using M blocking, our kernels show very similar performance to our previous implementations discussed in Chapter 5 and 6, because they have similar memory access patterns. Our M blocking kernels show a significant performance improvement for the PyFR tetrahedral and triangular element matrices. We experimentally determined that, for PyFR operator matrices, the optimum N and M blocking setting is $n_B = 2$, $m_B = 4$. At this setting, our implementation outperforms reference LIBXSMM by 63% and 38% for tetrahedral and triangular element matrices, respectively. The performance for quadrilateral and hexahedral element matrices remains similar to reference LIBXSMM. Similar to Chapter 6, we observed that a too big m_B could decrease the kernel performance. We suggest that the reference dense kernel performance could be limited by this as it always uses the maximum M blocking factors. In Chapter 8 later, we will discuss some potential causes for this behaviour.



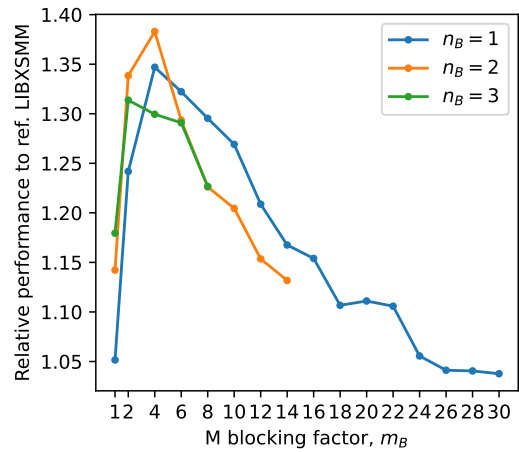
(a) Quadrilateral element matrices.



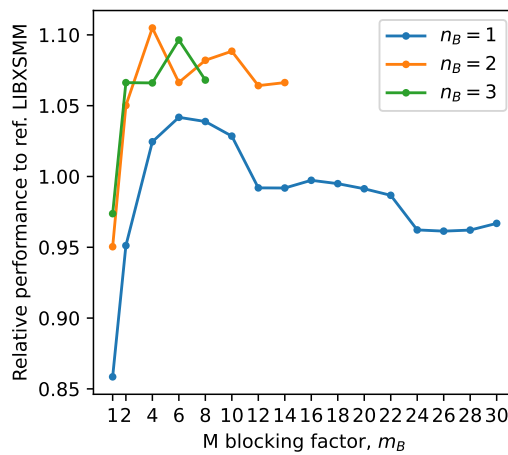
(b) Hexahedral element matrices.



(c) Tetrahedral element matrices.



(d) Triangular element matrices.



(e) Entire PyFR matrices set.

Figure 7.14: Performance of our caching \mathbf{B} strides routines with different N and M blocking factors evaluated using PyFR matrices set, relative to reference LIBXSMM.

Chapter 8

Possible Reasons for Why Too Large M Blocking Factors Decrease Kernel Performance

In Chapter 6 and 7 we presented two improved small and sparse GEMM implementations. For these implementations, utilisation of M blocking can achieve significant performance improvement for the dense PyFR tetrahedral and triangular element matrices. Counter-intuitively, we observed that the optimum M blocking factor is never the largest possible one, which uses the maximally available number of vector registers for accumulating C strides. In fact, the largest M blocking factor can severely decrease the kernel performance. In Chapter 7, we suggested this behaviour might be limiting the performance of the reference dense routine, which always tries to utilise as many vector registers available for multiple accumulations.

In this chapter, we present our investigation into potential causes of such behaviour.

8.1 Saturation of Write Buffer by Multiple Non-Temporal Stores

For modern computer hardware, memory access is generally an order of magnitude slower than CPU clock speed. Instead of initiating a write signal directly to the main memory and wait for the write to complete, modern CPUs usually writes the data to a write buffer [18]. The memory controller will handle the writing process from the write buffer so the CPU can continue on its execution.

As mentioned in Chapter 4, our kernels use non-temporal store instruction `vmovntpd` for streaming C strides from registers to main memory after GEMM computations. Comparing to the ordinary store instructions without the non-temporal hint, which

writes to the cache, the non-temporal store writes directly to the main memory so are prone to write stall. We suggest that with multiple accumulations, the amount of non-temporal stores issued in a small window can saturate the write buffer so that the CPU has to wait for the writes to complete.

Hypothesis Test

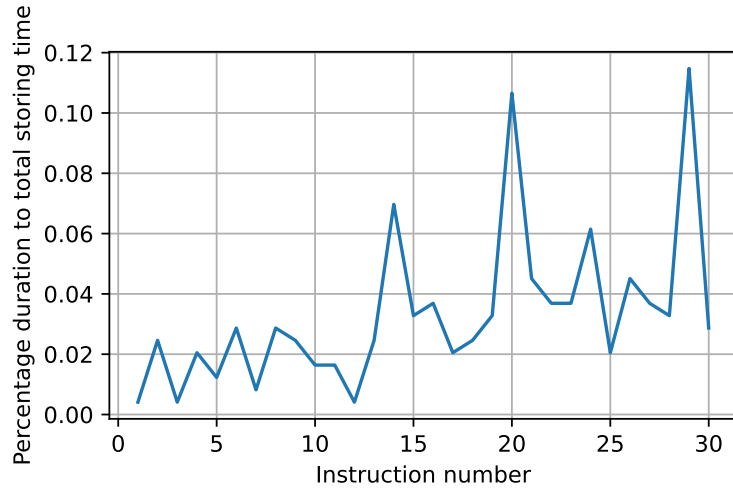


Figure 8.1: Percentage duration of each `vmovntpd` comparing to the total duration of 30 `vmovntpd` instructions.

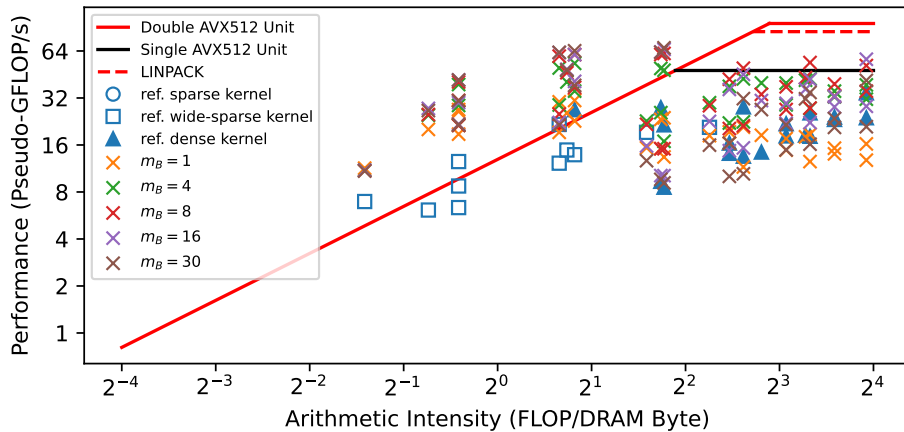


Figure 8.2: Roofline plots of our “broken” kernel with no non-temporal store evaluated using the PyFR tetrahedral element matrices.

For testing the hypothesis, we profiled our kernel discussed in Chapter 7 with a setting of $n_B = 1$ and $m_B = 30$ executing on the 6th order tetrahedral Shunn Ham m3 matrix using Intel VTune profiler running on the `c5n.metal` machine. With Intel VTune profiler, we were able to get the CPU time of each instruction. Figure 8.1 shows the percentage duration of each `vmovntpd` instruction, comparing to the total duration

of 30 `vmovntpd` instructions. As shown in the figure, the CPU spends less time on the initial 12 `vmovntpd` instructions than the later 18 instructions, suggesting possible write stalls.

We later prototyped a “broken” kernel which does not issue any non-temporal store instruction. Although this kernel does not compute correctly, the benchmark result should reflect if the non-temporal store instructions are the only cause for slow performance for big M blocking factors. The evaluation results are presented in Figure 8.2. As shown in the figure, $m_B = 8$ is still more performant than $m_B = 16$ and $m_B = 30$ for many data points, indicating other factors contributing to the slower performance for large m_{BS} .

8.2 CPU Stalls Due to Loading B Strides

As shown by the screenshots of our VTune profiler results in Figure 8.3 and 8.4, the `vmovupd` instructions for loading **B** strides usually cause the CPU to stall. The stall time is higher for $m_B = 30$ than $m_B = 8$. Additionally, the first `vmovupd` instruction is also blocking the subsequent `xor` instruction despite these two instructions have no data dependency. We suggested and explored two possible reasons for this:

1. $m_B = 30$ kernel is slower because CPU has a fewer number of free physical registers.
2. $m_B = 30$ kernel is slower because there are bigger FMA instruction “chunks”. The big chunk size prevents the out-of-order (OOO) execution engine to detect and issue `vmovupd` in advance to exploit memory-level parallelism.

Hypothesis Test - CPU Run Out of Free Registers

For testing this hypothesis, we prototyped a kernel which occupies all the remaining `zmm` registers using `xor` instructions. Therefore, regardless of the M blocking factors, the CPU should have the same number of free physical registers when executing different kernels.

Figure 8.5 shows the benchmark result of our modified kernel evaluated using the PyFR tetrahedral element matrices. As shown in the figure, $m_B = 8$ kernel still outperforms the $m_B = 30$ one, indicating CPU running out of free registers is not the cause.

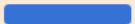

| Address ▲ | S... | Assembly | 🔥 CPU Time: Total |
|----------------|------|--|---|
| 0x7f4de27d4007 | | push r15 | 0.001s |
| 0x7f4de27d4009 | | vmovupd zmm0, zmmword ptr [rsi] | 0.002s |
| 0x7f4de27d400f | | vpxord zmm2, zmm2, zmm2 | 2.031s  |
| 0x7f4de27d4015 | | vpxord zmm3, zmm3, zmm3 | |
| 0x7f4de27d401b | | vpxord zmm4, zmm4, zmm4 | |
| 0x7f4de27d4021 | | vpxord zmm5, zmm5, zmm5 | |
| 0x7f4de27d4027 | | vpxord zmm6, zmm6, zmm6 | 0.005s |
| 0x7f4de27d402d | | vpxord zmm7, zmm7, zmm7 | |
| 0x7f4de27d4033 | | vpxord zmm8, zmm8, zmm8 | |
| 0x7f4de27d4039 | | vpxord zmm9, zmm9, zmm9 | |
| 0x7f4de27d403f | | vpxord zmm10, zmm10, zmm10 | 0.004s |
| 0x7f4de27d4045 | | vpxord zmm11, zmm11, zmm11 | |
| 0x7f4de27d404b | | vpxord zmm12, zmm12, zmm12 | |
| 0x7f4de27d4051 | | vpxord zmm13, zmm13, zmm13 | |
| 0x7f4de27d4057 | | vpxord zmm14, zmm14, zmm14 | 0.001s |
| 0x7f4de27d405d | | vpxord zmm15, zmm15, zmm15 | |
| 0x7f4de27d4063 | | vpxord zmm16, zmm16, zmm16 | |
| 0x7f4de27d4069 | | vpxord zmm17, zmm17, zmm17 | |
| 0x7f4de27d406f | | vpxord zmm18, zmm18, zmm18 | 0.002s |
| 0x7f4de27d4075 | | vpxord zmm19, zmm19, zmm19 | 0.001s |
| 0x7f4de27d407b | | vpxord zmm20, zmm20, zmm20 | |
| 0x7f4de27d4081 | | vpxord zmm21, zmm21, zmm21 | |
| 0x7f4de27d4087 | | vpxord zmm22, zmm22, zmm22 | 0.006s |
| 0x7f4de27d408d | | vpxord zmm23, zmm23, zmm23 | |
| 0x7f4de27d4093 | | vpxord zmm24, zmm24, zmm24 | |
| 0x7f4de27d4099 | | vpxord zmm25, zmm25, zmm25 | |
| 0x7f4de27d409f | | vpxord zmm26, zmm26, zmm26 | 0.011s |
| 0x7f4de27d40a5 | | vpxord zmm27, zmm27, zmm27 | |
| 0x7f4de27d40ab | | vpxord zmm28, zmm28, zmm28 | |
| 0x7f4de27d40b1 | | vpxord zmm29, zmm29, zmm29 | 0s |
| 0x7f4de27d40b7 | | vpxord zmm30, zmm30, zmm30 | 0.005s |
| 0x7f4de27d40bd | | vpxord zmm31, zmm31, zmm31 | |
| 0x7f4de27d40c3 | | vmovupd zmm1, zmmword ptr [rsi+0xce400] | 0.001s |
| 0x7f4de27d40cd | | vfmadd231pd zmm2, zmm0, qword ptr [rdi]{1to8} | 0.883s  |
| 0x7f4de27d40d3 | | vfnmadd231pd zmm3, zmm0, qword ptr [rdi+0x38]{1to8} | 0s |
| 0x7f4de27d40da | | vfnmadd231pd zmm4, zmm0, qword ptr [rdi+0x38]{1to8} | 0s |
| 0x7f4de27d40e1 | | vfnmadd231pd zmm5, zmm0, qword ptr [rdi+0x38]{1to8} | |
| 0x7f4de27d40e8 | | vfnmadd231pd zmm6, zmm0, qword ptr [rdi+0x170]{1to8} | 0.004s |
| 0x7f4de27d40ef | | vfnmadd231pd zmm7, zmm0, qword ptr [rdi+0x1a8]{1to8} | 0.002s |
| 0x7f4de27d40f6 | | vfnmadd231pd zmm8, zmm0, qword ptr [rdi+0x1a8]{1to8} | 0.002s |

Figure 8.3: The top section of Intel VTune profiler results of our kernel with $m_B = 30$ running on the 6th order tetrahedral Shunn Ham m3 matrix.

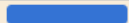



| Address ▲ | S... | Assembly | 🔥 CPU Time: Total |
|-----------------------|------|---|---|
| 0x7f65a0bfc000 | | Block 1: | |
| 0x7f65a0bfc000 | | push rbx | 0.001s |
| 0x7f65a0bfc001 | | push r12 | 0.003s |
| 0x7f65a0bfc003 | | push r13 | 0.002s |
| 0x7f65a0bfc005 | | push r14 | |
| 0x7f65a0bfc007 | | push r15 | 0.001s |
| 0x7f65a0bfc009 | | vmovupd zmm0, zmmword ptr [rsi] | 0.001s |
| 0x7f65a0bfc00f | | vpxord zmm24, zmm24, zmm24 | 1.021s  |
| 0x7f65a0bfc015 | | vpxord zmm25, zmm25, zmm25 | 0.001s |
| 0x7f65a0bfc01b | | vpxord zmm26, zmm26, zmm26 | |
| 0x7f65a0bfc021 | | vpxord zmm27, zmm27, zmm27 | |
| 0x7f65a0bfc027 | | vpxord zmm28, zmm28, zmm28 | 0.004s |
| 0x7f65a0bfc02d | | vpxord zmm29, zmm29, zmm29 | |
| 0x7f65a0bfc033 | | vpxord zmm30, zmm30, zmm30 | |
| 0x7f65a0bfc039 | | vpxord zmm31, zmm31, zmm31 | |
| 0x7f65a0bfc03f | | vmovupd zmm1, zmmword ptr [rsi+0xce400] | 0.001s |
| 0x7f65a0bfc049 | | vfmadd231pd zmm24, zmm0, qword ptr [rdi]{1to8} | 0.243s  |
| 0x7f65a0bfc04f | | vfnmadd231pd zmm25, zmm0, qword ptr [rdi+0x38]{1to8} | 0.004s |
| 0x7f65a0bfc056 | | vfnmadd231pd zmm26, zmm0, qword ptr [rdi+0x38]{1to8} | 0.002s |
| 0x7f65a0bfc05d | | vfnmadd231pd zmm27, zmm0, qword ptr [rdi+0x38]{1to8} | 0.004s |
| 0x7f65a0bfc064 | | vfnmadd231pd zmm28, zmm0, qword ptr [rdi+0x170]{1to8} | 0.007s |
| 0x7f65a0bfc06b | | vfnmadd231pd zmm29, zmm0, qword ptr [rdi+0x1a8]{1to8} | |
| 0x7f65a0bfc072 | | vfnmadd231pd zmm30, zmm0, qword ptr [rdi+0x1a8]{1to8} | 0s |
| 0x7f65a0bfc079 | | vfnmadd231pd zmm31, zmm0, qword ptr [rdi+0x1a8]{1to8} | |
| 0x7f65a0bfc080 | | vmovupd zmm0, zmmword ptr [rsi+0x19c800] | 0.002s |
| 0x7f65a0bfc08a | | vfnmadd231pd zmm24, zmm1, qword ptr [rdi+0x8]{1to8} | 0.114s  |
| 0x7f65a0bfc091 | | vfnmadd231pd zmm25, zmm1, qword ptr [rdi+0x48]{1to8} | 0.008s |
| 0x7f65a0bfc098 | | vfmadd231pd zmm26, zmm1, qword ptr [rdi+0x40]{1to8} | 0.001s |
| 0x7f65a0bfc09f | | vfnmadd231pd zmm27, zmm1, qword ptr [rdi+0x48]{1to8} | |
| 0x7f65a0bfc0a6 | | vfmadd231pd zmm28, zmm1, qword ptr [rdi+0x178]{1to8} | 0.003s |
| 0x7f65a0bfc0ad | | vfmadd231pd zmm29, zmm1, qword ptr [rdi+0x1b8]{1to8} | 0.001s |
| 0x7f65a0bfc0b4 | | vfnmadd231pd zmm30, zmm1, qword ptr [rdi+0x1b0]{1to8} | 0.003s |
| 0x7f65a0bfc0bb | | vfmadd231pd zmm31, zmm1, qword ptr [rdi+0x1b8]{1to8} | 0.002s |
| 0x7f65a0bfc0c2 | | vmovupd zmm1, zmmword ptr [rsi+0x26ac00] | 0.001s |
| 0x7f65a0bfc0cc | | vfnmadd231pd zmm24, zmm0, qword ptr [rdi+0x8]{1to8} | 0.074s  |
| 0x7f65a0bfc0d3 | | vfmadd231pd zmm25, zmm0, qword ptr [rdi+0x40]{1to8} | 0.008s |

Figure 8.4: The top section of Intel VTune profiler results of our kernel with $m_B = 8$ running on the 6th order tetrahedral Shunn Ham m3 matrix.

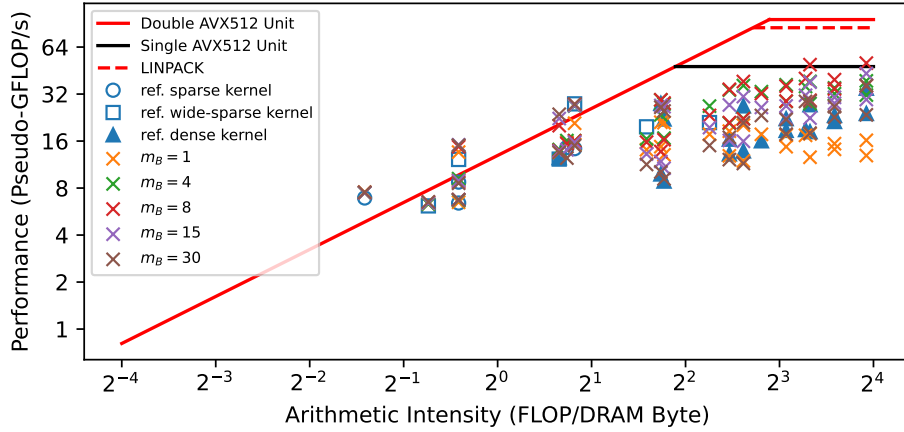


Figure 8.5: Roofline plots of our modified kernel which occupies all `zmm` registers using `xor` evaluated using the PyFR tetrahedral element matrices.

Hypothesis Test - OOO Engine Cannot Detect Loading Instructions In Advance

This hypothesis assumes that the CPU OOO unit can only detect instructions a certain number of instructions ahead of the current executed one. If the `vmovupd` instruction is too further ahead of the current one, because of the large chunk of FMAs, the OOO unit cannot issue a memory load in advance to exploit memory-level parallelism.

For OOO CPUs, the window size that the OOO unit can detect is determined by the size of the reservation station. For x86 processors such as the Skylake-SP one we were using, the reservation station acts like a buffer between the decoder and execution unit. It buffers the decoded micro-operations which are ready for OOO execution. As listed in [19], Intel Skylake microarchitecture features a 97-entry reservation station per core. As the FMA instructions count as simple operations [17], each FMA instruction is decoded into one micro-operation. Therefore, the 97-entry reservation station is large enough for accommodating multiple FMA instruction chunks so our hypothesis is not correct.

One other possible reason why big m_B introduces poor performance is that for smaller m_B , there are more micro-operations for loading \mathbf{B} strides present in the reservation buffer. This means the CPU could issues multiple memory read signals for loading multiple \mathbf{B} strides to exploit memory-level parallelism. However, this does not explain why the first \mathbf{B} loading instruction stalls the subsequent `xor` instruction.

8.3 Sanity Check - Are There Any More Factors?

In Section 8.1 and 8.2 we discussed two reasons why kernels with large m_B are slower than the ones with smaller m_B . However, we have not ruled out any other potential causes. For this purpose, we prototyped another broken kernel which does not issue any non-temporal stores for writing C strides or any moving instructions for loading B strides. Figure 8.6 shows the benchmark result of our modified kernel evaluated using the PyFR tetrahedral element matrices. As shown in the figure, $m_B = 8, 16, 30$ now have similar performance because they all can fully utilise the FMA pipeline. This indicates that there is no more factor contributing to slower kernel performance for large m_B .

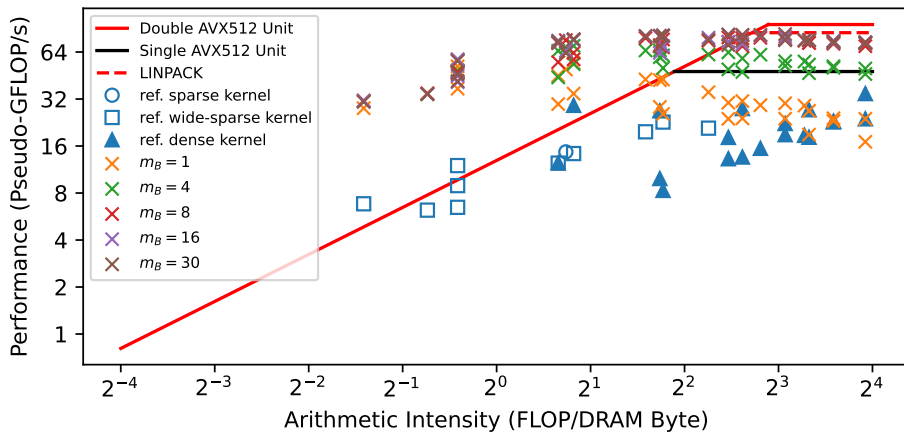


Figure 8.6: Roofline plots of our “broken” kernel with no non-temporal and `vmovupd`, evaluated using the PyFR tetrahedral element matrices.

Summary

In this chapter, we tried to answer one question: why are big M blocking factors bad for performance? With the help of Intel VTune profiler, we identified two instructions causing this problem - the non-temporal store instruction `vmovntpd` and the loading B instruction `vmovupd`. The former instruction hits the performance because it can easily saturate the write buffer for large m_B as it writes directly to the main memory. However, the reason why `vmovupd` hurts the performance is not clear. We suggested two hypotheses. The first one is related to the number of free physical registers available and the second one is about how wide the OOO instruction window is. However, we ruled out both hypotheses by either experimental or theoretical analysis. In the end, we completed a sanity check experimentally, confirmed there is no other cause to the problem apart from the two instructions we have discussed.

Chapter 9

Conclusion and Future Work

In this final chapter we will summarise and evaluate what our project has achieved. We will conclude our thesis with suggestions for directions of future work which can extend our research further.

9.1 Summary

We believe we have successfully accomplished the project objectives. The objectives of our project were to improve the small and sparse GEMM implementation of the Intel open-source library LIBXSMM, by 1. allowing a greater number of distinct absolute non-zero values present in the sparse matrix and 2. enhancing the kernel execution speed. The targeting platform was x86 CPUs with AVX-512 extension.

In Chapter 2, we introduced the context of our project - GEMM, LIBXSMM and PyFR. In Chapter 3 we reviewed two other matrix operation libraries GiMMiK and BLASFEO. We also presented that measurement bias is significant and unavoidable. In Chapter 4 we presented our comprehensive benchmark suite for sparse GEMM kernels. The benchmark suite was based on two matrices set - a set of complete 170 PyFR matrices and a set of 100 synthetic matrices with gradual changes of properties. In Chapter 5 we presented our first improved GEMM implementation which allows unlimited number of constants present in \mathbf{A} by runtime broadcasting packed \mathbf{A} constants from the main memory. Although this implementation is not faster than LIBXSMM's routines, it significantly increases the applicability of the sparse kernel for AVX2 architectures which originally only allows 15 constants present in \mathbf{A} . In Chapter 6 we explored two techniques allowing us to accumulate multiple \mathbf{C} strides - N blocking and M blocking. Both these techniques can improve the performance of our kernel. In Chapter 7, we presented our improved kernel which caches \mathbf{B} strides and runtime broadcasted packed \mathbf{A} elements from the main memory using AVX-512 FMA instructions. With the optimum N and M blocking setting, our kernel shows

significant superior performance than LIBXSMM’s methods. In Chapter 8, we tried to answer the question: why a too large M blocking factor hurts the performance? We successfully identified two memory access instructions causing this problem through micro-architectural analysis.

This thesis made the following contributions:

1. We developed an automatic benchmark suite for small and sparse GEMM routines based on the work from the previous year student Paribartan [8]. The test suite is based on two sets of matrices - a complete set of all the 170 PyFR operator matrices and a set of 100 synthetic matrices aiming to provide a broader coverage and a finer resolution of the matrices characteristics space.
2. We improved LIBXSMM’s small and sparse GEMM routine with the technique to runtime broadcast \mathbf{A} constants from memory. Comparing to the reference LIBXSMM routine, which allows maximally 240 double-precision or 480 single-precision constants present in \mathbf{A} for AVX-512, our improved routine allows an unlimited number of constants in the sparse matrix for both AVX-512 and AVX2.
3. We explored techniques to accumulate multiple \mathbf{C} strides in the free registers to exploit Instruction-Level Parallelism. We evaluated techniques - N blocking and M blocking, and determined the optimum blocking factors experimentally. We reported N blocking can provide a maximum speedup of 30% for \mathbf{A} s with densities larger than 0.1. M blocking can provide a maximum speedup of 60% for \mathbf{A} s with densities larger than 0.4.
4. We further improved our routine with techniques to cache and reuse \mathbf{B} strides in the free vector registers, and runtime broadcast \mathbf{A} elements using FMA instructions. This implementation is only supported by AVX-512. With the optimum N blocking and M blocking setting, comparing with reference LIBXSMM, our implementation shows a significant performance superiority up to 78% when executing on the dense PyFR operator matrices.
5. We observed that the kernels with large M blocking factors show unexpectedly low performance. We successfully identified the problem was caused by two instructions for memory writes and reads - `vmovntpd` and `vmovupd`. We conducted a systematic micro-architectural analysis and suggested possible underlining causes for this.

9.2 Future Work

We have successfully accomplished the project objectives. However we were unable to explore several promising ideas which can extend our thesis partially because of the time limitation. These ideas will be briefly discussed in this section.

Contribution to LIBXSMM Repository

It is unfortunate we were unable to get our prototyped improved kernel integrated with the upstream LIBXSMM repository. We suggest this should be one of the priorities for future work so scientific solvers such as PyFR can benefit from our achievements. Additionally, this will expose our method in an open-source format, allowing us to receive real-world practical significant feedback. We proposed the integration process can be realised in the following steps:

1. Implementation for AVX-512 In Chapter 7 we presented our small and sparse GEMM implementation which allows unlimited number of constants present in \mathbf{A} . This method caches some \mathbf{B} strides in the vector registers and achieves the runtime broadcasting using AVX-512 FMA instructions. Our evaluation shows our method is up to 78% faster than LIBXSMM's dense routine when executing the PyFR tetrahedral and triangular element operator matrices. Figure 9.1 presents our proposed changes to the LIBXSMM routine.

2. Implementation for AVX2 In Chapter 5 we discussed that platforms which only supports AVX2 can greatly benefit from our first improved GEMM routine. This routine runtime broadcasts packed \mathbf{A} constants from memory using VBROADCASTSS/D thus allows unlimited number of constants present in \mathbf{A} . Although the performance of our method on an AVX2 platform was not evaluated, we believed that being able to extend the applicability to \mathbf{A} containing more constants is extremely beneficial regardless of the performance change.

3. Improve the reference LIBXSMM routines with \mathbf{B} caching technique In Chapter 7 we present how we improved our first and second implementations with caching \mathbf{B} strides in vector registers for later reuses. Paring with M blocking, this technique provides significant performance increase, especially for dense matrices. In fact, this caching \mathbf{B} technique can also be applied to LIBXSMM's original and Paribartan's register packing kernels presented in Section 2.3. Figure 9.2 how the original LIBXSMM's small and sparse GEMM implementation can be integrating with our caching \mathbf{B} strides technique. Paribartan's register packing kernel can be improved with a similar method.

Improvement to the LIBXSMM Dense GEMM Routine

In Chapter 5 we pointed out that even for complete dense matrices, the reference LIBXSMM dense kernel can only achieve maximally 37% of the performance potential of our test machine. As the dense routine has the widest applicability among all the routines we have evaluated during our thesis, any further performance improvement

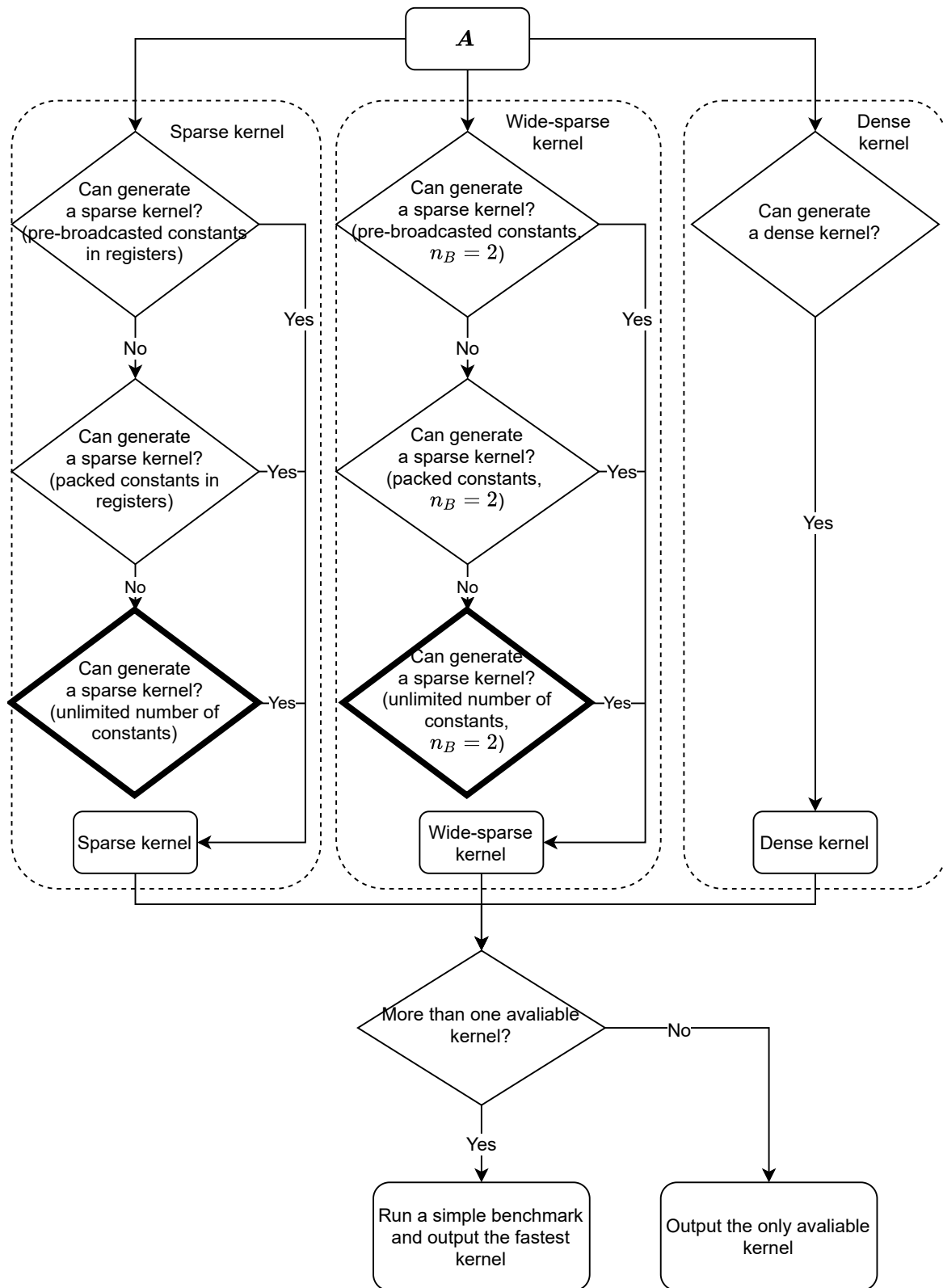


Figure 9.1: LIBXSMM small and sparse GEMM implementation integrated with our method. The bold diamond shapes indicate our proposed changes.

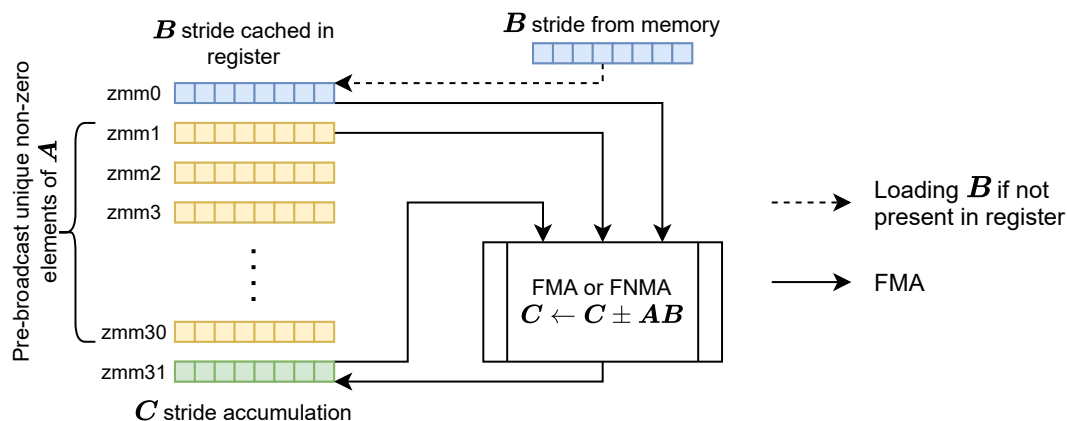


Figure 9.2: Improved original LIBXSMM’s small and sparse GEMM routine with caching B stride in register.

would be extremely beneficial. In Chapter 7 we observed that, although our best implementation can significantly outperform the dense routine with the optimal N and M blocking factors ($n_B = 2$, $m_B = 4$), if we set $m_B = 30$ to utilise all the vector registers available for multiple accumulations, we got far worse performance which is very similar to the performance of the dense routine. Coincidentally, the dense routine also tries to accumulate maximum number of C strides. Based on these observations, we suggested the performance of the dense kernel could be improved by decreasing the M blocking factor. It would be beneficial if this idea can be evaluated in future.

CPU Front-End Bottleneck Due to Large Instruction Size

During our evaluation process, we noticed that our best performant kernel, which was presented in Chapter 7, uses FMA instructions with variable code sizes. Figure 9.3 shows a section of our kernel generated for the 6th order Shun-Ham tetrahedral m_3 matrix with $n_B = 1$ and $m_B = 30$. As shown in the figure, some of our FMA instructions are 8-byte long while the others are 10-byte. As the Skylake-SP instruction fetch unit has a maximum throughput of 16 bytes per cycle [19], the 10-byte FMA instructions could easily saturate the fetch unit leading to CPU front-end bottleneck (front-end refers to the CPU components responsible for fetching and decoding). This idea was confirmed by Intel VTune profiler, which reported 10% front-end bound for some kernels. We noticed that the instruction length is determined by the memory displacement and the 10-byte encoding scheme is only used for displacements larger or equal to $0x400$, which is 1024 in decimal. We suggest this problem can be overcome by including an index register for addressing the memory location. Each time the displacement value reaches $0x400$, the index register is incremented by $0x400$ so we will always have the 8-byte encoding. As the LIBXSMM’s dense kernel also shows front-end bottleneck due to this instruction length problem, it is very beneficial to explore this idea in the future.

K Blocking

In Chapter 5 we observed that there are some kernels with performance very close to or even higher than the memory bandwidth roofline. The reason for this is that these kernels belong to problems with small matrices sizes, so parts of matrix \mathbf{B} are likely to remain inside the cache during the benchmark iterations. Throughout the entire project, we observed a general trend that smaller matrices tend to show higher performance. We suggest this is because smaller matrices have higher spacial locality. One technique used by many BLAS libraries including [2] is K blocking, which is the tiling in the k direction. Figure 9.4 shows how K blocking can be achieved for GEMM by dividing the entire MM into the summation of multiple independent and smaller MM problems. Because the sub-problems have smaller k dimensions, they have high temporal locality for accessing the matrices elements thus could result in better performance. One disadvantages of K blocking is that it introduces more intermediate \mathbf{C} strides which are stored in either the cache or the main memory. Because K blocking kernels have to access these intermediate values, they are more prone to memory bandwidth bottleneck. A good K blocking design should find the right balance between memory bandwidth bottleneck and the temporal locality.

For this thesis, we have done some preliminary evaluations of our kernels with K blocking. Our evaluation is rather incomplete, but it shows K blocking tends to hurt performance for sparse GEMM as the performance is limited by memory bandwidth. Because of the time constraint, we were unable to test K blocking together with M and N blocking. Because N blocking introduces more stress on the cache system as we discussed in Chapter 6, K blocking could be beneficial if pairing with it. We suggest this idea should be revisited in a more systematic manner for future development of LIBXSMM's small and sparse GEMM routines.

Multi-Thread Evaluation

One of the major limitations of our benchmark suite is it only evaluates the kernel performance running in a single-threaded setting. However, for realistic scientific computing tasks, it is almost certain that GEMM kernels are run as multi-threaded processes to exploit the massive multiprocessing capability of modern high-performance computers. As discussed in Chapter 4, multi-threaded environment establishes completely different design challenges because of sharing key hardware resources. It is worth exploring in this direction so we could further optimise our GEMM routines thus make them better suitable for real-life scientific tasks.

This concludes our thesis.

Excluding figures and appendix, this thesis consists of 70 pages.

```

c3: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
ca: 62 f2 fd 58 b8 17 vfmadd231pd zmm2,zmm0,QWORD PTR [rdi]{1to8}
d0: 62 f2 fd 58 bc 5f 07 vfnmadd231pd zmm3,zmm0,QWORD PTR [rdi+0x38]{1to8}
d7: 62 f2 fd 58 bc 67 07 vfnmadd231pd zmm4,zmm0,QWORD PTR [rdi+0x38]{1to8}
de: 62 f2 fd 58 bc 6f 07 vfnmadd231pd zmm5,zmm0,QWORD PTR [rdi+0x38]{1to8}
e5: 62 f2 fd 58 bc 77 2e vfnmadd231pd zmm6,zmm0,QWORD PTR [rdi+0x170]{1to8}
ec: 62 f2 fd 58 bc 7f 35 vfnmadd231pd zmm7,zmm0,QWORD PTR [rdi+0x1a8]{1to8}
f3: 62 72 fd 58 bc 47 35 vfnmadd231pd zmm8,zmm0,QWORD PTR [rdi+0x1a8]{1to8}
fa: 62 72 fd 58 bc 4f 35 vfnmadd231pd zmm9,zmm0,QWORD PTR [rdi+0x1a8]{1to8}
101: 62 72 fd 58 b8 57 5c vfmadd231pd zmm10,zmm0,QWORD PTR [rdi+0x2e0]{1to8}
108: 62 72 fd 58 b8 5f 63 vfmadd231pd zmm11,zmm0,QWORD PTR [rdi+0x318]{1to8}
10f: 62 72 fd 58 b8 67 63 vfmadd231pd zmm12,zmm0,QWORD PTR [rdi+0x318]{1to8}
116: 62 72 fd 58 b8 6f 63 vfmadd231pd zmm13,zmm0,QWORD PTR [rdi+0x318]{1to8}
11d: 62 72 fd 58 bc b7 50 vfnmadd231pd zmm14,zmm0,QWORD PTR [rdi+0x450]{1to8}
124: 04 00 00
127: 62 72 fd 58 bc bf 50 vfnmadd231pd zmm15,zmm0,QWORD PTR [rdi+0x450]{1to8}
12e: 04 00 00
131: 62 e2 fd 58 b8 87 d0 vfmadd231pd zmm16,zmm0,QWORD PTR [rdi+0x4d0]{1to8}
138: 04 00 00
13b: 62 e2 fd 58 b8 8f d0 vfmadd231pd zmm17,zmm0,QWORD PTR [rdi+0x4d0]{1to8}
142: 04 00 00
145: 62 e2 fd 58 bc 97 50 vfnmadd231pd zmm18,zmm0,QWORD PTR [rdi+0x450]{1to8}
14c: 04 00 00
14f: 62 e2 fd 58 b8 9f d0 vfmadd231pd zmm19,zmm0,QWORD PTR [rdi+0x4d0]{1to8}
156: 04 00 00
159: 62 e2 fd 58 bc a7 50 vfnmadd231pd zmm20,zmm0,QWORD PTR [rdi+0x650]{1to8}
160: 06 00 00
163: 62 e2 fd 58 bc af 50 vfnmadd231pd zmm21,zmm0,QWORD PTR [rdi+0x650]{1to8}
16a: 06 00 00
16d: 62 e2 fd 58 b8 b7 d0 vfmadd231pd zmm22,zmm0,QWORD PTR [rdi+0x6d0]{1to8}
174: 06 00 00
177: 62 e2 fd 58 b8 bf d0 vfmadd231pd zmm23,zmm0,QWORD PTR [rdi+0x6d0]{1to8}
17e: 06 00 00
181: 62 62 fd 58 bc 87 50 vfnmadd231pd zmm24,zmm0,QWORD PTR [rdi+0x650]{1to8}
188: 06 00 00
18b: 62 62 fd 58 b8 8f d0 vfmadd231pd zmm25,zmm0,QWORD PTR [rdi+0x6d0]{1to8}
192: 06 00 00
195: 62 62 fd 58 b8 97 50 vfmadd231pd zmm26,zmm0,QWORD PTR [rdi+0x850]{1to8}
19c: 08 00 00
19f: 62 62 fd 58 b8 9f d0 vfmadd231pd zmm27,zmm0,QWORD PTR [rdi+0x8d0]{1to8}
1a6: 08 00 00
1a9: 62 62 fd 58 bc a7 b0 vfnmadd231pd zmm28,zmm0,QWORD PTR [rdi+0x9b0]{1to8}
1b0: 09 00 00
1b3: 62 62 fd 58 b8 af d0 vfmadd231pd zmm29,zmm0,QWORD PTR [rdi+0x8d0]{1to8}
1ba: 08 00 00
1bd: 62 62 fd 58 bc b7 b0 vfnmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x9b0]{1to8}
1c4: 09 00 00
1c7: 62 62 fd 58 b8 bf d0 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x8d0]{1to8}
1ce: 08 00 00

```

Figure 9.3: Section of our kernel example disassembled using `objdump`. Each line is in the format: [Instruction offset]: [Raw binary instruction] [Disassembled instruction].

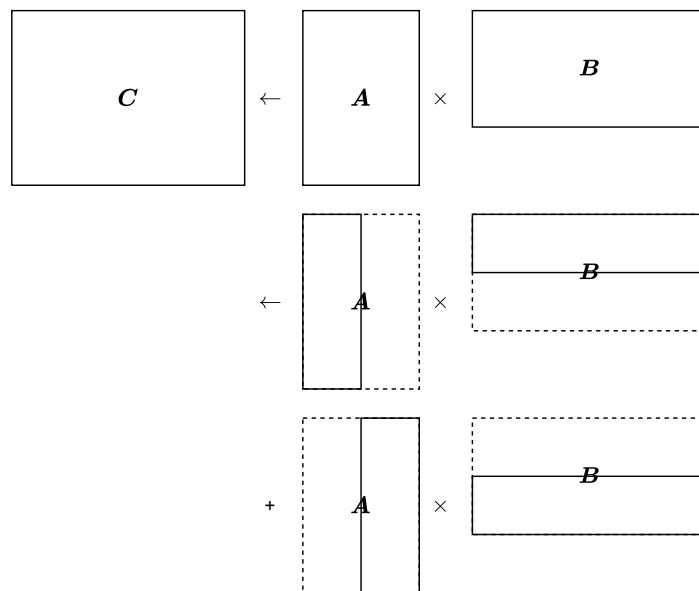


Figure 9.4: K blocking technique.

Bibliography

- [1] Heinecke A, Henry G, Hutchinson M, Pabst H. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In: SC16 Int. Conf. High Perform. Comput. Networking, Storage Anal. November. IEEE; 2016. p. 981–991.
- [2] Goto K, van de Geijn RA. Anatomy of high-performance matrix multiplication. ACM Trans Math Softw. 2008 may;34(3):1–25. Available from: <https://dl.acm.org/doi/10.1145/1356052.1356053>.
- [3] Witherden FD, Farrington AM, Vincent PE. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. Comput Phys Commun. 2014 nov;185(11):3028–3040. Available from: <https://linkinghub.elsevier.com/retrieve/pii/S0010465514002549>.
- [4] Witherden F, Vincent P, Vermeire B, Trojak W, Park JS, Ntemos G, et al.. PyFR/PyFR: PyFR; 2021. [Accessed 09/08/2021]. Available from: <https://github.com/PyFR/PyFR>.
- [5] Huynh HT. A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods. In: 18th AIAA Comput. Fluid Dyn. Conf. vol. 1. Reston, Virginia: American Institute of Aeronautics and Astronautics; 2007. p. 698–739. Available from: <https://arc.aiaa.org/doi/10.2514/6.2007-4079>.
- [6] LIBXSMM Contributors. hfp/libxsmm: Library for specialized dense and sparse matrix operations, and deep learning primitives.; 2021. [Accessed 09/08/2021]. Available from: <https://github.com/hfp/libxsmm>.
- [7] Wozniak BD, Witherden FD, Russell FP, Vincent PE, Kelly PHJ. GiMMiK - Generating bespoke matrix multiplication kernels for accelerators: Application to high-order Computational Fluid Dynamics. Comput Phys Commun. 2016 may;202:12–22.
- [8] Paribartan MV. Using Register Packing for Small Sparse Matrix Multiplication [MEng Thesis]. Imperial College London; 2020.
- [9] LIBXSMM Contributors. LIBXSMM; 2021. [Accessed 25/15/2021]. Available from: <https://libxsmm.readthedocs.io/en/latest/>.

- [10] LIBXSMM Contributors. Attempt to JIT double width kernels in dfsspm. by FreddieWetherden · Pull Request #487 · hfp/libxsmm; 2021. [Accessed 29/08/2021]. Available from: <https://github.com/hfp/libxsmm/pull/487>.
- [11] GiMMiK Contributors. PyFR/GiMMiK; 2021. [Accessed 31/08/2021]. Available from: <https://github.com/PyFR/GiMMiK>.
- [12] Frison G, Kouzoupis D, Sartor T, Zanelli A, Diehl M. BLASFEO: basic linear algebra subroutines for embedded optimization. *ACM Trans Math Softw.* 2017 apr;44(4):1–30. Available from: <https://dl.acm.org/doi/10.1145/3210754>.
- [13] Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Not.* 2009 feb;44(3):265–276. Available from: <https://dl.acm.org/doi/10.1145/1508284.1508275>.
- [14] Intel Corporation. Intel Math Kernel Library (Intel MKL) Benchmarks Suite; 2021. [Accessed 08/08/2021]. Available from: <https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-benchmarks-suite.html>.
- [15] McCalpin J. STREAM: Sustainable Memory Bandwidth in High Performance Computers;. [Accessed 22/08/2021]. Available from: <https://www.cs.virginia.edu/stream/>.
- [16] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer’s Manual; 2021. Available from: <https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html>.
- [17] Fog A. 4. Instruction tables; 2021. [Accessed 13/08/2021]. Available from: https://www.agner.org/optimize/instruction_tables.pdf.
- [18] Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach*. Elsevier; 2011.
- [19] Fog A. The microarchitecture of Intel, AMD, and VIA CPUs; 2021. [Accessed 28/08/2021].

Appendix A

Characteristics of PyFR operator matrices

This appendix lists the major characteristics of all 170 PyFR operator matrices. Following notations are used:

- R** Number of rows;
- C** Number of columns;
- ρ Density, calculated as $\frac{\text{num. of non-zeros}}{\text{size}}$;
- U** Number of unique absolute non-zero values.

The PyFR operator matrices are available from *samples/pyfr/mats/* in LIBXSMM repository [6].

| Order | Matrix | R | C | ρ | U |
|--------|--------|----|----|--------|----|
| First | m0 | 8 | 4 | 0.500 | 2 |
| | m3 | 4 | 8 | 0.500 | 2 |
| | m6 | 8 | 8 | 0.250 | 2 |
| | m132 | 4 | 8 | 0.500 | 1 |
| | m460 | 8 | 4 | 0.500 | 1 |
| Second | m0 | 12 | 9 | 0.333 | 3 |
| | m3 | 9 | 12 | 0.333 | 3 |
| | m6 | 18 | 12 | 0.167 | 3 |
| | m132 | 9 | 18 | 0.296 | 4 |
| | m460 | 18 | 9 | 0.296 | 4 |
| Third | m0 | 16 | 16 | 0.250 | 4 |
| | m3 | 16 | 16 | 0.250 | 4 |
| | m6 | 32 | 16 | 0.125 | 4 |
| | m132 | 16 | 32 | 0.250 | 8 |
| | m460 | 32 | 16 | 0.250 | 8 |
| Fourth | m0 | 20 | 25 | 0.200 | 5 |
| | m3 | 25 | 20 | 0.200 | 5 |
| | m6 | 50 | 20 | 0.100 | 5 |
| | m132 | 25 | 50 | 0.192 | 12 |
| | m460 | 50 | 25 | 0.192 | 12 |
| Fifth | m0 | 24 | 36 | 0.167 | 6 |
| | m3 | 36 | 24 | 0.167 | 6 |
| | m6 | 72 | 24 | 0.083 | 6 |
| | m132 | 36 | 72 | 0.167 | 18 |
| | m460 | 72 | 36 | 0.167 | 18 |
| Sixth | m0 | 28 | 49 | 0.143 | 7 |
| | m3 | 49 | 28 | 0.143 | 7 |
| | m6 | 98 | 28 | 0.071 | 7 |
| | m132 | 49 | 98 | 0.140 | 24 |
| | m460 | 98 | 49 | 0.140 | 24 |

(a) Using Gauss-Legendre quadrature.

| Order | Matrix | R | C | ρ | U |
|--------|--------|----|----|--------|----|
| Second | m0 | 12 | 9 | 0.111 | 1 |
| | m3 | 9 | 12 | 0.333 | 3 |
| | m6 | 18 | 12 | 0.167 | 3 |
| | m132 | 9 | 18 | 0.296 | 3 |
| | m460 | 18 | 9 | 0.296 | 3 |
| Third | m0 | 16 | 16 | 0.063 | 1 |
| | m3 | 16 | 16 | 0.250 | 3 |
| | m6 | 32 | 16 | 0.125 | 3 |
| | m132 | 16 | 32 | 0.219 | 7 |
| | m460 | 32 | 16 | 0.219 | 7 |
| Fourth | m0 | 20 | 25 | 0.040 | 1 |
| | m3 | 25 | 20 | 0.200 | 4 |
| | m6 | 50 | 20 | 0.100 | 4 |
| | m132 | 25 | 50 | 0.176 | 11 |
| | m460 | 50 | 25 | 0.176 | 11 |
| Fifth | m0 | 24 | 36 | 0.028 | 1 |
| | m3 | 36 | 24 | 0.167 | 4 |
| | m6 | 72 | 24 | 0.083 | 4 |
| | m132 | 36 | 72 | 0.148 | 16 |
| | m460 | 72 | 36 | 0.148 | 16 |
| Sixth | m0 | 28 | 49 | 0.020 | 1 |
| | m3 | 49 | 28 | 0.143 | 5 |
| | m6 | 98 | 28 | 0.071 | 5 |
| | m132 | 49 | 98 | 0.128 | 22 |
| | m460 | 98 | 49 | 0.128 | 22 |

(b) Using Gauss-Legendre-Lobatto quadrature.

Table A.1: Characteristics of quadrilateral operator matrices.

| Order | Matrix | R | C | ρ | U |
|--------|--------|------|------|--------|----|
| First | m0 | 24 | 8 | 0.250 | 2 |
| | m3 | 8 | 24 | 0.250 | 2 |
| | m6 | 24 | 24 | 0.833 | 2 |
| | m132 | 8 | 24 | 0.250 | 1 |
| | m460 | 24 | 8 | 0.250 | 1 |
| Second | m0 | 54 | 27 | 0.111 | 3 |
| | m3 | 27 | 54 | 0.111 | 3 |
| | m6 | 81 | 54 | 0.037 | 3 |
| | m132 | 27 | 81 | 0.099 | 4 |
| | m460 | 81 | 27 | 0.099 | 4 |
| Third | m0 | 96 | 64 | 0.063 | 4 |
| | m3 | 64 | 96 | 0.063 | 4 |
| | m6 | 192 | 96 | 0.021 | 4 |
| | m132 | 64 | 192 | 0.063 | 8 |
| | m460 | 192 | 64 | 0.063 | 8 |
| Fourth | m0 | 150 | 125 | 0.040 | 5 |
| | m3 | 125 | 150 | 0.040 | 5 |
| | m6 | 375 | 150 | 0.013 | 5 |
| | m132 | 125 | 375 | 0.038 | 12 |
| | m460 | 375 | 125 | 0.038 | 12 |
| Fifth | m0 | 216 | 216 | 0.028 | 6 |
| | m3 | 216 | 216 | 0.028 | 6 |
| | m6 | 648 | 216 | 0.009 | 6 |
| | m132 | 216 | 648 | 0.028 | 18 |
| | m460 | 648 | 216 | 0.028 | 18 |
| Sixth | m0 | 294 | 343 | 0.020 | 7 |
| | m3 | 343 | 294 | 0.020 | 7 |
| | m6 | 1029 | 294 | 0.007 | 7 |
| | m132 | 343 | 1029 | 0.020 | 24 |
| | m460 | 1029 | 343 | 0.020 | 24 |

(a) Using Gauss-Legendre quadrature.

| Order | Matrix | R | C | ρ | U |
|--------|--------|------|------|--------|----|
| Second | m0 | 54 | 27 | 0.037 | 1 |
| | m3 | 27 | 54 | 0.111 | 3 |
| | m6 | 81 | 54 | 0.037 | 3 |
| | m132 | 27 | 81 | 0.099 | 3 |
| | m460 | 81 | 27 | 0.099 | 3 |
| Third | m0 | 96 | 64 | 0.016 | 1 |
| | m3 | 64 | 96 | 0.063 | 3 |
| | m6 | 192 | 96 | 0.021 | 3 |
| | m132 | 64 | 192 | 0.055 | 7 |
| | m460 | 192 | 64 | 0.055 | 7 |
| Fourth | m0 | 150 | 125 | 0.008 | 1 |
| | m3 | 125 | 150 | 0.040 | 4 |
| | m6 | 375 | 150 | 0.013 | 4 |
| | m132 | 125 | 375 | 0.035 | 11 |
| | m460 | 375 | 125 | 0.035 | 11 |
| Fifth | m0 | 216 | 216 | 0.005 | 1 |
| | m3 | 216 | 216 | 0.028 | 4 |
| | m6 | 648 | 216 | 0.009 | 4 |
| | m132 | 216 | 648 | 0.025 | 16 |
| | m460 | 648 | 216 | 0.025 | 16 |
| Sixth | m0 | 294 | 343 | 0.003 | 1 |
| | m3 | 343 | 294 | 0.020 | 5 |
| | m6 | 1029 | 294 | 0.007 | 5 |
| | m132 | 343 | 1029 | 0.018 | 22 |
| | m460 | 1029 | 343 | 0.018 | 22 |

(b) Using Gauss-Legendre-Lobatto quadrature.

Table A.2: Characteristics of hexahedral operator matrices.

| Order | Matrix | R | C | ρ | U |
|--------|--------|-----|-----|--------|------|
| First | m0 | 12 | 4 | 1.00 | 3 |
| | m3 | 4 | 12 | 1.00 | 6 |
| | m6 | 12 | 12 | 0.50 | 3 |
| | m132 | 4 | 12 | 0.50 | 1 |
| | m460 | 12 | 4 | 0.50 | 1 |
| Second | m0 | 24 | 10 | 1.00 | 14 |
| | m3 | 10 | 24 | 1.00 | 28 |
| | m6 | 30 | 24 | 0.50 | 14 |
| | m132 | 10 | 30 | 0.84 | 23 |
| | m460 | 30 | 10 | 0.84 | 23 |
| Third | m0 | 40 | 20 | 1.00 | 40 |
| | m3 | 20 | 40 | 1.00 | 80 |
| | m6 | 60 | 40 | 0.50 | 40 |
| | m132 | 20 | 60 | 0.91 | 100 |
| | m460 | 60 | 20 | 0.91 | 100 |
| Fourth | m0 | 60 | 35 | 1.00 | 101 |
| | m3 | 35 | 60 | 1.00 | 202 |
| | m6 | 105 | 60 | 0.50 | 101 |
| | m132 | 35 | 105 | 0.93 | 304 |
| | m460 | 105 | 35 | 0.93 | 304 |
| Fifth | m0 | 84 | 56 | 1.00 | 214 |
| | m3 | 56 | 84 | 1.00 | 428 |
| | m6 | 168 | 84 | 0.50 | 214 |
| | m132 | 56 | 168 | 0.95 | 784 |
| | m460 | 168 | 56 | 0.95 | 784 |
| Sixth | m0 | 112 | 84 | 1.00 | 425 |
| | m3 | 84 | 112 | 1.00 | 850 |
| | m6 | 252 | 112 | 0.50 | 425 |
| | m132 | 84 | 252 | 0.96 | 1760 |
| | m460 | 252 | 84 | 0.96 | 1760 |

Table A.3: Characteristics of tetrahedral Shunn-Ham operator matrices.

| Order | Matrix | R | C | ρ | U |
|--------|--------|----|----|--------|-----|
| First | m0 | 6 | 3 | 1.00 | 3 |
| | m3 | 3 | 6 | 1.00 | 6 |
| | m6 | 6 | 6 | 0.67 | 3 |
| | m132 | 3 | 6 | 0.67 | 1 |
| | m460 | 6 | 3 | 0.67 | 1 |
| Second | m0 | 9 | 6 | 1.00 | 10 |
| | m3 | 6 | 9 | 1.00 | 20 |
| | m6 | 12 | 9 | 0.67 | 10 |
| | m132 | 6 | 12 | 0.89 | 14 |
| | m460 | 12 | 6 | 0.89 | 14 |
| Third | m0 | 12 | 10 | 1.00 | 20 |
| | m3 | 10 | 12 | 1.00 | 40 |
| | m6 | 20 | 12 | 0.67 | 20 |
| | m132 | 10 | 20 | 0.96 | 48 |
| | m460 | 20 | 10 | 0.96 | 48 |
| Fourth | m0 | 15 | 15 | 1.00 | 39 |
| | m3 | 15 | 15 | 1.00 | 78 |
| | m6 | 30 | 15 | 0.67 | 39 |
| | m132 | 15 | 30 | 0.96 | 108 |
| | m460 | 30 | 15 | 0.96 | 108 |
| Fifth | m0 | 18 | 21 | 1.00 | 63 |
| | m3 | 21 | 18 | 1.00 | 126 |
| | m6 | 42 | 18 | 0.67 | 63 |
| | m132 | 21 | 42 | 0.98 | 216 |
| | m460 | 42 | 21 | 0.98 | 216 |
| Sixth | m0 | 21 | 28 | 1.00 | 100 |
| | m3 | 28 | 21 | 1.00 | 200 |
| | m6 | 56 | 21 | 0.67 | 100 |
| | m132 | 28 | 56 | 0.98 | 384 |
| | m460 | 56 | 28 | 0.98 | 384 |

Table A.4: Characteristics of triangular Williams-Shunn operator matrices.

Appendix B

Kernel Examples

In this appendix we provide example kernels for the matrix multiplication routines discussed in this report. All the kernels apply to a simple GEMM operation:

$$\mathbf{C} \leftarrow \alpha \mathbf{A} \times \mathbf{B} + \beta \mathbf{C},$$

with $\alpha = 1$ and $\beta = 0$.

For the matrices:

$$\mathbf{A} = \begin{bmatrix} 1.1 & 2.2 & 3.3 & 4.4 \\ 0 & -1.1 & -2.2 & -3.3 \\ 0 & 0 & 1.1 & 2.2 \\ 0 & 0 & 0 & -1.1 \\ 1.1 & 2.2 & 3.3 & 4.4 \\ 0 & -1.1 & -2.2 & -3.3 \\ 0 & 0 & 1.1 & 2.2 \\ 0 & 0 & 0 & -1.1 \end{bmatrix},$$

and \mathbf{B} is 8×48 in shape.

As LIBXSMM does not directly generate kernel as textual assemblies, we set the environment with `LIBXSMM_VERBOSE=-1` to enable binary kernel dump. The binary kernels were disassembled by `objdump` with `-D -b binary -m i386 -M x86-64,intel` flags.

B.1 Reference LIBXSMM kernels

For the reference sparse and wide-sparse kernels, the \mathbf{A} constants are originally stored in the instruction stream. These constants are loaded into the vector registers at the beginning of kernel execution by `VMOVUPS` instructions relative to the RIP register. This is why there are uninterpretable instructions in the beginning of these two kernels as `objdump` sees these data as instructions.

Sparse Kernel

```
libxsmm_skx_f64_nn_8x8x4_0_48_48_a1_b0_p0.sreg:      file format binary
```

```
Disassembly of section .data:
```

```
00000000 <.data>:
 0: 53                push   rbx
 1: 41 54            push   r12
 3: 41 55            push   r13
 5: 41 56            push   r14
 7: 41 57            push   r15
 9: eb 40           jmp    0x4b
 b: 9a              (bad)
 c: 99             cdq
 d: 99             cdq
 e: 99             cdq
 f: 99             cdq
10: 99             cdq
11: f1             icebp
12: 3f             (bad)
13: 9a             (bad)
14: 99             cdq
15: 99             cdq
16: 99             cdq
17: 99             cdq
18: 99             cdq
19: f1             icebp
1a: 3f             (bad)
1b: 9a             (bad)
1c: 99             cdq
1d: 99             cdq
1e: 99             cdq
1f: 99             cdq
20: 99             cdq
21: f1             icebp
22: 3f             (bad)
23: 9a             (bad)
24: 99             cdq
25: 99             cdq
26: 99             cdq
27: 99             cdq
28: 99             cdq
29: f1             icebp
2a: 3f             (bad)
2b: 9a             (bad)
2c: 99             cdq
2d: 99             cdq
2e: 99             cdq
2f: 99             cdq
30: 99             cdq
31: f1             icebp
```

```

32: 3f                (bad)
33: 9a                (bad)
34: 99                cdq
35: 99                cdq
36: 99                cdq
37: 99                cdq
38: 99                cdq
39: f1                icebp
3a: 3f                (bad)
3b: 9a                (bad)
3c: 99                cdq
3d: 99                cdq
3e: 99                cdq
3f: 99                cdq
40: 99                cdq
41: f1                icebp
42: 3f                (bad)
43: 9a                (bad)
44: 99                cdq
45: 99                cdq
46: 99                cdq
47: 99                cdq
48: 99                cdq
49: f1                icebp
4a: 3f                (bad)
4b: 62 f1 7c 48 10 05 b6 vmovups zmm0,ZMMWORD PTR [rip+0xffffffffffffb6]
    # 0xb
52: ff ff ff
55: eb 40              jmp     0x97
57: 9a                (bad)
58: 99                cdq
59: 99                cdq
5a: 99                cdq
5b: 99                cdq
5c: 99                cdq
5d: 01 40 9a          add     DWORD PTR [rax-0x66],eax
60: 99                cdq
61: 99                cdq
62: 99                cdq
63: 99                cdq
64: 99                cdq
65: 01 40 9a          add     DWORD PTR [rax-0x66],eax
68: 99                cdq
69: 99                cdq
6a: 99                cdq
6b: 99                cdq
6c: 99                cdq
6d: 01 40 9a          add     DWORD PTR [rax-0x66],eax
70: 99                cdq
71: 99                cdq
72: 99                cdq
73: 99                cdq
74: 99                cdq
75: 01 40 9a          add     DWORD PTR [rax-0x66],eax
78: 99                cdq
79: 99                cdq
7a: 99                cdq
7b: 99                cdq
7c: 99                cdq
7d: 01 40 9a          add     DWORD PTR [rax-0x66],eax
80: 99                cdq
81: 99                cdq
82: 99                cdq
83: 99                cdq
84: 99                cdq
85: 01 40 9a          add     DWORD PTR [rax-0x66],eax
88: 99                cdq
89: 99                cdq
8a: 99                cdq

```

```

8b: 99          cdq
8c: 99          cdq
8d: 01 40 9a   add     DWORD PTR [rax-0x66],eax
90: 99          cdq
91: 99          cdq
92: 99          cdq
93: 99          cdq
94: 99          cdq
95: 01 40 62   add     DWORD PTR [rax+0x62],eax
98: f1          icebp
99: 7c 48      jl     0xe3
9b: 10 0d b6 ff ff ff  adc   BYTE PTR [rip+0xffffffffffffb6],cl      # 0x57
a1: eb 40      jmp   0xe3
a3: 66 66 66 66 66 66 0a  data16 data16 data16 data16 data16 data16 or al,BYTE PTR
    [rax+0x66]
aa: 40 66
ac: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
ab: 66
b4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
bb: 66
bc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
c3: 66
c4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
cb: 66
cc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
d3: 66
d4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
db: 66
dc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x62]
e3: 62
e4: f1          icebp
e5: 7c 48      jl     0x12f
e7: 10 15 b6 ff ff ff  adc   BYTE PTR [rip+0xffffffffffffb6],dl      # 0xa3
ed: eb 40      jmp   0x12f
ef: 9a          (bad)
f0: 99          cdq
f1: 99          cdq
f2: 99          cdq
f3: 99          cdq
f4: 99          cdq
f5: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
f8: 99          cdq
f9: 99          cdq
fa: 99          cdq
fb: 99          cdq
fc: 99          cdq
fd: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
100: 99          cdq
101: 99          cdq
102: 99          cdq
103: 99          cdq
104: 99          cdq
105: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
108: 99          cdq
109: 99          cdq
10a: 99          cdq
10b: 99          cdq
10c: 99          cdq
10d: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
110: 99          cdq
111: 99          cdq
112: 99          cdq

```

```

113: 99                                cdq
114: 99                                cdq
115: 11 40 9a                          adc     DWORD PTR [rax-0x66],eax
118: 99                                cdq
119: 99                                cdq
11a: 99                                cdq
11b: 99                                cdq
11c: 99                                cdq
11d: 11 40 9a                          adc     DWORD PTR [rax-0x66],eax
120: 99                                cdq
121: 99                                cdq
122: 99                                cdq
123: 99                                cdq
124: 99                                cdq
125: 11 40 9a                          adc     DWORD PTR [rax-0x66],eax
128: 99                                cdq
129: 99                                cdq
12a: 99                                cdq
12b: 99                                cdq
12c: 99                                cdq
12d: 11 40 62                          adc     DWORD PTR [rax+0x62],eax
130: f1                                icebp
131: 7c 48                              jl     0x17b
133: 10 1d b6 ff ff ff                  adc     BYTE PTR [rip+0xffffffffffffb6],bl      # 0xef
139: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
13f: 62 62 fd 48 b8 3e                  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi]
145: 0f 18 5e 40                        prefetcht2 BYTE PTR [rsi+0x40]
149: 62 62 f5 48 b8 7e 06              vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x180]
150: 0f 18 9e c0 01 00 00              prefetcht2 BYTE PTR [rsi+0x1c0]
157: 62 62 ed 48 b8 7e 0c              vfmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x300]
15e: 0f 18 9e 40 03 00 00              prefetcht2 BYTE PTR [rsi+0x340]
165: 62 62 e5 48 b8 7e 12              vfmadd231pd zmm31,zmm3,ZMMWORD PTR [rsi+0x480]
16c: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]
173: 62 61 fd 48 2b 3a                  vmovntpd ZMMWORD PTR [rdx],zmm31
179: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
17f: 62 62 fd 48 bc 7e 06              vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
186: 0f 18 9e c0 01 00 00              prefetcht2 BYTE PTR [rsi+0x1c0]
18d: 62 62 f5 48 bc 7e 0c              vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x300]
194: 0f 18 9e 40 03 00 00              prefetcht2 BYTE PTR [rsi+0x340]
19b: 62 62 ed 48 bc 7e 12              vfnmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x480]
1a2: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]
1a9: 62 61 fd 48 2b 7a 06              vmovntpd ZMMWORD PTR [rdx+0x180],zmm31
1b0: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
1b6: 62 62 fd 48 b8 7e 0c              vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
1bd: 0f 18 9e 40 03 00 00              prefetcht2 BYTE PTR [rsi+0x340]
1c4: 62 62 f5 48 b8 7e 12              vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
1cb: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]
1d2: 62 61 fd 48 2b 7a 0c              vmovntpd ZMMWORD PTR [rdx+0x300],zmm31
1d9: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
1df: 62 62 fd 48 bc 7e 12              vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
1e6: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]
1ed: 62 61 fd 48 2b 7a 12              vmovntpd ZMMWORD PTR [rdx+0x480],zmm31
1f4: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
1fa: 62 62 fd 48 b8 3e                  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi]
200: 0f 18 5e 40                        prefetcht2 BYTE PTR [rsi+0x40]
204: 62 62 f5 48 b8 7e 06              vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x180]
20b: 0f 18 9e c0 01 00 00              prefetcht2 BYTE PTR [rsi+0x1c0]
212: 62 62 ed 48 b8 7e 0c              vfmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x300]
219: 0f 18 9e 40 03 00 00              prefetcht2 BYTE PTR [rsi+0x340]
220: 62 62 e5 48 b8 7e 12              vfmadd231pd zmm31,zmm3,ZMMWORD PTR [rsi+0x480]
227: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]
22e: 62 61 fd 48 2b 7a 18              vmovntpd ZMMWORD PTR [rdx+0x600],zmm31
235: 62 01 05 40 ef ff                  vpxord zmm31,zmm31,zmm31
23b: 62 62 fd 48 bc 7e 06              vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
242: 0f 18 9e c0 01 00 00              prefetcht2 BYTE PTR [rsi+0x1c0]
249: 62 62 f5 48 bc 7e 0c              vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x300]
250: 0f 18 9e 40 03 00 00              prefetcht2 BYTE PTR [rsi+0x340]
257: 62 62 ed 48 bc 7e 12              vfnmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x480]
25e: 0f 18 9e c0 04 00 00              prefetcht2 BYTE PTR [rsi+0x4c0]

```

```

265: 62 61 fd 48 2b 7a 1e  vmovntpd ZMMWORD PTR [rdx+0x780],zmm31
26c: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
272: 62 62 fd 48 b8 7e 0c  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
279: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
280: 62 62 f5 48 b8 7e 12  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
287: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
28e: 62 61 fd 48 2b 7a 24  vmovntpd ZMMWORD PTR [rdx+0x900],zmm31
295: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
29b: 62 62 fd 48 bc 7e 12  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
2a2: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
2a9: 62 61 fd 48 2b 7a 2a  vmovntpd ZMMWORD PTR [rdx+0xa80],zmm31
2b0: 41 5f                pop     r15
2b2: 41 5e                pop     r14
2b4: 41 5d                pop     r13
2b6: 41 5c                pop     r12
2b8: 5b                  pop     rbx
2b9: c3                  ret

```

Wide-sparse Kernel

```
libxsmm_skx_f64_nn_8x16x4_0_48_48_a1_b0_p0.sreg:    file format binary
```

Disassembly of section .data:

```

00000000 <.data>:
 0: 53                push   rbx
 1: 41 54            push   r12
 3: 41 55            push   r13
 5: 41 56            push   r14
 7: 41 57            push   r15
 9: eb 40            jmp    0x4b
 b: 9a                (bad)
 c: 99                cdq
 d: 99                cdq
 e: 99                cdq
 f: 99                cdq
10: 99                cdq
11: f1                icebp
12: 3f                (bad)
13: 9a                (bad)
14: 99                cdq
15: 99                cdq
16: 99                cdq
17: 99                cdq
18: 99                cdq
19: f1                icebp
1a: 3f                (bad)
1b: 9a                (bad)
1c: 99                cdq
1d: 99                cdq
1e: 99                cdq
1f: 99                cdq
20: 99                cdq
21: f1                icebp
22: 3f                (bad)
23: 9a                (bad)
24: 99                cdq
25: 99                cdq
26: 99                cdq
27: 99                cdq
28: 99                cdq
29: f1                icebp
2a: 3f                (bad)
2b: 9a                (bad)

```



```

2c: 99          cdq
2d: 99          cdq
2e: 99          cdq
2f: 99          cdq
30: 99          cdq
31: f1          icebp
32: 3f          (bad)
33: 9a          (bad)
34: 99          cdq
35: 99          cdq
36: 99          cdq
37: 99          cdq
38: 99          cdq
39: f1          icebp
3a: 3f          (bad)
3b: 9a          (bad)
3c: 99          cdq
3d: 99          cdq
3e: 99          cdq
3f: 99          cdq
40: 99          cdq
41: f1          icebp
42: 3f          (bad)
43: 9a          (bad)
44: 99          cdq
45: 99          cdq
46: 99          cdq
47: 99          cdq
48: 99          cdq
49: f1          icebp
4a: 3f          (bad)
4b: 62 f1 7c 48 10 05 b6 vmovups zmm0,ZMMWORD PTR [rip+0xffffffffffffb6]
    # 0xb
52: ff ff ff
55: eb 40          jmp     0x97
57: 9a          (bad)
58: 99          cdq
59: 99          cdq
5a: 99          cdq
5b: 99          cdq
5c: 99          cdq
5d: 01 40 9a      add     DWORD PTR [rax-0x66],eax
60: 99          cdq
61: 99          cdq
62: 99          cdq
63: 99          cdq
64: 99          cdq
65: 01 40 9a      add     DWORD PTR [rax-0x66],eax
68: 99          cdq
69: 99          cdq
6a: 99          cdq
6b: 99          cdq
6c: 99          cdq
6d: 01 40 9a      add     DWORD PTR [rax-0x66],eax
70: 99          cdq
71: 99          cdq
72: 99          cdq
73: 99          cdq
74: 99          cdq
75: 01 40 9a      add     DWORD PTR [rax-0x66],eax
78: 99          cdq
79: 99          cdq
7a: 99          cdq
7b: 99          cdq
7c: 99          cdq
7d: 01 40 9a      add     DWORD PTR [rax-0x66],eax
80: 99          cdq
81: 99          cdq
82: 99          cdq

```

```

83: 99          cdq
84: 99          cdq
85: 01 40 9a    add     DWORD PTR [rax-0x66],eax
88: 99          cdq
89: 99          cdq
8a: 99          cdq
8b: 99          cdq
8c: 99          cdq
8d: 01 40 9a    add     DWORD PTR [rax-0x66],eax
90: 99          cdq
91: 99          cdq
92: 99          cdq
93: 99          cdq
94: 99          cdq
95: 01 40 62    add     DWORD PTR [rax+0x62],eax
98: f1          icebp
99: 7c 48       jl     0xe3
9b: 10 0d b6 ff ff ff  adc     BYTE PTR [rip+0xffffffffffffb6],cl      # 0x57
a1: eb 40       jmp     0xe3
a3: 66 66 66 66 66 66 0a  data16 data16 data16 data16 data16 data16 or al,BYTE PTR
    [rax+0x66]
aa: 40 66
ac: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
b3: 66
b4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
bb: 66
bc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
c3: 66
c4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
cb: 66
cc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
d3: 66
d4: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x66]
db: 66
dc: 66 66 66 66 66 0a 40  data16 data16 data16 data16 data16 or al,BYTE PTR [rax+0
    x62]
e3: 62
e4: f1          icebp
e5: 7c 48       jl     0x12f
e7: 10 15 b6 ff ff ff  adc     BYTE PTR [rip+0xffffffffffffb6],dl      # 0xa3
ed: eb 40       jmp     0x12f
ef: 9a          (bad)
f0: 99          cdq
f1: 99          cdq
f2: 99          cdq
f3: 99          cdq
f4: 99          cdq
f5: 11 40 9a    adc     DWORD PTR [rax-0x66],eax
f8: 99          cdq
f9: 99          cdq
fa: 99          cdq
fb: 99          cdq
fc: 99          cdq
fd: 11 40 9a    adc     DWORD PTR [rax-0x66],eax
100: 99         cdq
101: 99         cdq
102: 99         cdq
103: 99         cdq
104: 99         cdq
105: 11 40 9a    adc     DWORD PTR [rax-0x66],eax
108: 99         cdq
109: 99         cdq
10a: 99         cdq

```

```

10b: 99          cdq
10c: 99          cdq
10d: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
110: 99          cdq
111: 99          cdq
112: 99          cdq
113: 99          cdq
114: 99          cdq
115: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
118: 99          cdq
119: 99          cdq
11a: 99          cdq
11b: 99          cdq
11c: 99          cdq
11d: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
120: 99          cdq
121: 99          cdq
122: 99          cdq
123: 99          cdq
124: 99          cdq
125: 11 40 9a   adc     DWORD PTR [rax-0x66],eax
128: 99          cdq
129: 99          cdq
12a: 99          cdq
12b: 99          cdq
12c: 99          cdq
12d: 11 40 62   adc     DWORD PTR [rax+0x62],eax
130: f1          icebp
131: 7c 48       jl     0x17b
133: 10 1d b6 ff ff ff   adc     BYTE PTR [rip+0xffffffffffffb6],bl      # 0xef
139: 62 01 0d 40 ef f6   vpxord zmm30,zmm30,zmm30
13f: 62 01 05 40 ef ff   vpxord zmm31,zmm31,zmm31
145: 62 62 fd 48 b8 36   vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
14b: 0f 18 5e 40       prefetcht2 BYTE PTR [rsi+0x40]
14f: 62 62 fd 48 b8 7e 01 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x40]
156: 0f 18 9e 80 00 00 00 prefetcht2 BYTE PTR [rsi+0x80]
15d: 62 62 f5 48 b8 76 06 vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x180]
164: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
16b: 62 62 f5 48 b8 7e 07 vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x1c0]
172: 0f 18 9e 00 02 00 00 prefetcht2 BYTE PTR [rsi+0x200]
179: 62 62 ed 48 b8 76 0c vfmadd231pd zmm30,zmm2,ZMMWORD PTR [rsi+0x300]
180: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
187: 62 62 ed 48 b8 7e 0d vfmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x340]
18e: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
195: 62 62 e5 48 b8 76 12 vfmadd231pd zmm30,zmm3,ZMMWORD PTR [rsi+0x480]
19c: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
1a3: 62 62 e5 48 b8 7e 13 vfmadd231pd zmm31,zmm3,ZMMWORD PTR [rsi+0x4c0]
1aa: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
1b1: 62 61 fd 48 2b 32   vmovntpd ZMMWORD PTR [rdx],zmm30
1b7: 62 61 fd 48 2b 7a 01 vmovntpd ZMMWORD PTR [rdx+0x40],zmm31
1be: 62 01 0d 40 ef f6   vpxord zmm30,zmm30,zmm30
1c4: 62 01 05 40 ef ff   vpxord zmm31,zmm31,zmm31
1ca: 62 62 fd 48 bc 76 06 vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
1d1: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
1d8: 62 62 fd 48 bc 7e 07 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
1df: 0f 18 9e 00 02 00 00 prefetcht2 BYTE PTR [rsi+0x200]
1e6: 62 62 f5 48 bc 76 0c vfnmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x300]
1ed: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
1f4: 62 62 f5 48 bc 7e 0d vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x340]
1fb: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
202: 62 62 ed 48 bc 76 12 vfnmadd231pd zmm30,zmm2,ZMMWORD PTR [rsi+0x480]
209: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
210: 62 62 ed 48 bc 7e 13 vfnmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x4c0]
217: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
21e: 62 61 fd 48 2b 72 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm30
225: 62 61 fd 48 2b 7a 07 vmovntpd ZMMWORD PTR [rdx+0x1c0],zmm31
22c: 62 01 0d 40 ef f6   vpxord zmm30,zmm30,zmm30
232: 62 01 05 40 ef ff   vpxord zmm31,zmm31,zmm31
238: 62 62 fd 48 b8 76 0c vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]

```

```

23f: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
246: 62 62 fd 48 b8 7e 0d  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
24d: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
254: 62 62 f5 48 b8 76 12  vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
25b: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
262: 62 62 f5 48 b8 7e 13  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
269: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
270: 62 61 fd 48 2b 72 0c  vmovntpd ZMMWORD PTR [rdx+0x300],zmm30
277: 62 61 fd 48 2b 7a 0d  vmovntpd ZMMWORD PTR [rdx+0x340],zmm31
27e: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
284: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
28a: 62 62 fd 48 bc 76 12  vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
291: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
298: 62 62 fd 48 bc 7e 13  vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
29f: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
2a6: 62 61 fd 48 2b 72 12  vmovntpd ZMMWORD PTR [rdx+0x480],zmm30
2ad: 62 61 fd 48 2b 7a 13  vmovntpd ZMMWORD PTR [rdx+0x4c0],zmm31
2b4: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
2ba: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
2c0: 62 62 fd 48 b8 36     vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
2c6: 0f 18 5e 40           prefetcht2 BYTE PTR [rsi+0x40]
2ca: 62 62 fd 48 b8 7e 01  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x40]
2d1: 0f 18 9e 80 00 00 00  prefetcht2 BYTE PTR [rsi+0x80]
2d8: 62 62 f5 48 b8 76 06  vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x180]
2df: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
2e6: 62 62 f5 48 b8 7e 07  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x1c0]
2ed: 0f 18 9e 00 02 00 00  prefetcht2 BYTE PTR [rsi+0x200]
2f4: 62 62 ed 48 b8 76 0c  vfmadd231pd zmm30,zmm2,ZMMWORD PTR [rsi+0x300]
2fb: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
302: 62 62 ed 48 b8 7e 0d  vfmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x340]
309: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
310: 62 62 e5 48 b8 76 12  vfmadd231pd zmm30,zmm3,ZMMWORD PTR [rsi+0x480]
317: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
31e: 62 62 e5 48 b8 7e 13  vfmadd231pd zmm31,zmm3,ZMMWORD PTR [rsi+0x4c0]
325: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
32c: 62 61 fd 48 2b 72 18  vmovntpd ZMMWORD PTR [rdx+0x600],zmm30
333: 62 61 fd 48 2b 7a 19  vmovntpd ZMMWORD PTR [rdx+0x640],zmm31
33a: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
340: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
346: 62 62 fd 48 bc 76 06  vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
34d: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
354: 62 62 fd 48 bc 7e 07  vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
35b: 0f 18 9e 00 02 00 00  prefetcht2 BYTE PTR [rsi+0x200]
362: 62 62 f5 48 bc 76 0c  vfnmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x300]
369: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
370: 62 62 f5 48 bc 7e 0d  vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x340]
377: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
37e: 62 62 ed 48 bc 76 12  vfnmadd231pd zmm30,zmm2,ZMMWORD PTR [rsi+0x480]
385: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
38c: 62 62 ed 48 bc 7e 13  vfnmadd231pd zmm31,zmm2,ZMMWORD PTR [rsi+0x4c0]
393: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
39a: 62 61 fd 48 2b 72 1e  vmovntpd ZMMWORD PTR [rdx+0x780],zmm30
3a1: 62 61 fd 48 2b 7a 1f  vmovntpd ZMMWORD PTR [rdx+0x7c0],zmm31
3a8: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
3ae: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
3b4: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
3bb: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
3c2: 62 62 fd 48 b8 7e 0d  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
3c9: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
3d0: 62 62 f5 48 b8 76 12  vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
3d7: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
3de: 62 62 f5 48 b8 7e 13  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
3e5: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
3ec: 62 61 fd 48 2b 72 24  vmovntpd ZMMWORD PTR [rdx+0x900],zmm30
3f3: 62 61 fd 48 2b 7a 25  vmovntpd ZMMWORD PTR [rdx+0x940],zmm31
3fa: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
400: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
406: 62 62 fd 48 bc 76 12  vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
40d: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]

```

```

414: 62 62 fd 48 bc 7e 13 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
41b: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
422: 62 61 fd 48 2b 72 2a vmovntpd ZMMWORD PTR [rdx+0xa80],zmm30
429: 62 61 fd 48 2b 7a 2b vmovntpd ZMMWORD PTR [rdx+0xac0],zmm31
430: 41 5f pop r15
432: 41 5e pop r14
434: 41 5d pop r13
436: 41 5c pop r12
438: 5b pop rbx
439: c3 ret

```

Dense Kernel

```

libxsmm_skx_f64_nn_8x8x4_48_4_48_a1_b0_p0_br0_uh0_si0_tc-
  abid_avnni0_bvnni0_cvnni0_decompress_A0_spfactor1.mxm: file format binary

```

Disassembly of section .data:

```

00000000 <.data>:
 0: 53 push rbx
 1: 41 54 push r12
 3: 41 55 push r13
 5: 41 56 push r14
 7: 41 57 push r15
 9: 49 c7 c3 00 00 00 00 mov r11,0x0
10: 49 83 c3 08 add r11,0x8
14: 49 c7 c2 00 00 00 00 mov r10,0x0
1b: 49 83 c2 08 add r10,0x8
1f: 62 01 3d 40 ef c0 vpxord zmm24,zmm24,zmm24
25: 62 01 35 40 ef c9 vpxord zmm25,zmm25,zmm25
2b: 62 01 2d 40 ef d2 vpxord zmm26,zmm26,zmm26
31: 62 01 25 40 ef db vpxord zmm27,zmm27,zmm27
37: 62 01 1d 40 ef e4 vpxord zmm28,zmm28,zmm28
3d: 62 01 15 40 ef ed vpxord zmm29,zmm29,zmm29
43: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
49: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
4f: 62 a1 7d 40 ef c0 vpxord zmm16,zmm16,zmm16
55: 62 a1 75 40 ef c9 vpxord zmm17,zmm17,zmm17
5b: 62 a1 6d 40 ef d2 vpxord zmm18,zmm18,zmm18
61: 62 a1 65 40 ef db vpxord zmm19,zmm19,zmm19
67: 62 a1 5d 40 ef e4 vpxord zmm20,zmm20,zmm20
6d: 62 a1 55 40 ef ed vpxord zmm21,zmm21,zmm21
73: 62 a1 4d 40 ef f6 vpxord zmm22,zmm22,zmm22
79: 62 a1 45 40 ef ff vpxord zmm23,zmm23,zmm23
7f: 62 f1 fd 48 10 07 vmovupd zmm0,ZMMWORD PTR [rdi]
85: 62 f1 fd 48 10 4f 06 vmovupd zmm1,ZMMWORD PTR [rdi+0x180]
8c: 62 62 fd 58 b8 06 vfmadd231pd zmm24,zmm0,QWORD PTR [rsi]{1to8}
92: 62 62 fd 58 b8 4e 04 vfmadd231pd zmm25,zmm0,QWORD PTR [rsi+0x20]{1to8}
99: 62 62 fd 58 b8 56 08 vfmadd231pd zmm26,zmm0,QWORD PTR [rsi+0x40]{1to8}
a0: 62 62 fd 58 b8 5e 0c vfmadd231pd zmm27,zmm0,QWORD PTR [rsi+0x60]{1to8}
a7: 62 62 fd 58 b8 66 10 vfmadd231pd zmm28,zmm0,QWORD PTR [rsi+0x80]{1to8}
ae: 62 62 fd 58 b8 6e 14 vfmadd231pd zmm29,zmm0,QWORD PTR [rsi+0xa0]{1to8}
b5: 62 62 fd 58 b8 76 18 vfmadd231pd zmm30,zmm0,QWORD PTR [rsi+0xc0]{1to8}
bc: 62 62 fd 58 b8 7e 1c vfmadd231pd zmm31,zmm0,QWORD PTR [rsi+0xe0]{1to8}
c3: 62 f1 fd 48 10 47 0c vmovupd zmm0,ZMMWORD PTR [rdi+0x300]
ca: 62 e2 f5 58 b8 46 01 vfmadd231pd zmm16,zmm1,QWORD PTR [rsi+0x8]{1to8}
d1: 62 e2 f5 58 b8 4e 05 vfmadd231pd zmm17,zmm1,QWORD PTR [rsi+0x28]{1to8}
d8: 62 e2 f5 58 b8 56 09 vfmadd231pd zmm18,zmm1,QWORD PTR [rsi+0x48]{1to8}
df: 62 e2 f5 58 b8 5e 0d vfmadd231pd zmm19,zmm1,QWORD PTR [rsi+0x68]{1to8}
e6: 62 e2 f5 58 b8 66 11 vfmadd231pd zmm20,zmm1,QWORD PTR [rsi+0x88]{1to8}
ed: 62 e2 f5 58 b8 6e 15 vfmadd231pd zmm21,zmm1,QWORD PTR [rsi+0xa8]{1to8}
f4: 62 e2 f5 58 b8 76 19 vfmadd231pd zmm22,zmm1,QWORD PTR [rsi+0xc8]{1to8}
fb: 62 e2 f5 58 b8 7e 1d vfmadd231pd zmm23,zmm1,QWORD PTR [rsi+0xe8]{1to8}
102: 62 f1 fd 48 10 4f 12 vmovupd zmm1,ZMMWORD PTR [rdi+0x480]

```

```

109: 62 62 fd 58 b8 46 02 vfmadd231pd zmm24, zmm0, QWORD PTR [rsi+0x10]{1to8}
110: 62 62 fd 58 b8 4e 06 vfmadd231pd zmm25, zmm0, QWORD PTR [rsi+0x30]{1to8}
117: 62 62 fd 58 b8 56 0a vfmadd231pd zmm26, zmm0, QWORD PTR [rsi+0x50]{1to8}
11e: 62 62 fd 58 b8 5e 0e vfmadd231pd zmm27, zmm0, QWORD PTR [rsi+0x70]{1to8}
125: 62 62 fd 58 b8 66 12 vfmadd231pd zmm28, zmm0, QWORD PTR [rsi+0x90]{1to8}
12c: 62 62 fd 58 b8 6e 16 vfmadd231pd zmm29, zmm0, QWORD PTR [rsi+0xb0]{1to8}
133: 62 62 fd 58 b8 76 1a vfmadd231pd zmm30, zmm0, QWORD PTR [rsi+0xd0]{1to8}
13a: 62 62 fd 58 b8 7e 1e vfmadd231pd zmm31, zmm0, QWORD PTR [rsi+0xf0]{1to8}
141: 48 81 c7 00 06 00 00 add rdi, 0x600
148: 62 e2 f5 58 b8 46 03 vfmadd231pd zmm16, zmm1, QWORD PTR [rsi+0x18]{1to8}
14f: 62 e2 f5 58 b8 4e 07 vfmadd231pd zmm17, zmm1, QWORD PTR [rsi+0x38]{1to8}
156: 62 e2 f5 58 b8 56 0b vfmadd231pd zmm18, zmm1, QWORD PTR [rsi+0x58]{1to8}
15d: 62 e2 f5 58 b8 5e 0f vfmadd231pd zmm19, zmm1, QWORD PTR [rsi+0x78]{1to8}
164: 62 e2 f5 58 b8 66 13 vfmadd231pd zmm20, zmm1, QWORD PTR [rsi+0x98]{1to8}
16b: 62 e2 f5 58 b8 6e 17 vfmadd231pd zmm21, zmm1, QWORD PTR [rsi+0xb8]{1to8}
172: 62 e2 f5 58 b8 76 1b vfmadd231pd zmm22, zmm1, QWORD PTR [rsi+0xd8]{1to8}
179: 62 e2 f5 58 b8 7e 1f vfmadd231pd zmm23, zmm1, QWORD PTR [rsi+0xf8]{1to8}
180: 62 21 bd 40 58 c0 vaddpd zmm24, zmm24, zmm16
186: 62 21 b5 40 58 c9 vaddpd zmm25, zmm25, zmm17
18c: 62 21 ad 40 58 d2 vaddpd zmm26, zmm26, zmm18
192: 62 21 a5 40 58 db vaddpd zmm27, zmm27, zmm19
198: 62 21 9d 40 58 e4 vaddpd zmm28, zmm28, zmm20
19e: 62 21 95 40 58 ed vaddpd zmm29, zmm29, zmm21
1a4: 62 21 8d 40 58 f6 vaddpd zmm30, zmm30, zmm22
1aa: 62 21 85 40 58 ff vaddpd zmm31, zmm31, zmm23
1b0: 62 61 fd 48 2b 02 vmovntpd ZMMWORD PTR [rdx], zmm24
1b6: 62 61 fd 48 2b 4a 06 vmovntpd ZMMWORD PTR [rdx+0x180], zmm25
1bd: 62 61 fd 48 2b 52 0c vmovntpd ZMMWORD PTR [rdx+0x300], zmm26
1c4: 62 61 fd 48 2b 5a 12 vmovntpd ZMMWORD PTR [rdx+0x480], zmm27
1cb: 62 61 fd 48 2b 62 18 vmovntpd ZMMWORD PTR [rdx+0x600], zmm28
1d2: 62 61 fd 48 2b 6a 1e vmovntpd ZMMWORD PTR [rdx+0x780], zmm29
1d9: 62 61 fd 48 2b 72 24 vmovntpd ZMMWORD PTR [rdx+0x900], zmm30
1e0: 62 61 fd 48 2b 7a 2a vmovntpd ZMMWORD PTR [rdx+0xa80], zmm31
1e7: 48 83 c2 40 add rdx, 0x40
1eb: 48 81 ef c0 05 00 00 sub rdi, 0x5c0
1f2: 49 83 fa 08 cmp r10, 0x8
1f6: 0f 8c 1f fe ff ff jl 0x1b
1fc: 48 81 c2 c0 0b 00 00 add rdx, 0xbc0
203: 48 81 c6 00 01 00 00 add rsi, 0x100
20a: 48 83 ef 40 sub rdi, 0x40
20e: 49 83 fb 08 cmp r11, 0x8
212: 0f 8c f8 fd ff ff jl 0x10
218: 41 5f pop r15
21a: 41 5e pop r14
21c: 41 5d pop r13
21e: 41 5c pop r12
220: 5b pop rbx
221: c3 ret

```

B.2 GiMMiK Kernel

```

void
gimmik_mm(int ncol,
           const double* restrict b, int ldb,
           double* restrict c, int ldc)
{
    double dotp;

    #pragma omp parallel for simd private(dotp)
    for (int i = 0; i < ncol; i++)
    {
        dotp = 1.1*b[i + 0*ldb] + 2.2*b[i + 1*ldb] + 3.3*b[i + 2*ldb] + 4.4*b[i + 3*
            ldb];
        c[i + 0*ldc] = dotp;
    }
}

```

```

dotp = -1.1*b[i + 1*ldb] + -2.2*b[i + 2*ldb] + -3.3*b[i + 3*ldb];
c[i + 1*ldc] = dotp;
dotp = 1.1*b[i + 2*ldb] + 2.2*b[i + 3*ldb];
c[i + 2*ldc] = dotp;
dotp = -1.1*b[i + 3*ldb];
c[i + 3*ldc] = dotp;
dotp = 1.1*b[i + 0*ldb] + 2.2*b[i + 1*ldb] + 3.3*b[i + 2*ldb] + 4.4*b[i + 3*
ldb];
c[i + 4*ldc] = dotp;
dotp = -1.1*b[i + 1*ldb] + -2.2*b[i + 2*ldb] + -3.3*b[i + 3*ldb];
c[i + 5*ldc] = dotp;
dotp = 1.1*b[i + 2*ldb] + 2.2*b[i + 3*ldb];
c[i + 6*ldc] = dotp;
dotp = -1.1*b[i + 3*ldb];
c[i + 7*ldc] = dotp;
}
}

```

B.3 Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory

No Multiple Accumulation

```
libxsmm_skx_f64_nn_8x8x4_0_48_48_a1_b0_p0.sreg: file format binary
```

```
Disassembly of section .data:
```

```

00000000 <.data>:
0: 53          push  rbx
1: 41 54       push  r12
3: 41 55       push  r13
5: 41 56       push  r14
7: 41 57       push  r15
9: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
f: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
15: 62 62 fd 48 b8 3e vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi]
1b: 0f 18 5e 40   prefetcht2 BYTE PTR [rsi+0x40]
1f: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
26: 62 62 fd 48 b8 7e 06 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
2d: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
34: 62 f2 fd 48 19 47 02 vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
3b: 62 62 fd 48 b8 7e 0c vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
42: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
49: 62 f2 fd 48 19 47 03 vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
50: 62 62 fd 48 b8 7e 12 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
57: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
5e: 62 61 fd 48 2b 3a vmovntpd ZMMWORD PTR [rdx],zmm31
64: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
6a: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
70: 62 62 fd 48 bc 7e 06 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
77: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
7e: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
85: 62 62 fd 48 bc 7e 0c vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
8c: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
93: 62 f2 fd 48 19 47 02 vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
9a: 62 62 fd 48 bc 7e 12 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
a1: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
a8: 62 61 fd 48 2b 7a 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm31

```

```

af: 62 01 05 40 ef ff      vpxord zmm31,zmm31,zmm31
b5: 62 f2 fd 48 19 07      vbroadcastsd zmm0,QWORD PTR [rdi]
bb: 62 62 fd 48 b8 7e 0c    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
c2: 0f 18 9e 40 03 00 00    prefetcht2 BYTE PTR [rsi+0x340]
c9: 62 f2 fd 48 19 47 01    vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
d0: 62 62 fd 48 b8 7e 12    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
d7: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
de: 62 61 fd 48 2b 7a 0c    vmovntpd ZMMWORD PTR [rdx+0x300],zmm31
e5: 62 01 05 40 ef ff      vpxord zmm31,zmm31,zmm31
eb: 62 f2 fd 48 19 07      vbroadcastsd zmm0,QWORD PTR [rdi]
f1: 62 62 fd 48 bc 7e 12    vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
f8: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
ff: 62 61 fd 48 2b 7a 12    vmovntpd ZMMWORD PTR [rdx+0x480],zmm31
106: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
10c: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
112: 62 62 fd 48 b8 3e     vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi]
118: 0f 18 5e 40           prefetcht2 BYTE PTR [rsi+0x40]
11c: 62 f2 fd 48 19 47 01    vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
123: 62 62 fd 48 b8 7e 06    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
12a: 0f 18 9e c0 01 00 00    prefetcht2 BYTE PTR [rsi+0x1c0]
131: 62 f2 fd 48 19 47 02    vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
138: 62 62 fd 48 b8 7e 0c    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
13f: 0f 18 9e 40 03 00 00    prefetcht2 BYTE PTR [rsi+0x340]
146: 62 f2 fd 48 19 47 03    vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
14d: 62 62 fd 48 b8 7e 12    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
154: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
15b: 62 61 fd 48 2b 7a 18    vmovntpd ZMMWORD PTR [rdx+0x600],zmm31
162: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
168: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
16e: 62 62 fd 48 bc 7e 06    vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x180]
175: 0f 18 9e c0 01 00 00    prefetcht2 BYTE PTR [rsi+0x1c0]
17c: 62 f2 fd 48 19 47 01    vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
183: 62 62 fd 48 bc 7e 0c    vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
18a: 0f 18 9e 40 03 00 00    prefetcht2 BYTE PTR [rsi+0x340]
191: 62 f2 fd 48 19 47 02    vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
198: 62 62 fd 48 bc 7e 12    vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
19f: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
1a6: 62 61 fd 48 2b 7a 1e    vmovntpd ZMMWORD PTR [rdx+0x780],zmm31
1ad: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
1b3: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
1b9: 62 62 fd 48 b8 7e 0c    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x300]
1c0: 0f 18 9e 40 03 00 00    prefetcht2 BYTE PTR [rsi+0x340]
1c7: 62 f2 fd 48 19 47 01    vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
1ce: 62 62 fd 48 b8 7e 12    vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
1d5: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
1dc: 62 61 fd 48 2b 7a 24    vmovntpd ZMMWORD PTR [rdx+0x900],zmm31
1e3: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
1e9: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
1ef: 62 62 fd 48 bc 7e 12    vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x480]
1f6: 0f 18 9e c0 04 00 00    prefetcht2 BYTE PTR [rsi+0x4c0]
1fd: 62 61 fd 48 2b 7a 2a    vmovntpd ZMMWORD PTR [rdx+0xa80],zmm31
204: 41 5f                  pop     r15
206: 41 5e                  pop     r14
208: 41 5d                  pop     r13
20a: 41 5c                  pop     r12
20c: 5b                    pop     rbx
20d: c3                    ret

```

N Blocking of 2

```
libxsmm_skx_f64_nn_8x16x4_0_48_48_a1_b0_p0.sreg:      file format binary
```

```
Disassembly of section .data:
```



```

00000000 <.data>:
 0: 53          push   rbx
 1: 41 54       push   r12
 3: 41 55       push   r13
 5: 41 56       push   r14
 7: 41 57       push   r15
 9: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
 f: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
15: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
1b: 62 62 fd 48 b8 36 vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
21: 0f 18 5e 40      prefetcht2 BYTE PTR [rsi+0x40]
25: 62 62 fd 48 b8 7e 01 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x40]
2c: 0f 18 9e 80 00 00 00 prefetcht2 BYTE PTR [rsi+0x80]
33: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
3a: 62 62 fd 48 b8 76 06 vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
41: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
48: 62 62 fd 48 b8 7e 07 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
4f: 0f 18 9e 00 02 00 00 prefetcht2 BYTE PTR [rsi+0x200]
56: 62 f2 fd 48 19 47 02 vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
5d: 62 62 fd 48 b8 76 0c vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
64: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
6b: 62 62 fd 48 b8 7e 0d vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
72: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
79: 62 f2 fd 48 19 47 03 vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
80: 62 62 fd 48 b8 76 12 vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
87: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
8e: 62 62 fd 48 b8 7e 13 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
95: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
9c: 62 61 fd 48 2b 32      vmovntpd ZMMWORD PTR [rdx],zmm30
a2: 62 61 fd 48 2b 7a 01 vmovntpd ZMMWORD PTR [rdx+0x40],zmm31
a9: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
af: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
b5: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
bb: 62 62 fd 48 bc 76 06 vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
c2: 0f 18 9e c0 01 00 00 prefetcht2 BYTE PTR [rsi+0x1c0]
c9: 62 62 fd 48 bc 7e 07 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
d0: 0f 18 9e 00 02 00 00 prefetcht2 BYTE PTR [rsi+0x200]
d7: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
de: 62 62 fd 48 bc 76 0c vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
e5: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
ec: 62 62 fd 48 bc 7e 0d vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
f3: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
fa: 62 f2 fd 48 19 47 02 vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
101: 62 62 fd 48 bc 76 12 vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
108: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
10f: 62 62 fd 48 bc 7e 13 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
116: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
11d: 62 61 fd 48 2b 72 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm30
124: 62 61 fd 48 2b 7a 07 vmovntpd ZMMWORD PTR [rdx+0x1c0],zmm31
12b: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
131: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
137: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
13d: 62 62 fd 48 b8 76 0c vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
144: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
14b: 62 62 fd 48 b8 7e 0d vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
152: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
159: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
160: 62 62 fd 48 b8 76 12 vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
167: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
16e: 62 62 fd 48 b8 7e 13 vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
175: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
17c: 62 61 fd 48 2b 72 0c vmovntpd ZMMWORD PTR [rdx+0x300],zmm30
183: 62 61 fd 48 2b 7a 0d vmovntpd ZMMWORD PTR [rdx+0x340],zmm31
18a: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
190: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
196: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
19c: 62 62 fd 48 bc 76 12 vfnmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
1a3: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
1aa: 62 62 fd 48 bc 7e 13 vfnmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]

```

```

1b1: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
1b8: 62 61 fd 48 2b 72 12  vmovntpd ZMMWORD PTR [rdx+0x480],zmm30
1bf: 62 61 fd 48 2b 7a 13  vmovntpd ZMMWORD PTR [rdx+0x4c0],zmm31
1c6: 62 01 0d 40 ef f6    vpxord zmm30,zmm30,zmm30
1cc: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
1d2: 62 f2 fd 48 19 07    vbroadcastsd zmm0,QWORD PTR [rdi]
1d8: 62 62 fd 48 b8 36    vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
1de: 0f 18 5e 40          prefetcht2 BYTE PTR [rsi+0x40]
1e2: 62 62 fd 48 b8 7e 01  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x40]
1e9: 0f 18 9e 80 00 00 00  prefetcht2 BYTE PTR [rsi+0x80]
1f0: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
1f7: 62 62 fd 48 b8 76 06  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
1fe: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
205: 62 62 fd 48 b8 7e 07  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
20c: 0f 18 9e 00 02 00 00  prefetcht2 BYTE PTR [rsi+0x200]
213: 62 f2 fd 48 19 47 02  vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
21a: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
221: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
228: 62 62 fd 48 b8 7e 0d  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
22f: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
236: 62 f2 fd 48 19 47 03  vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
23d: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
244: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
24b: 62 62 fd 48 b8 7e 13  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
252: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
259: 62 61 fd 48 2b 72 18  vmovntpd ZMMWORD PTR [rdx+0x600],zmm30
260: 62 61 fd 48 2b 7a 19  vmovntpd ZMMWORD PTR [rdx+0x640],zmm31
267: 62 01 0d 40 ef f6    vpxord zmm30,zmm30,zmm30
26d: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
273: 62 f2 fd 48 19 07    vbroadcastsd zmm0,QWORD PTR [rdi]
279: 62 62 fd 48 bc 76 06  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
280: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
287: 62 62 fd 48 bc 7e 07  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x1c0]
28e: 0f 18 9e 00 02 00 00  prefetcht2 BYTE PTR [rsi+0x200]
295: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
29c: 62 62 fd 48 bc 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
2a3: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
2aa: 62 62 fd 48 bc 7e 0d  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
2b1: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
2b8: 62 f2 fd 48 19 47 02  vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
2bf: 62 62 fd 48 bc 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
2c6: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
2cd: 62 62 fd 48 bc 7e 13  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
2d4: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
2db: 62 61 fd 48 2b 72 1e  vmovntpd ZMMWORD PTR [rdx+0x780],zmm30
2e2: 62 61 fd 48 2b 7a 1f  vmovntpd ZMMWORD PTR [rdx+0x7c0],zmm31
2e9: 62 01 0d 40 ef f6    vpxord zmm30,zmm30,zmm30
2ef: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
2f5: 62 f2 fd 48 19 07    vbroadcastsd zmm0,QWORD PTR [rdi]
2fb: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
302: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
309: 62 62 fd 48 b8 7e 0d  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x340]
310: 0f 18 9e 80 03 00 00  prefetcht2 BYTE PTR [rsi+0x380]
317: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
31e: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
325: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
32c: 62 62 fd 48 b8 7e 13  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
333: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
33a: 62 61 fd 48 2b 72 24  vmovntpd ZMMWORD PTR [rdx+0x900],zmm30
341: 62 61 fd 48 2b 7a 25  vmovntpd ZMMWORD PTR [rdx+0x940],zmm31
348: 62 01 0d 40 ef f6    vpxord zmm30,zmm30,zmm30
34e: 62 01 05 40 ef ff    vpxord zmm31,zmm31,zmm31
354: 62 f2 fd 48 19 07    vbroadcastsd zmm0,QWORD PTR [rdi]
35a: 62 62 fd 48 bc 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
361: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
368: 62 62 fd 48 bc 7e 13  vfmadd231pd zmm31,zmm0,ZMMWORD PTR [rsi+0x4c0]
36f: 0f 18 9e 00 05 00 00  prefetcht2 BYTE PTR [rsi+0x500]
376: 62 61 fd 48 2b 72 2a  vmovntpd ZMMWORD PTR [rdx+0xa80],zmm30
37d: 62 61 fd 48 2b 7a 2b  vmovntpd ZMMWORD PTR [rdx+0xac0],zmm31

```

```

384: 41 5f          pop     r15
386: 41 5e          pop     r14
388: 41 5d          pop     r13
38a: 41 5c          pop     r12
38c: 5b           pop     rbx
38d: c3           ret

```

M Blocking of 2

```
libxsmm_skx_f64_nn_8x8x4_0_48_48_a1_b0_p0.sreg:    file format binary
```

```
Disassembly of section .data:
```

```

00000000 <.data>:
 0: 53           push   rbx
 1: 41 54       push   r12
 3: 41 55       push   r13
 5: 41 56       push   r14
 7: 41 57       push   r15
 9: 62 01 0d 40 ef f6  vpxord zmm30,zmm30,zmm30
 f: 62 01 05 40 ef ff  vpxord zmm31,zmm31,zmm31
15: 62 f2 fd 48 19 07  vbroadcastsd zmm0,QWORD PTR [rdi]
1b: 62 62 fd 48 b8 36  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
21: 0f 18 5e 40       prefetcht2 BYTE PTR [rsi+0x40]
25: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
2c: 62 62 fd 48 b8 76 06  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]
33: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
3a: 62 f2 fd 48 19 0f       vbroadcastsd zmm1,QWORD PTR [rdi]
40: 62 62 f5 48 bc 7e 06  vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x180]
47: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
4e: 62 f2 fd 48 19 47 02  vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
55: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
5c: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
63: 62 f2 fd 48 19 4f 01  vbroadcastsd zmm1,QWORD PTR [rdi+0x8]
6a: 62 62 f5 48 bc 7e 0c  vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x300]
71: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
78: 62 f2 fd 48 19 47 03  vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
7f: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
86: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
8d: 62 f2 fd 48 19 4f 02  vbroadcastsd zmm1,QWORD PTR [rdi+0x10]
94: 62 62 f5 48 bc 7e 12  vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
9b: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
a2: 62 61 fd 48 2b 32       vmovntpd ZMMWORD PTR [rdx],zmm30
a8: 62 61 fd 48 2b 7a 06  vmovntpd ZMMWORD PTR [rdx+0x180],zmm31
af: 62 01 0d 40 ef f6  vpxord zmm30,zmm30,zmm30
b5: 62 01 05 40 ef ff  vpxord zmm31,zmm31,zmm31
bb: 62 f2 fd 48 19 07  vbroadcastsd zmm0,QWORD PTR [rdi]
c1: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
c8: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
cf: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
d6: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
dd: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
e4: 62 f2 fd 48 19 0f       vbroadcastsd zmm1,QWORD PTR [rdi]
ea: 62 62 f5 48 bc 7e 12  vfnmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
f1: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
f8: 62 61 fd 48 2b 72 0c  vmovntpd ZMMWORD PTR [rdx+0x300],zmm30
ff: 62 61 fd 48 2b 7a 12  vmovntpd ZMMWORD PTR [rdx+0x480],zmm31
106: 62 01 0d 40 ef f6  vpxord zmm30,zmm30,zmm30
10c: 62 01 05 40 ef ff  vpxord zmm31,zmm31,zmm31
112: 62 f2 fd 48 19 07  vbroadcastsd zmm0,QWORD PTR [rdi]
118: 62 62 fd 48 b8 36  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi]
11e: 0f 18 5e 40       prefetcht2 BYTE PTR [rsi+0x40]
122: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
129: 62 62 fd 48 b8 76 06  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x180]

```

```

130: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
137: 62 f2 fd 48 19 0f      vbroadcastsd zmm1,QWORD PTR [rdi]
13d: 62 62 f5 48 bc 7e 06  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x180]
144: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
14b: 62 f2 fd 48 19 47 02  vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
152: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
159: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
160: 62 f2 fd 48 19 4f 01  vbroadcastsd zmm1,QWORD PTR [rdi+0x8]
167: 62 62 f5 48 bc 7e 0c  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x300]
16e: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
175: 62 f2 fd 48 19 47 03  vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
17c: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
183: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
18a: 62 f2 fd 48 19 4f 02  vbroadcastsd zmm1,QWORD PTR [rdi+0x10]
191: 62 62 f5 48 bc 7e 12  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
198: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
19f: 62 61 fd 48 2b 72 18  vmovntpd ZMMWORD PTR [rdx+0x600],zmm30
1a6: 62 61 fd 48 2b 7a 1e  vmovntpd ZMMWORD PTR [rdx+0x780],zmm31
1ad: 62 01 0d 40 ef f6      vpxord zmm30,zmm30,zmm30
1b3: 62 01 05 40 ef ff      vpxord zmm31,zmm31,zmm31
1b9: 62 f2 fd 48 19 07      vbroadcastsd zmm0,QWORD PTR [rdi]
1bf: 62 62 fd 48 b8 76 0c  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x300]
1c6: 0f 18 9e 40 03 00 00  prefetcht2 BYTE PTR [rsi+0x340]
1cd: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
1d4: 62 62 fd 48 b8 76 12  vfmadd231pd zmm30,zmm0,ZMMWORD PTR [rsi+0x480]
1db: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
1e2: 62 f2 fd 48 19 0f      vbroadcastsd zmm1,QWORD PTR [rdi]
1e8: 62 62 f5 48 bc 7e 12  vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x480]
1ef: 0f 18 9e c0 04 00 00  prefetcht2 BYTE PTR [rsi+0x4c0]
1f6: 62 61 fd 48 2b 72 24  vmovntpd ZMMWORD PTR [rdx+0x900],zmm30
1fd: 62 61 fd 48 2b 7a 2a  vmovntpd ZMMWORD PTR [rdx+0xa80],zmm31
204: 41 5f                    pop     r15
206: 41 5e                    pop     r14
208: 41 5d                    pop     r13
20a: 41 5c                    pop     r12
20c: 5b                      pop     rbx
20d: c3                      ret

```

N Blocking of 2 and M Blocking of 2

```
libxsmm_skx_f64_nn_8x16x4_0_48_48_a1_b0_p0.sreg:      file format binary
```

```
Disassembly of section .data:
```

```

00000000 <.data>:
 0: 53                      push   rbx
 1: 41 54                   push   r12
 3: 41 55                   push   r13
 5: 41 56                   push   r14
 7: 41 57                   push   r15
 9: 62 01 1d 40 ef e4      vpxord zmm28,zmm28,zmm28
 f: 62 01 15 40 ef ed      vpxord zmm29,zmm29,zmm29
15: 62 01 0d 40 ef f6      vpxord zmm30,zmm30,zmm30
1b: 62 01 05 40 ef ff      vpxord zmm31,zmm31,zmm31
21: 62 f2 fd 48 19 07      vbroadcastsd zmm0,QWORD PTR [rdi]
27: 62 62 fd 48 b8 26      vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi]
2d: 0f 18 5e 40            prefetcht2 BYTE PTR [rsi+0x40]
31: 62 62 fd 48 b8 6e 01  vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x40]
38: 0f 18 9e 80 00 00 00  prefetcht2 BYTE PTR [rsi+0x80]
3f: 62 f2 fd 48 19 47 01  vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
46: 62 62 fd 48 b8 66 06  vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x180]
4d: 0f 18 9e c0 01 00 00  prefetcht2 BYTE PTR [rsi+0x1c0]
54: 62 62 fd 48 b8 6e 07  vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x1c0]
5b: 0f 18 9e 00 02 00 00  prefetcht2 BYTE PTR [rsi+0x200]

```

```

62: 62 f2 fd 48 19 0f      vbroadcastsd zmm1,QWORD PTR [rdi]
68: 62 62 f5 48 bc 76 06   vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x180]
6f: 0f 18 9e c0 01 00 00   prefetcht2 BYTE PTR [rsi+0x1c0]
76: 62 62 f5 48 bc 7e 07   vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x1c0]
7d: 0f 18 9e 00 02 00 00   prefetcht2 BYTE PTR [rsi+0x200]
84: 62 f2 fd 48 19 47 02   vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
8b: 62 62 fd 48 b8 66 0c   vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x300]
92: 0f 18 9e 40 03 00 00   prefetcht2 BYTE PTR [rsi+0x340]
99: 62 62 fd 48 b8 6e 0d   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x340]
a0: 0f 18 9e 80 03 00 00   prefetcht2 BYTE PTR [rsi+0x380]
a7: 62 f2 fd 48 19 4f 01   vbroadcastsd zmm1,QWORD PTR [rdi+0x8]
ae: 62 62 f5 48 bc 76 0c   vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x300]
b5: 0f 18 9e 40 03 00 00   prefetcht2 BYTE PTR [rsi+0x340]
bc: 62 62 f5 48 bc 7e 0d   vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x340]
c3: 0f 18 9e 80 03 00 00   prefetcht2 BYTE PTR [rsi+0x380]
ca: 62 f2 fd 48 19 47 03   vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
d1: 62 62 fd 48 b8 66 12   vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x480]
d8: 0f 18 9e c0 04 00 00   prefetcht2 BYTE PTR [rsi+0x4c0]
df: 62 62 fd 48 b8 6e 13   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x4c0]
e6: 0f 18 9e 00 05 00 00   prefetcht2 BYTE PTR [rsi+0x500]
ed: 62 f2 fd 48 19 4f 02   vbroadcastsd zmm1,QWORD PTR [rdi+0x10]
f4: 62 62 f5 48 bc 76 12   vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
fb: 0f 18 9e c0 04 00 00   prefetcht2 BYTE PTR [rsi+0x4c0]
102: 62 62 f5 48 bc 7e 13   vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
109: 0f 18 9e 00 05 00 00   prefetcht2 BYTE PTR [rsi+0x500]
110: 62 61 fd 48 2b 22      vmovntpd ZMMWORD PTR [rdx],zmm28
116: 62 61 fd 48 2b 6a 01   vmovntpd ZMMWORD PTR [rdx+0x40],zmm29
11d: 62 61 fd 48 2b 72 06   vmovntpd ZMMWORD PTR [rdx+0x180],zmm30
124: 62 61 fd 48 2b 7a 07   vmovntpd ZMMWORD PTR [rdx+0x1c0],zmm31
12b: 62 01 1d 40 ef e4     vpxord zmm28,zmm28,zmm28
131: 62 01 15 40 ef ed     vpxord zmm29,zmm29,zmm29
137: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
13d: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
143: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
149: 62 62 fd 48 b8 66 0c   vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x300]
150: 0f 18 9e 40 03 00 00   prefetcht2 BYTE PTR [rsi+0x340]
157: 62 62 fd 48 b8 6e 0d   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x340]
15e: 0f 18 9e 80 03 00 00   prefetcht2 BYTE PTR [rsi+0x380]
165: 62 f2 fd 48 19 47 01   vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
16c: 62 62 fd 48 b8 66 12   vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x480]
173: 0f 18 9e c0 04 00 00   prefetcht2 BYTE PTR [rsi+0x4c0]
17a: 62 62 fd 48 b8 6e 13   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x4c0]
181: 0f 18 9e 00 05 00 00   prefetcht2 BYTE PTR [rsi+0x500]
188: 62 f2 fd 48 19 0f     vbroadcastsd zmm1,QWORD PTR [rdi]
18e: 62 62 f5 48 bc 76 12   vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
195: 0f 18 9e c0 04 00 00   prefetcht2 BYTE PTR [rsi+0x4c0]
19c: 62 62 f5 48 bc 7e 13   vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
1a3: 0f 18 9e 00 05 00 00   prefetcht2 BYTE PTR [rsi+0x500]
1aa: 62 61 fd 48 2b 62 0c   vmovntpd ZMMWORD PTR [rdx+0x300],zmm28
1b1: 62 61 fd 48 2b 6a 0d   vmovntpd ZMMWORD PTR [rdx+0x340],zmm29
1b8: 62 61 fd 48 2b 72 12   vmovntpd ZMMWORD PTR [rdx+0x480],zmm30
1bf: 62 61 fd 48 2b 7a 13   vmovntpd ZMMWORD PTR [rdx+0x4c0],zmm31
1c6: 62 01 1d 40 ef e4     vpxord zmm28,zmm28,zmm28
1cc: 62 01 15 40 ef ed     vpxord zmm29,zmm29,zmm29
1d2: 62 01 0d 40 ef f6     vpxord zmm30,zmm30,zmm30
1d8: 62 01 05 40 ef ff     vpxord zmm31,zmm31,zmm31
1de: 62 f2 fd 48 19 07     vbroadcastsd zmm0,QWORD PTR [rdi]
1e4: 62 62 fd 48 b8 26     vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi]
1ea: 0f 18 5e 40           prefetcht2 BYTE PTR [rsi+0x40]
1ee: 62 62 fd 48 b8 6e 01   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x40]
1f5: 0f 18 9e 80 00 00 00   prefetcht2 BYTE PTR [rsi+0x80]
1fc: 62 f2 fd 48 19 47 01   vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
203: 62 62 fd 48 b8 66 06   vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x180]
20a: 0f 18 9e c0 01 00 00   prefetcht2 BYTE PTR [rsi+0x1c0]
211: 62 62 fd 48 b8 6e 07   vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x1c0]
218: 0f 18 9e 00 02 00 00   prefetcht2 BYTE PTR [rsi+0x200]
21f: 62 f2 fd 48 19 0f     vbroadcastsd zmm1,QWORD PTR [rdi]
225: 62 62 f5 48 bc 76 06   vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x180]
22c: 0f 18 9e c0 01 00 00   prefetcht2 BYTE PTR [rsi+0x1c0]

```

```

233: 62 62 f5 48 bc 7e 07 vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x1c0]
23a: 0f 18 9e 00 02 00 00 prefetcht2 BYTE PTR [rsi+0x200]
241: 62 f2 fd 48 19 47 02 vbroadcastsd zmm0,QWORD PTR [rdi+0x10]
248: 62 62 fd 48 b8 66 0c vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x300]
24f: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
256: 62 62 fd 48 b8 6e 0d vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x340]
25d: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
264: 62 f2 fd 48 19 4f 01 vbroadcastsd zmm1,QWORD PTR [rdi+0x8]
26b: 62 62 f5 48 bc 76 0c vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x300]
272: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
279: 62 62 f5 48 bc 7e 0d vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x340]
280: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
287: 62 f2 fd 48 19 47 03 vbroadcastsd zmm0,QWORD PTR [rdi+0x18]
28e: 62 62 fd 48 b8 66 12 vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x480]
295: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
29c: 62 62 fd 48 b8 6e 13 vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x4c0]
2a3: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
2aa: 62 f2 fd 48 19 4f 02 vbroadcastsd zmm1,QWORD PTR [rdi+0x10]
2b1: 62 62 f5 48 bc 76 12 vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
2b8: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
2bf: 62 62 f5 48 bc 7e 13 vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
2c6: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
2cd: 62 61 fd 48 2b 62 18 vmovntpd ZMMWORD PTR [rdx+0x600],zmm28
2d4: 62 61 fd 48 2b 6a 19 vmovntpd ZMMWORD PTR [rdx+0x640],zmm29
2db: 62 61 fd 48 2b 72 1e vmovntpd ZMMWORD PTR [rdx+0x780],zmm30
2e2: 62 61 fd 48 2b 7a 1f vmovntpd ZMMWORD PTR [rdx+0x7c0],zmm31
2e9: 62 01 1d 40 ef e4 vpxord zmm28,zmm28,zmm28
2ef: 62 01 15 40 ef ed vpxord zmm29,zmm29,zmm29
2f5: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
2fb: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
301: 62 f2 fd 48 19 07 vbroadcastsd zmm0,QWORD PTR [rdi]
307: 62 62 fd 48 b8 66 0c vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x300]
30e: 0f 18 9e 40 03 00 00 prefetcht2 BYTE PTR [rsi+0x340]
315: 62 62 fd 48 b8 6e 0d vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x340]
31c: 0f 18 9e 80 03 00 00 prefetcht2 BYTE PTR [rsi+0x380]
323: 62 f2 fd 48 19 47 01 vbroadcastsd zmm0,QWORD PTR [rdi+0x8]
32a: 62 62 fd 48 b8 66 12 vfmadd231pd zmm28,zmm0,ZMMWORD PTR [rsi+0x480]
331: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
338: 62 62 fd 48 b8 6e 13 vfmadd231pd zmm29,zmm0,ZMMWORD PTR [rsi+0x4c0]
33f: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
346: 62 f2 fd 48 19 0f vbroadcastsd zmm1,QWORD PTR [rdi]
34c: 62 62 f5 48 bc 76 12 vfmadd231pd zmm30,zmm1,ZMMWORD PTR [rsi+0x480]
353: 0f 18 9e c0 04 00 00 prefetcht2 BYTE PTR [rsi+0x4c0]
35a: 62 62 f5 48 bc 7e 13 vfmadd231pd zmm31,zmm1,ZMMWORD PTR [rsi+0x4c0]
361: 0f 18 9e 00 05 00 00 prefetcht2 BYTE PTR [rsi+0x500]
368: 62 61 fd 48 2b 62 24 vmovntpd ZMMWORD PTR [rdx+0x900],zmm28
36f: 62 61 fd 48 2b 6a 25 vmovntpd ZMMWORD PTR [rdx+0x940],zmm29
376: 62 61 fd 48 2b 72 2a vmovntpd ZMMWORD PTR [rdx+0xa80],zmm30
37d: 62 61 fd 48 2b 7a 2b vmovntpd ZMMWORD PTR [rdx+0xac0],zmm31
384: 41 5f pop r15
386: 41 5e pop r14
388: 41 5d pop r13
38a: 41 5c pop r12
38c: 5b pop rbx
38d: c3 ret

```

B.4 Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory and Caching B Strides in Vector Registers

No Multiple Accumulation

libxsmm_skx_f64_nn_8x8x4_0_48_48_a1_b0_p0.sreg: file format binary

Disassembly of section .data:

```
00000000 <.data>:
 0: 53                push   rbx
 1: 41 54             push   r12
 3: 41 55             push   r13
 5: 41 56             push   r14
 7: 41 57             push   r15
 9: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
 f: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
15: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
1c: 62 62 fd 58 b8 3f vfmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
22: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
29: 62 62 f5 58 b8 7f 01 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
30: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
37: 62 62 fd 58 b8 7f 02 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x10]{1to8}
3e: 62 62 f5 58 b8 7f 03 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x18]{1to8}
45: 62 61 fd 48 2b 3a vmovntpd ZMMWORD PTR [rdx],zmm31
4b: 62 f1 fd 48 10 46 06 vmovupd zmm0,ZMMWORD PTR [rsi+0x180]
52: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
58: 62 f1 fd 48 10 4e 0c vmovupd zmm1,ZMMWORD PTR [rsi+0x300]
5f: 62 62 fd 58 bc 3f vfnmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
65: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
6c: 62 62 f5 58 bc 7f 01 vfnmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
73: 62 62 fd 58 bc 7f 02 vfnmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x10]{1to8}
7a: 62 61 fd 48 2b 7a 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm31
81: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
88: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
8e: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
95: 62 62 fd 58 b8 3f vfmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
9b: 62 62 f5 58 b8 7f 01 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
a2: 62 61 fd 48 2b 7a 0c vmovntpd ZMMWORD PTR [rdx+0x300],zmm31
a9: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
b0: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
b6: 62 62 fd 58 bc 3f vfnmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
bc: 62 61 fd 48 2b 7a 12 vmovntpd ZMMWORD PTR [rdx+0x480],zmm31
c3: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
c9: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
cf: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
d6: 62 62 fd 58 b8 3f vfmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
dc: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
e3: 62 62 f5 58 b8 7f 01 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
ea: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
f1: 62 62 fd 58 b8 7f 02 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x10]{1to8}
f8: 62 62 f5 58 b8 7f 03 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x18]{1to8}
ff: 62 61 fd 48 2b 7a 18 vmovntpd ZMMWORD PTR [rdx+0x600],zmm31
106: 62 f1 fd 48 10 46 06 vmovupd zmm0,ZMMWORD PTR [rsi+0x180]
10d: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
113: 62 f1 fd 48 10 4e 0c vmovupd zmm1,ZMMWORD PTR [rsi+0x300]
11a: 62 62 fd 58 bc 3f vfnmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
120: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
127: 62 62 f5 58 bc 7f 01 vfnmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
```

```

12e: 62 62 fd 58 bc 7f 02 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x10]{1to8}
135: 62 61 fd 48 2b 7a 1e vmovntpd ZMMWORD PTR [rdx+0x780],zmm31
13c: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
143: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
149: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
150: 62 62 fd 58 b8 3f vfmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
156: 62 62 f5 58 b8 7f 01 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x8]{1to8}
15d: 62 61 fd 48 2b 7a 24 vmovntpd ZMMWORD PTR [rdx+0x900],zmm31
164: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
16b: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
171: 62 62 fd 58 bc 3f vfmadd231pd zmm31,zmm0,QWORD PTR [rdi]{1to8}
177: 62 61 fd 48 2b 7a 2a vmovntpd ZMMWORD PTR [rdx+0xa80],zmm31
17e: 41 5f pop r15
180: 41 5e pop r14
182: 41 5d pop r13
184: 41 5c pop r12
186: 5b pop rbx
187: c3 ret

```

N Blocking of 2

```
libxsmm_skx_f64_nn_8x16x4_0_48_48_a1_b0_p0.sreg: file format binary
```

Disassembly of section .data:

```

00000000 <.data>:
0: 53 push rbx
1: 41 54 push r12
3: 41 55 push r13
5: 41 56 push r14
7: 41 57 push r15
9: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
f: 62 f1 fd 48 10 56 01 vmovupd zmm2,ZMMWORD PTR [rsi+0x40]
16: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
1c: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
22: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
29: 62 f1 fd 48 10 5e 07 vmovupd zmm3,ZMMWORD PTR [rsi+0x1c0]
30: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
36: 62 62 ed 58 b8 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
3c: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
43: 62 f1 fd 48 10 56 0d vmovupd zmm2,ZMMWORD PTR [rsi+0x340]
4a: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
51: 62 62 e5 58 b8 77 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}
58: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
5f: 62 f1 fd 48 10 5e 13 vmovupd zmm3,ZMMWORD PTR [rsi+0x4c0]
66: 62 62 fd 58 b8 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}
6d: 62 62 ed 58 b8 7f 02 vfmadd231pd zmm31,zmm2,QWORD PTR [rdi+0x10]{1to8}
74: 62 62 f5 58 b8 77 03 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x18]{1to8}
7b: 62 62 e5 58 b8 7f 03 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x18]{1to8}
82: 62 61 fd 48 2b 32 vmovntpd ZMMWORD PTR [rdx],zmm30
88: 62 61 fd 48 2b 7a 01 vmovntpd ZMMWORD PTR [rdx+0x40],zmm31
8f: 62 f1 fd 48 10 46 06 vmovupd zmm0,ZMMWORD PTR [rsi+0x180]
96: 62 f1 fd 48 10 56 07 vmovupd zmm2,ZMMWORD PTR [rsi+0x1c0]
9d: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
a3: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
a9: 62 f1 fd 48 10 4e 0c vmovupd zmm1,ZMMWORD PTR [rsi+0x300]
b0: 62 f1 fd 48 10 5e 0d vmovupd zmm3,ZMMWORD PTR [rsi+0x340]
b7: 62 62 fd 58 bc 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
bd: 62 62 ed 58 bc 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
c3: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
ca: 62 f1 fd 48 10 56 13 vmovupd zmm2,ZMMWORD PTR [rsi+0x4c0]
d1: 62 62 f5 58 bc 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
d8: 62 62 e5 58 bc 7f 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}
df: 62 62 fd 58 bc 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}

```



```

e6: 62 62 ed 58 bc 7f 02 vfmadd231pd zmm31,zmm2,QWORD PTR [rdi+0x10]{1to8}
ed: 62 61 fd 48 2b 72 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm30
f4: 62 61 fd 48 2b 7a 07 vmovntpd ZMMWORD PTR [rdx+0x1c0],zmm31
fb: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
102: 62 f1 fd 48 10 56 0d vmovupd zmm2,ZMMWORD PTR [rsi+0x340]
109: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
10f: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
115: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
11c: 62 f1 fd 48 10 5e 13 vmovupd zmm3,ZMMWORD PTR [rsi+0x4c0]
123: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
129: 62 62 ed 58 b8 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
12f: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
136: 62 62 e5 58 b8 7f 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}
13d: 62 61 fd 48 2b 72 0c vmovntpd ZMMWORD PTR [rdx+0x300],zmm30
144: 62 61 fd 48 2b 7a 0d vmovntpd ZMMWORD PTR [rdx+0x340],zmm31
14b: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
152: 62 f1 fd 48 10 56 13 vmovupd zmm2,ZMMWORD PTR [rsi+0x4c0]
159: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
15f: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
165: 62 62 fd 58 bc 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
16b: 62 62 ed 58 bc 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
171: 62 61 fd 48 2b 72 12 vmovntpd ZMMWORD PTR [rdx+0x480],zmm30
178: 62 61 fd 48 2b 7a 13 vmovntpd ZMMWORD PTR [rdx+0x4c0],zmm31
17f: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
185: 62 f1 fd 48 10 56 01 vmovupd zmm2,ZMMWORD PTR [rsi+0x40]
18c: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
192: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
198: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
19f: 62 f1 fd 48 10 5e 07 vmovupd zmm3,ZMMWORD PTR [rsi+0x1c0]
1a6: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
1ac: 62 62 ed 58 b8 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
1b2: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
1b9: 62 f1 fd 48 10 56 0d vmovupd zmm2,ZMMWORD PTR [rsi+0x340]
1c0: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
1c7: 62 62 e5 58 b8 7f 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}
1ce: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
1d5: 62 f1 fd 48 10 5e 13 vmovupd zmm3,ZMMWORD PTR [rsi+0x4c0]
1dc: 62 62 fd 58 b8 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}
1e3: 62 62 ed 58 b8 7f 02 vfmadd231pd zmm31,zmm2,QWORD PTR [rdi+0x10]{1to8}
1ea: 62 62 f5 58 b8 77 03 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x18]{1to8}
1f1: 62 62 e5 58 b8 7f 03 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x18]{1to8}
1f8: 62 61 fd 48 2b 72 18 vmovntpd ZMMWORD PTR [rdx+0x600],zmm30
1ff: 62 61 fd 48 2b 7a 19 vmovntpd ZMMWORD PTR [rdx+0x640],zmm31
206: 62 f1 fd 48 10 46 06 vmovupd zmm0,ZMMWORD PTR [rsi+0x180]
20d: 62 f1 fd 48 10 56 07 vmovupd zmm2,ZMMWORD PTR [rsi+0x1c0]
214: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
21a: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
220: 62 f1 fd 48 10 4e 0c vmovupd zmm1,ZMMWORD PTR [rsi+0x300]
227: 62 f1 fd 48 10 5e 0d vmovupd zmm3,ZMMWORD PTR [rsi+0x340]
22e: 62 62 fd 58 bc 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
234: 62 62 ed 58 bc 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
23a: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
241: 62 f1 fd 48 10 56 13 vmovupd zmm2,ZMMWORD PTR [rsi+0x4c0]
248: 62 62 f5 58 bc 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
24f: 62 62 e5 58 bc 7f 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}
256: 62 62 fd 58 bc 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}
25d: 62 62 ed 58 bc 7f 02 vfmadd231pd zmm31,zmm2,QWORD PTR [rdi+0x10]{1to8}
264: 62 61 fd 48 2b 72 1e vmovntpd ZMMWORD PTR [rdx+0x780],zmm30
26b: 62 61 fd 48 2b 7a 1f vmovntpd ZMMWORD PTR [rdx+0x7c0],zmm31
272: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
279: 62 f1 fd 48 10 56 0d vmovupd zmm2,ZMMWORD PTR [rsi+0x340]
280: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
286: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
28c: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
293: 62 f1 fd 48 10 5e 13 vmovupd zmm3,ZMMWORD PTR [rsi+0x4c0]
29a: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
2a0: 62 62 ed 58 b8 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
2a6: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
2ad: 62 62 e5 58 b8 7f 01 vfmadd231pd zmm31,zmm3,QWORD PTR [rdi+0x8]{1to8}

```

```

2b4: 62 61 fd 48 2b 72 24 vmovntpd ZMMWORD PTR [rdx+0x900],zmm30
2bb: 62 61 fd 48 2b 7a 25 vmovntpd ZMMWORD PTR [rdx+0x940],zmm31
2c2: 62 f1 fd 48 10 46 12 vmovupd zmm0,ZMMWORD PTR [rsi+0x480]
2c9: 62 f1 fd 48 10 56 13 vmovupd zmm2,ZMMWORD PTR [rsi+0x4c0]
2d0: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
2d6: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
2dc: 62 62 fd 58 bc 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
2e2: 62 62 ed 58 bc 3f vfmadd231pd zmm31,zmm2,QWORD PTR [rdi]{1to8}
2e8: 62 61 fd 48 2b 72 2a vmovntpd ZMMWORD PTR [rdx+0xa80],zmm30
2ef: 62 61 fd 48 2b 7a 2b vmovntpd ZMMWORD PTR [rdx+0xac0],zmm31
2f6: 41 5f pop r15
2f8: 41 5e pop r14
2fa: 41 5d pop r13
2fc: 41 5c pop r12
2fe: 5b pop rbx
2ff: c3 ret

```

M Blocking of 2

```
libxsmm_skx_f64_nn_8x8x4_0_48_48_a1_b0_p0.sreg: file format binary
```

```
Disassembly of section .data:
```

```
00000000 <.data>:
```

```

0: 53 push rbx
1: 41 54 push r12
3: 41 55 push r13
5: 41 56 push r14
7: 41 57 push r15
9: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
f: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
15: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
1b: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
22: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
28: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
2f: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
36: 62 62 f5 58 bc 3f vfmadd231pd zmm31,zmm1,QWORD PTR [rdi]{1to8}
3c: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
43: 62 62 fd 58 b8 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}
4a: 62 62 fd 58 bc 7f 01 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x8]{1to8}
51: 62 62 f5 58 b8 77 03 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x18]{1to8}
58: 62 62 f5 58 bc 7f 02 vfmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x10]{1to8}
5f: 62 61 fd 48 2b 32 vmovntpd ZMMWORD PTR [rdx],zmm30
65: 62 61 fd 48 2b 7a 06 vmovntpd ZMMWORD PTR [rdx+0x180],zmm31
6c: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
73: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
79: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
7f: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
86: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
8c: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
93: 62 62 f5 58 bc 3f vfmadd231pd zmm31,zmm1,QWORD PTR [rdi]{1to8}
99: 62 61 fd 48 2b 72 0c vmovntpd ZMMWORD PTR [rdx+0x300],zmm30
a0: 62 61 fd 48 2b 7a 12 vmovntpd ZMMWORD PTR [rdx+0x480],zmm31
a7: 62 f1 fd 48 10 06 vmovupd zmm0,ZMMWORD PTR [rsi]
ad: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
b3: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
b9: 62 f1 fd 48 10 4e 06 vmovupd zmm1,ZMMWORD PTR [rsi+0x180]
c0: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
c6: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
cd: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
d4: 62 62 f5 58 bc 3f vfmadd231pd zmm31,zmm1,QWORD PTR [rdi]{1to8}
da: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
e1: 62 62 fd 58 b8 77 02 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi+0x10]{1to8}
e8: 62 62 fd 58 bc 7f 01 vfmadd231pd zmm31,zmm0,QWORD PTR [rdi+0x8]{1to8}

```

```

ef: 62 62 f5 58 b8 77 03 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x18]{1to8}
f6: 62 62 f5 58 bc 7f 02 vfnmadd231pd zmm31,zmm1,QWORD PTR [rdi+0x10]{1to8}
fd: 62 61 fd 48 2b 72 18 vmovntpd ZMMWORD PTR [rdx+0x600],zmm30
104: 62 61 fd 48 2b 7a 1e vmovntpd ZMMWORD PTR [rdx+0x780],zmm31
10b: 62 f1 fd 48 10 46 0c vmovupd zmm0,ZMMWORD PTR [rsi+0x300]
112: 62 01 0d 40 ef f6 vpxord zmm30,zmm30,zmm30
118: 62 01 05 40 ef ff vpxord zmm31,zmm31,zmm31
11e: 62 f1 fd 48 10 4e 12 vmovupd zmm1,ZMMWORD PTR [rsi+0x480]
125: 62 62 fd 58 b8 37 vfmadd231pd zmm30,zmm0,QWORD PTR [rdi]{1to8}
12b: 62 62 f5 58 b8 77 01 vfmadd231pd zmm30,zmm1,QWORD PTR [rdi+0x8]{1to8}
132: 62 62 f5 58 bc 3f vfnmadd231pd zmm31,zmm1,QWORD PTR [rdi]{1to8}
138: 62 61 fd 48 2b 72 24 vmovntpd ZMMWORD PTR [rdx+0x900],zmm30
13f: 62 61 fd 48 2b 7a 2a vmovntpd ZMMWORD PTR [rdx+0xa80],zmm31
146: 41 5f pop r15
148: 41 5e pop r14
14a: 41 5d pop r13
14c: 41 5c pop r12
14e: 5b pop rbx
14f: c3 ret

```

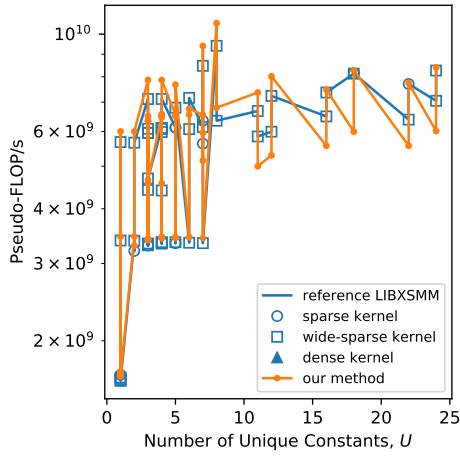
Appendix C

Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed \mathbf{A} Constants from Memory

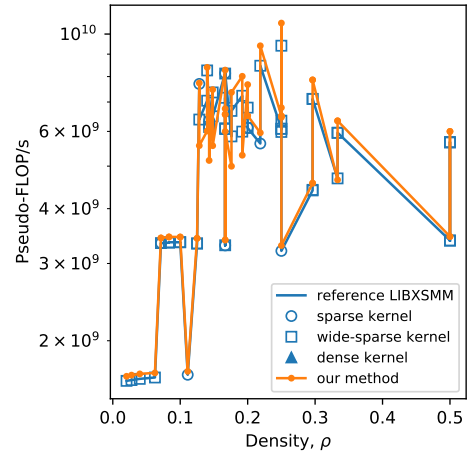
This appendix presents the complete benchmark results for our GEMM method which runtime broadcasts packed \mathbf{A} constants from memory. As explained in Chapter 4, two sets of \mathbf{A} matrices were used for the testing - a set of all PyFR operator matrices, and a set of synthetic matrices. We evaluated our kernel on two testing machines - a Skylake machine (c5n.xlarge) and a Cascade Lake machine (m5n.xlarge). Please refer to Chapter 5 for how the kernel implements GEMM.

C.1 PyFR Operator Matrices

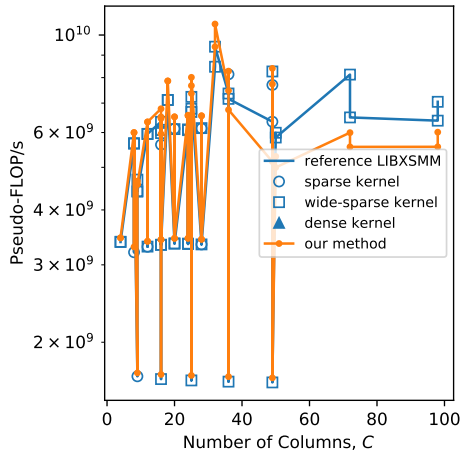
Benchmark Run on c5n.xlarge



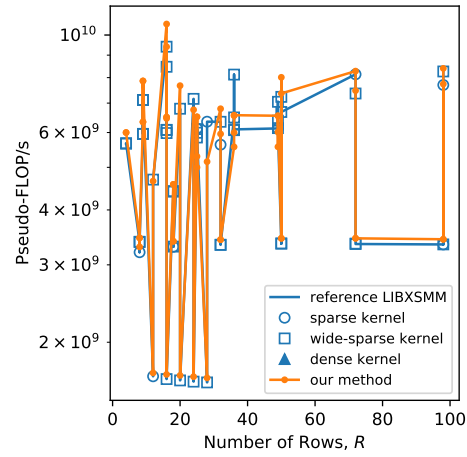
(a) Performance against number of unique \mathbf{A} constants.



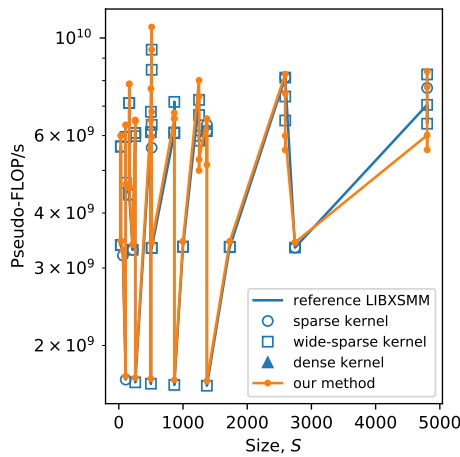
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.1: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on c5n.xlarge machine.

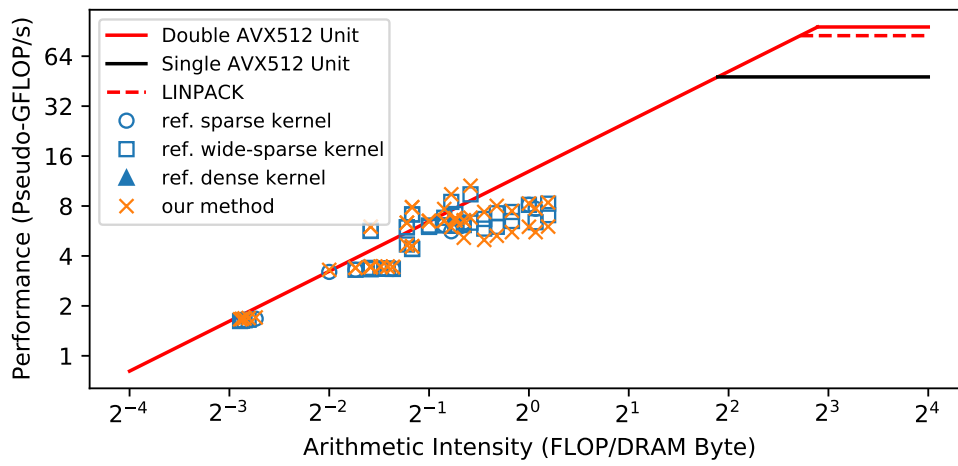
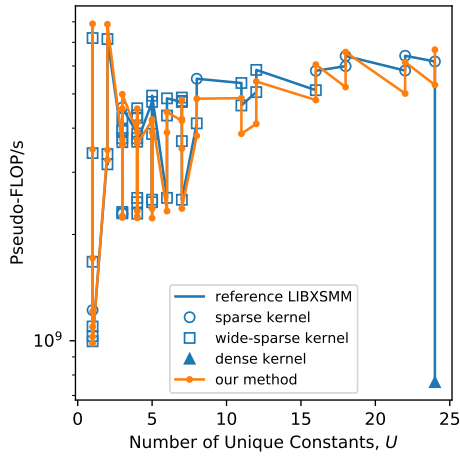
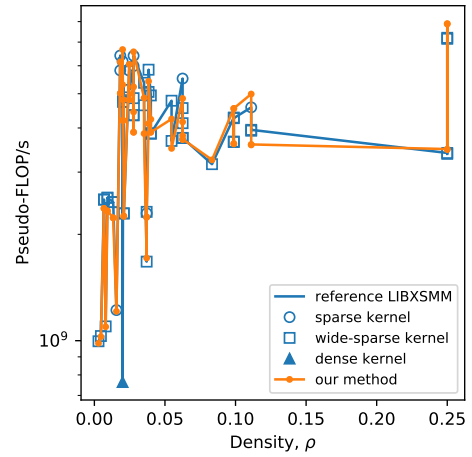


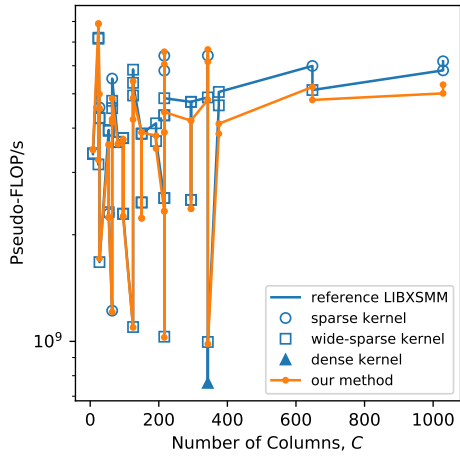
Figure C.2: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



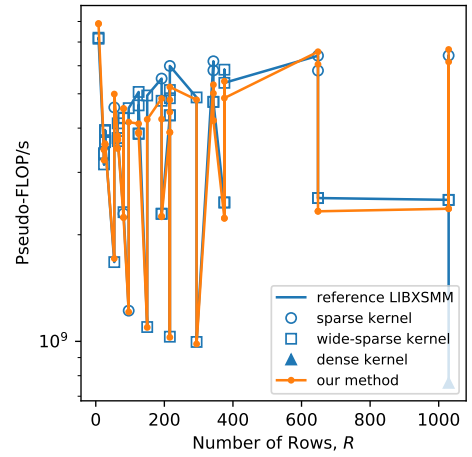
(a) Performance against number of unique \mathbf{A} constants.



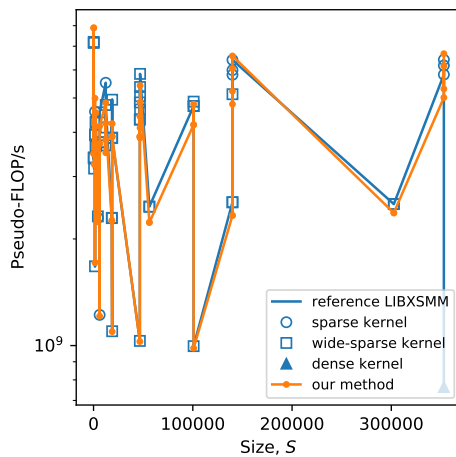
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.3: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on c5n.xlarge machine.

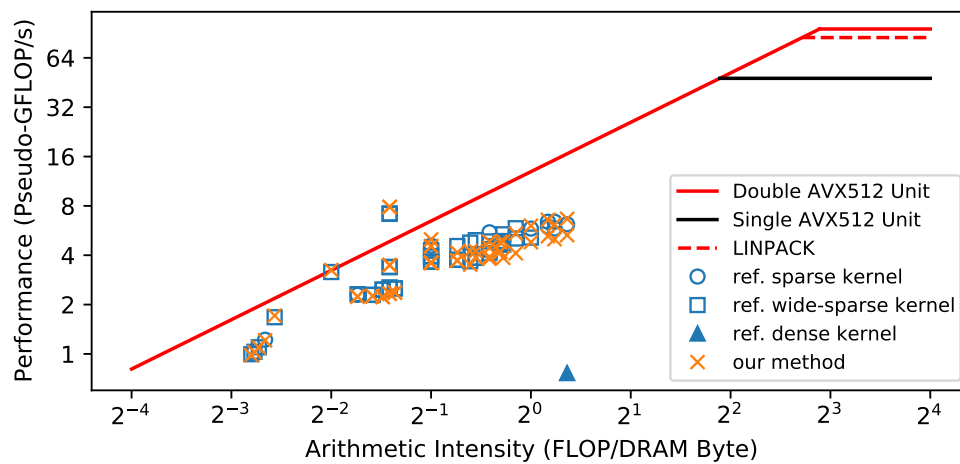
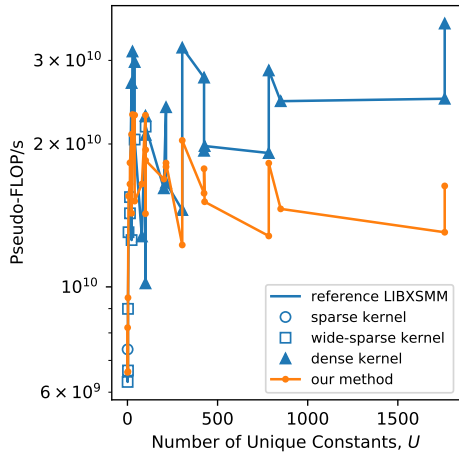
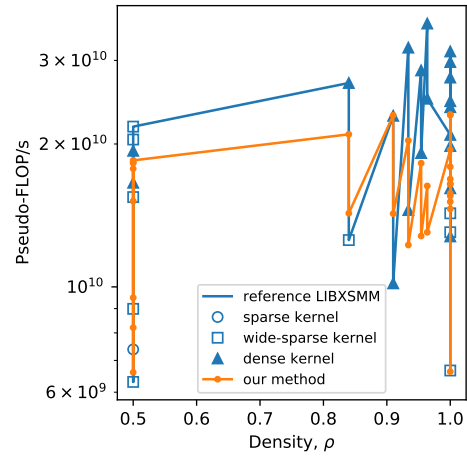


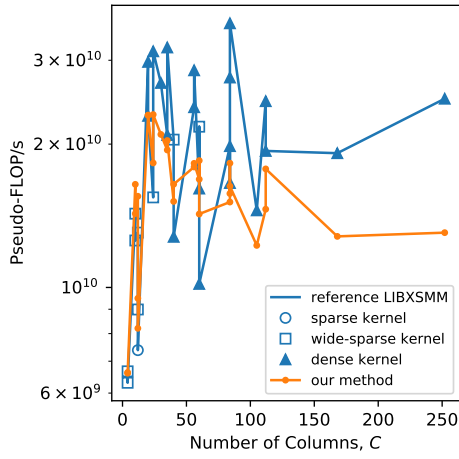
Figure C.4: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



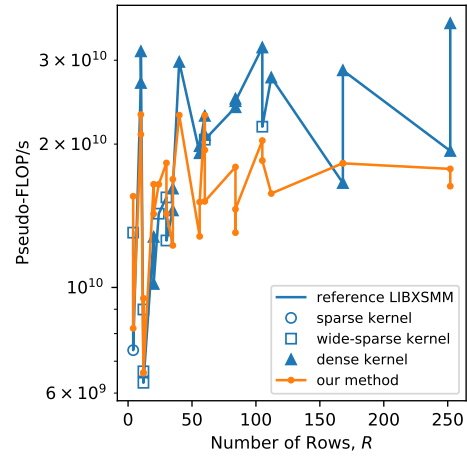
(a) Performance against number of unique \mathbf{A} constants.



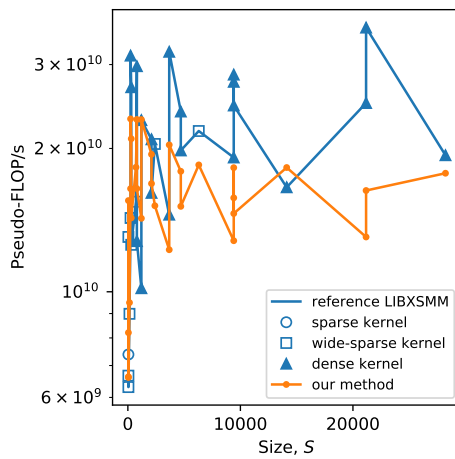
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.5: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on c5n.xlarge machine.

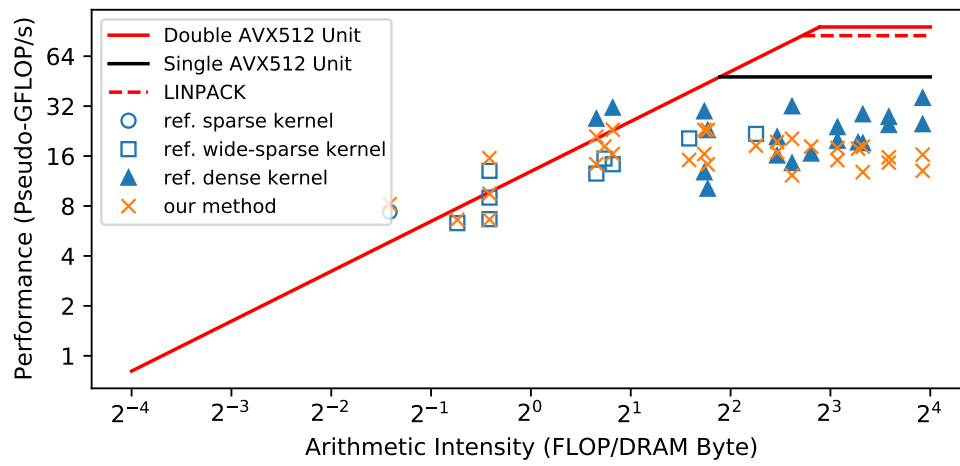
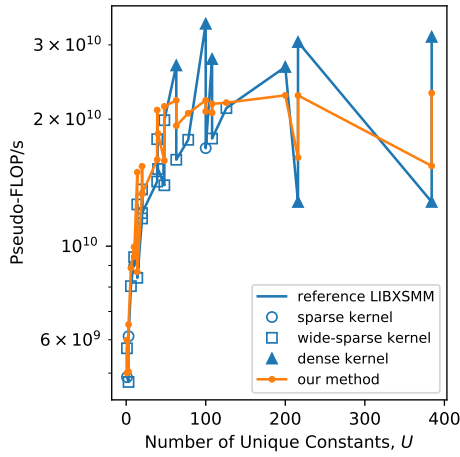
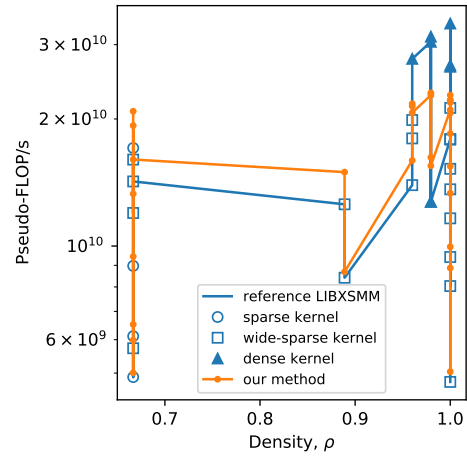


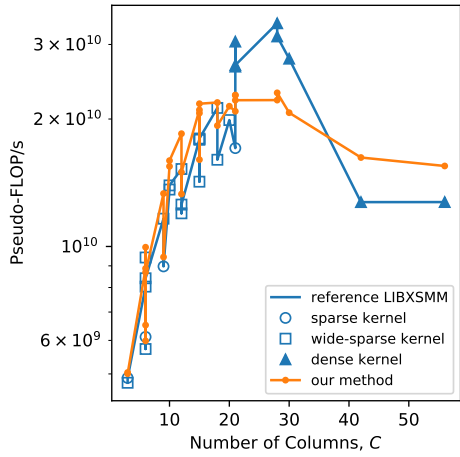
Figure C.6: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



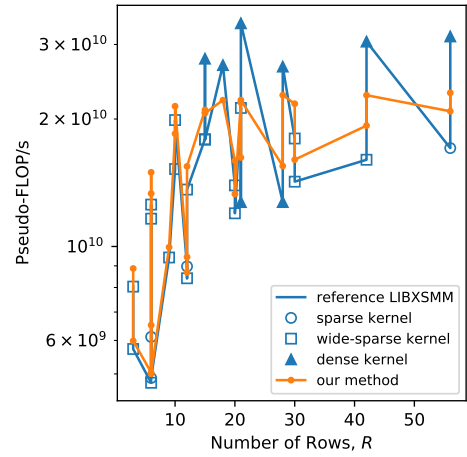
(a) Performance against number of unique \mathbf{A} constants.



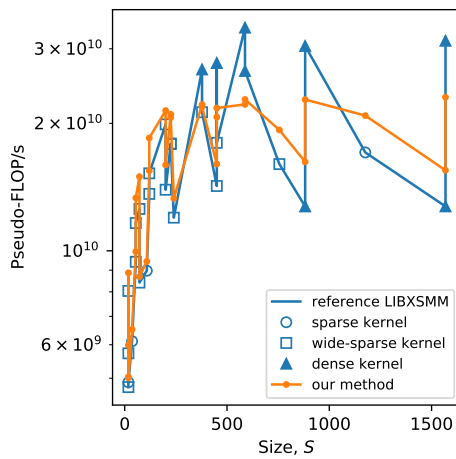
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.7: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on c5n.xlarge machine.

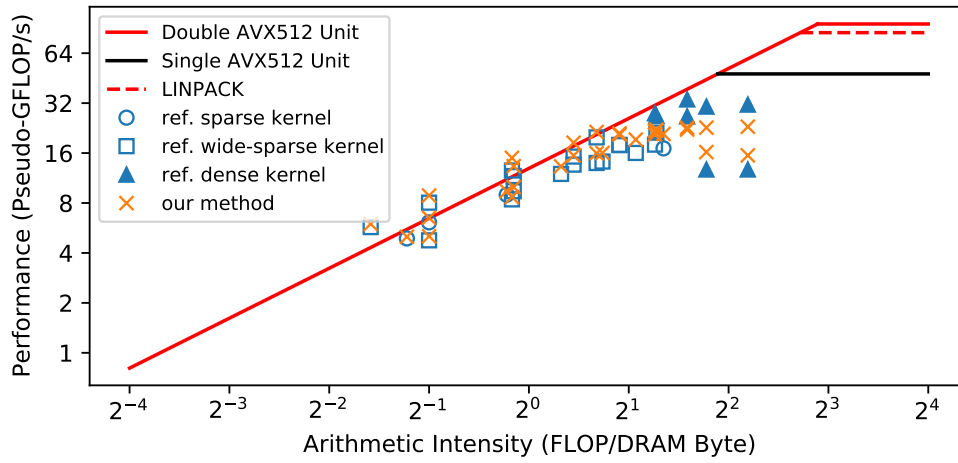
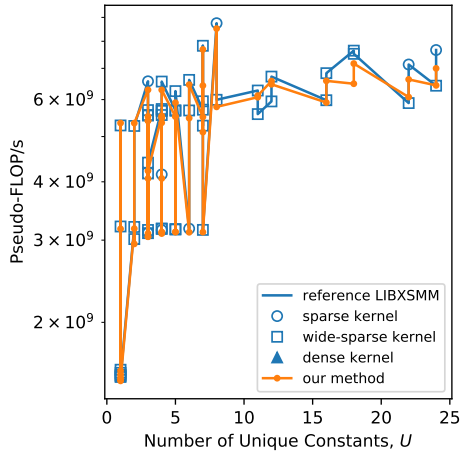
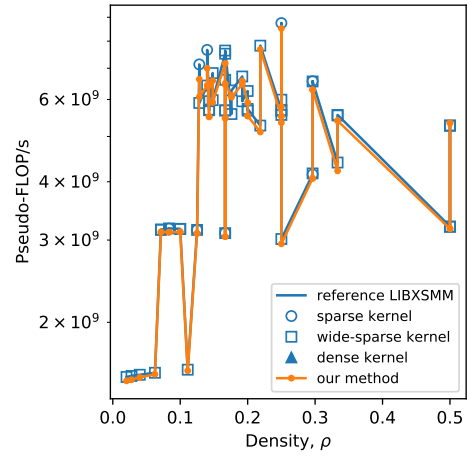


Figure C.8: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.

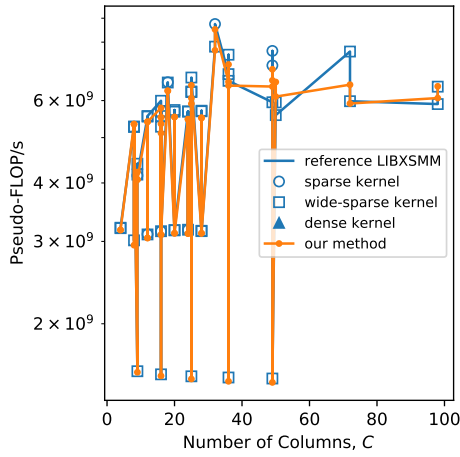
Benchmark Run on m5n.xlarge



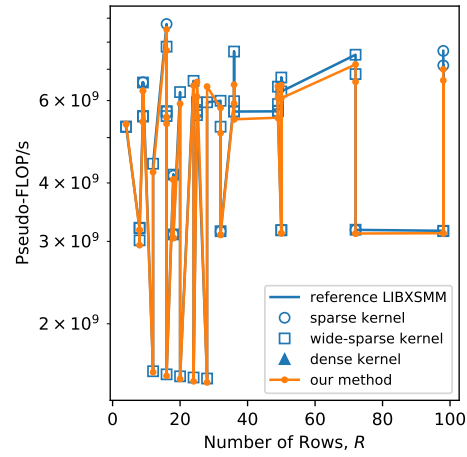
(a) Performance against number of unique \mathbf{A} constants.



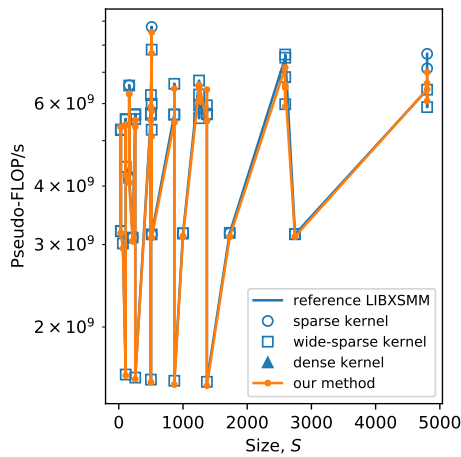
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.9: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on m5n.xlarge machine.

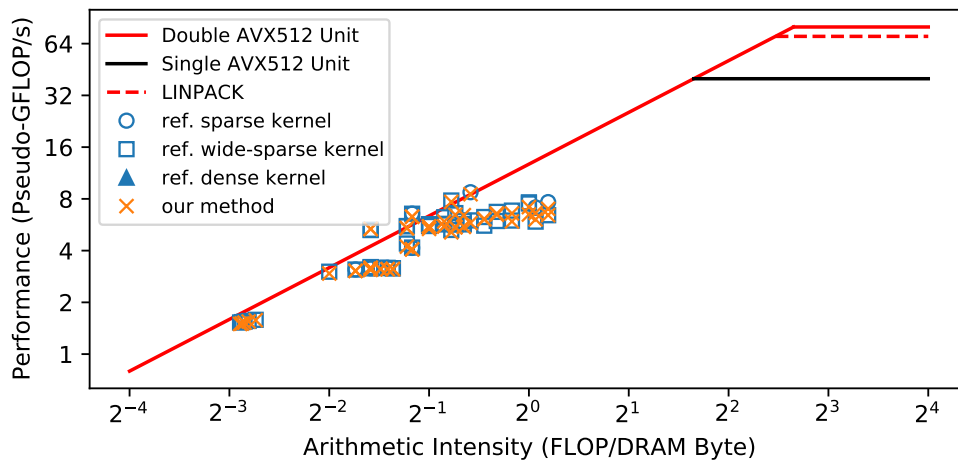
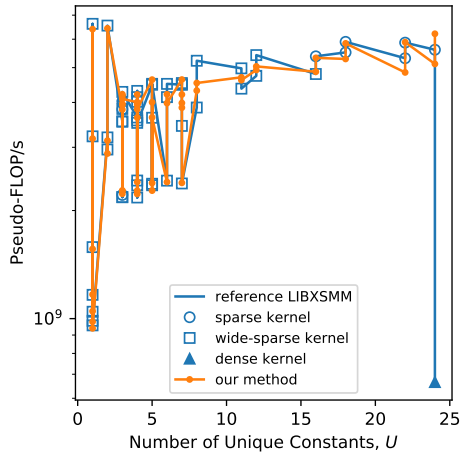
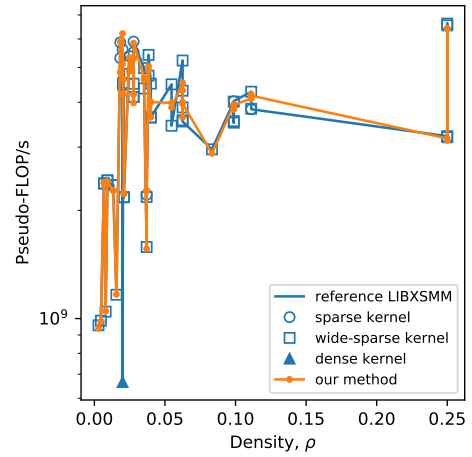


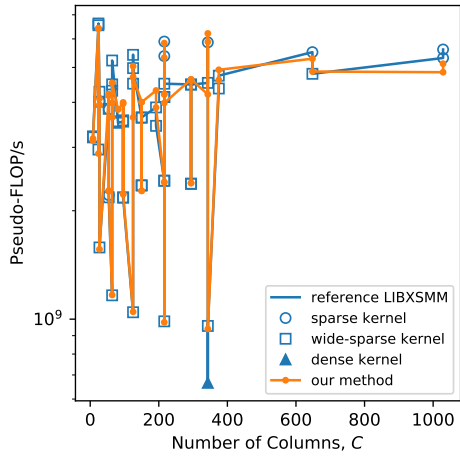
Figure C.10: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



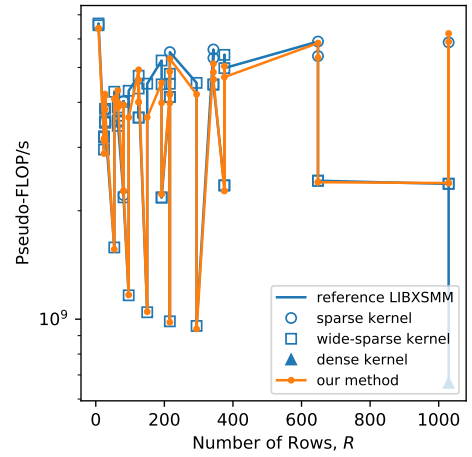
(a) Performance against number of unique \mathbf{A} constants.



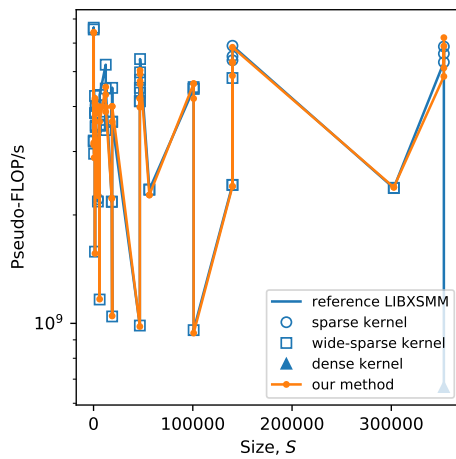
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.11: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on m5n.xlarge machine.

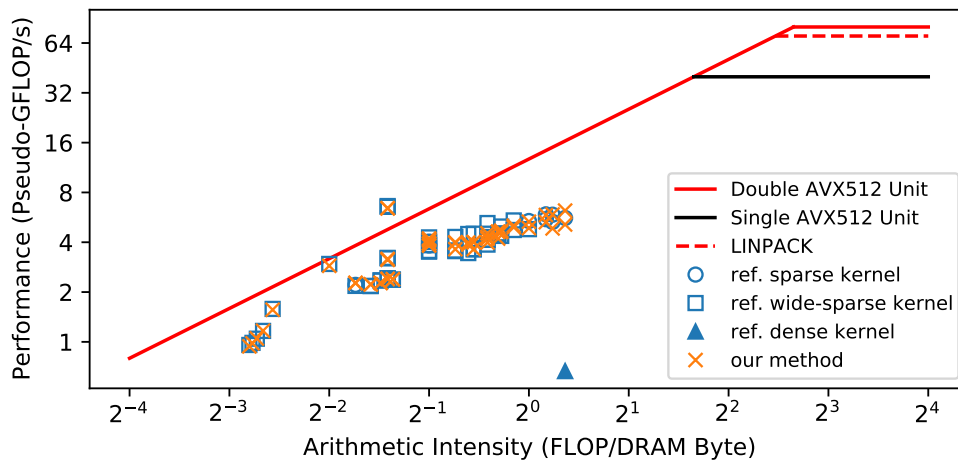
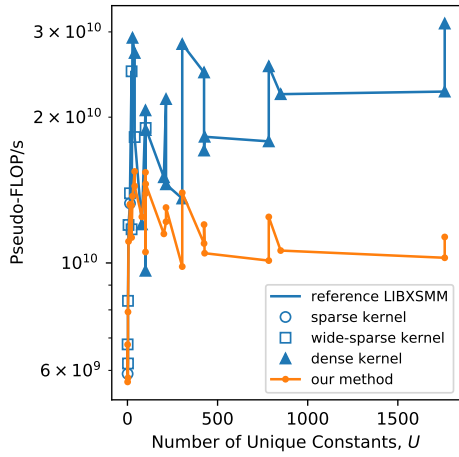
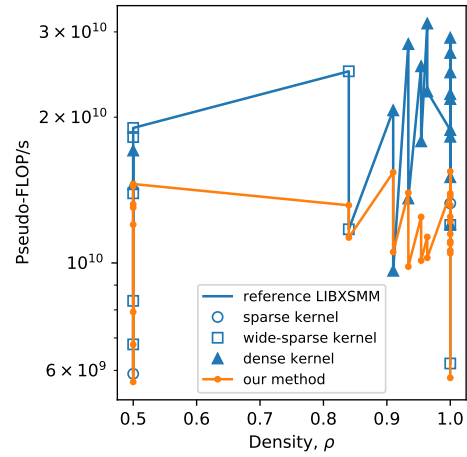


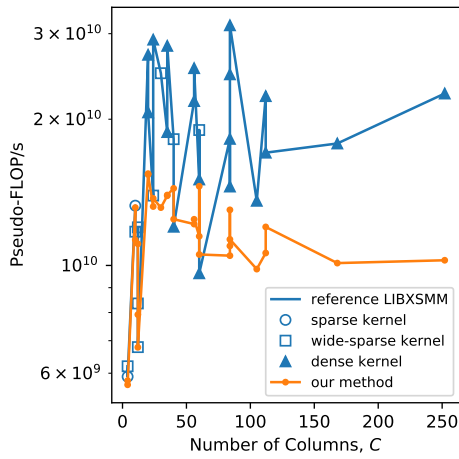
Figure C.12: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



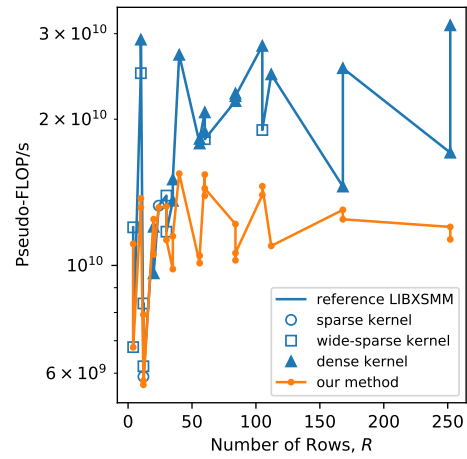
(a) Performance against number of unique \mathbf{A} constants.



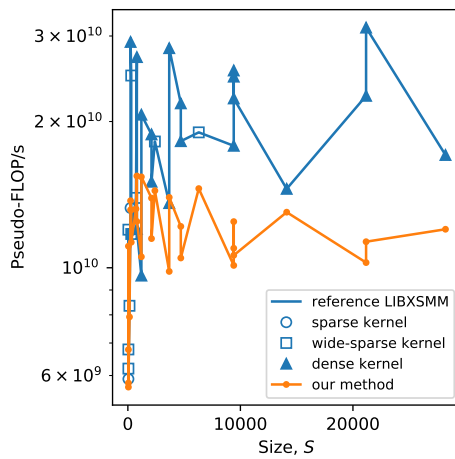
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.13: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on m5n.xlarge machine.

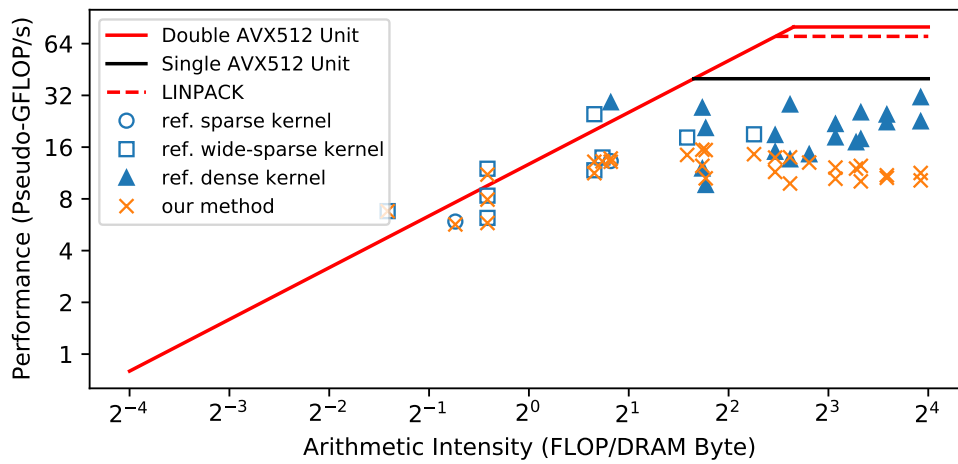
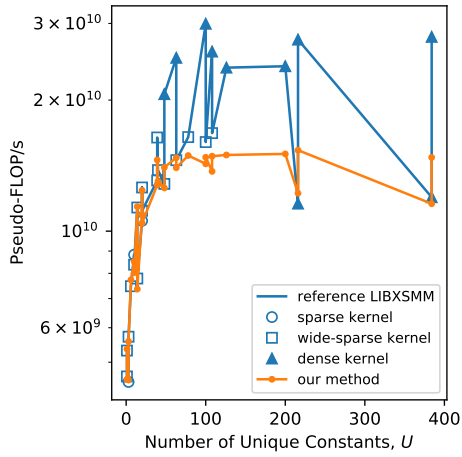
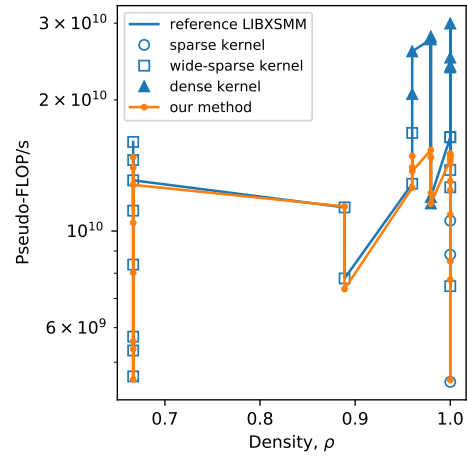


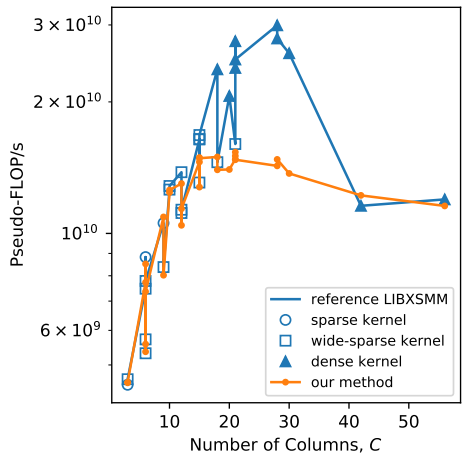
Figure C.14: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



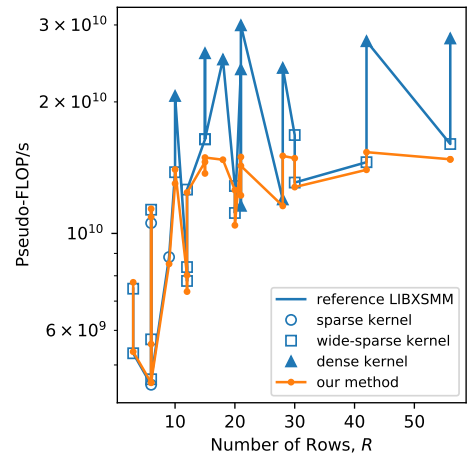
(a) Performance against number of unique \mathbf{A} constants.



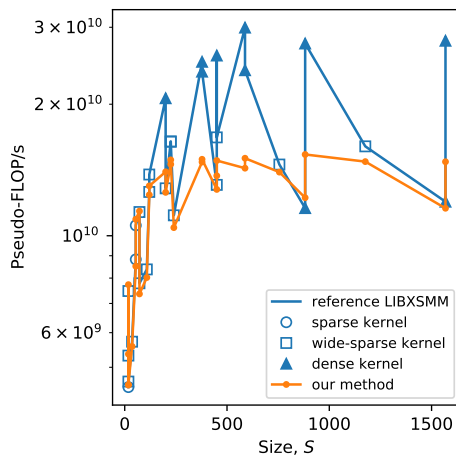
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure C.15: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on m5n.xlarge machine.

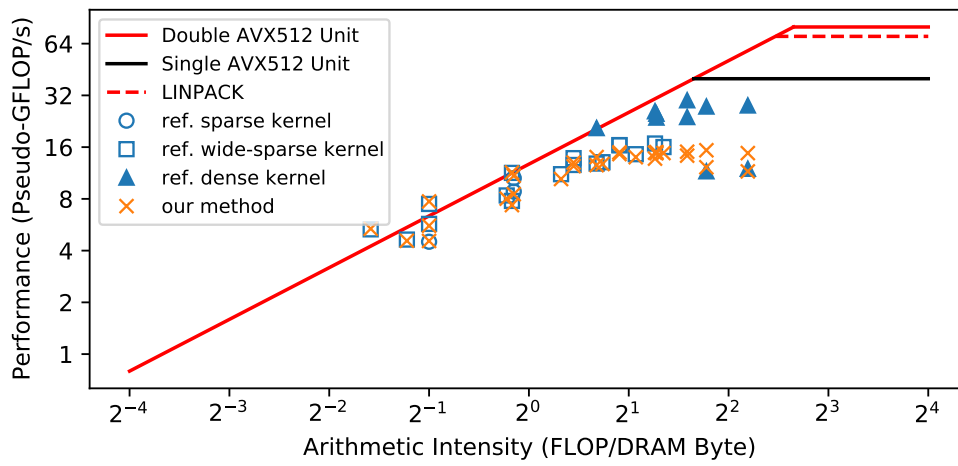
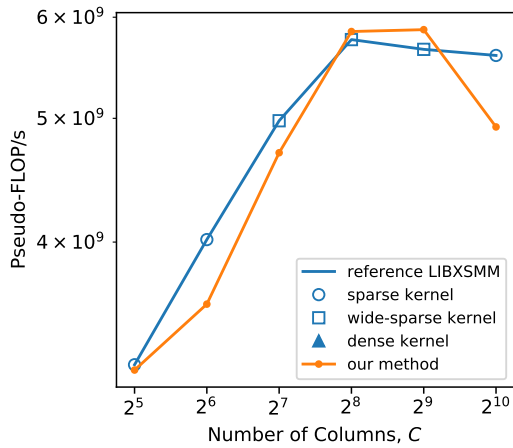


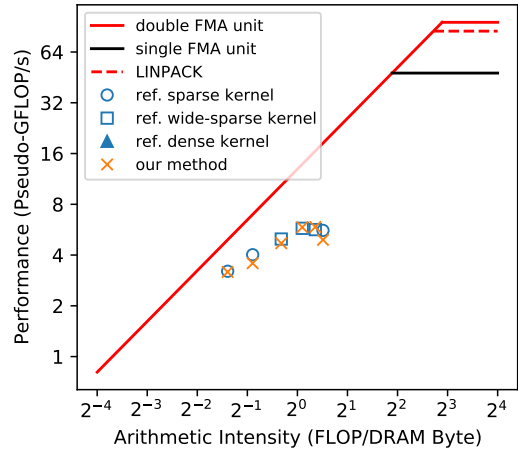
Figure C.16: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.

C.2 Synthetic Matrices

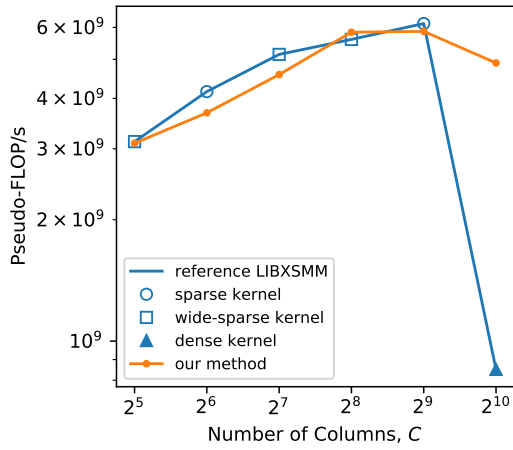
Benchmark Run on c5n.xlarge



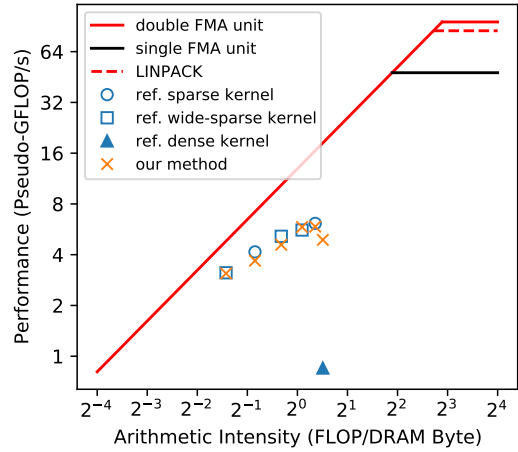
(a) Performance vs. number of columns, $U = 16$.



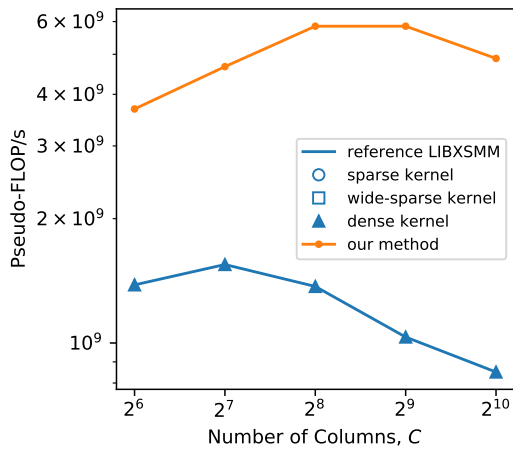
(b) Roofline plot, $U = 16$.



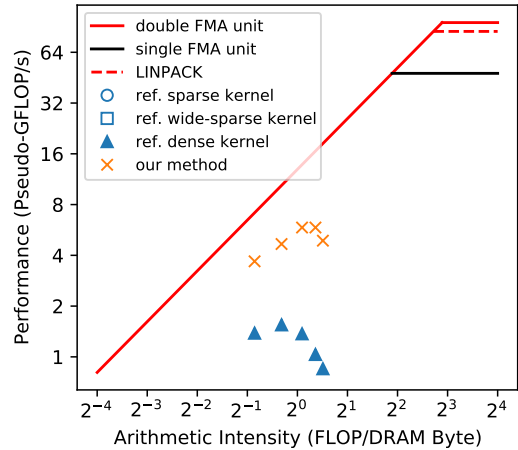
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

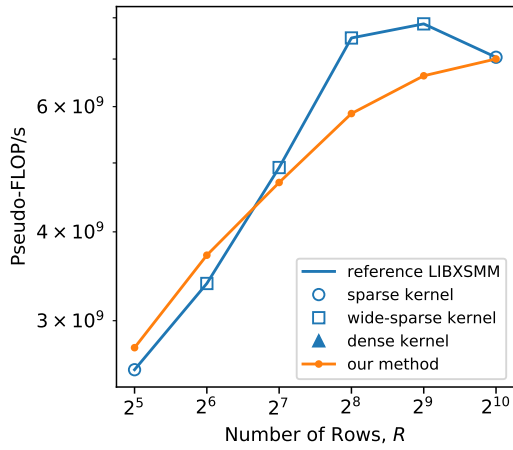


(e) Performance vs. number of columns, $U = 256$.

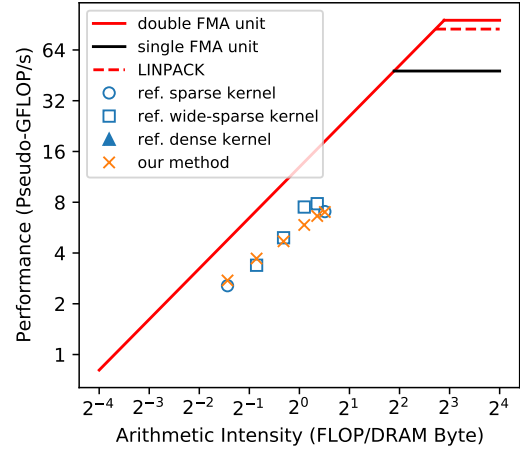


(f) Roofline plot, $U = 256$.

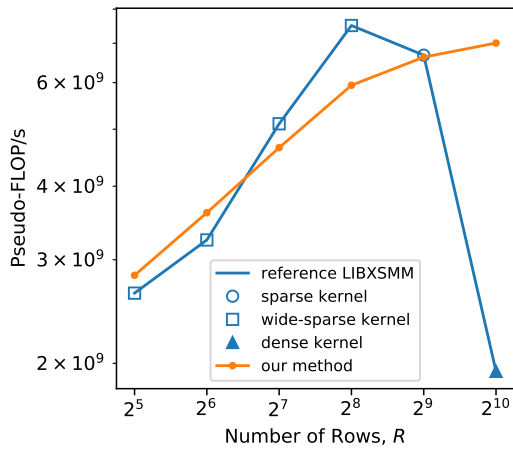
Figure C.17: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Experiment run on c5n.xlarge.



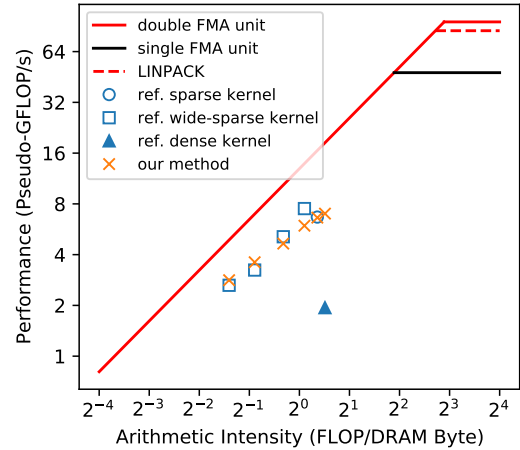
(a) Performance vs. number of rows, $U = 16$.



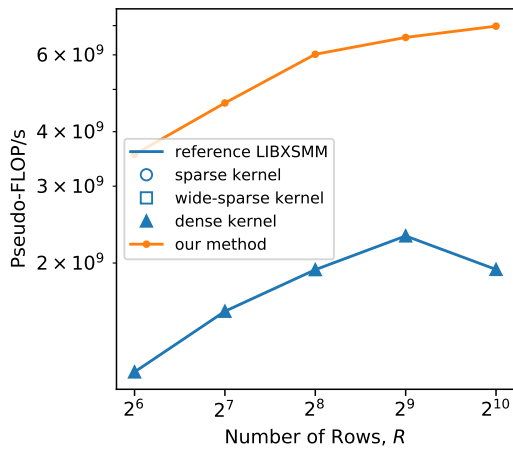
(b) Roofline plot, $U = 16$.



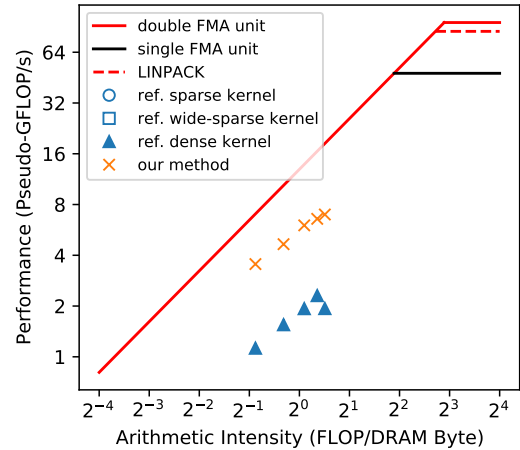
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

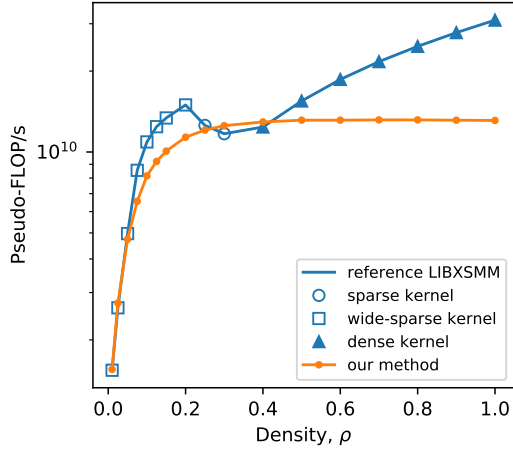


(e) Performance vs. number of rows, $U = 256$.

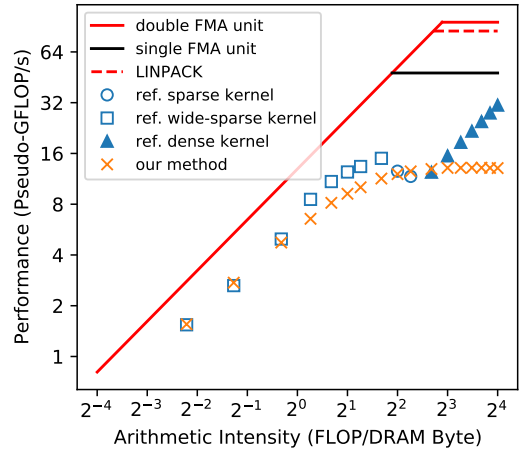


(f) Roofline plot, $U = 256$.

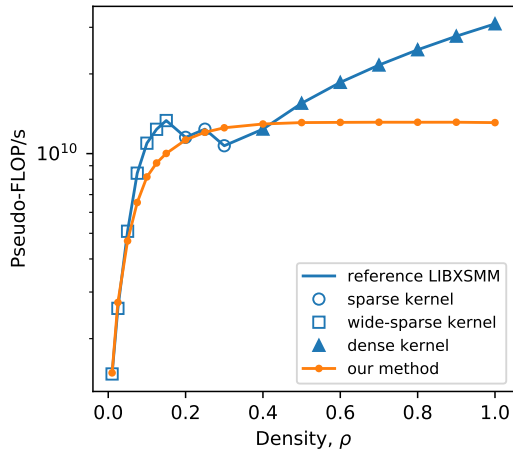
Figure C.18: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Experiment run on c5n.xlarge.



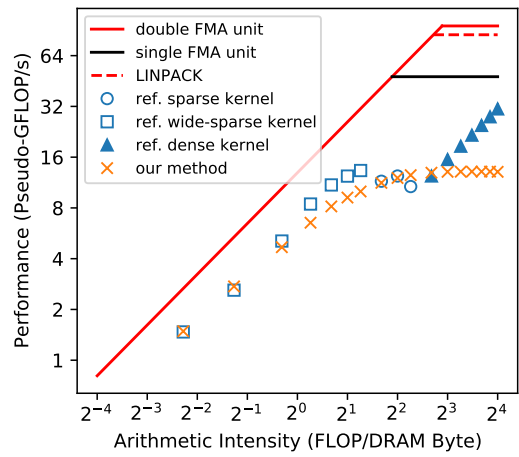
(a) Performance vs. density, $U = 16$.



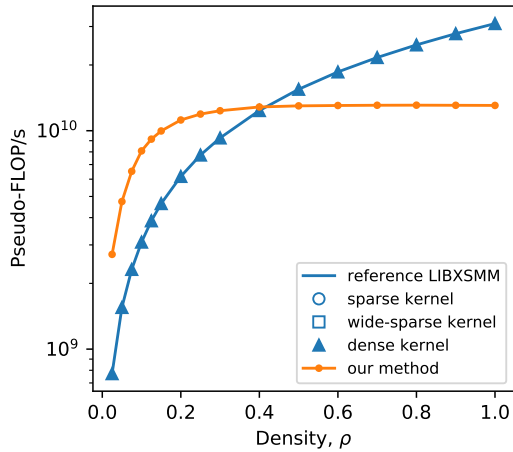
(b) Roofline plot, $U = 16$.



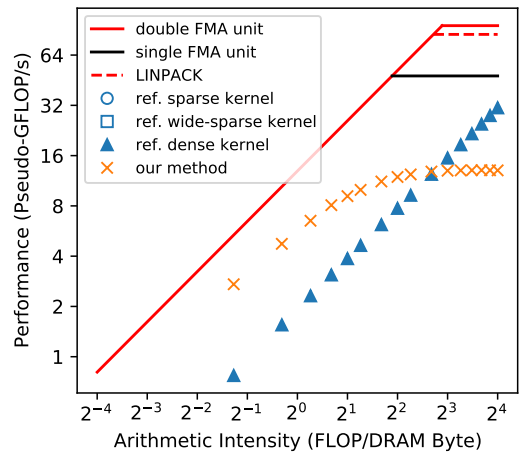
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

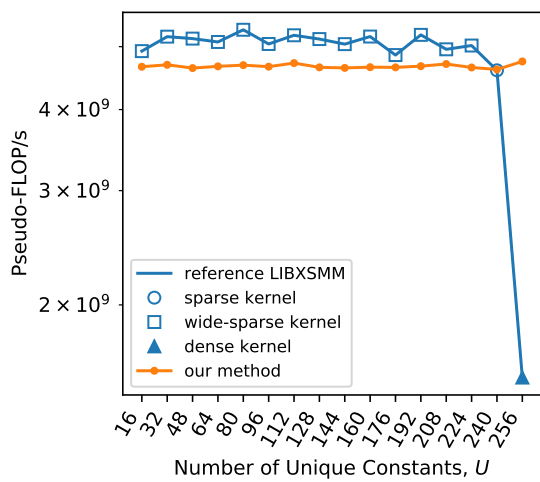


(e) Performance vs. density, $U = 256$.

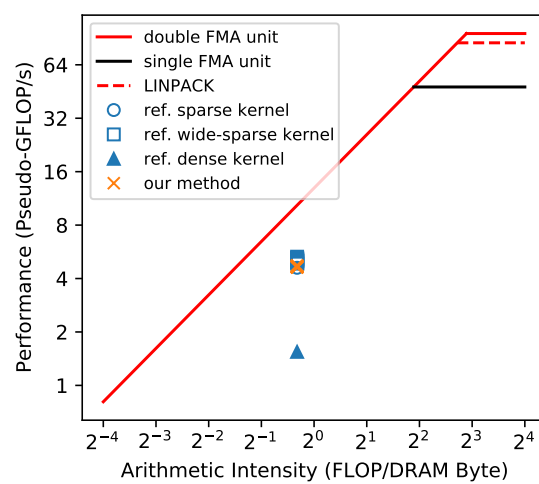


(f) Roofline plot, $U = 256$.

Figure C.19: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Experiment run on c5n.xlarge.



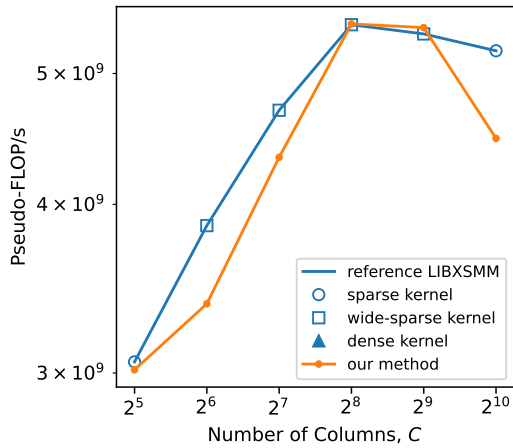
(a) Performance vs. number of unique constants.



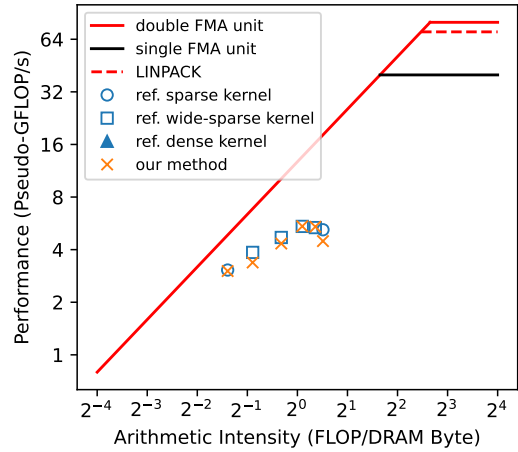
(b) Roofline plot.

Figure C.20: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of unique absolute non-zero constants in \mathbf{A} . Experiment run on c5n.xlarge.

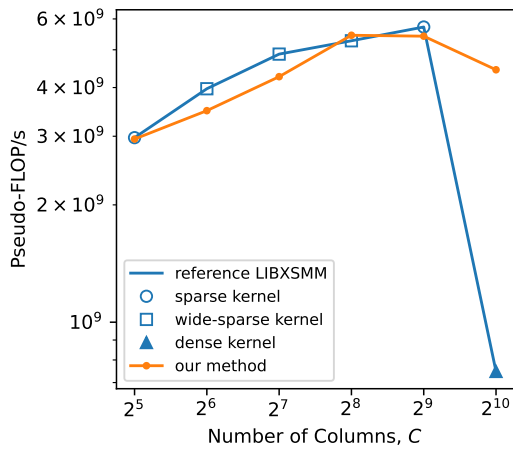
Benchmark Run on m5n.xlarge



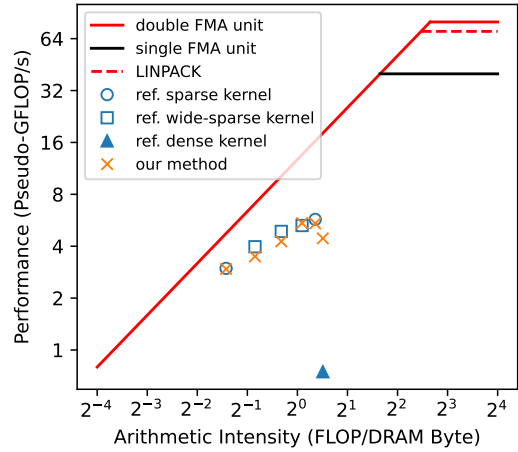
(a) Performance vs. number of columns, $U = 16$.



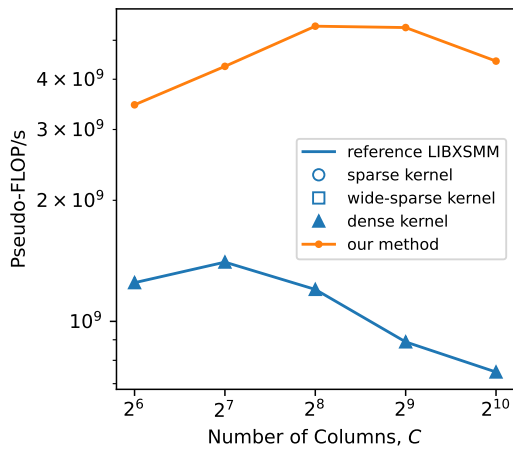
(b) Roofline plot, $U = 16$.



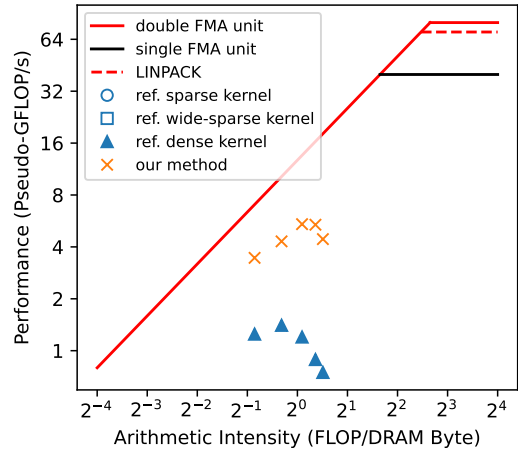
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

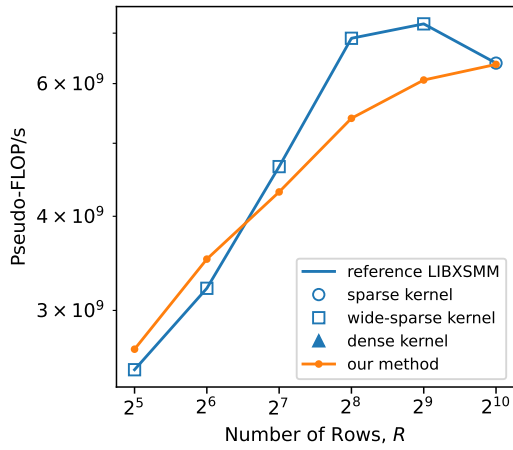


(e) Performance vs. number of columns, $U = 256$.

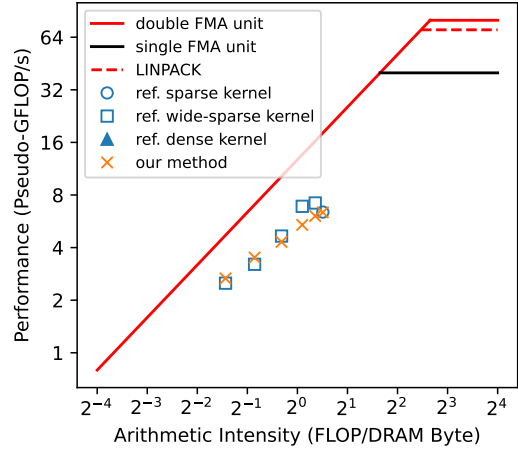


(f) Roofline plot, $U = 256$.

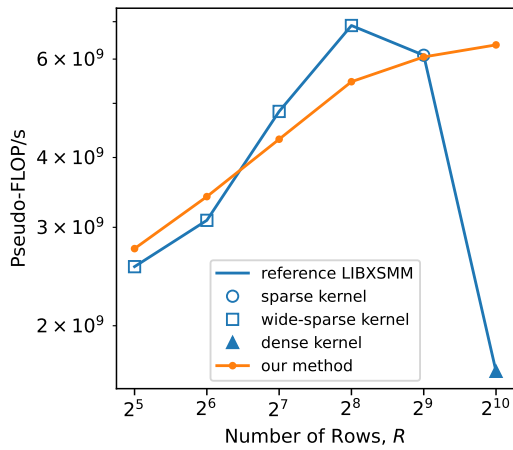
Figure C.21: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Experiment run on c5n.xlarge.



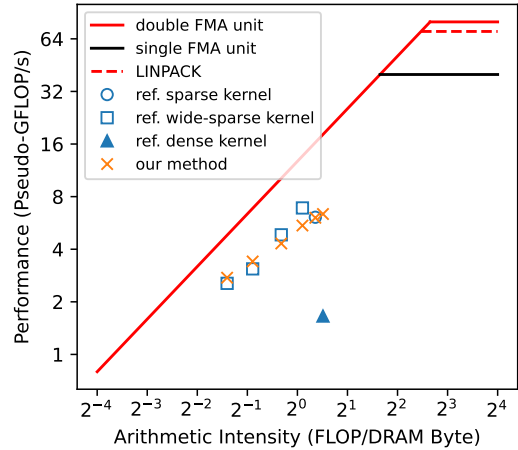
(a) Performance vs. number of rows, $U = 16$.



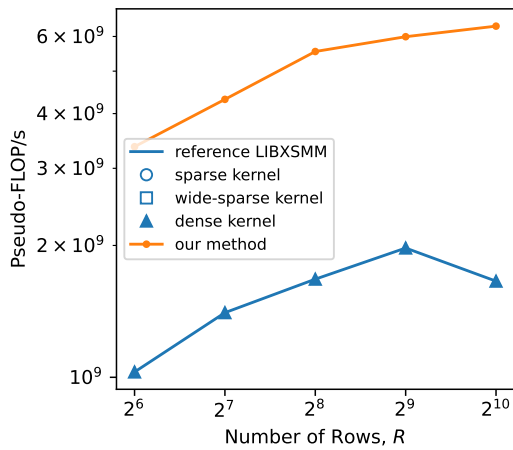
(b) Roofline plot, $U = 16$.



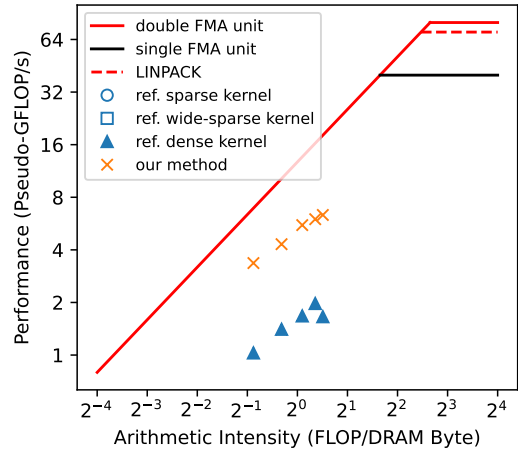
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

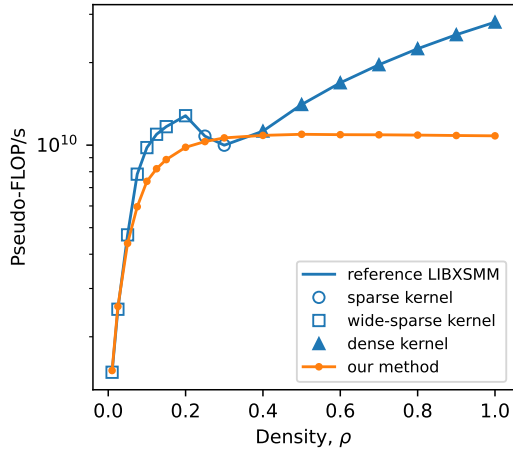


(e) Performance vs. number of rows, $U = 256$.

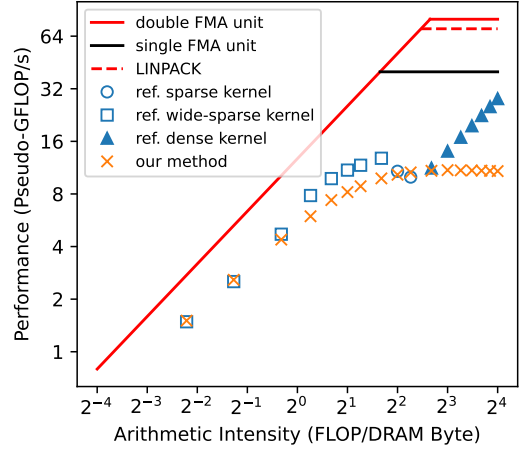


(f) Roofline plot, $U = 256$.

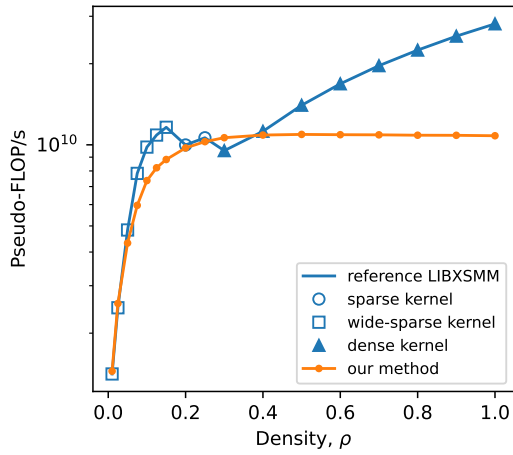
Figure C.22: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Experiment run on c5n.xlarge.



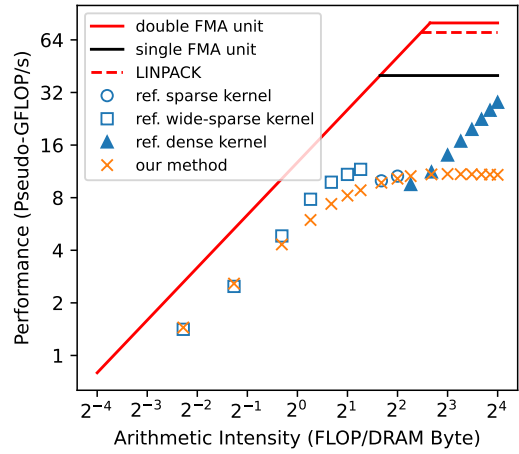
(a) Performance vs. density, $U = 16$.



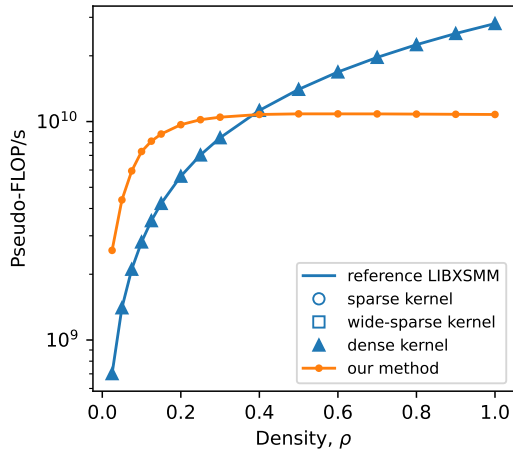
(b) Roofline plot, $U = 16$.



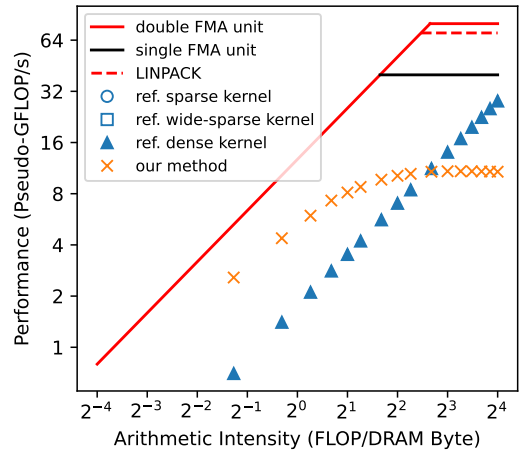
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

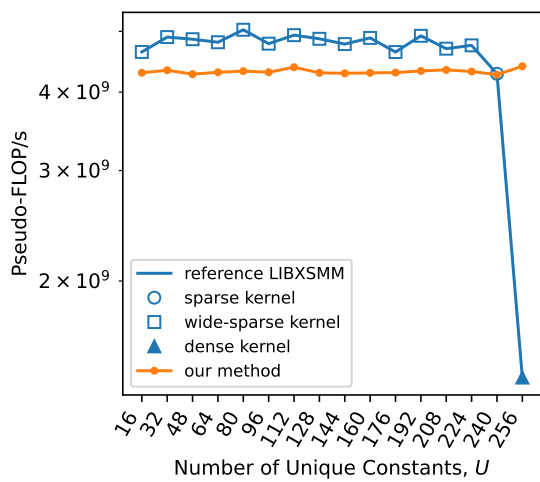


(e) Performance vs. density, $U = 256$.

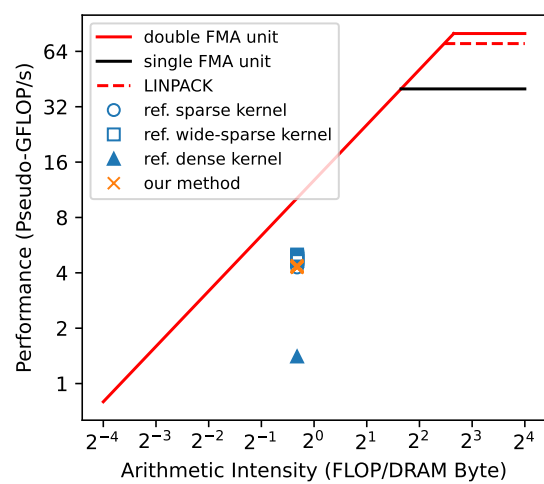


(f) Roofline plot, $U = 256$.

Figure C.23: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Experiment run on c5n.xlarge.



(a) Performance vs. number of unique constants.



(b) Roofline plot.

Figure C.24: Runtime broadcasting with loading \mathbf{A} from memory vs. reference LIBXSMM implementations, for synthetic matrices with varying number of unique absolute non-zero constants in \mathbf{A} . Experiment run on c5n.xlarge.

Appendix D

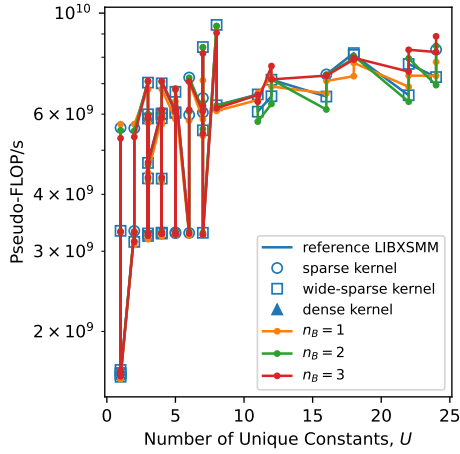
Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed \mathbf{A} Constants from Memory and Multiple Accumulations

This appendix presents the complete benchmark results for our GEMM method which runtime broadcasts packed \mathbf{A} constants from memory and uses multiple accumulations. As explained in Chapter 4, two sets of \mathbf{A} matrices were used for the testing - a set of all PyFR operator matrices, and a set of synthetic matrices. We evaluated our kernel on two testing machines - a Skylake machine (c5n.xlarge) and a Cascade Lake machine (m5n.xlarge). Please refer to Chapter 6 for how the kernel implements GEMM.

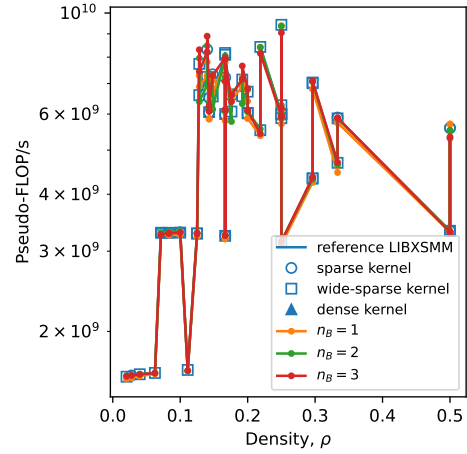
D.1 N Blocking

PyFR Operator Matrices

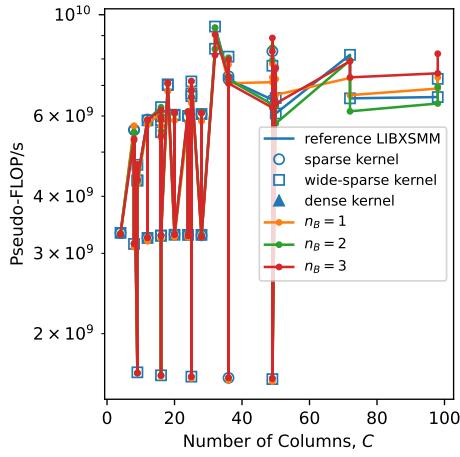
Benchmark Run on c5n.xlarge



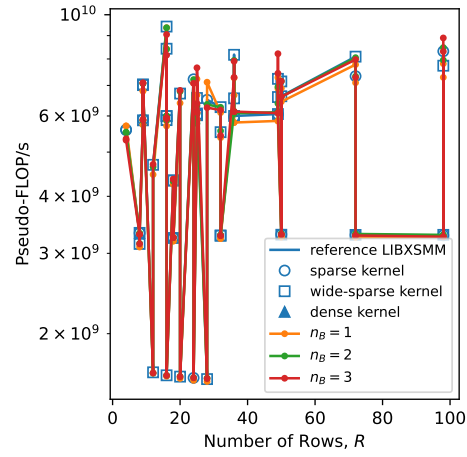
(a) Performance against number of unique \mathbf{A} constants.



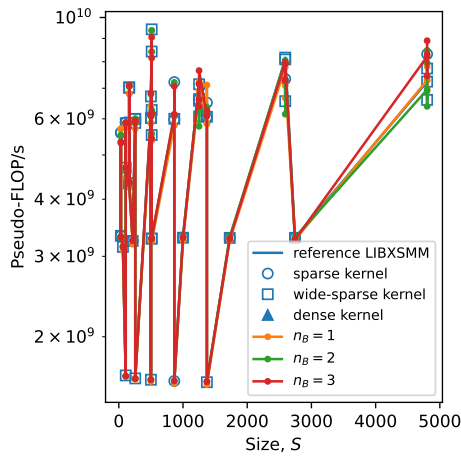
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.1: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on c5n.xlarge machine.

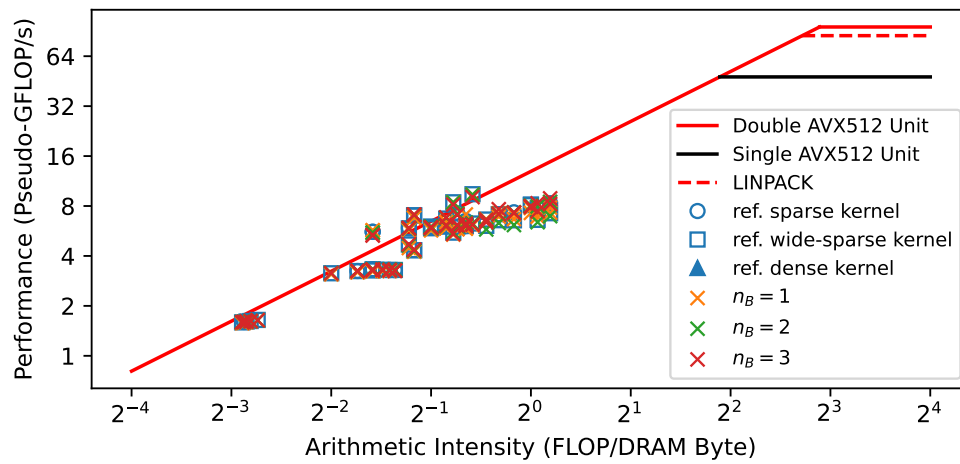
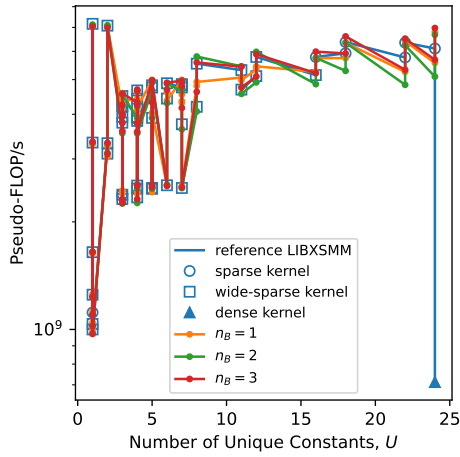
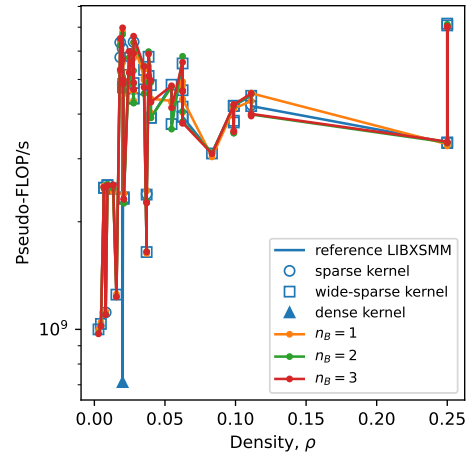


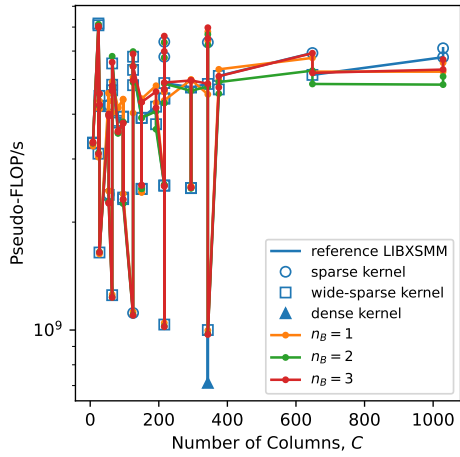
Figure D.2: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



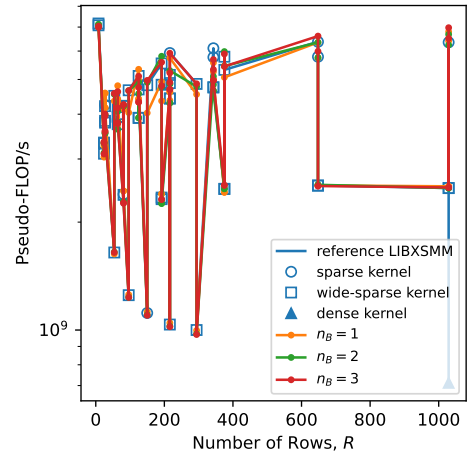
(a) Performance against number of unique \mathbf{A} constants.



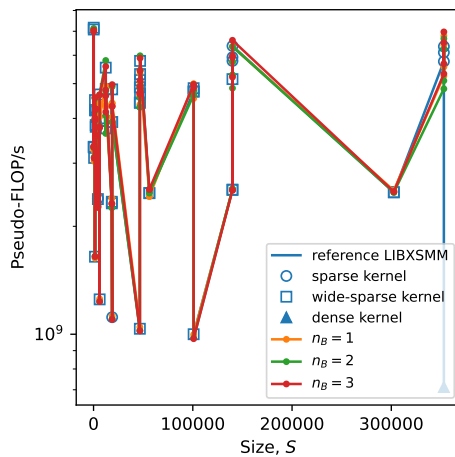
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.3: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on c5n.xlarge machine.

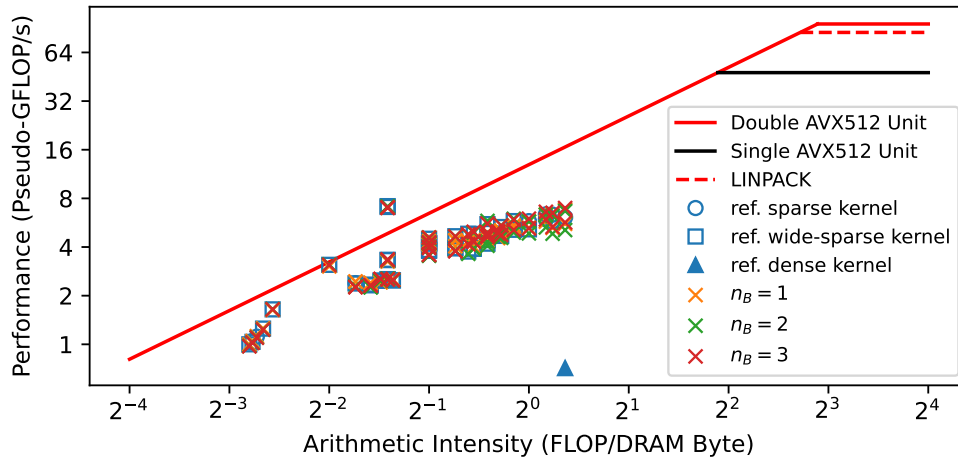
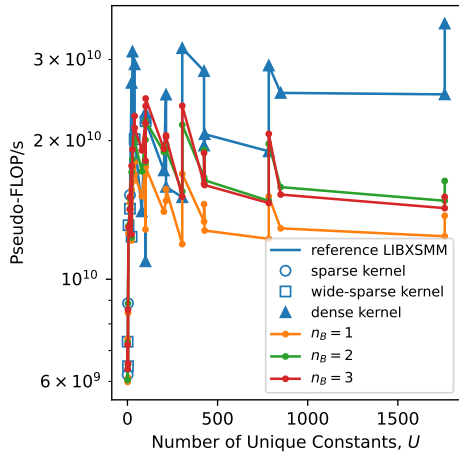
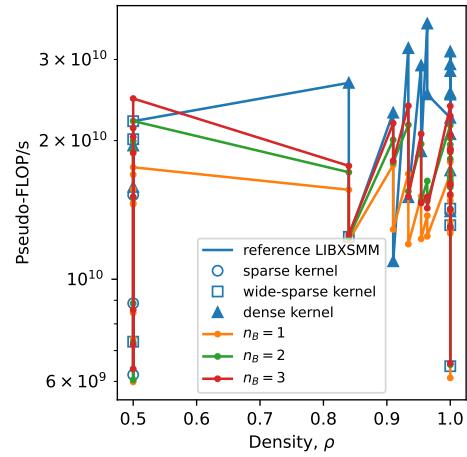


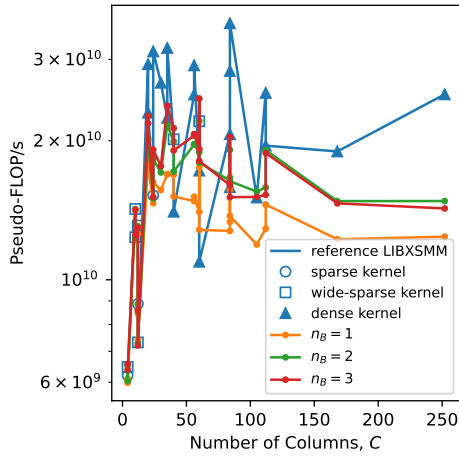
Figure D.4: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



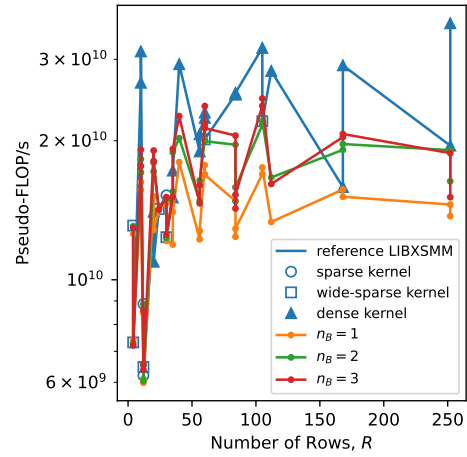
(a) Performance against number of unique \mathbf{A} constants.



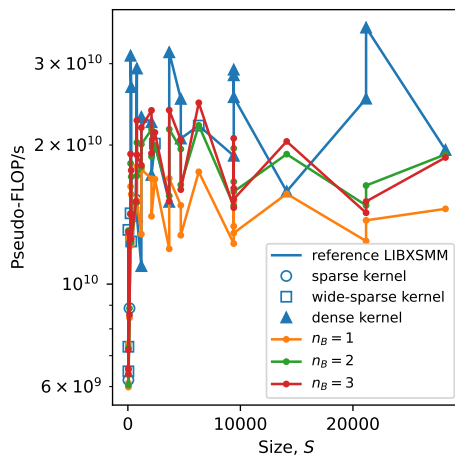
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.5: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on c5n.xlarge machine.

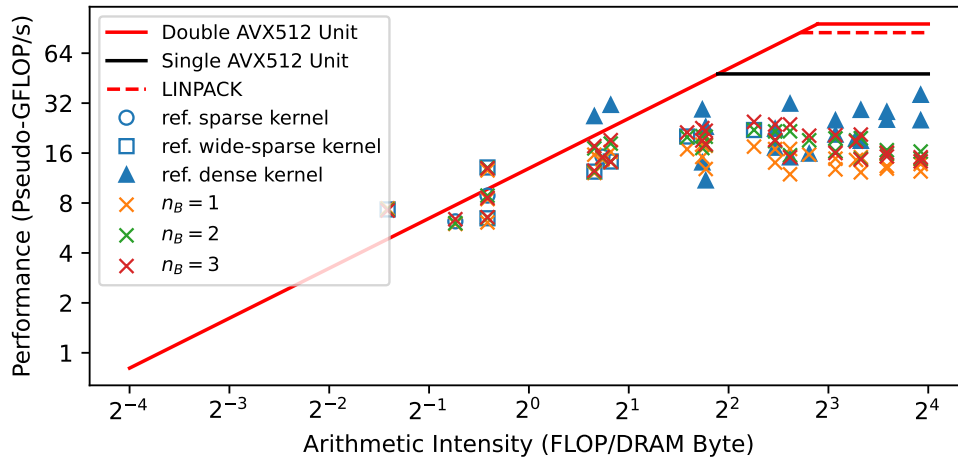
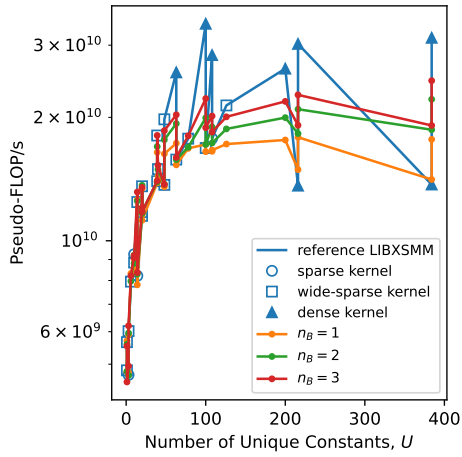
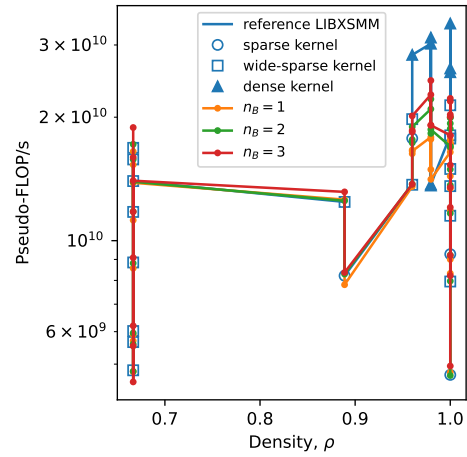


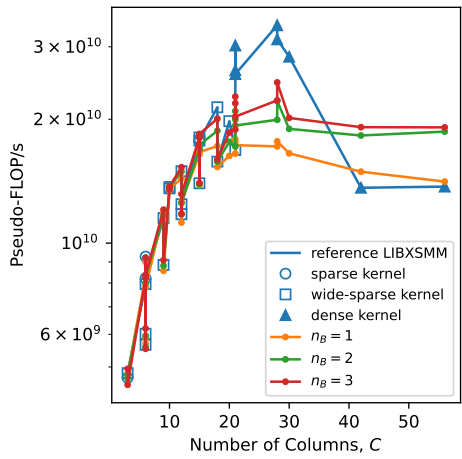
Figure D.6: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



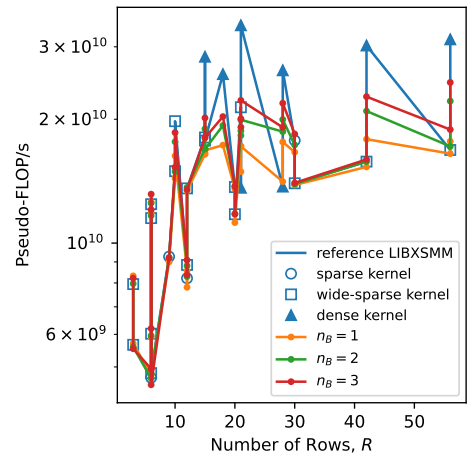
(a) Performance against number of unique \mathbf{A} constants.



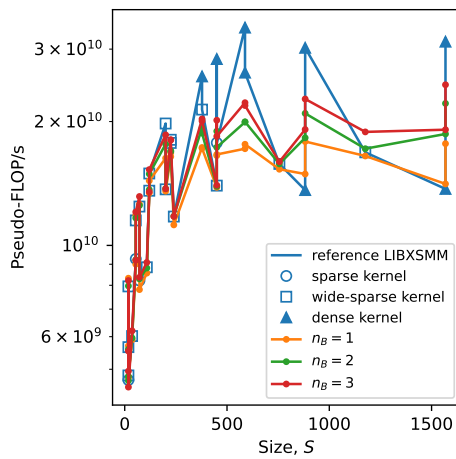
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.7: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on c5n.xlarge machine.

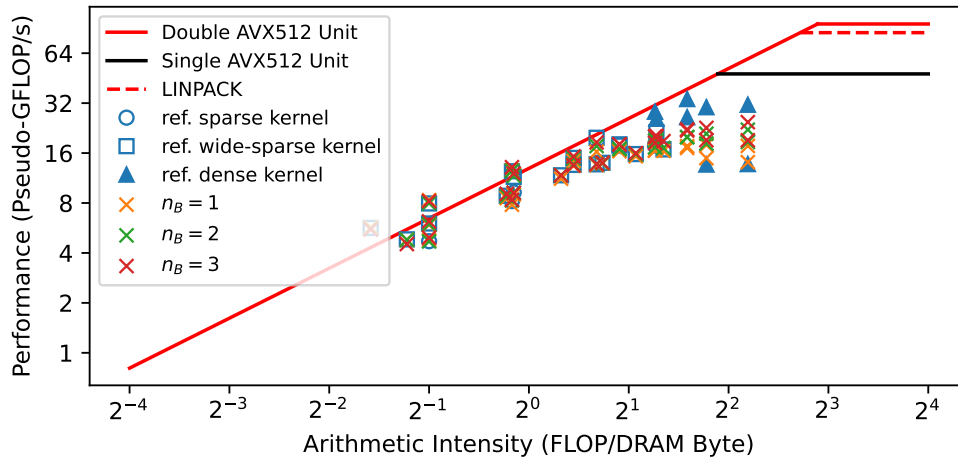
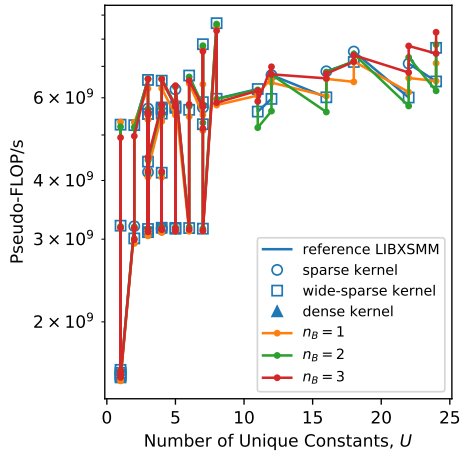
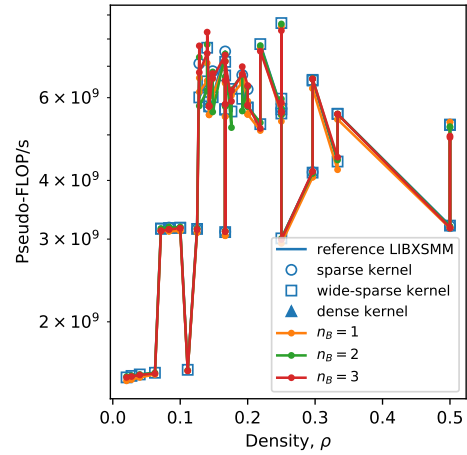


Figure D.8: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.

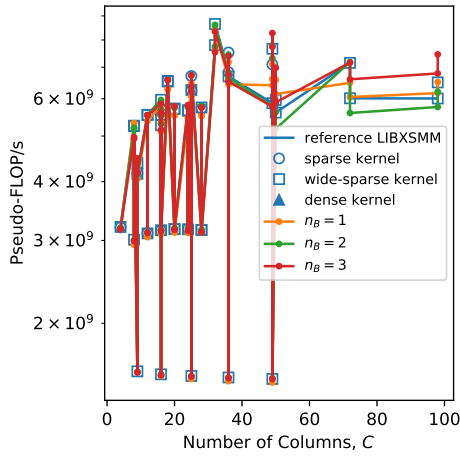
Benchmark Run on m5n.xlarge



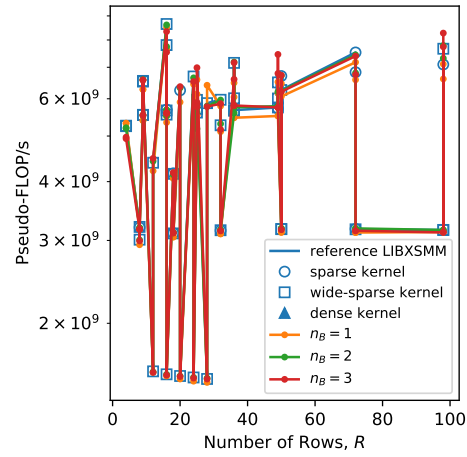
(a) Performance against number of unique \mathbf{A} constants.



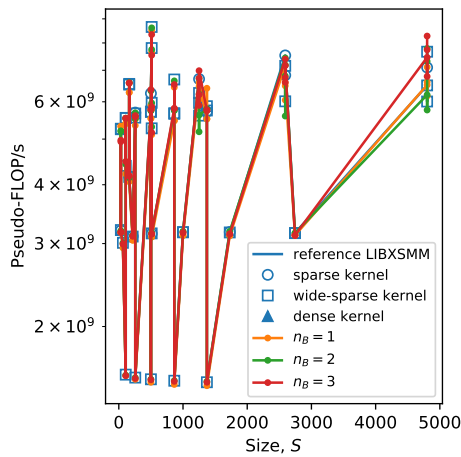
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.9: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on m5n.xlarge machine.

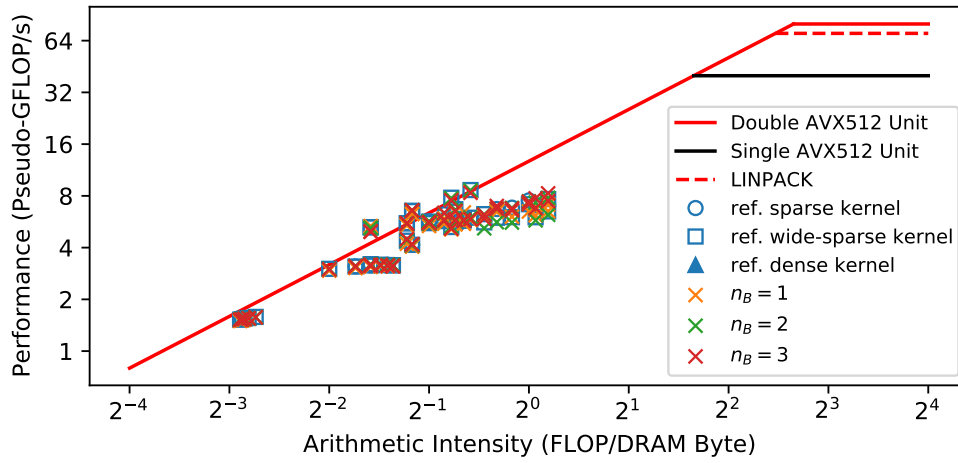
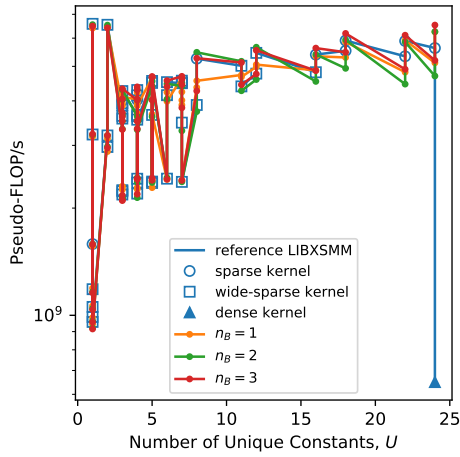
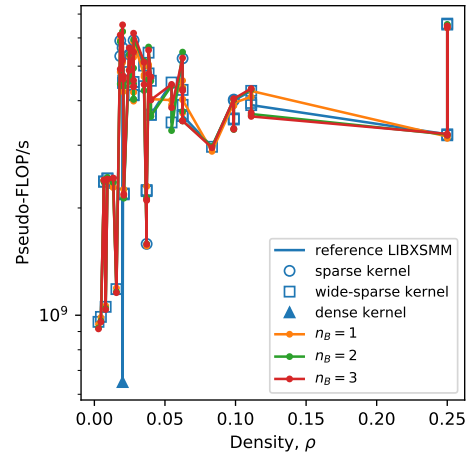


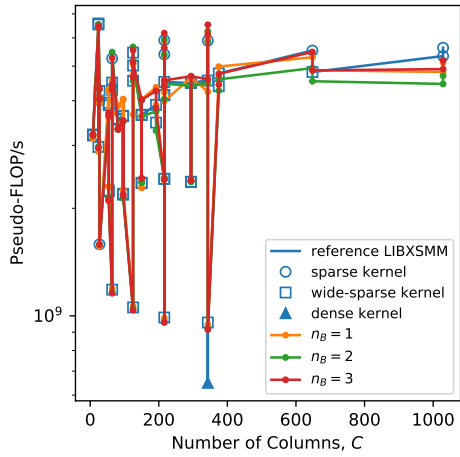
Figure D.10: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



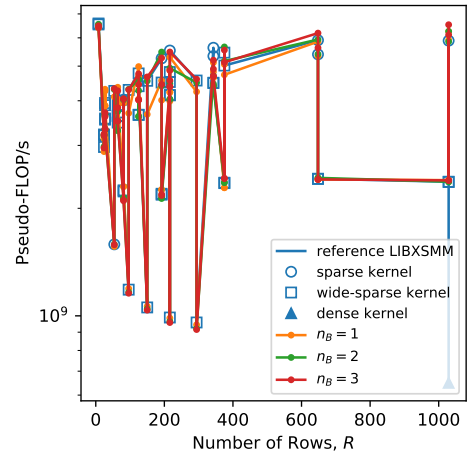
(a) Performance against number of unique \mathbf{A} constants.



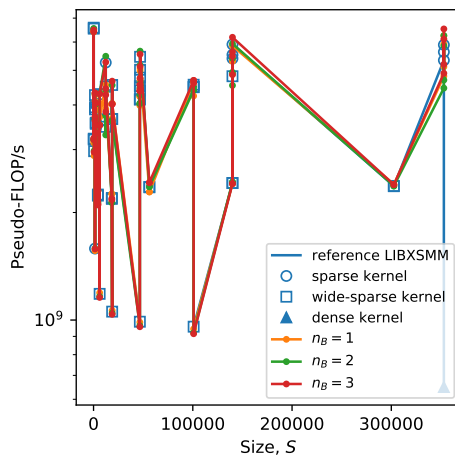
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.11: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on m5n.xlarge machine.

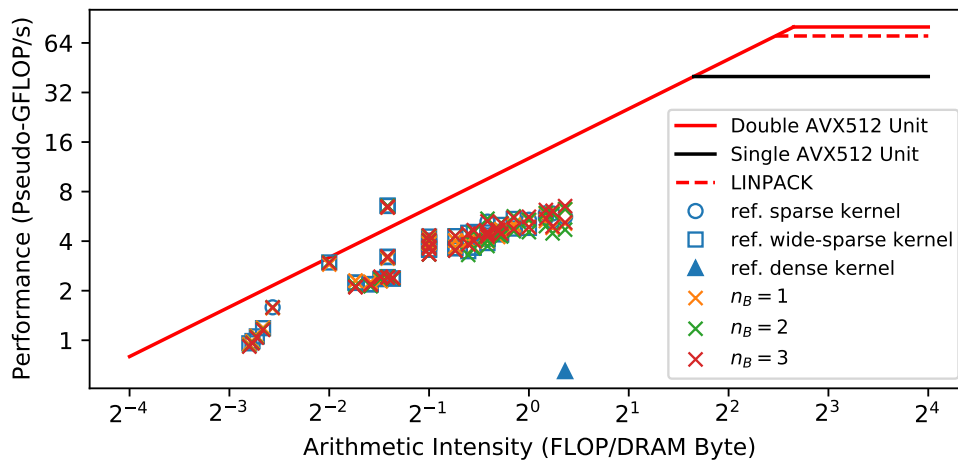
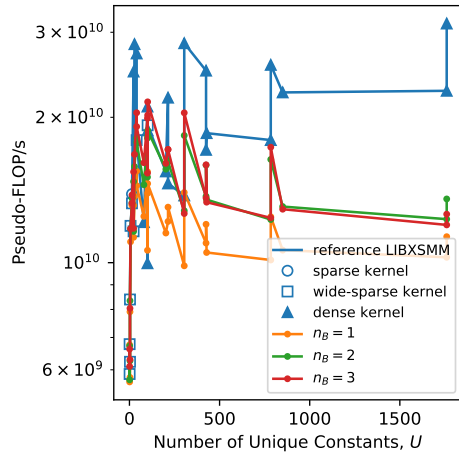
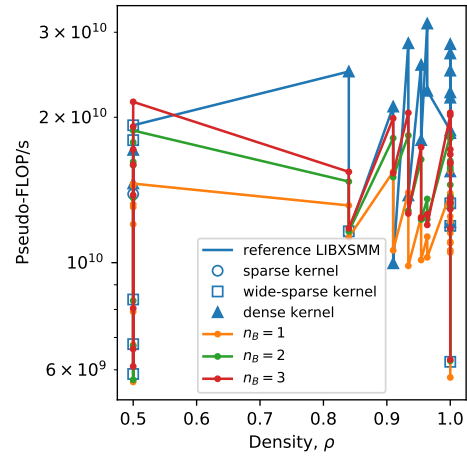


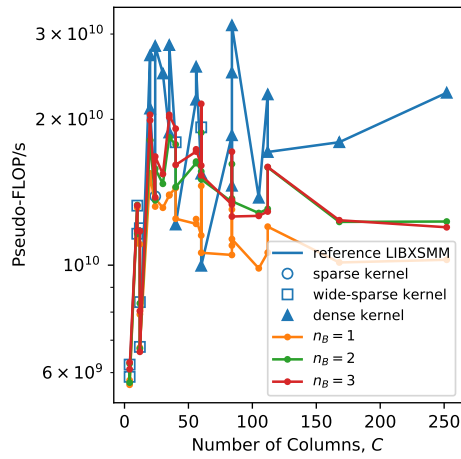
Figure D.12: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



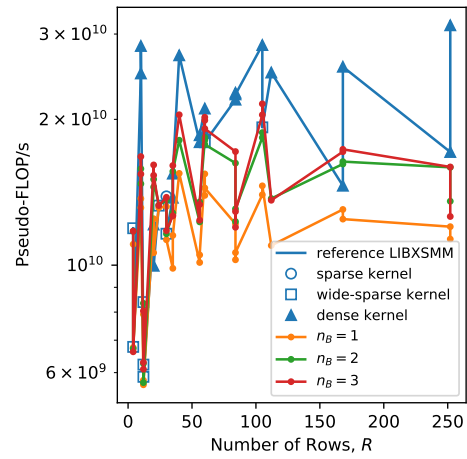
(a) Performance against number of unique \mathbf{A} constants.



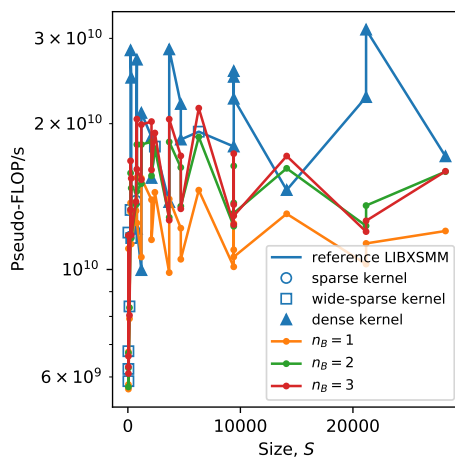
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.13: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on m5n.xlarge machine.

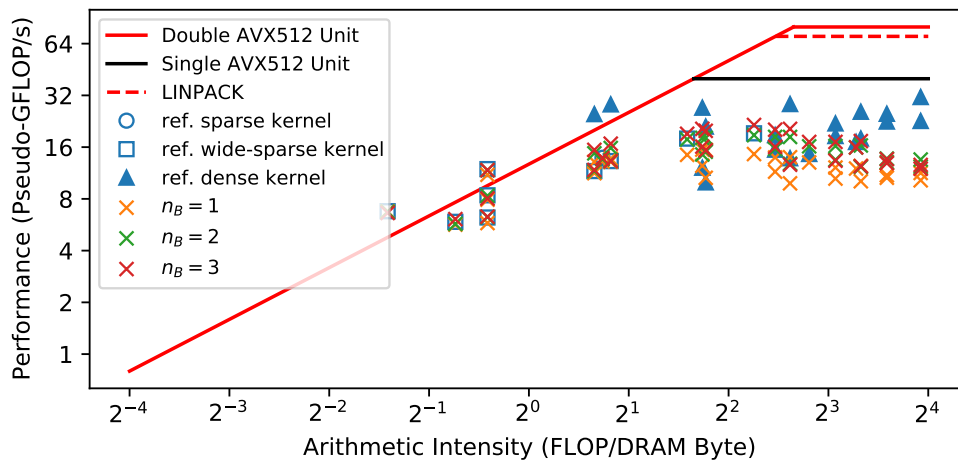
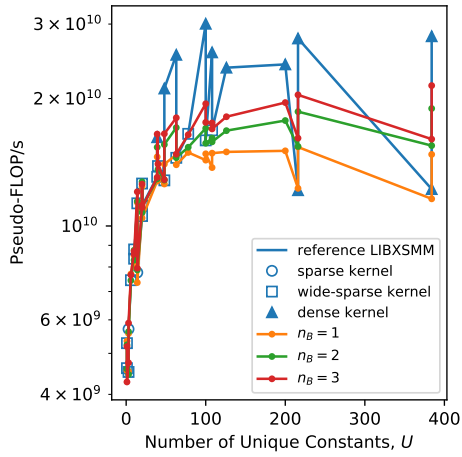
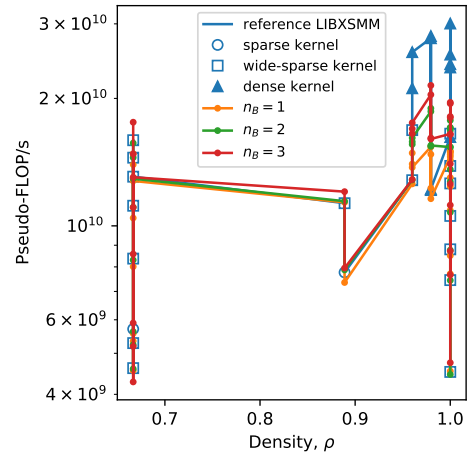


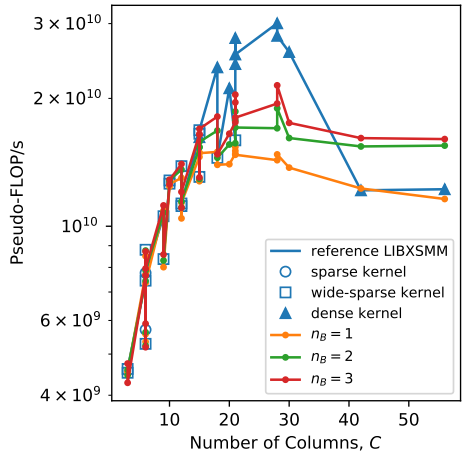
Figure D.14: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



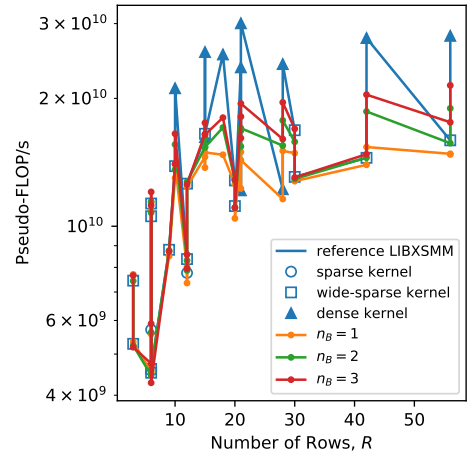
(a) Performance against number of unique \mathbf{A} constants.



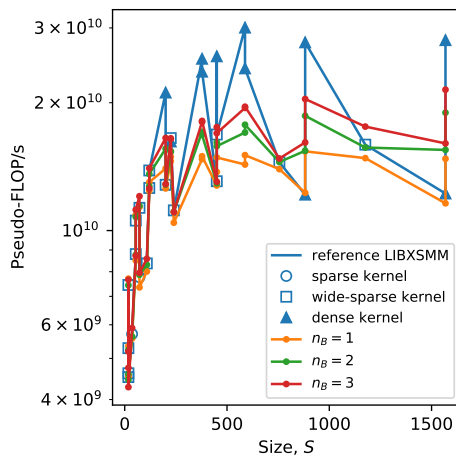
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.15: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on m5n.xlarge machine.

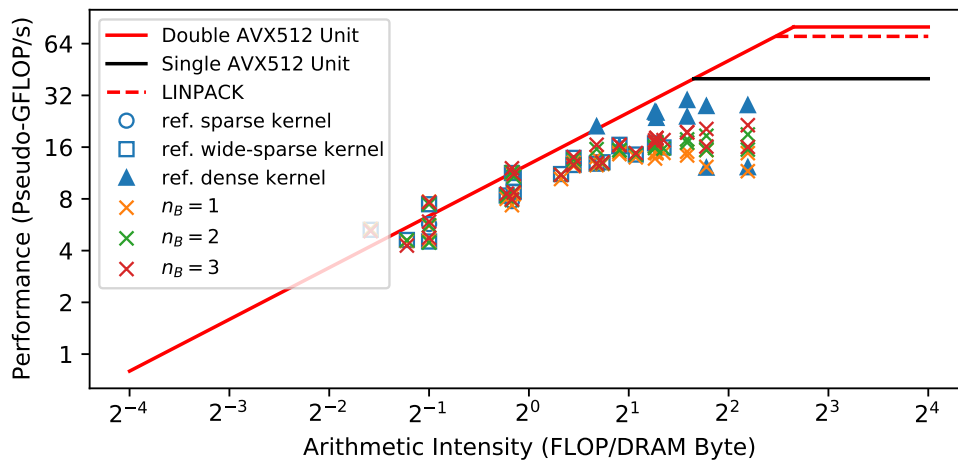
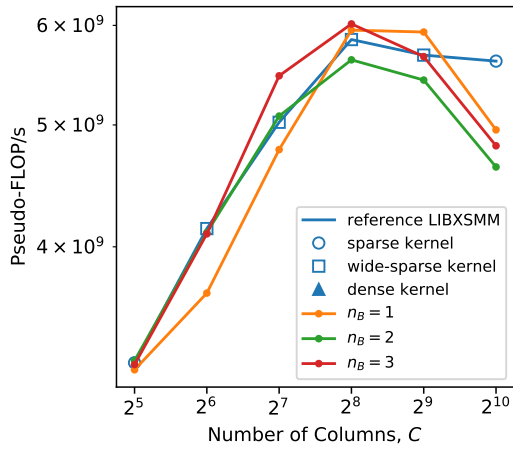


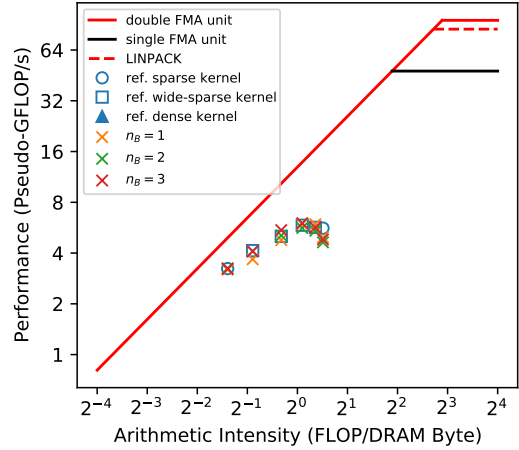
Figure D.16: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.

Synthetic Matrices

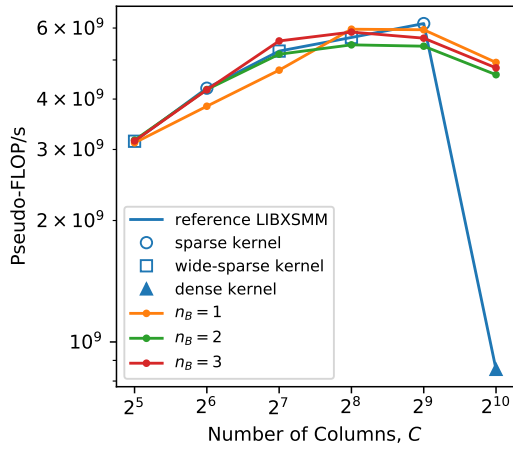
Benchmark Run on c5n.xlarge



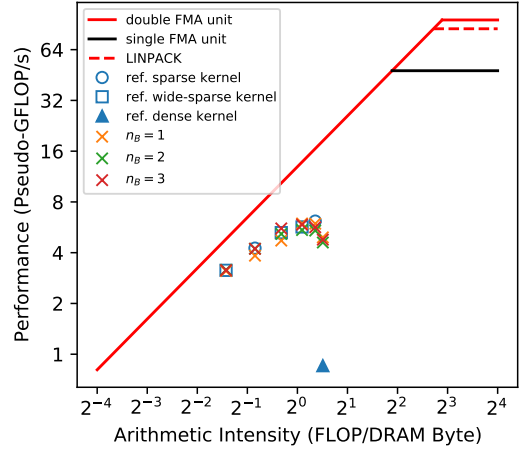
(a) Performance vs. number of columns, $U = 16$.



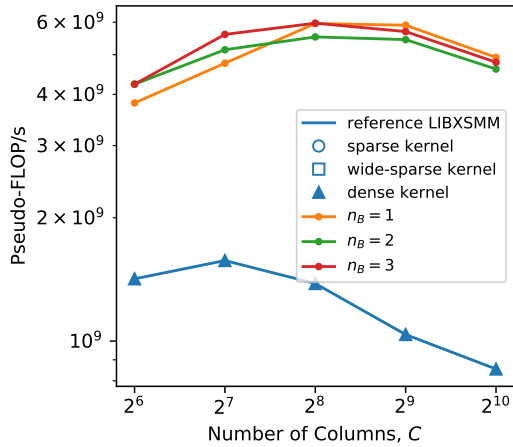
(b) Roofline plot, $U = 16$.



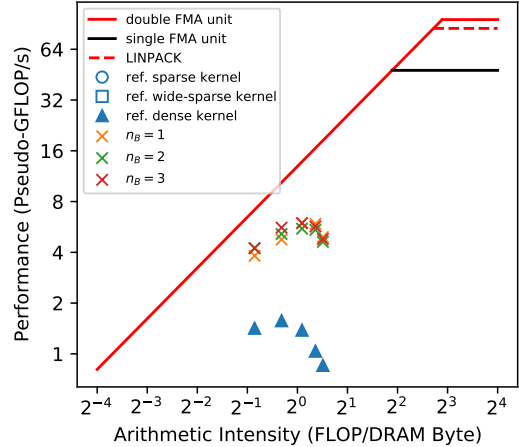
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

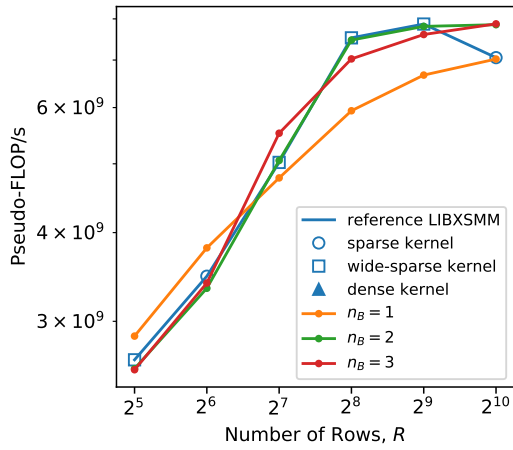


(e) Performance vs. number of columns, $U = 256$.

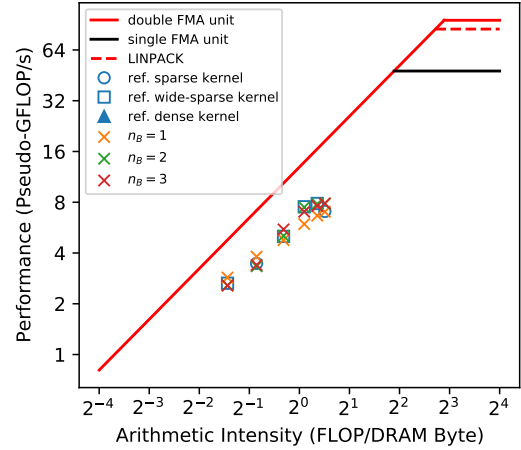


(f) Roofline plot, $U = 256$.

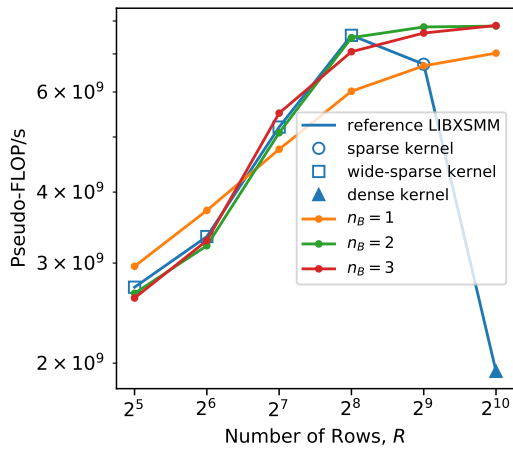
Figure D.17: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on c5n.xlarge machine



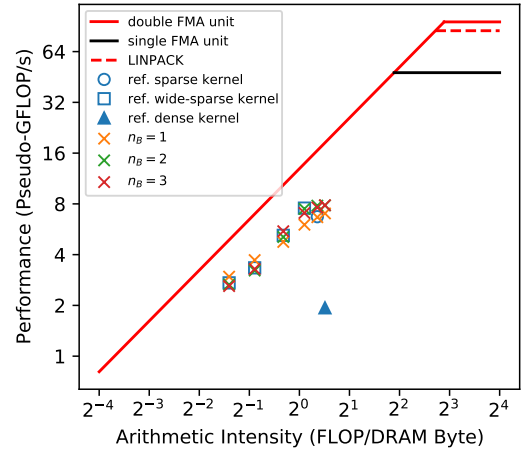
(a) Performance vs. number of rows, $U = 16$.



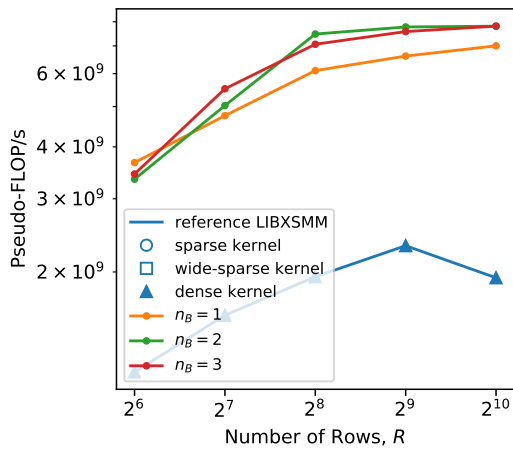
(b) Roofline plot, $U = 16$.



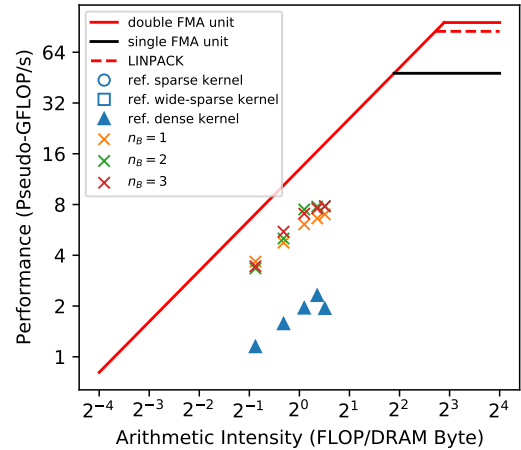
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

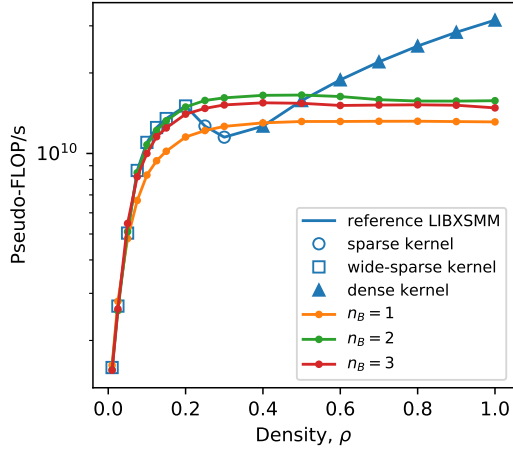


(e) Performance vs. number of rows, $U = 256$.

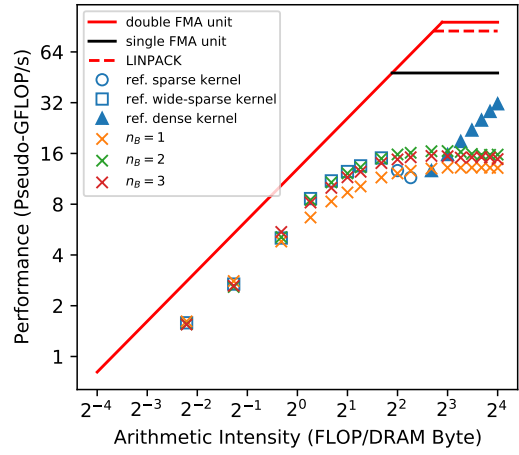


(f) Roofline plot, $U = 256$.

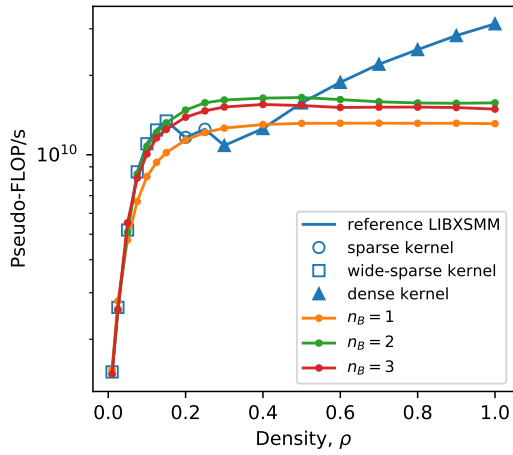
Figure D.18: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on c5n.xlarge machine



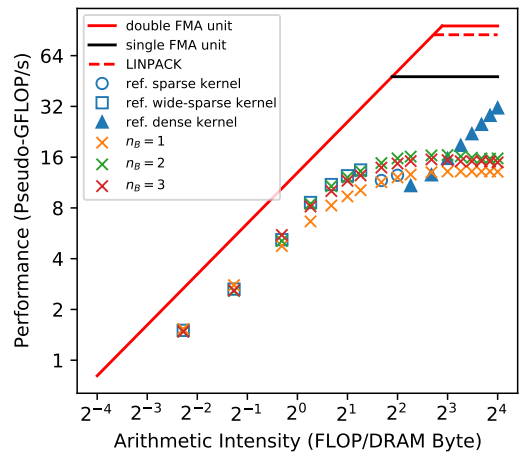
(a) Performance vs. density, $U = 16$.



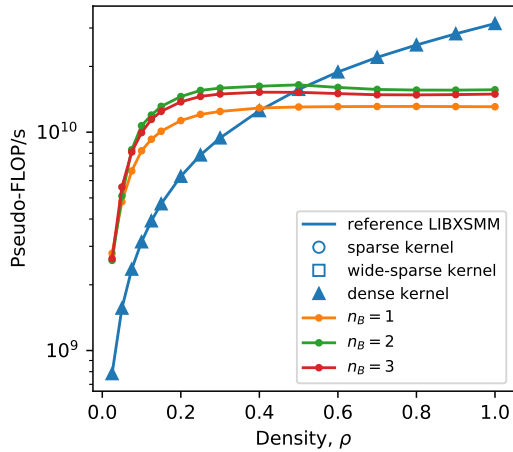
(b) Roofline plot, $U = 16$.



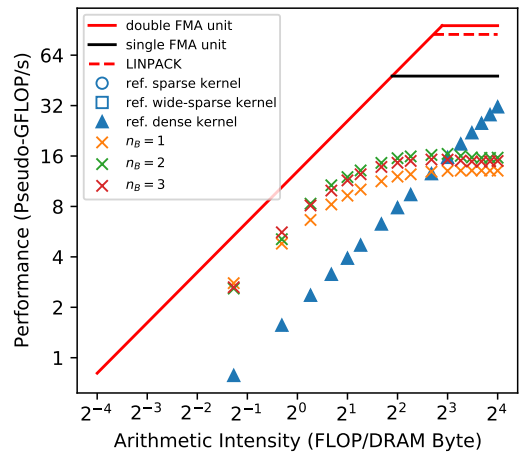
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

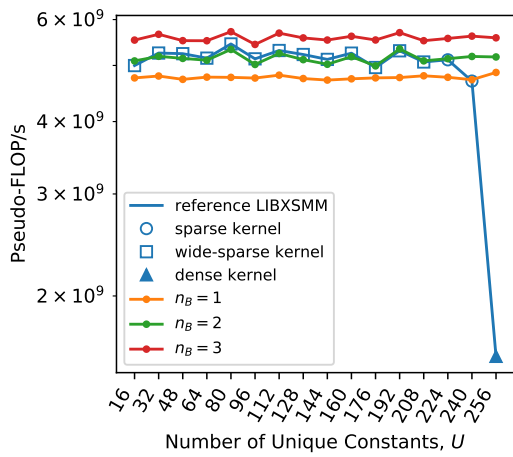


(e) Performance vs. density, $U = 256$.

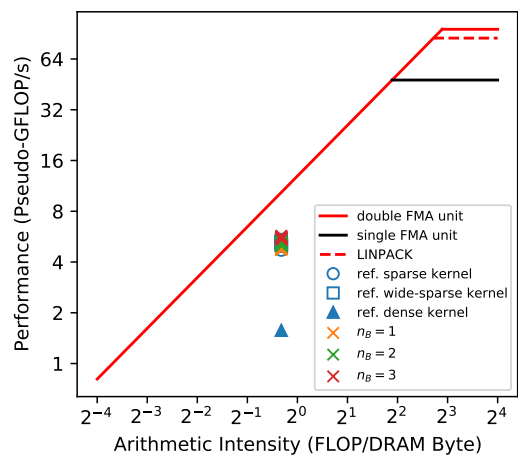


(f) Roofline plot, $U = 256$.

Figure D.19: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on c5n.xlarge machine



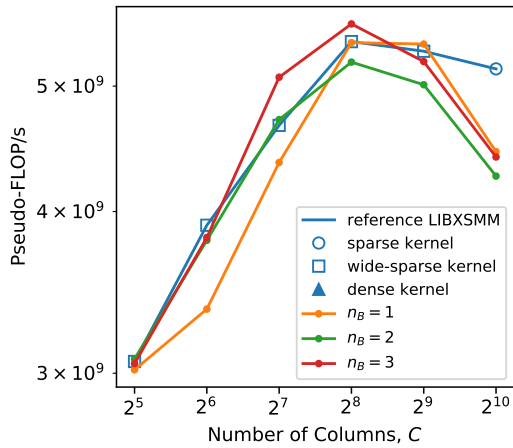
(a) Performance vs. number of unique absolute non-zero values.



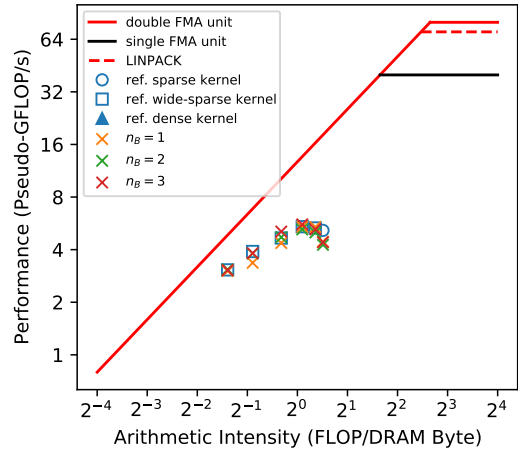
(b) Roofline plot.

Figure D.20: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine

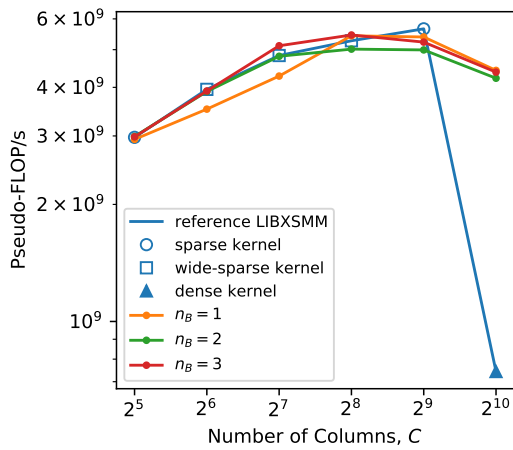
Benchmark Run on m5n.xlarge



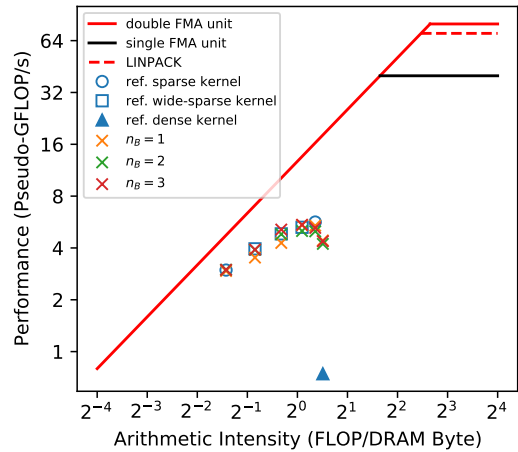
(a) Performance vs. number of columns, $U = 16$.



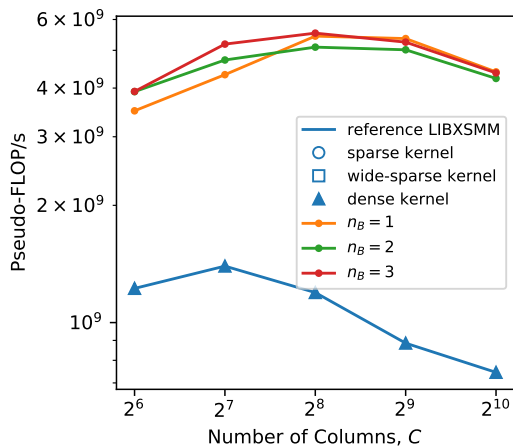
(b) Roofline plot, $U = 16$.



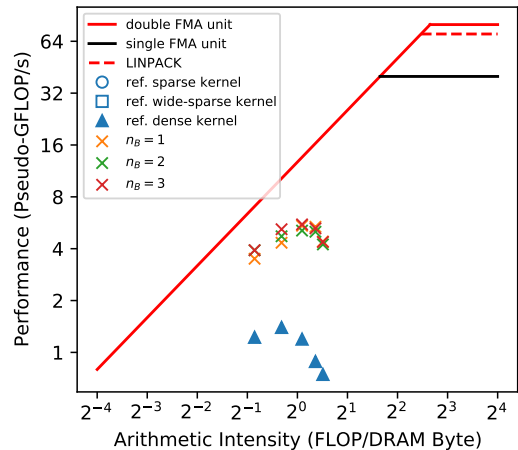
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

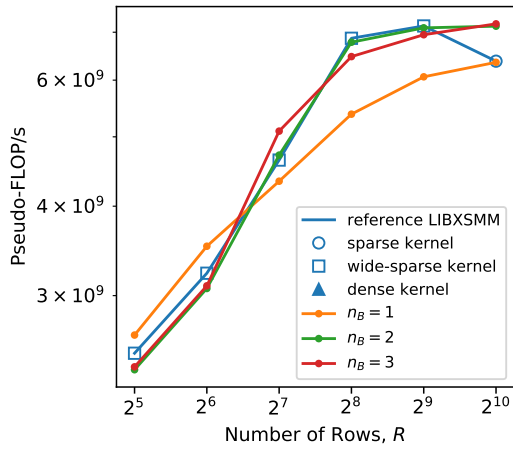


(e) Performance vs. number of columns, $U = 256$.

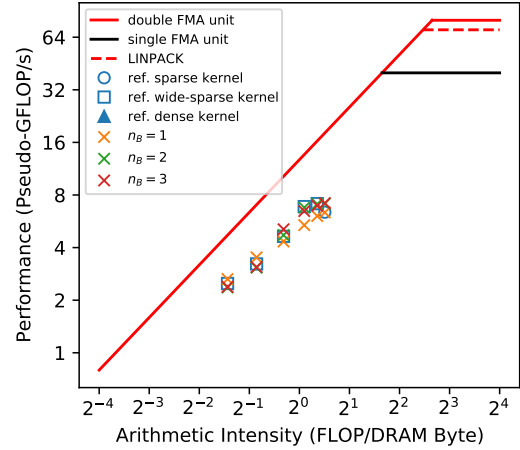


(f) Roofline plot, $U = 256$.

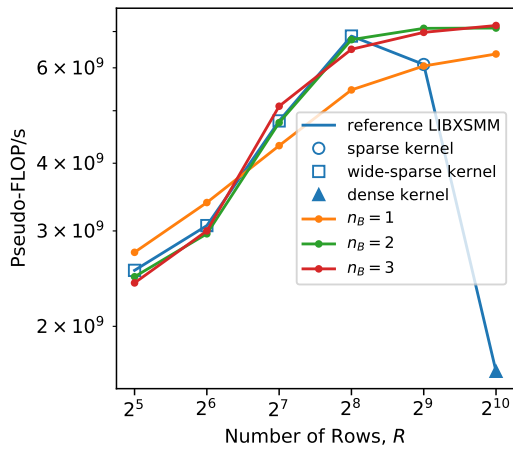
Figure D.21: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on m5n.xlarge machine



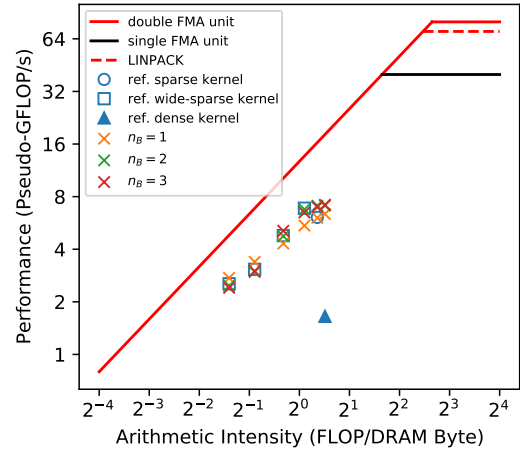
(a) Performance vs. number of rows, $U = 16$.



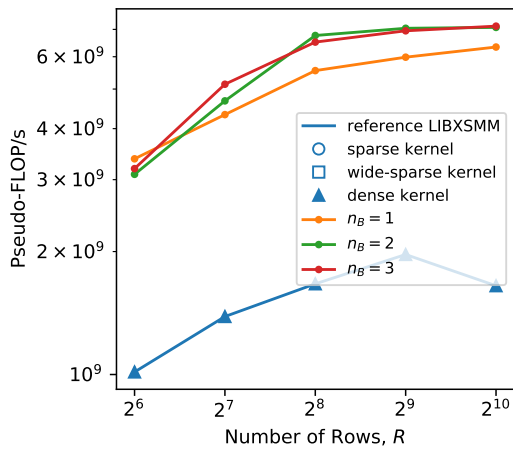
(b) Roofline plot, $U = 16$.



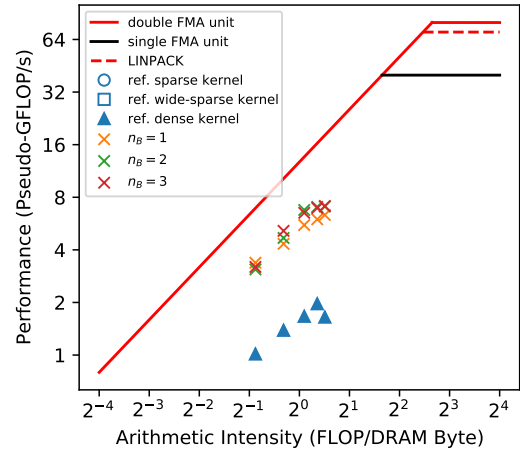
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

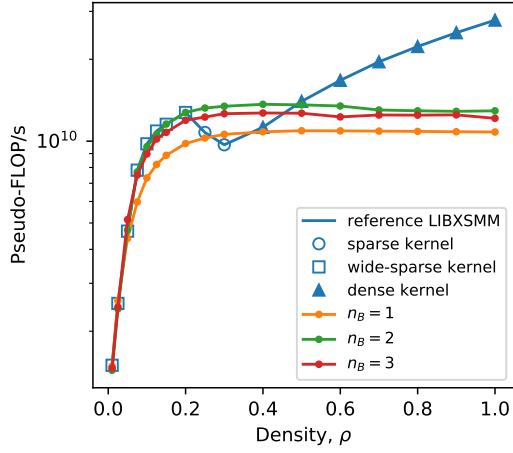


(e) Performance vs. number of rows, $U = 256$.

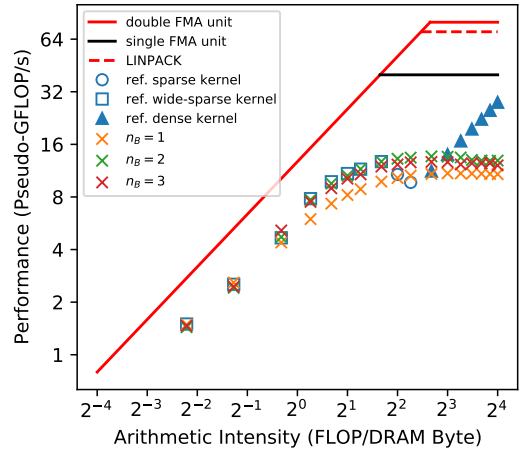


(f) Roofline plot, $U = 256$.

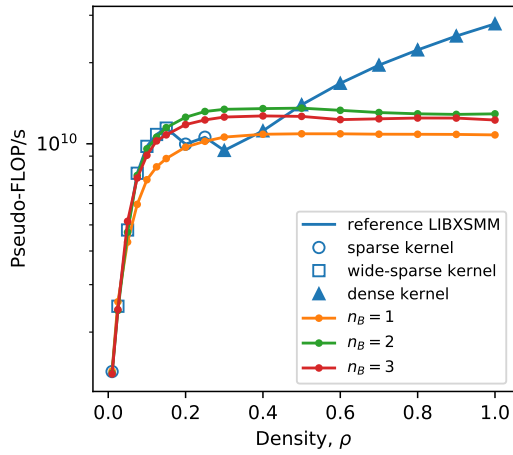
Figure D.22: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on m5n.xlarge machine



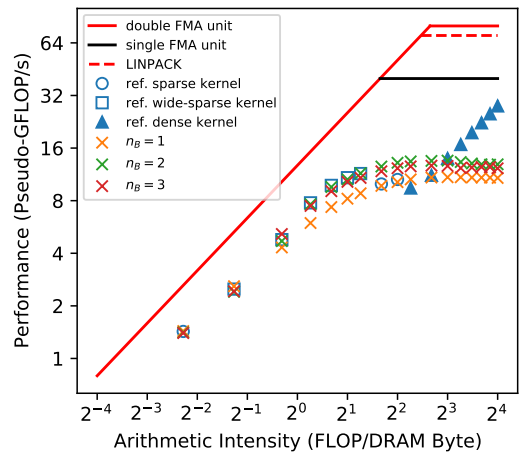
(a) Performance vs. density, $U = 16$.



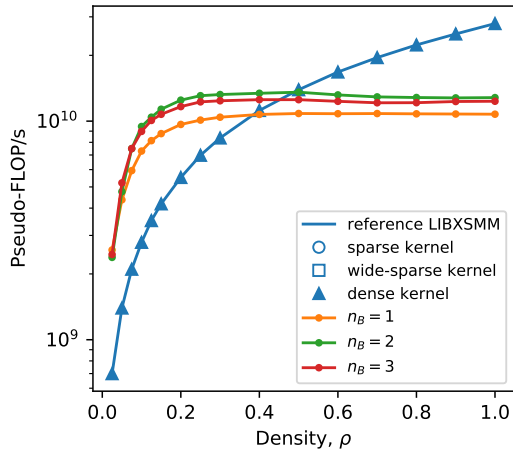
(b) Roofline plot, $U = 16$.



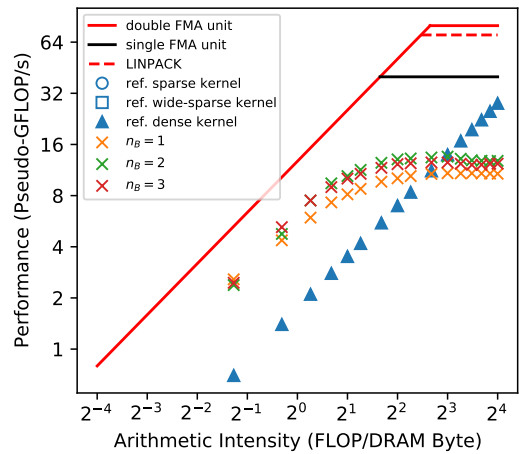
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

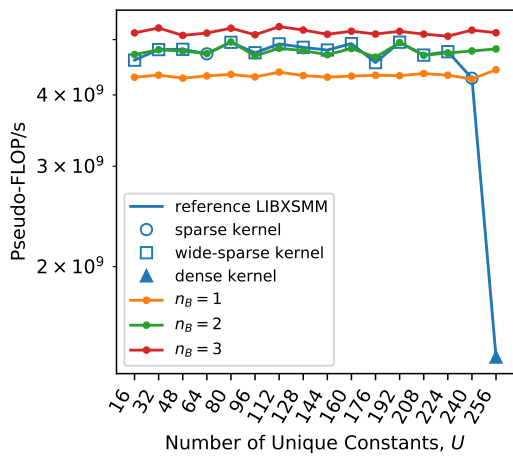


(e) Performance vs. density, $U = 256$.

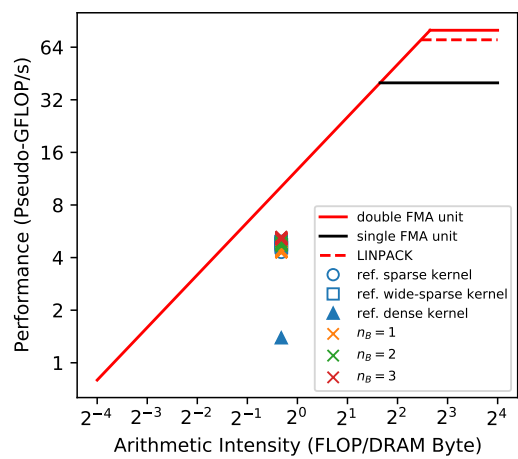


(f) Roofline plot, $U = 256$.

Figure D.23: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on m5n.xlarge machine



(a) Performance vs. number of unique absolute non-zero values.



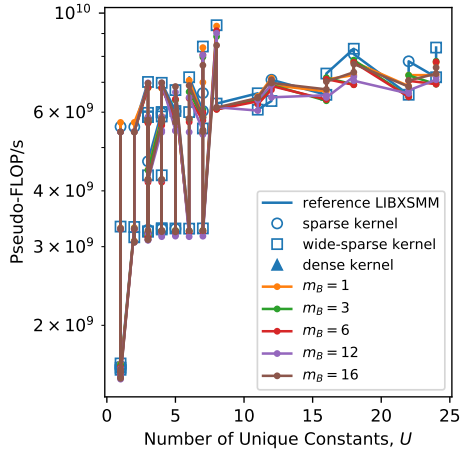
(b) Roofline plot.

Figure D.24: Runtime broadcasting with loading \mathbf{A} from memory and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on `m5n.xlarge` machine

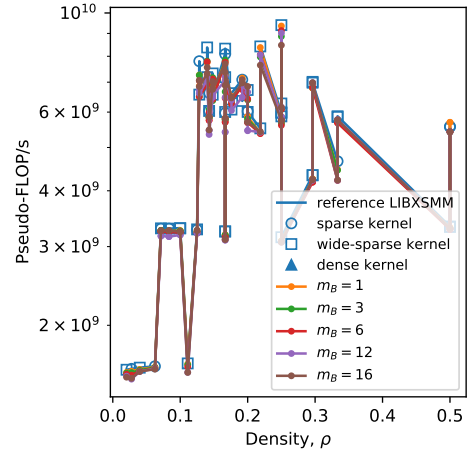
D.2 M Blocking

PyFR Operator Matrices

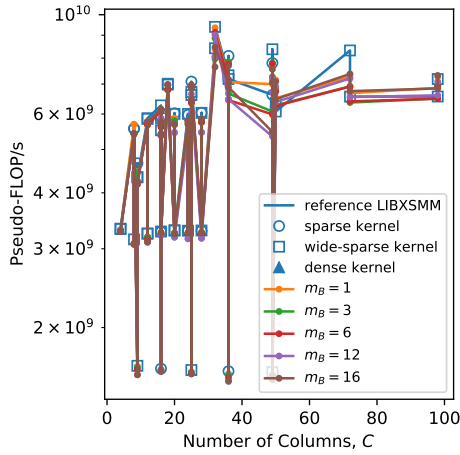
Benchmark Run on c5n.xlarge



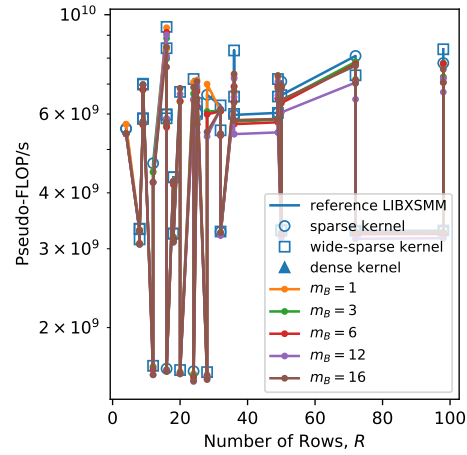
(a) Performance against number of unique \mathbf{A} constants.



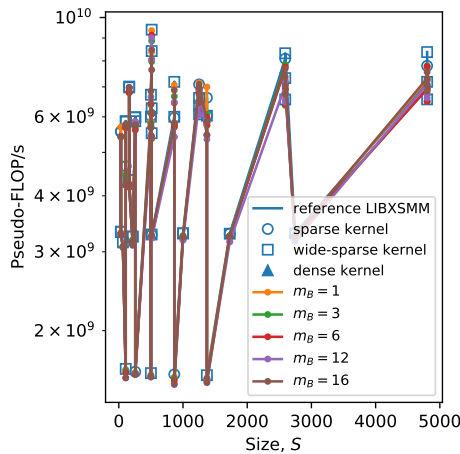
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.25: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on c5n.xlarge machine.

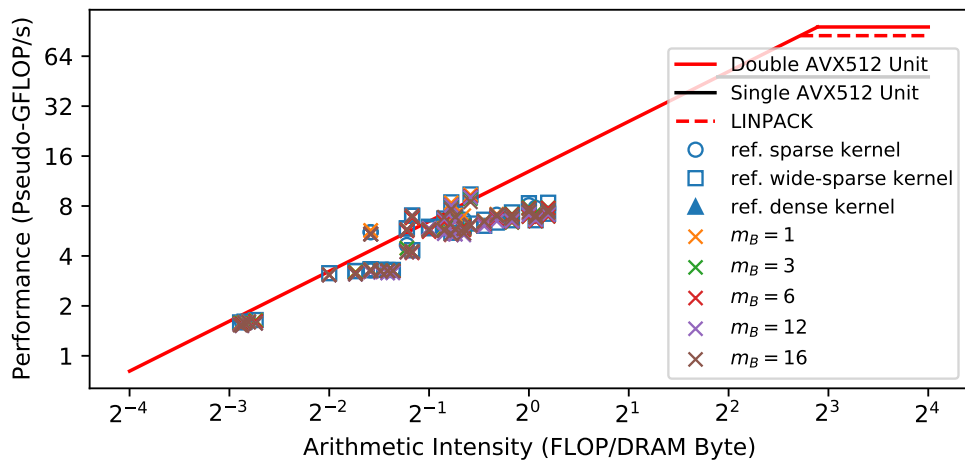
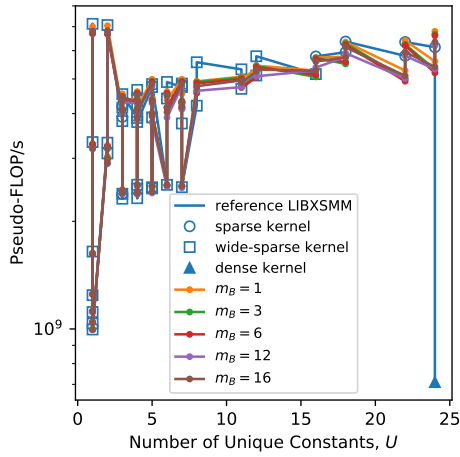
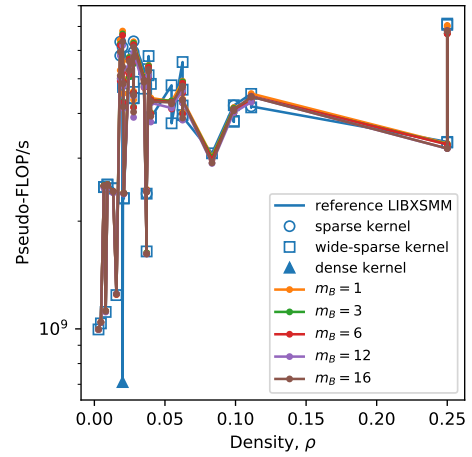


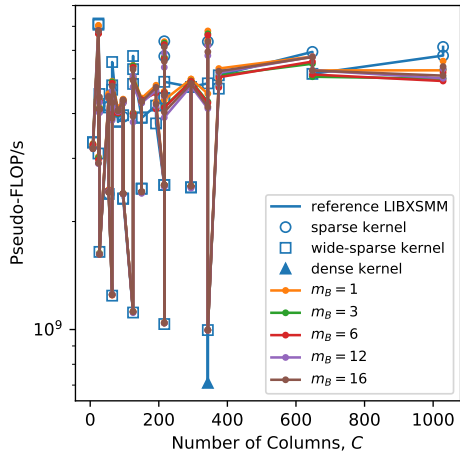
Figure D.26: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



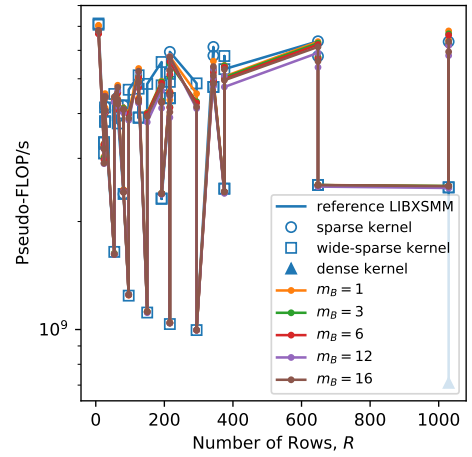
(a) Performance against number of unique \mathbf{A} constants.



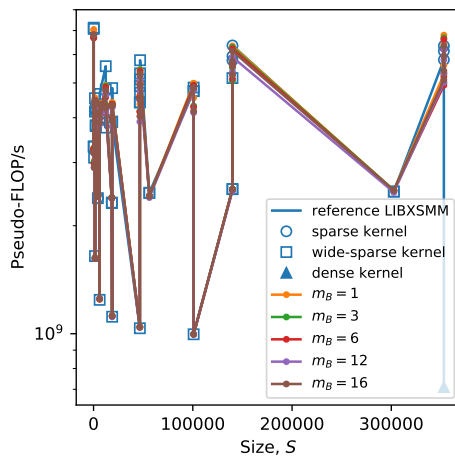
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.27: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on c5n.xlarge machine.

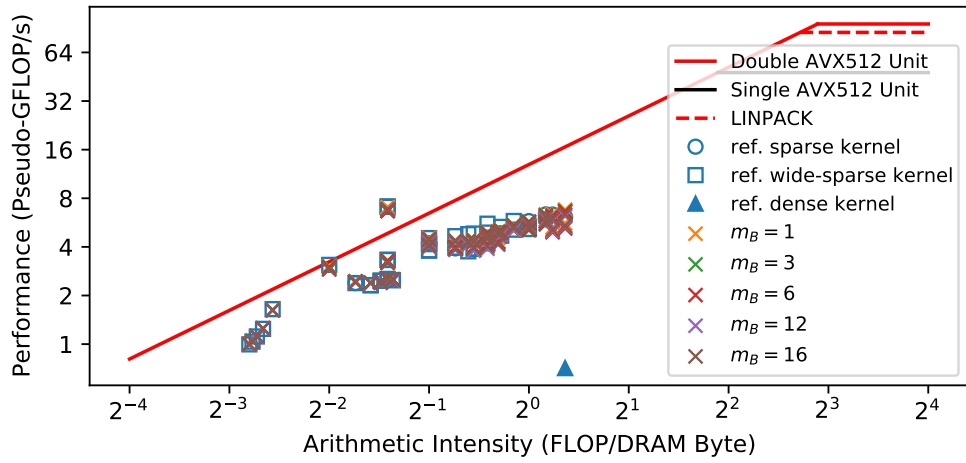
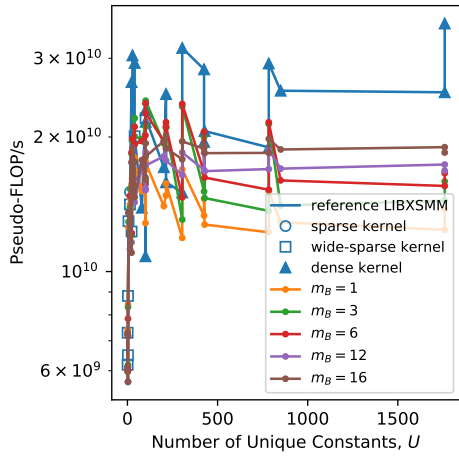
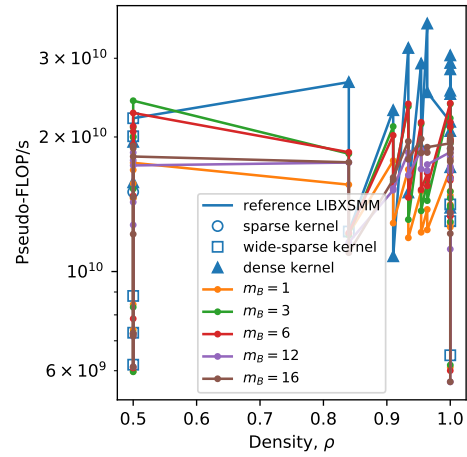


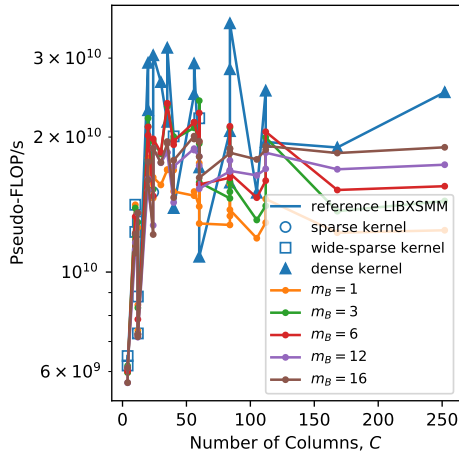
Figure D.28: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



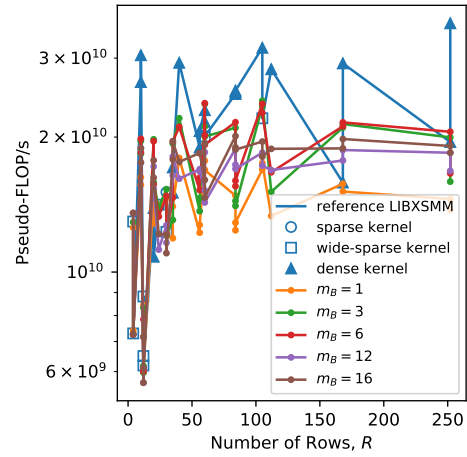
(a) Performance against number of unique \mathbf{A} constants.



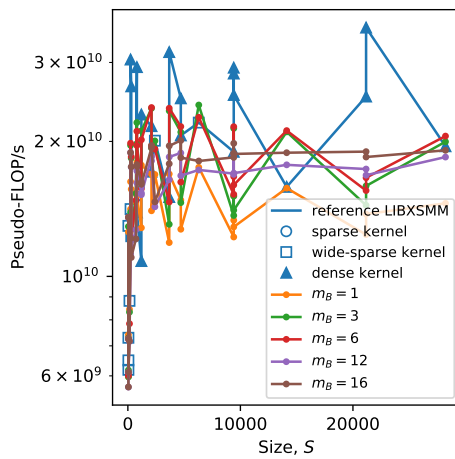
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.29: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on c5n.xlarge machine.

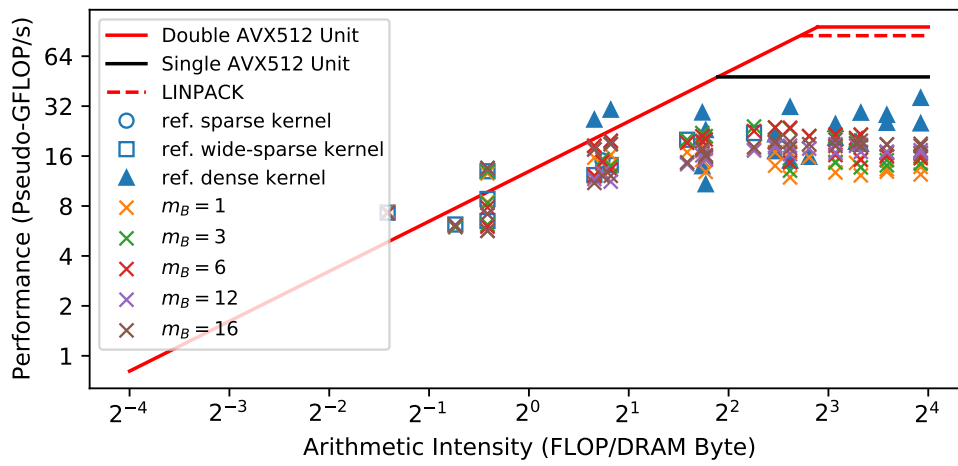
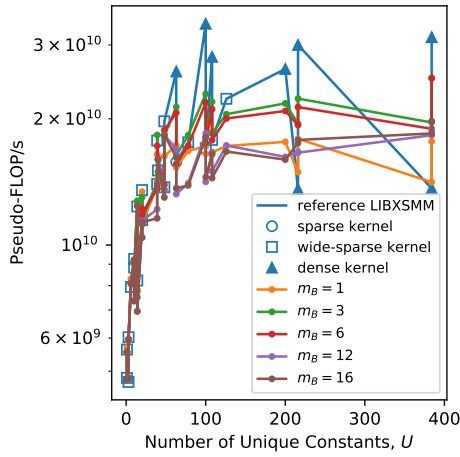
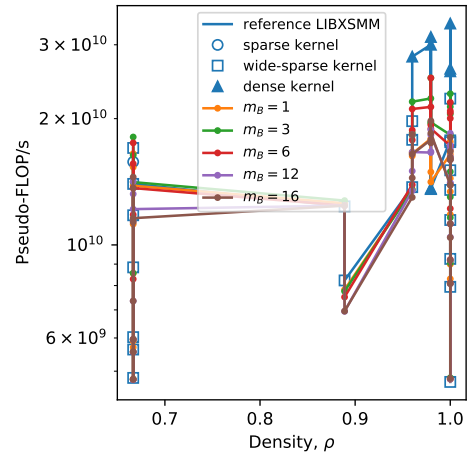


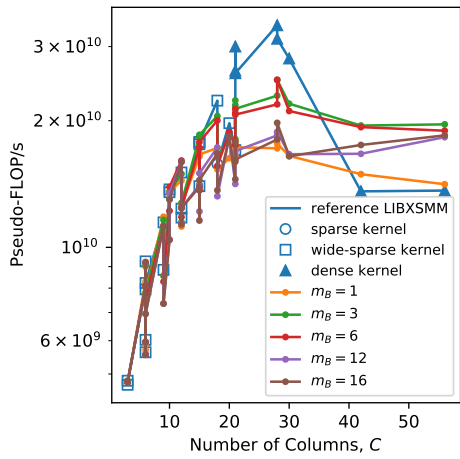
Figure D.30: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



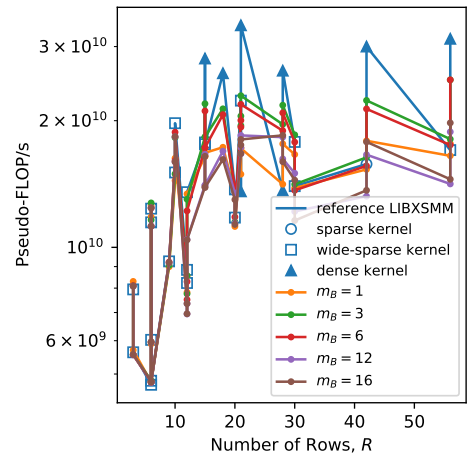
(a) Performance against number of unique \mathbf{A} constants.



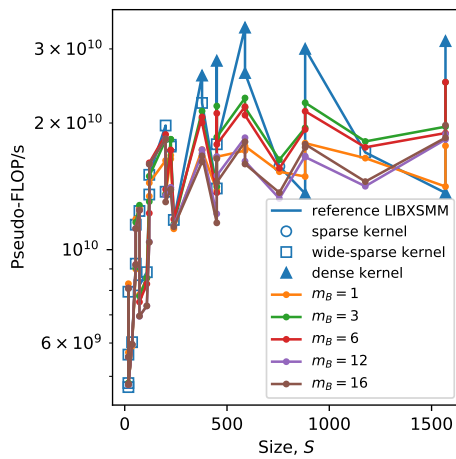
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.31: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on c5n.xlarge machine.

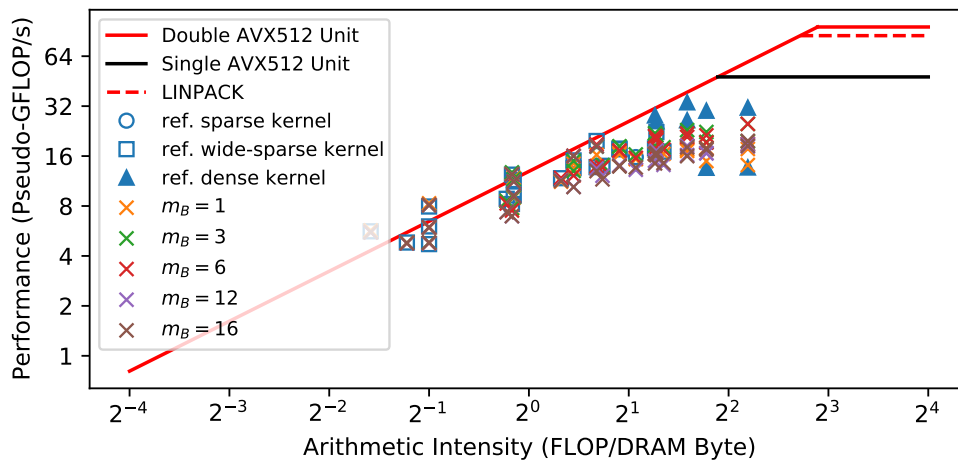
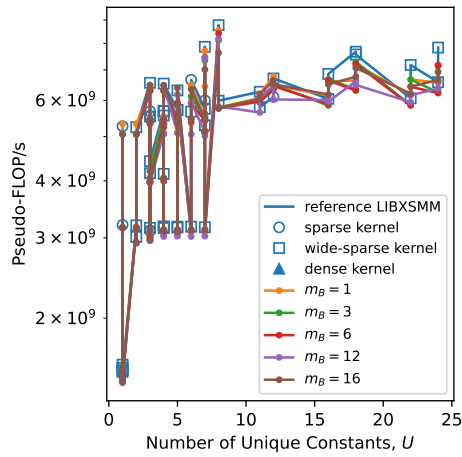
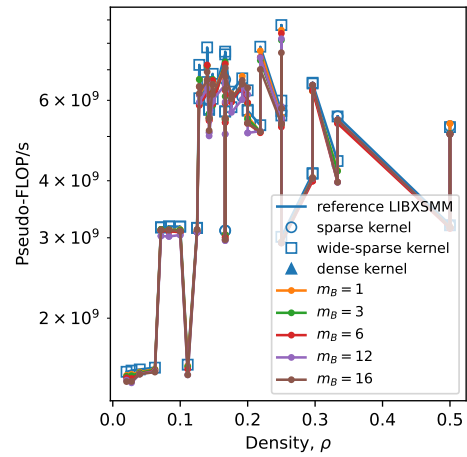


Figure D.32: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.

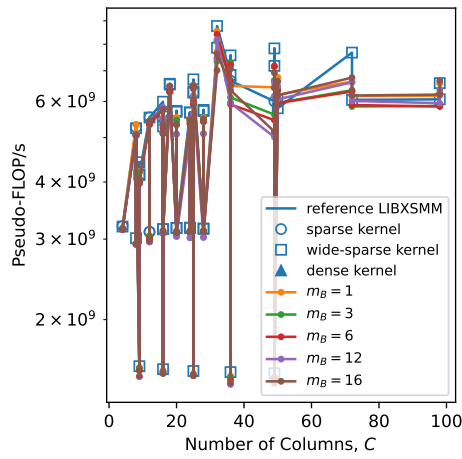
Benchmark Run on m5n.xlarge



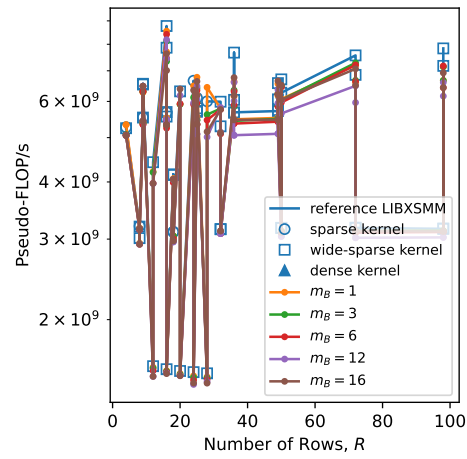
(a) Performance against number of unique \mathbf{A} constants.



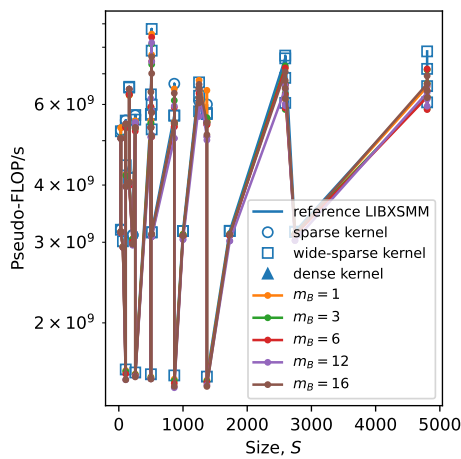
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.33: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral operator matrices. Benchmark run on m5n.xlarge machine.

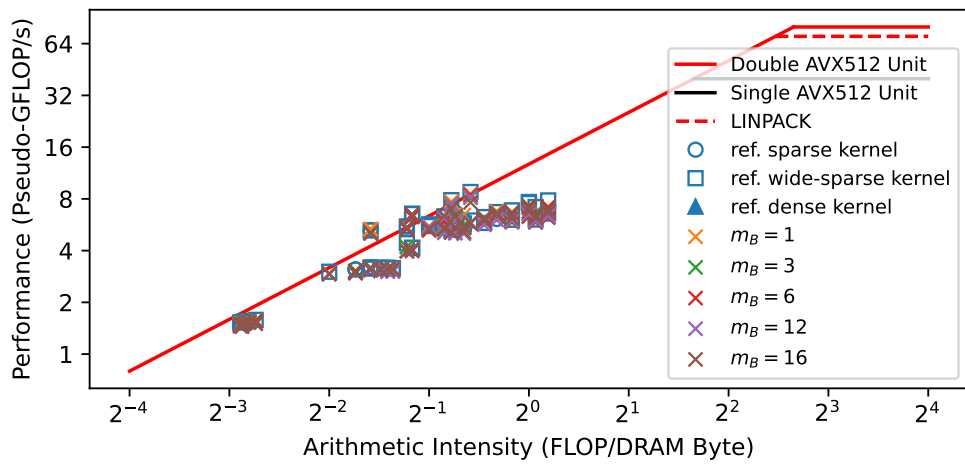
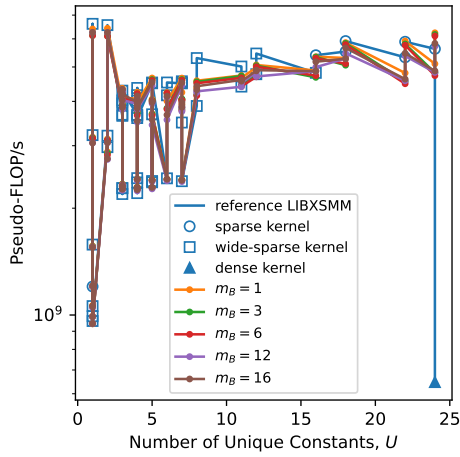
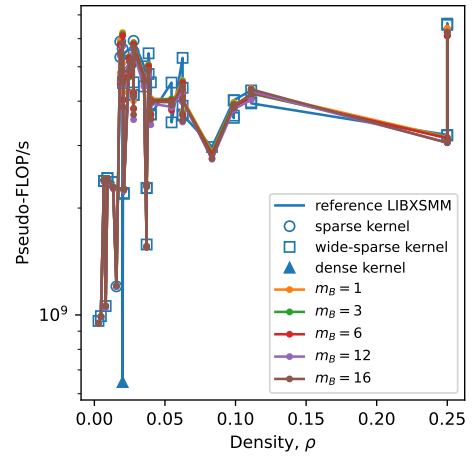


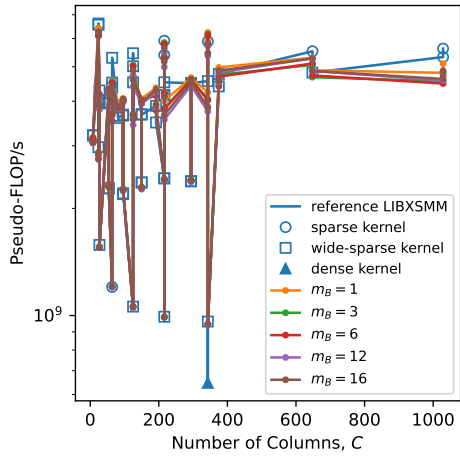
Figure D.34: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR quadrilateral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



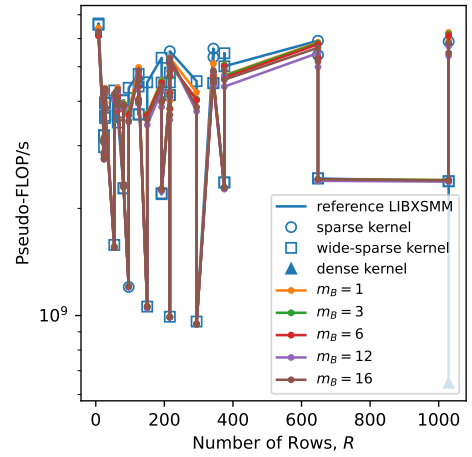
(a) Performance against number of unique \mathbf{A} constants.



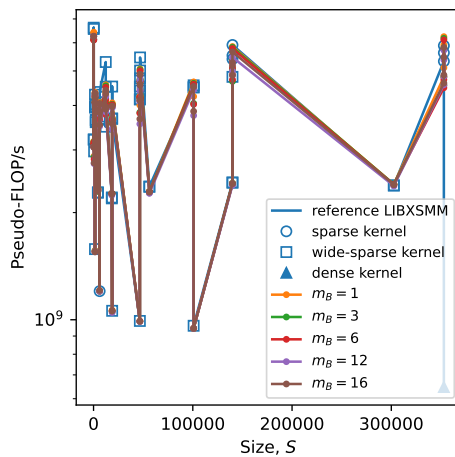
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.35: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR hexahedral operator matrices. Benchmark run on m5n.xlarge machine.

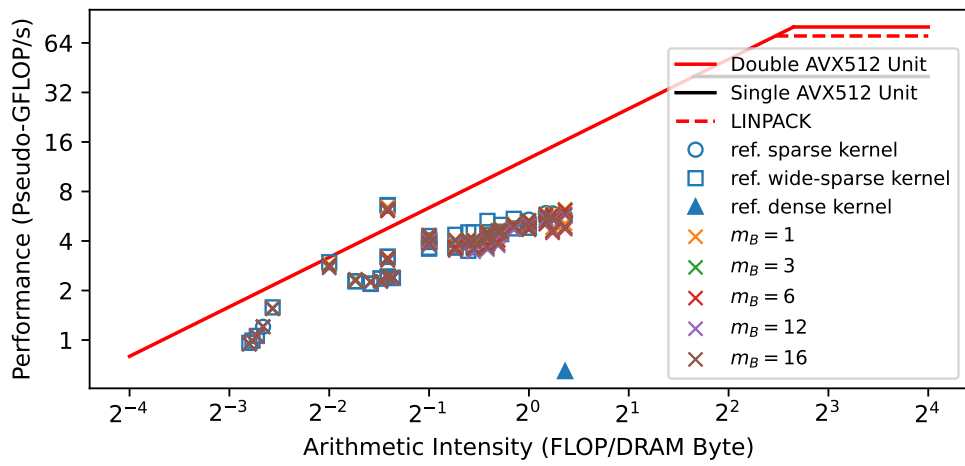
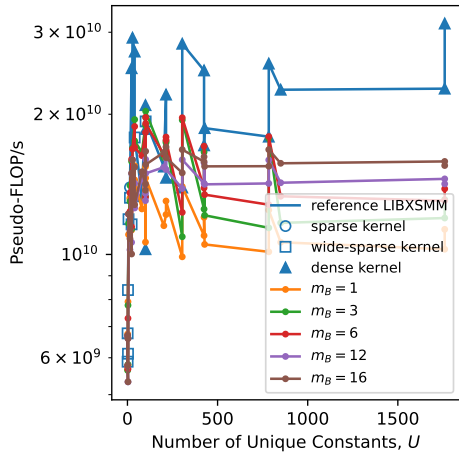
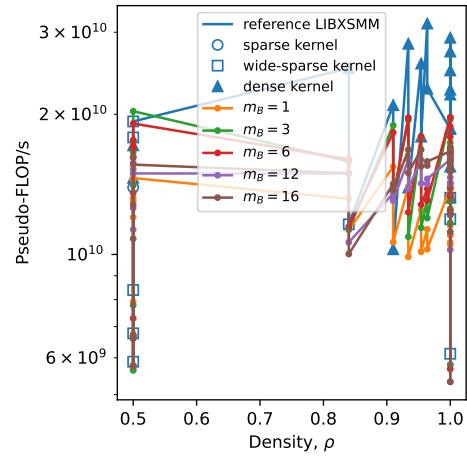


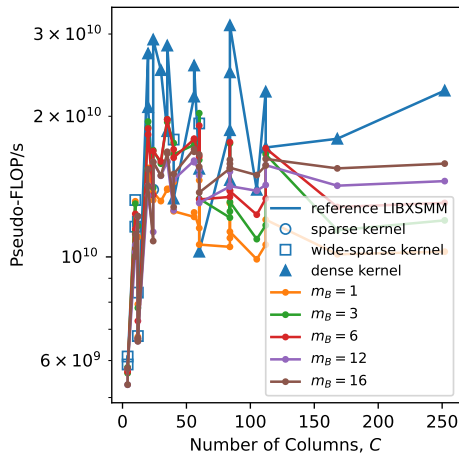
Figure D.36: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR hexahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



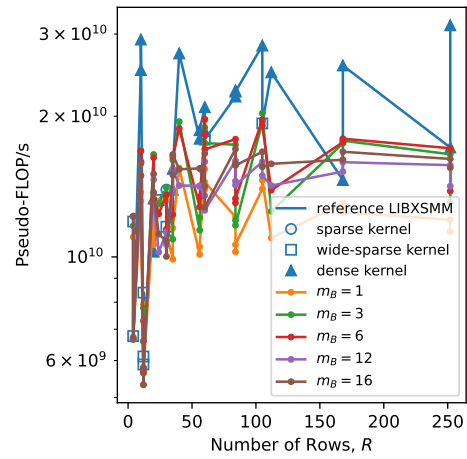
(a) Performance against number of unique \mathbf{A} constants.



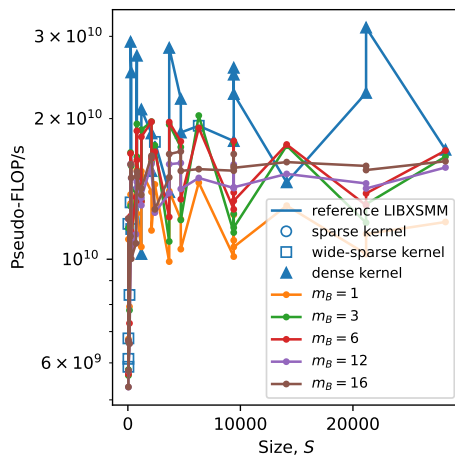
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.37: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral operator matrices. Benchmark run on m5n.xlarge machine.

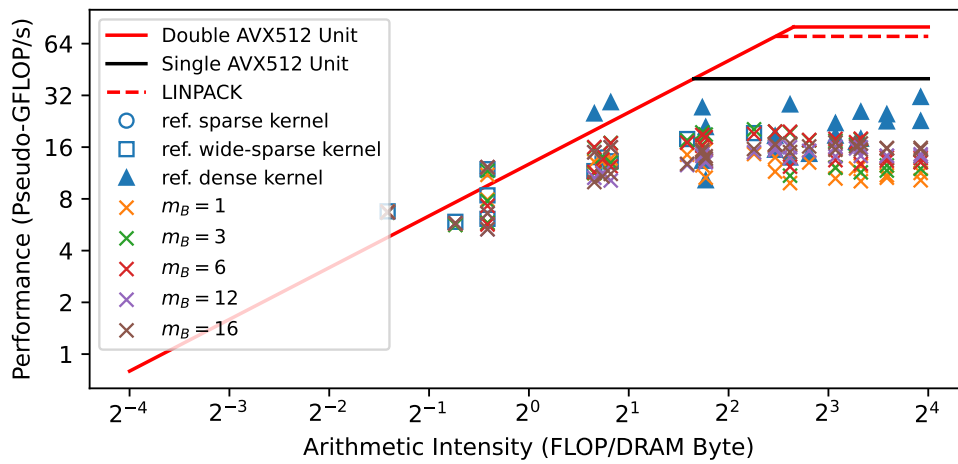
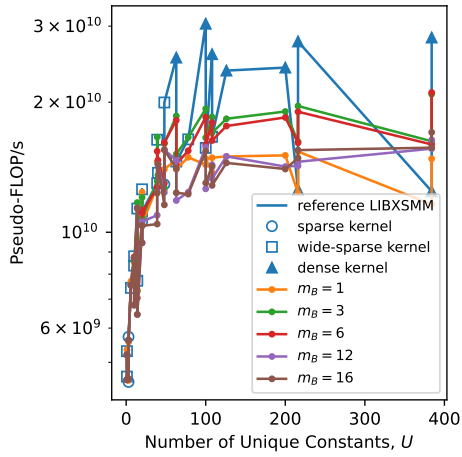
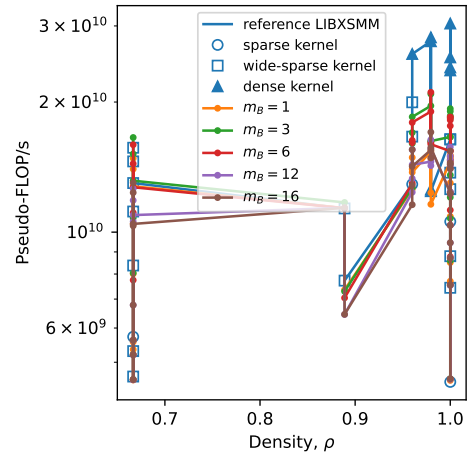


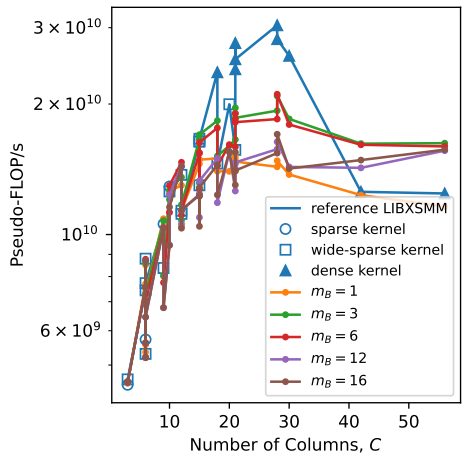
Figure D.38: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR tetrahedral operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



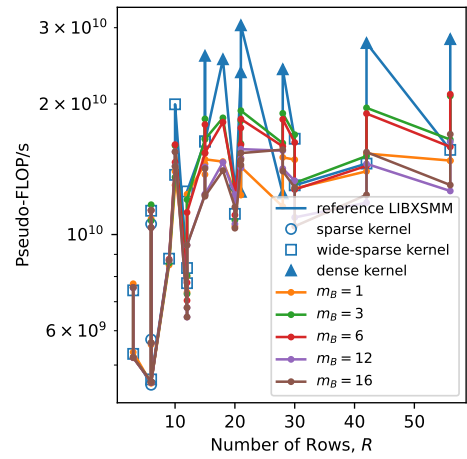
(a) Performance against number of unique \mathbf{A} constants.



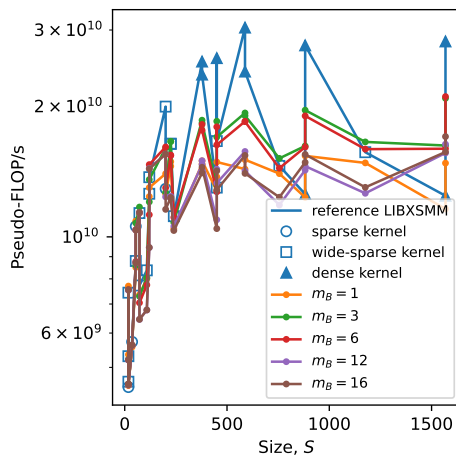
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure D.39: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for PyFR triangular operator matrices. Benchmark run on m5n.xlarge machine.

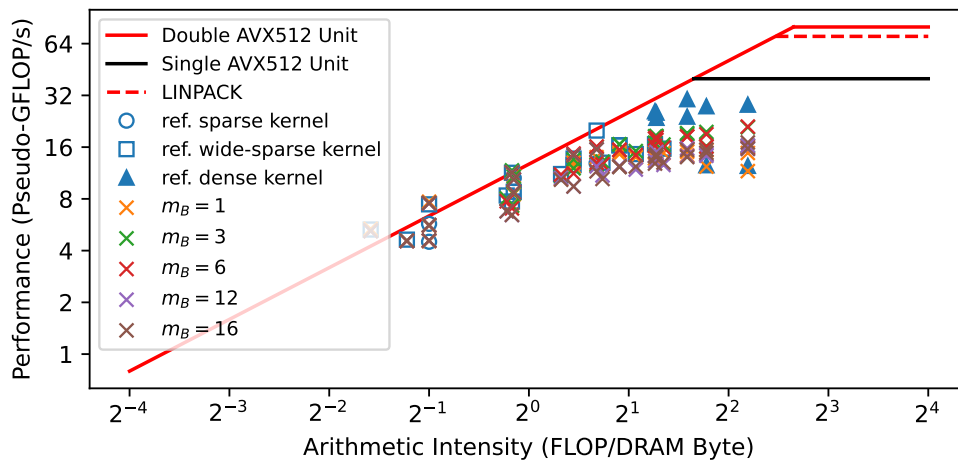
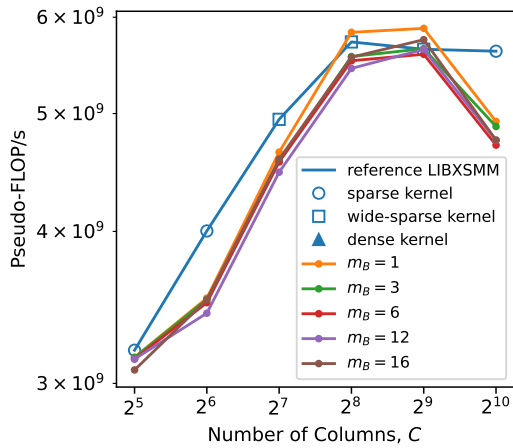


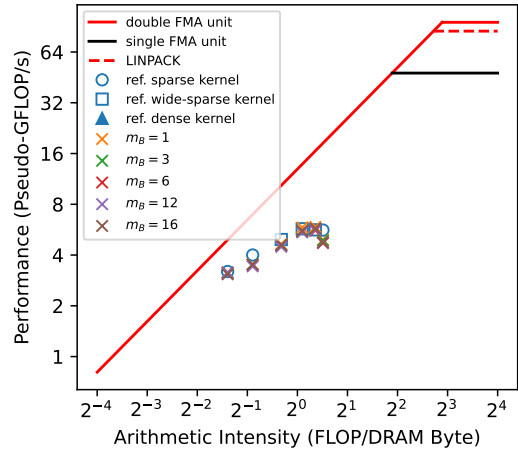
Figure D.40: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations for PyFR triangular operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.

Synthetic Matrices

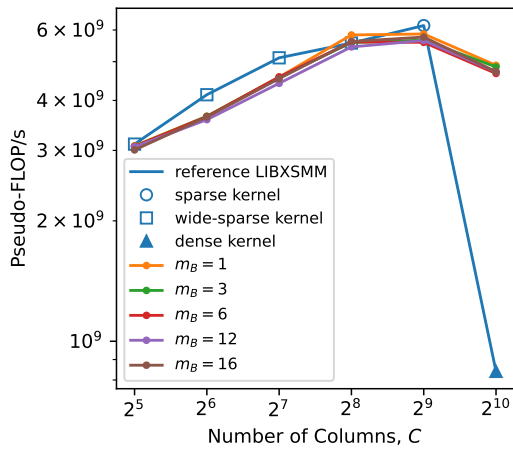
Benchmark Run on c5n.xlarge



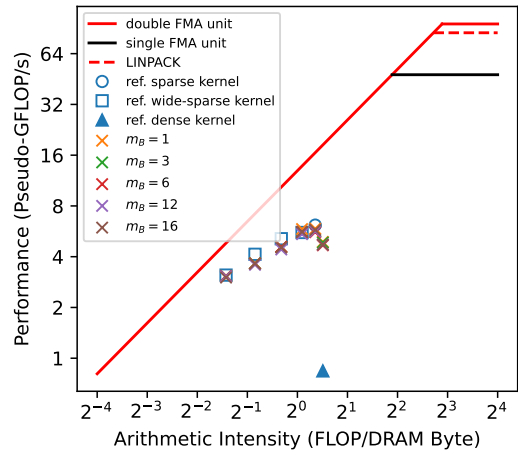
(a) Performance vs. number of columns, $U = 16$.



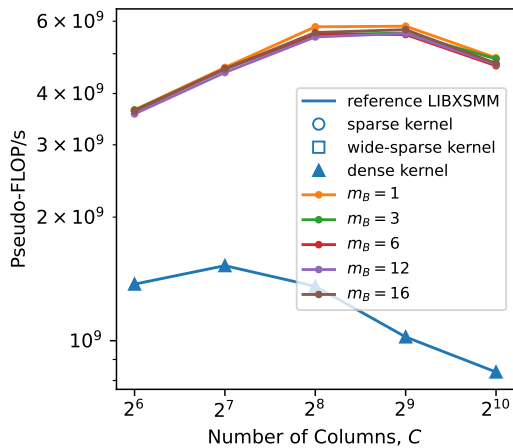
(b) Roofline plot, $U = 16$.



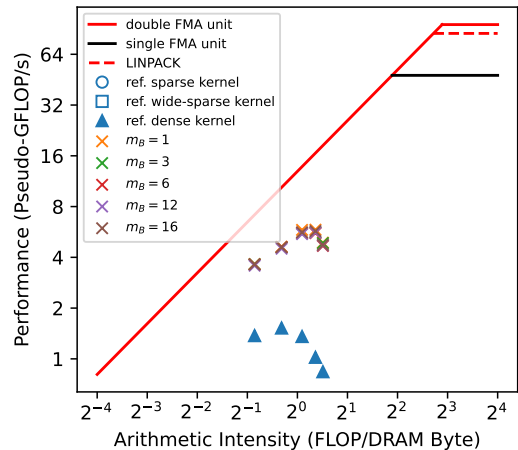
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

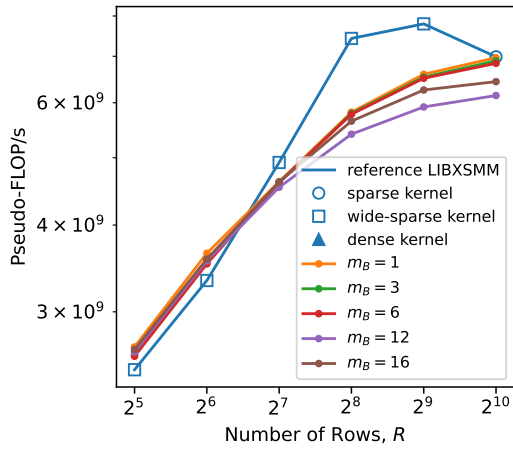


(e) Performance vs. number of columns, $U = 256$.

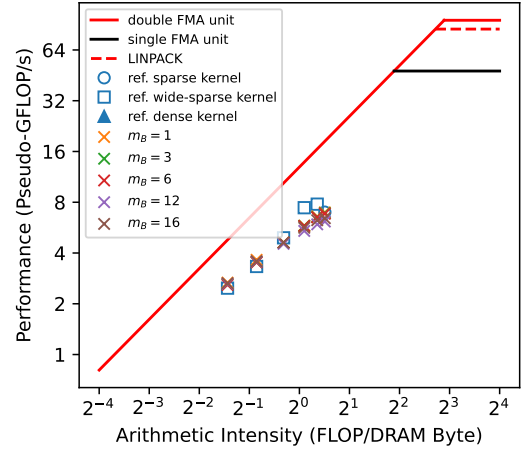


(f) Roofline plot, $U = 256$.

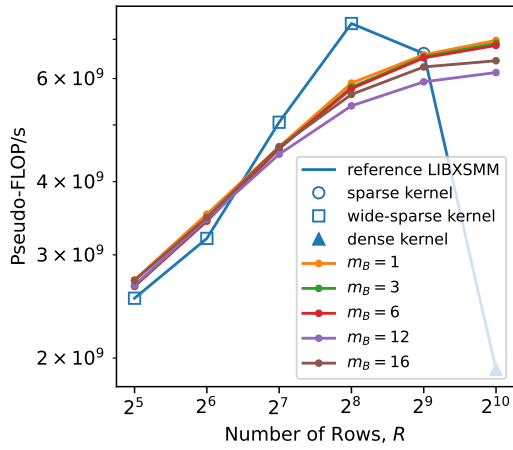
Figure D.41: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on c5n.xlarge machine.



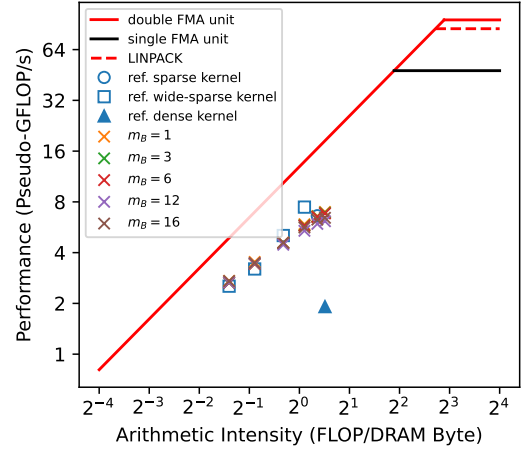
(a) Performance vs. number of rows, $U = 16$.



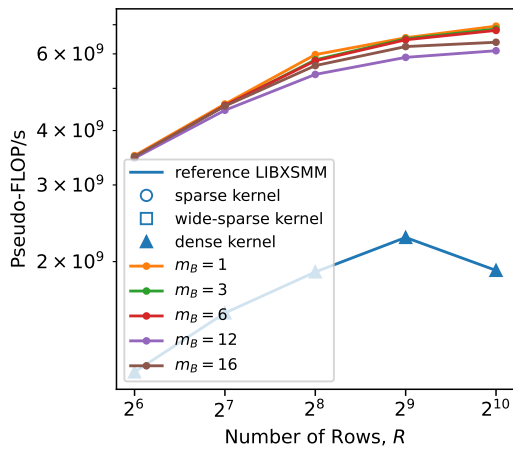
(b) Roofline plot, $U = 16$.



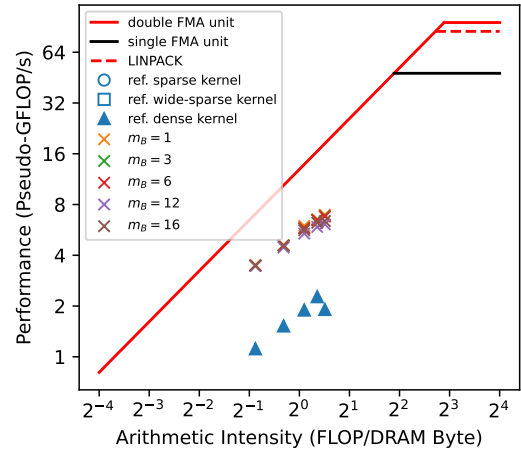
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

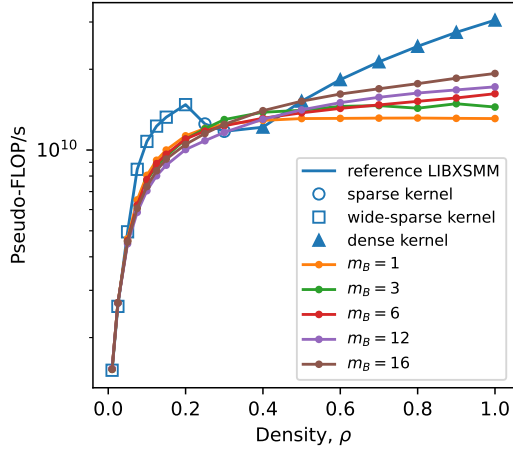


(e) Performance vs. number of rows, $U = 256$.

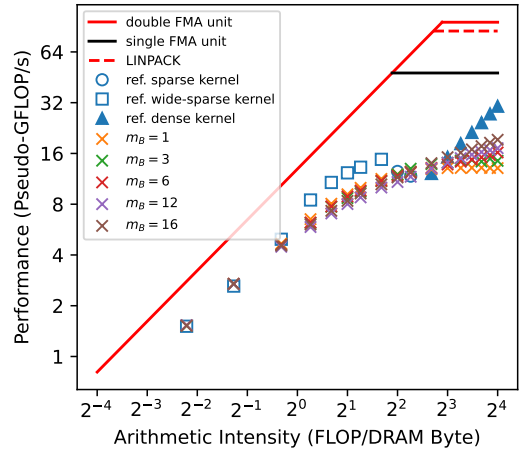


(f) Roofline plot, $U = 256$.

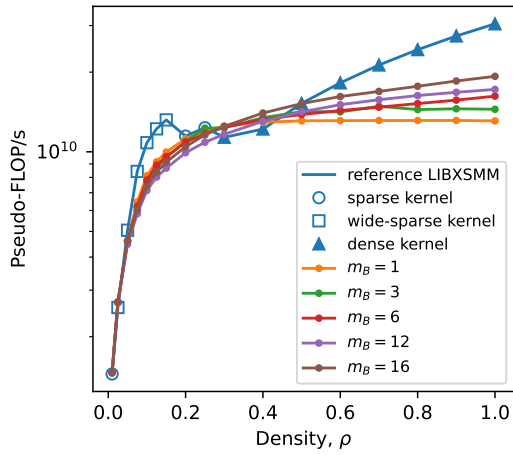
Figure D.42: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} row. Benchmark run on c5n.xlarge machine.



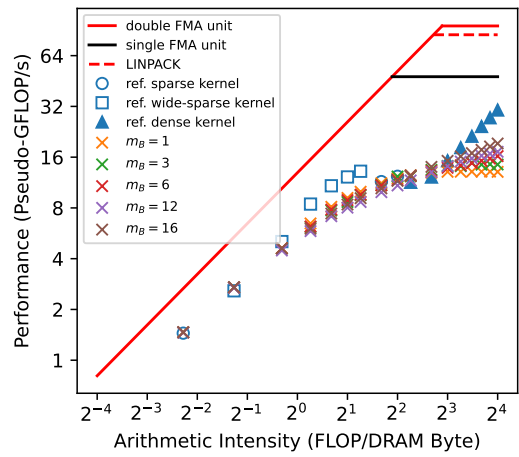
(a) Performance vs. density, $U = 16$.



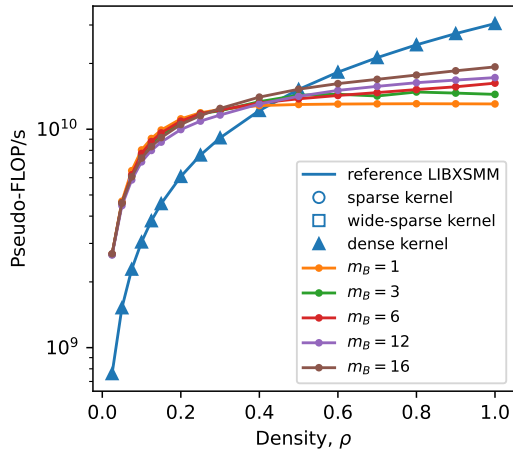
(b) Roofline plot, $U = 16$.



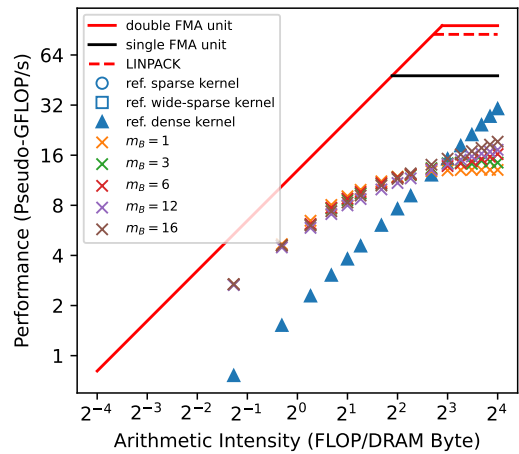
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

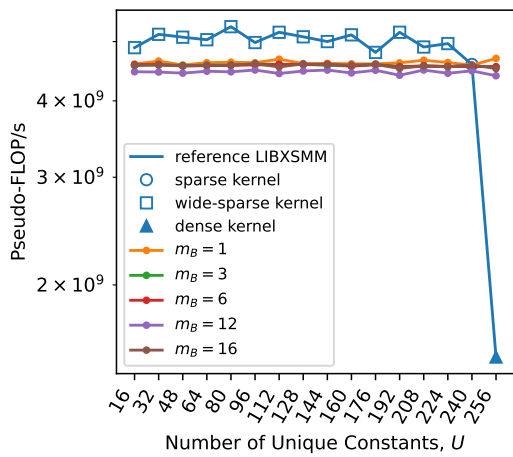


(e) Performance vs. density, $U = 256$.

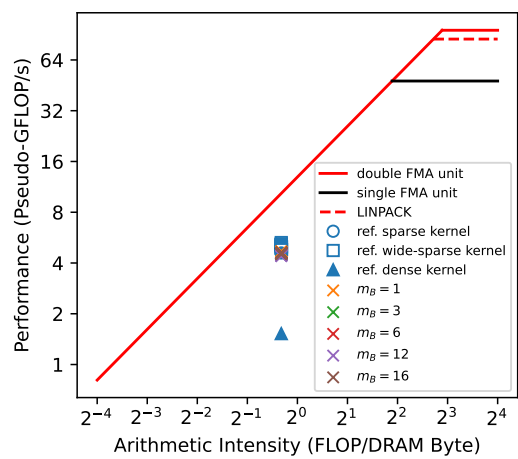


(f) Roofline plot, $U = 256$.

Figure D.43: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with \mathbf{A} density. Benchmark run on c5n.xlarge machine.



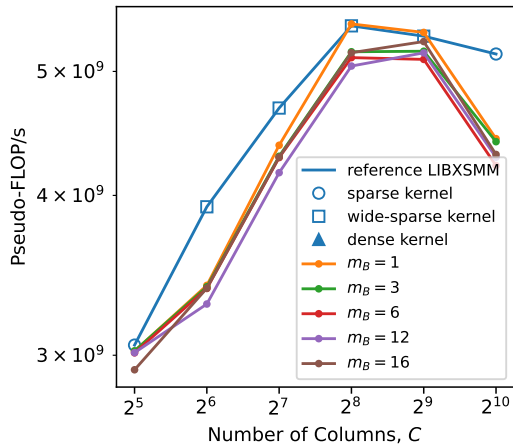
(a) Performance vs. number of absolute non-zero values.



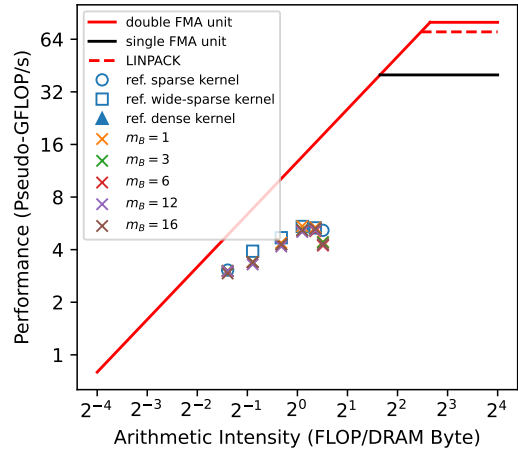
(b) Roofline plot.

Figure D.44: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine.

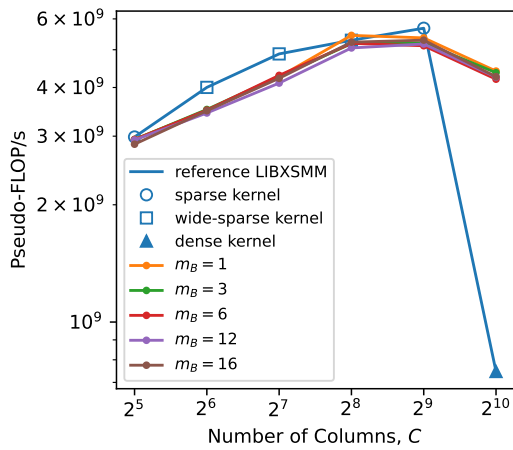
Benchmark Run on m5n.xlarge



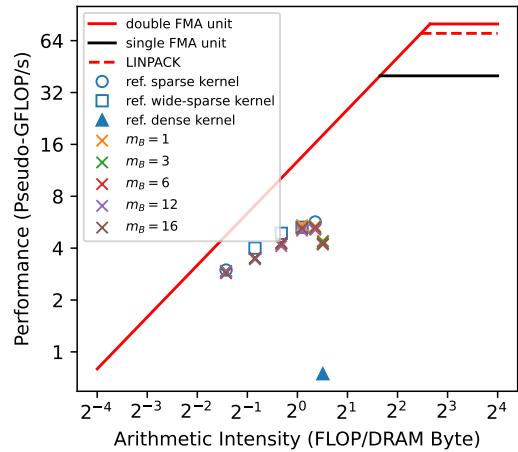
(a) Performance vs. number of columns, $U = 16$.



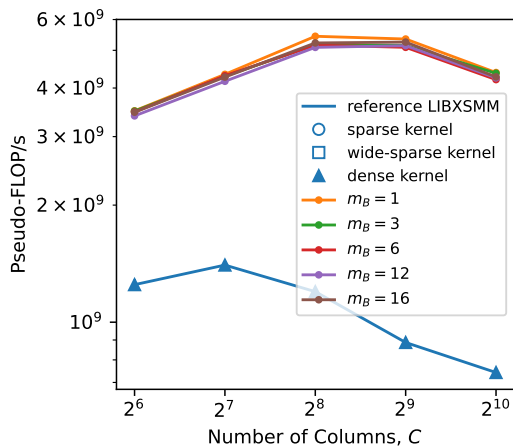
(b) Roofline plot, $U = 16$.



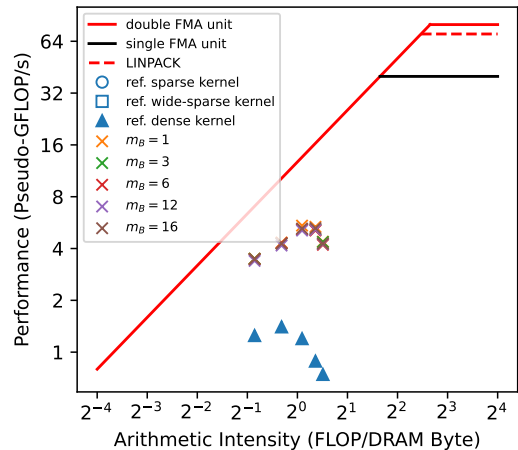
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

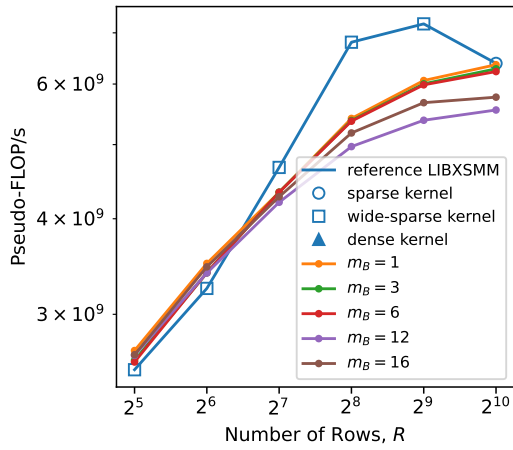


(e) Performance vs. number of columns, $U = 256$.

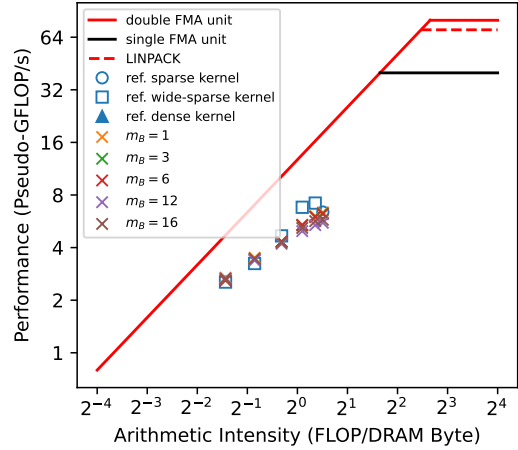


(f) Roofline plot, $U = 256$.

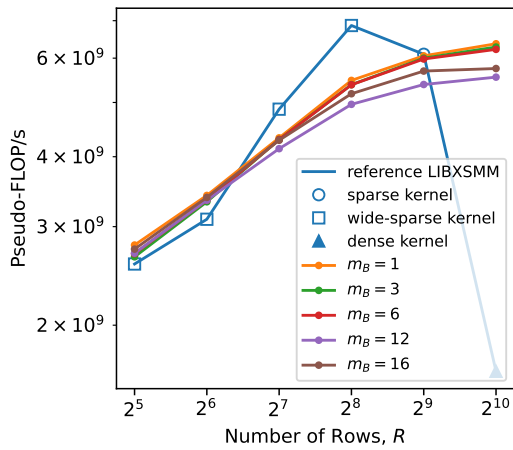
Figure D.45: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on m5n.xlarge machine.



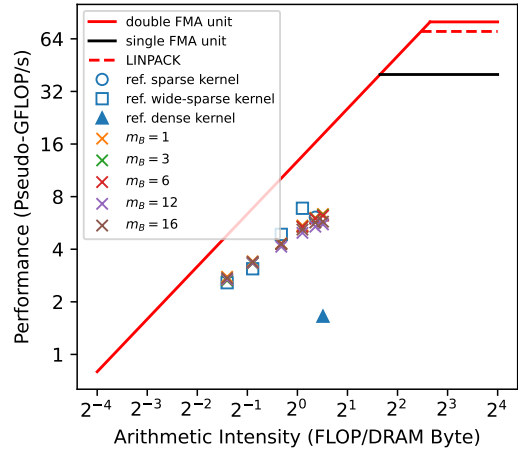
(a) Performance vs. number of rows, $U = 16$.



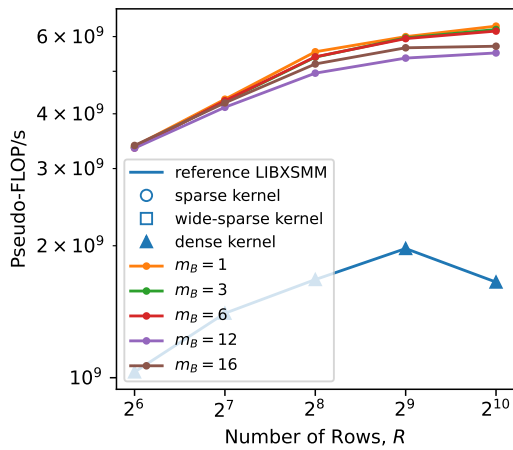
(b) Roofline plot, $U = 16$.



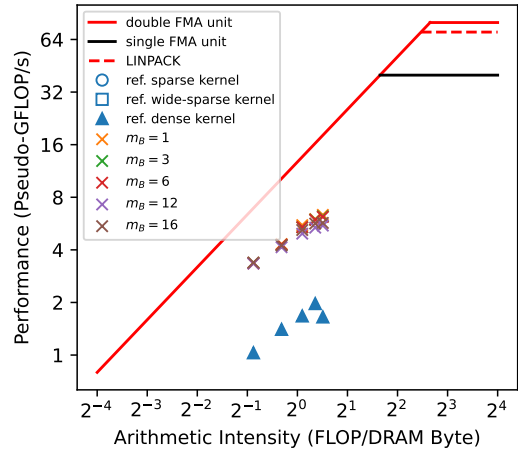
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

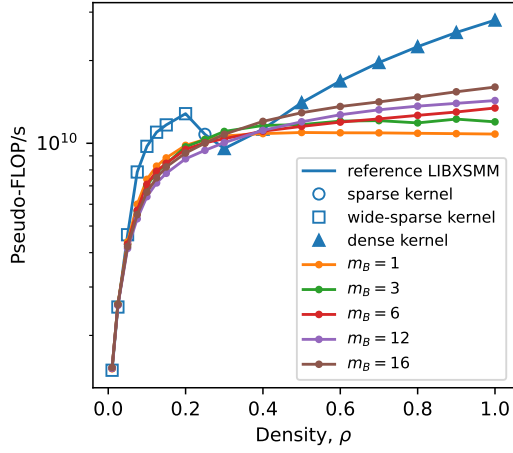


(e) Performance vs. number of rows, $U = 256$.

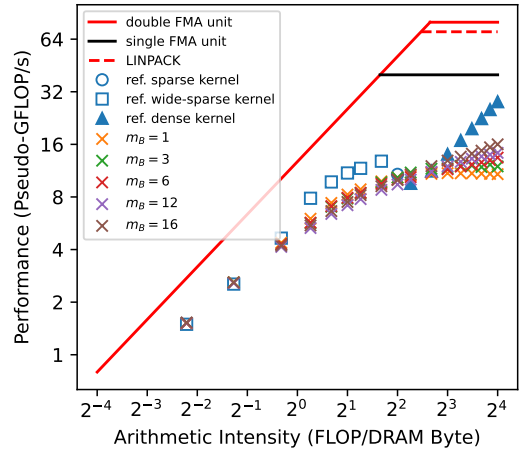


(f) Roofline plot, $U = 256$.

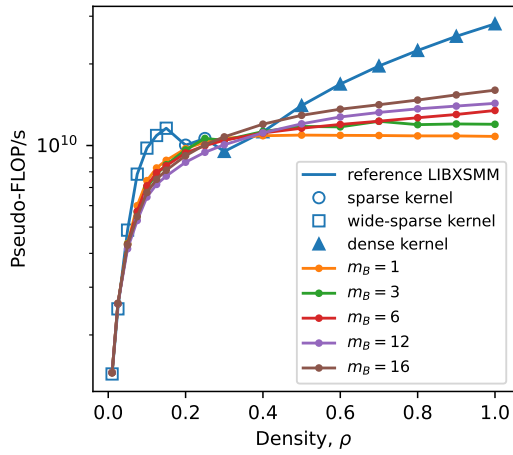
Figure D.46: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} row. Benchmark run on m5n.xlarge machine.



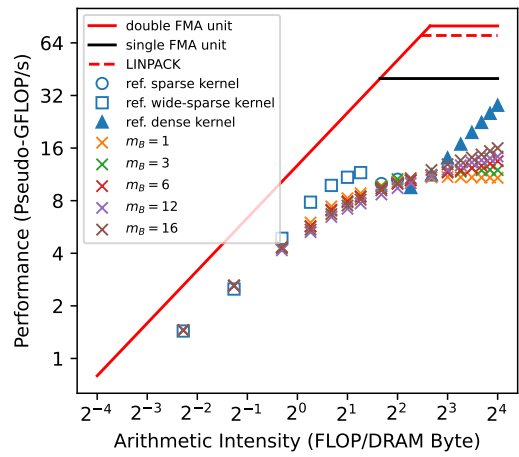
(a) Performance vs. density, $U = 16$.



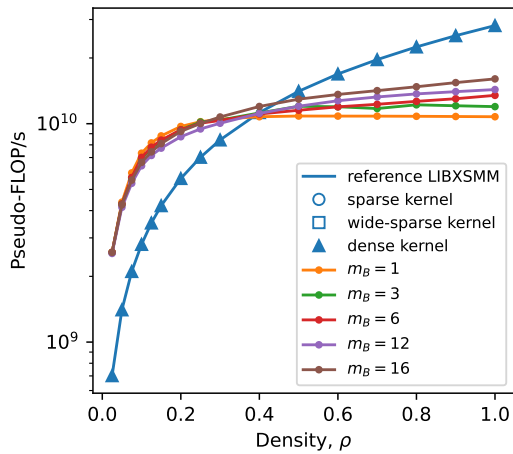
(b) Roofline plot, $U = 16$.



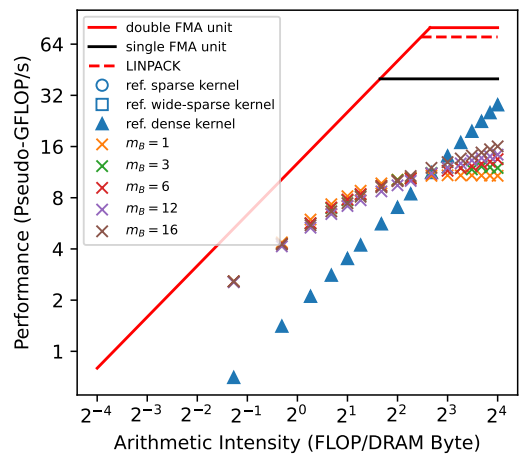
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

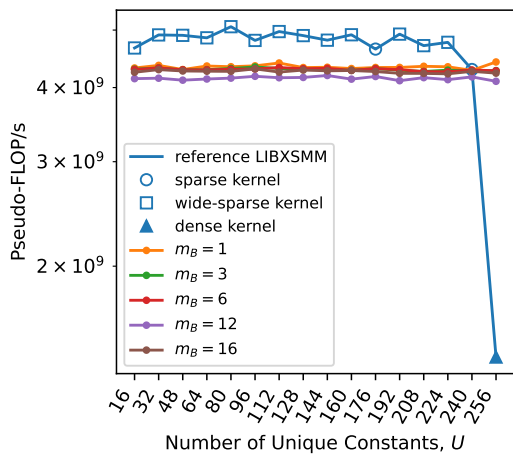


(e) Performance vs. density, $U = 256$.

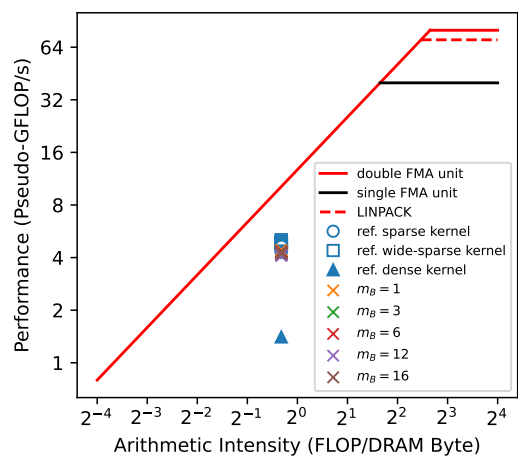


(f) Roofline plot, $U = 256$.

Figure D.47: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with \mathbf{A} density. Benchmark run on m5n.xlarge machine.



(a) Performance vs. number of absolute non-zero values.



(b) Roofline plot.

Figure D.48: Runtime broadcasting with loading \mathbf{A} from memory and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on m5n.xlarge machine.

Appendix E

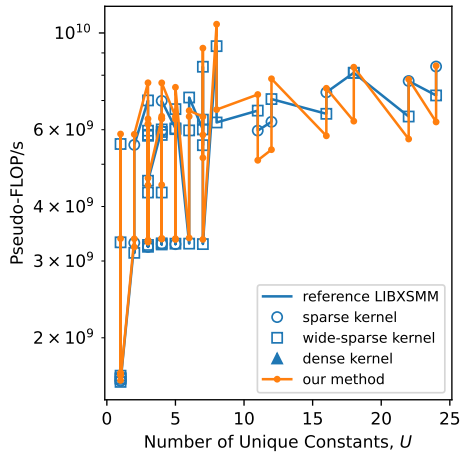
Complete Experiment Results for Testing the Small and Sparse GEMM Kernel with Runtime Broadcasting Packed A Constants from Memory and Caching B Strides in Vector Register

This appendix present the complete evaluation results for our GEMM routine which runtime broadcasts packed A constants from memory and caches B strides in the vector register. Chapter 7 details how this routine was implemented. As explained in Chapter 4, two sets of A matrices were used for the testing - a set of complete PyFR operator matrices, and a set of synthetic matrices. We evaluated our kernel on two testing machines - a Skylake machine (c5n.xlarge) and a Cascade Lake machine (m5n.xlarge).

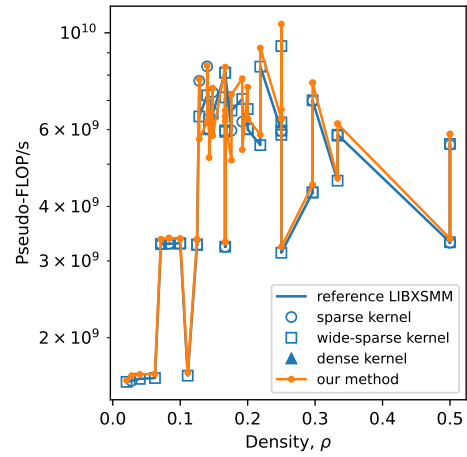
E.1 Single Accumulation

FyFR Operator Matrices

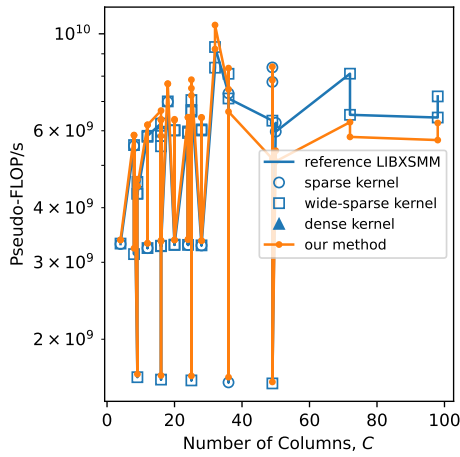
Benchmark Run on c5n.xlarge



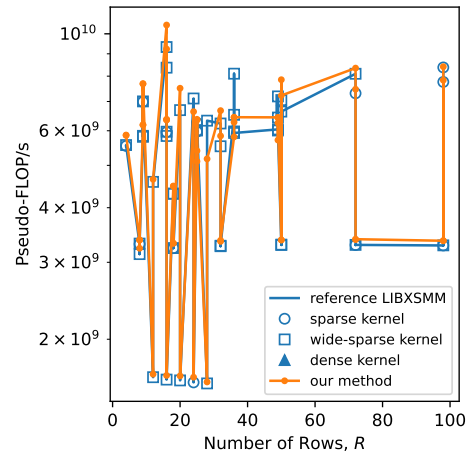
(a) Performance against number of unique \mathbf{A} constants.



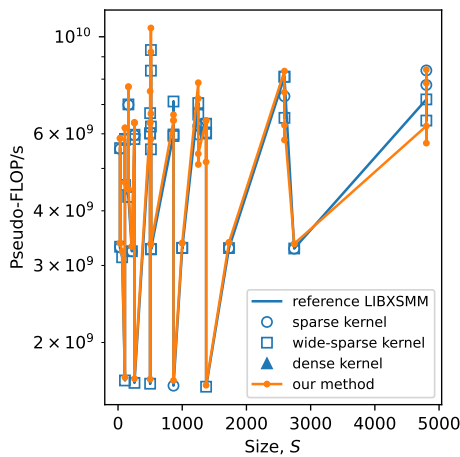
(b) Performance against \mathbf{A} density.



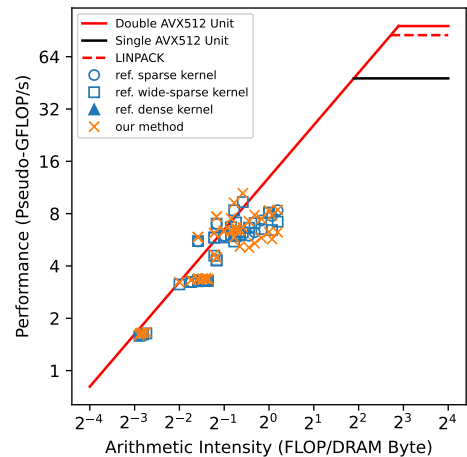
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

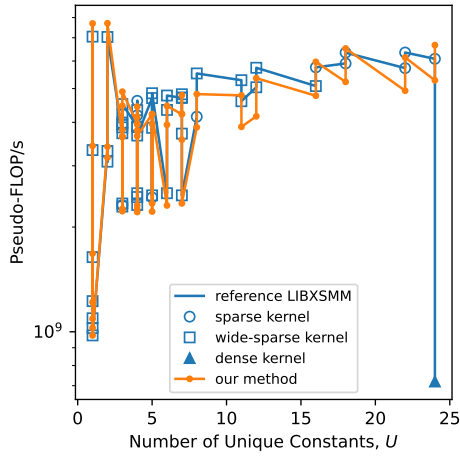


(e) Performance against number of \mathbf{A} size.

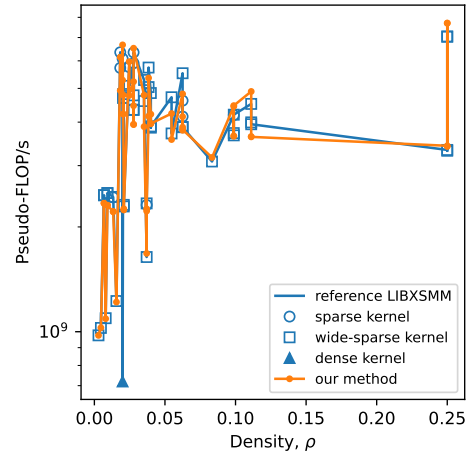


(f) Roofline plot.

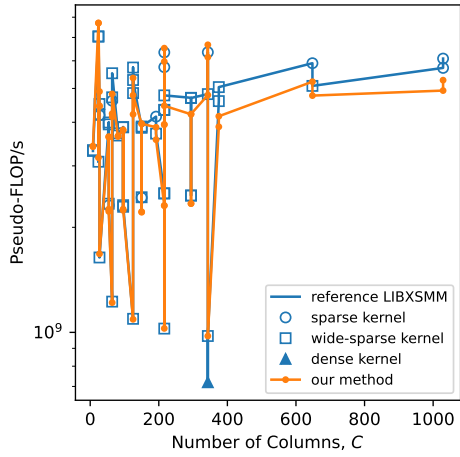
Figure E.1: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations, for PyFR quadrilateral element matrices. Benchmark run on c5n.xlarge machine.



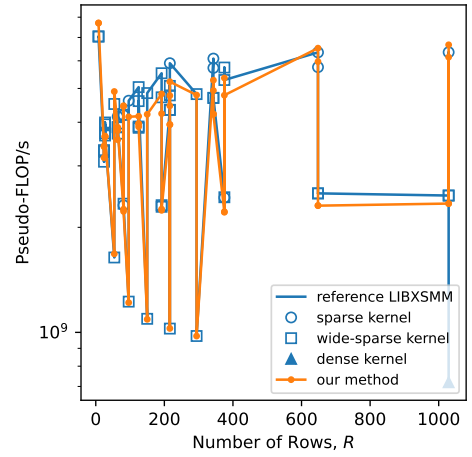
(a) Performance against number of unique \mathbf{A} constants.



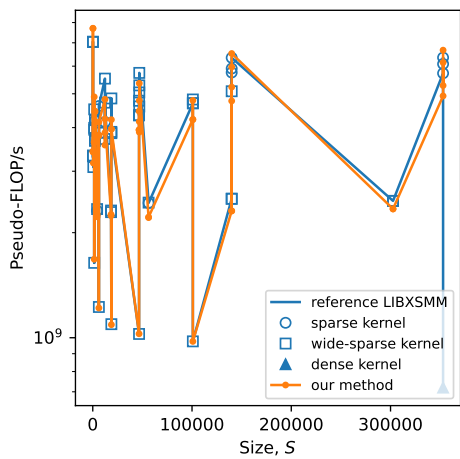
(b) Performance against \mathbf{A} density.



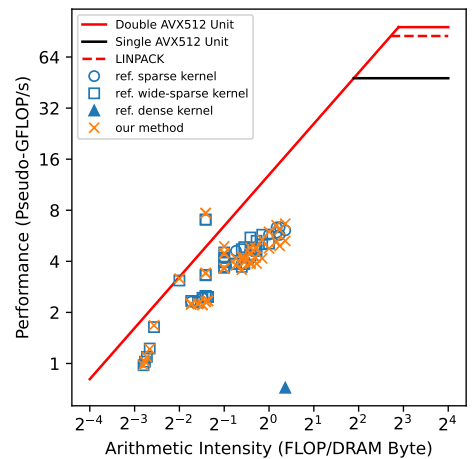
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

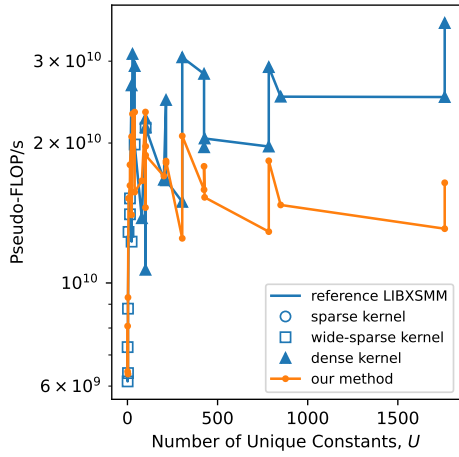


(e) Performance against number of \mathbf{A} size.

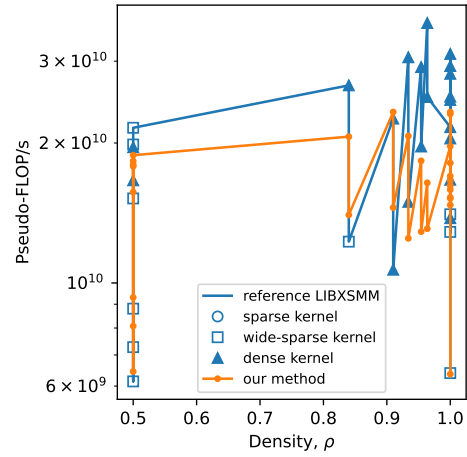


(f) Roofline plot.

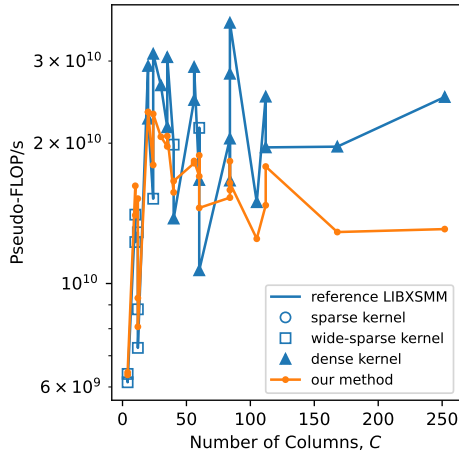
Figure E.2: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations, for PyFR hexahedral element matrices. Benchmark run on c5n.xlarge machine.



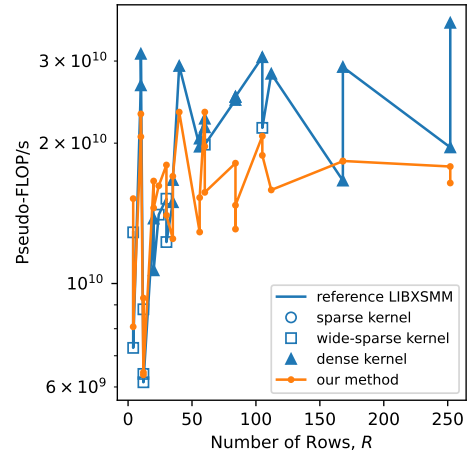
(a) Performance against number of unique A constants.



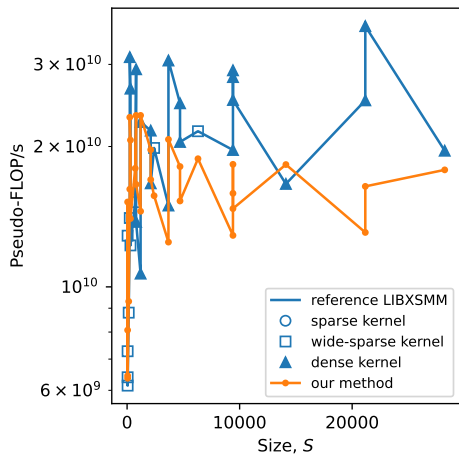
(b) Performance against A density.



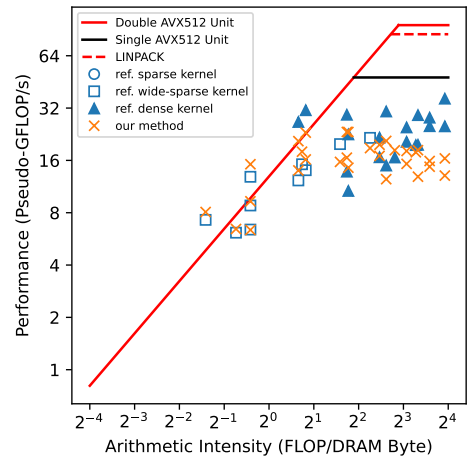
(c) Performance against number of A columns.



(d) Performance against number of A rows.

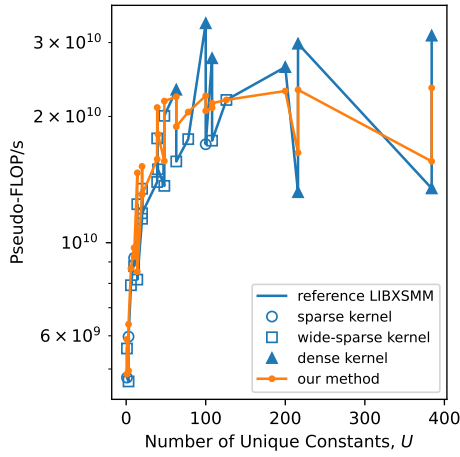


(e) Performance against number of A size.

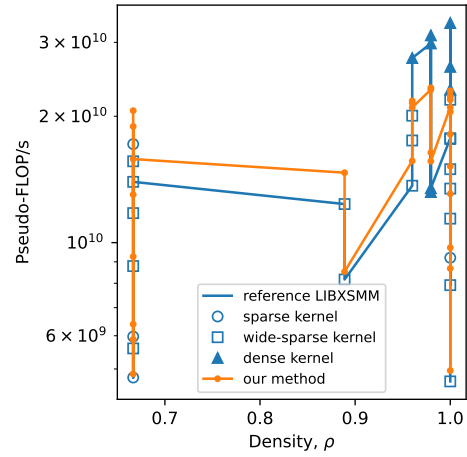


(f) Roofline plot.

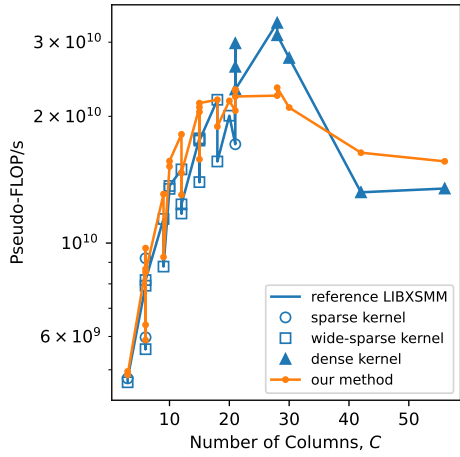
Figure E.3: Runtime broadcasting with loading A from memory and caching B strides in vector registers vs. reference LIBXSMM implementations, for PyFR tetrahedral element matrices. Benchmark run on c5n.xlarge machine.



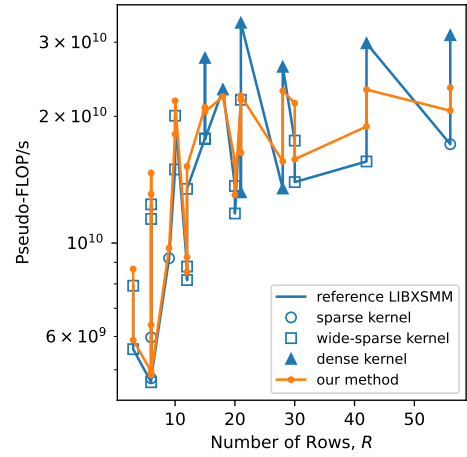
(a) Performance against number of unique A constants.



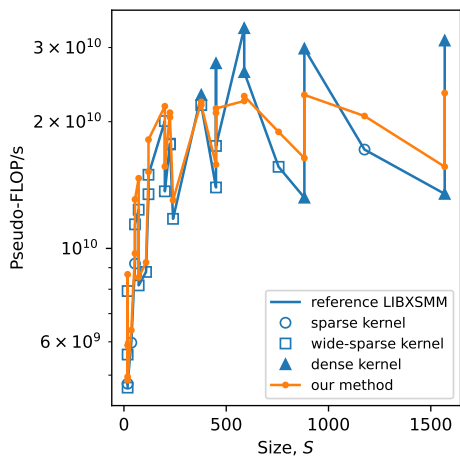
(b) Performance against A density.



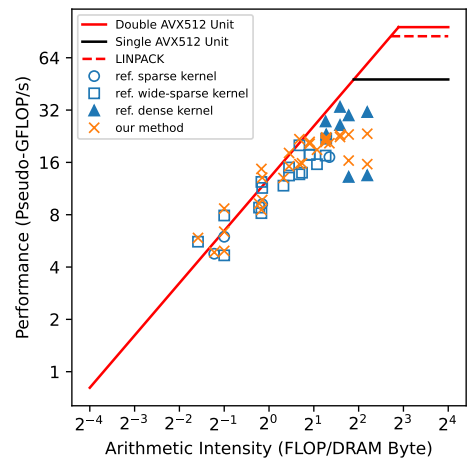
(c) Performance against number of A columns.



(d) Performance against number of A rows.



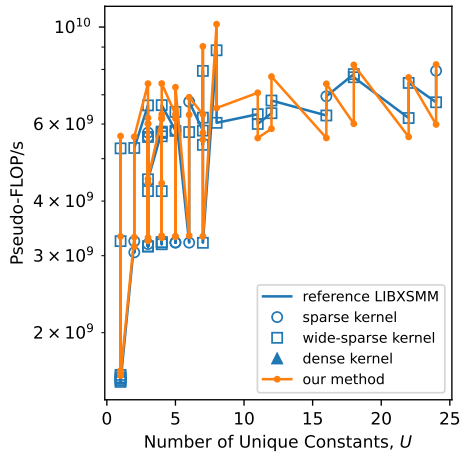
(e) Performance against number of A size.



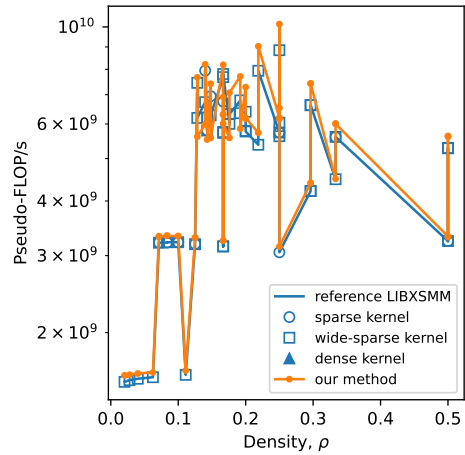
(f) Roofline plot.

Figure E.4: Runtime broadcasting with loading A from memory and caching B strides in vector registers vs. reference LIBXSMM implementations, for PyFR triangular element matrices. Benchmark run on c5n.xlarge machine.

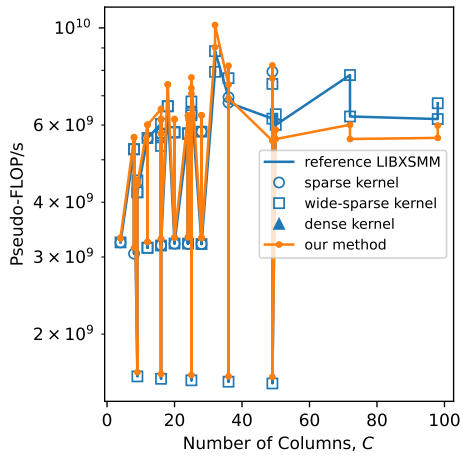
Benchmark Run on m5n.xlarge



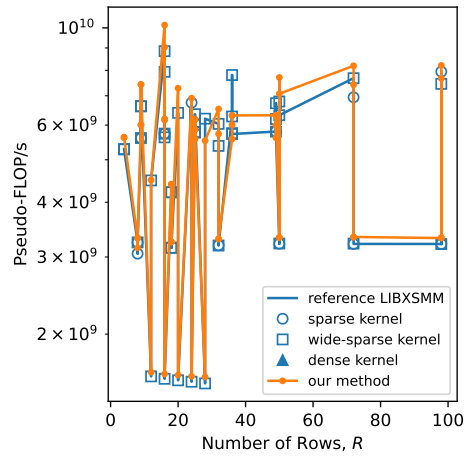
(a) Performance against number of unique \mathbf{A} constants.



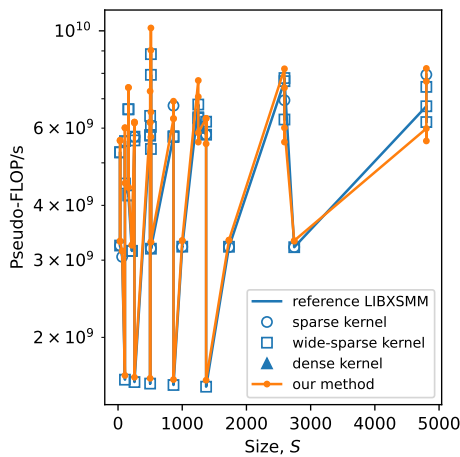
(b) Performance against \mathbf{A} density.



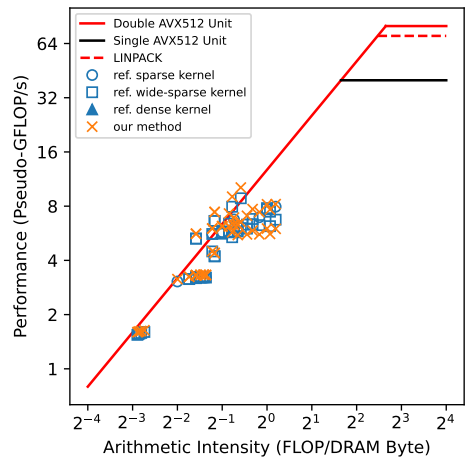
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

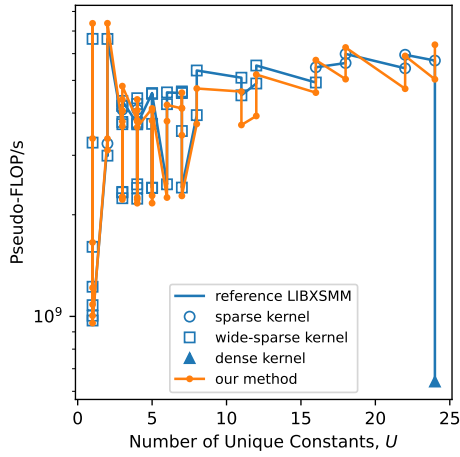


(e) Performance against number of \mathbf{A} size.

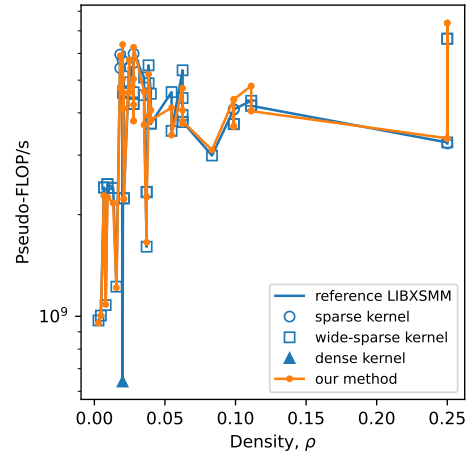


(f) Roofline plot.

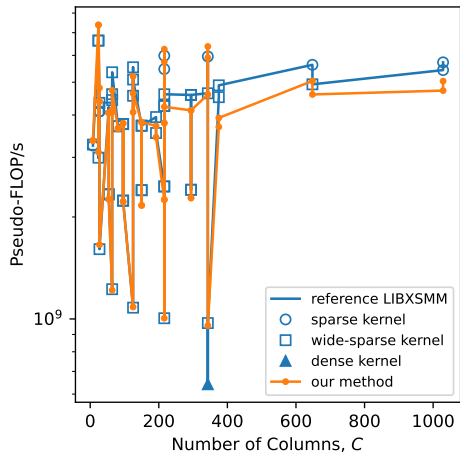
Figure E.5: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations, for PyFR quadrilateral element matrices. Benchmark run on m5n.xlarge machine.



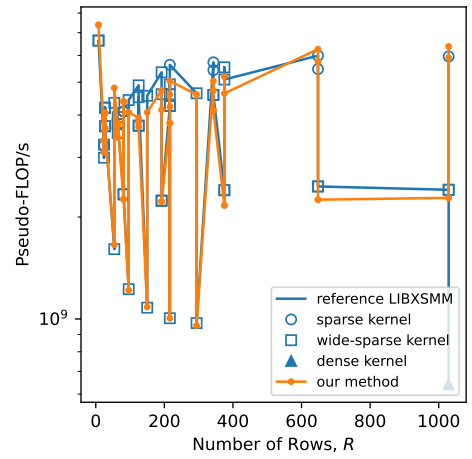
(a) Performance against number of unique A constants.



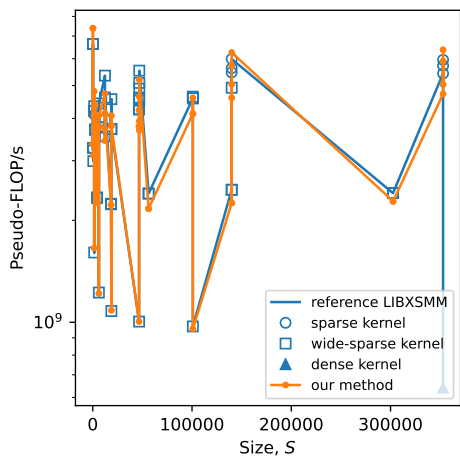
(b) Performance against A density.



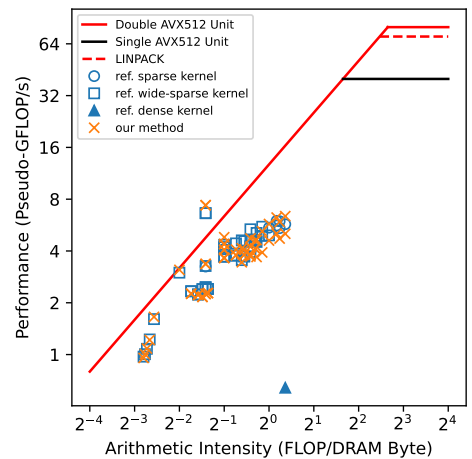
(c) Performance against number of A columns.



(d) Performance against number of A rows.

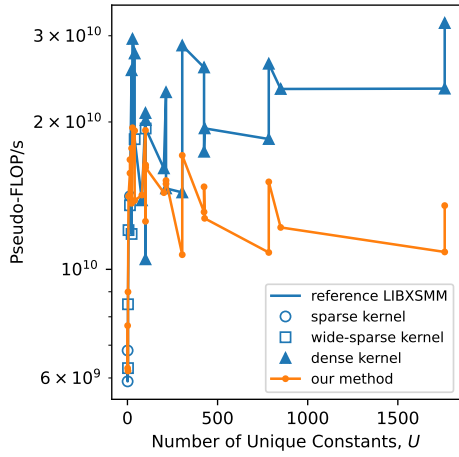


(e) Performance against number of A size.

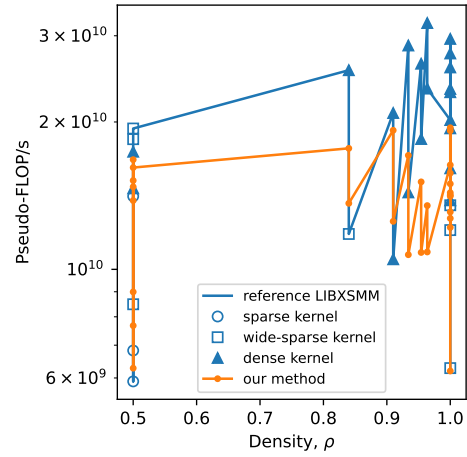


(f) Roofline plot.

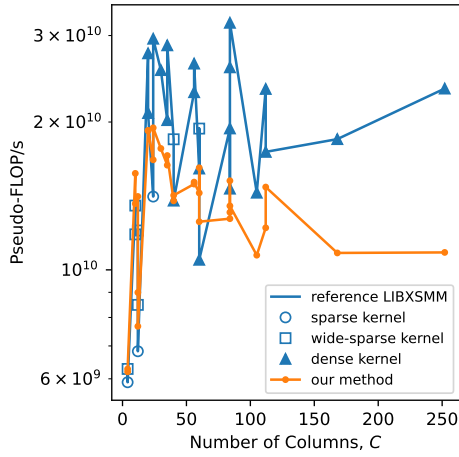
Figure E.6: Runtime broadcasting with loading A from memory and caching B strides in vector registers vs. reference LIBXSMM implementations, for PyFR hexahedral element matrices. Benchmark run on m5n.xlarge machine.



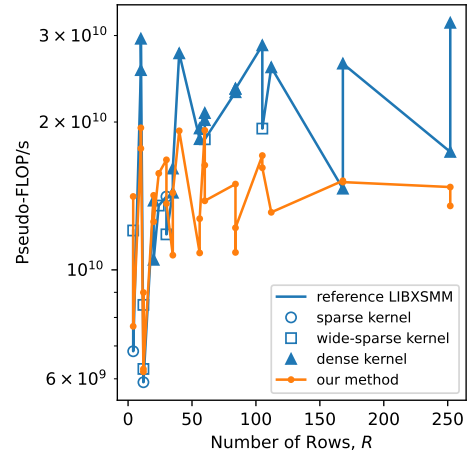
(a) Performance against number of unique A constants.



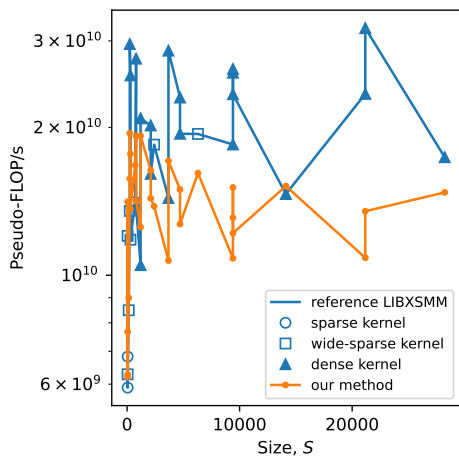
(b) Performance against A density.



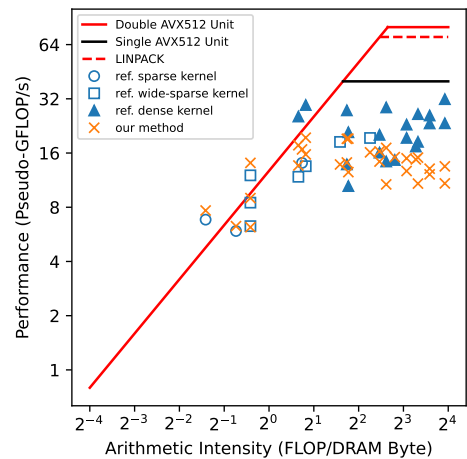
(c) Performance against number of A columns.



(d) Performance against number of A rows.

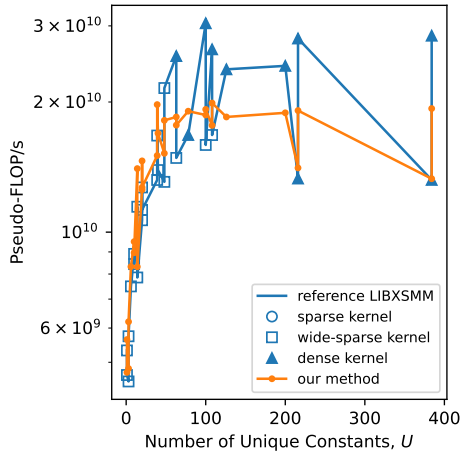


(e) Performance against number of A size.

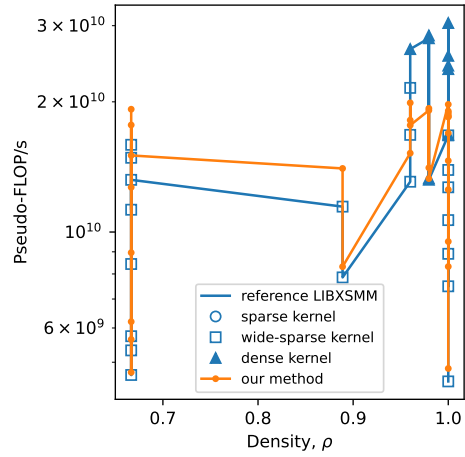


(f) Roofline plot.

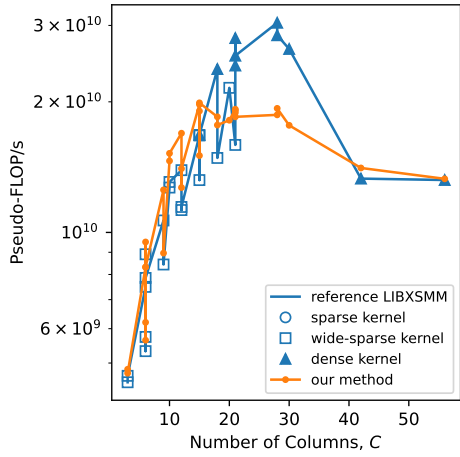
Figure E.7: Runtime broadcasting with loading A from memory and caching B strides in vector registers vs. reference LIBXSMM implementations, for PyFR tetrahedral element matrices. Benchmark run on m5n.xlarge machine.



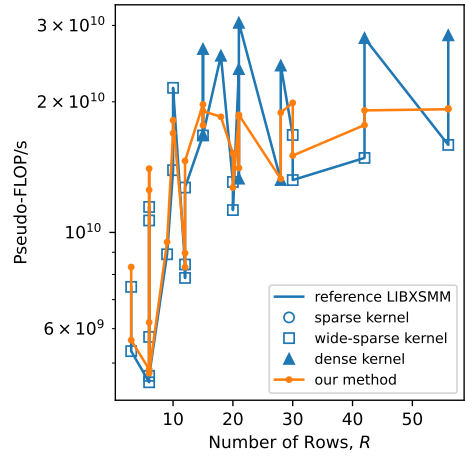
(a) Performance against number of unique \mathbf{A} constants.



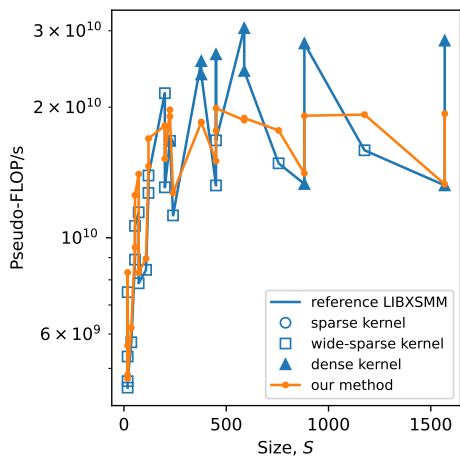
(b) Performance against \mathbf{A} density.



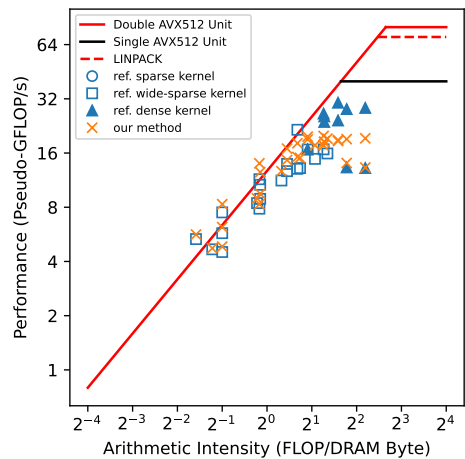
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

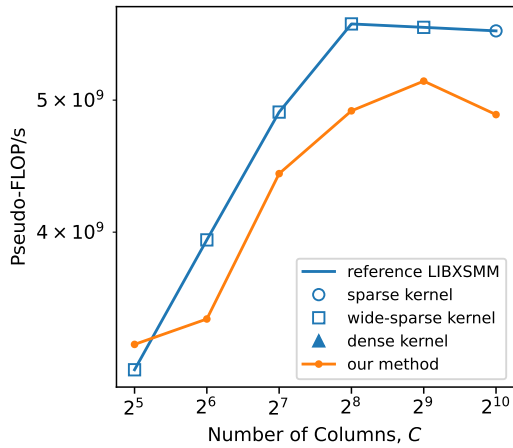


(f) Roofline plot.

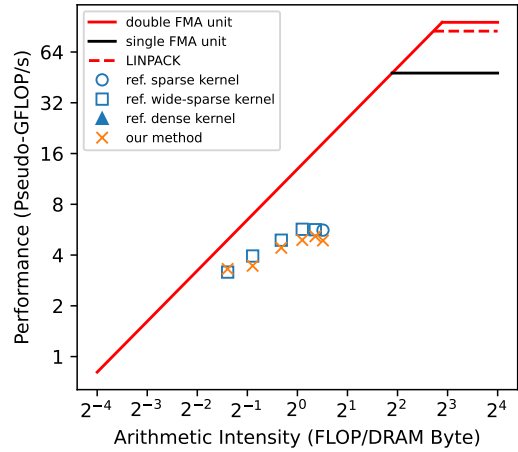
Figure E.8: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides in vector registers vs. reference LIBXSMM implementations, for PyFR triangular element matrices. Benchmark run on m5n.xlarge machine.

Synthetic Matrices

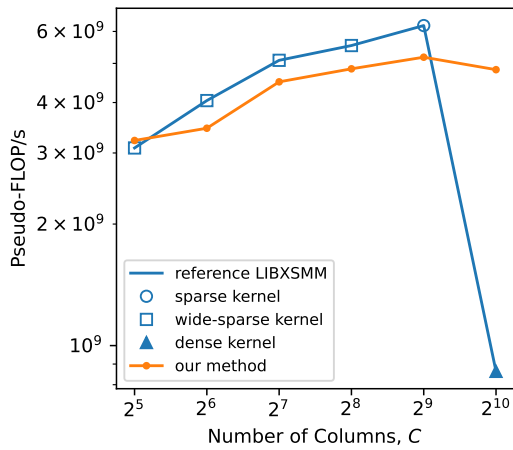
Benchmark Run on c5n.xlarge



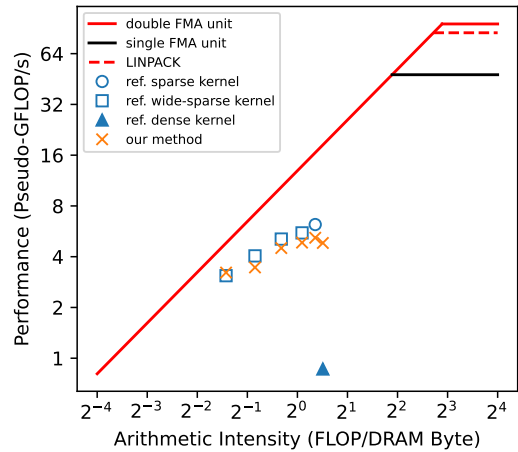
(a) Performance vs. number of columns, $U = 16$.



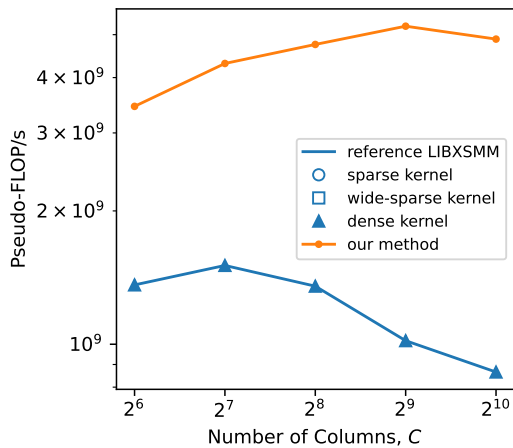
(b) Roofline plot, $U = 16$.



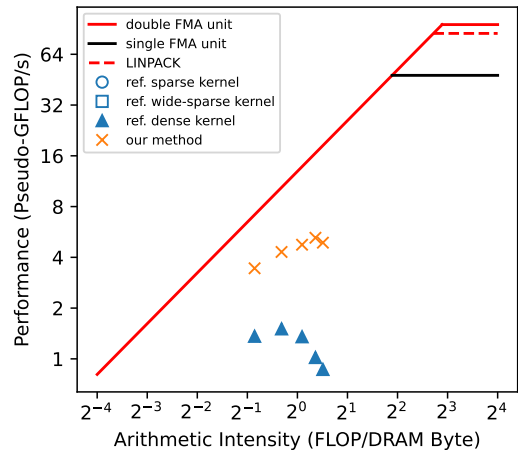
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

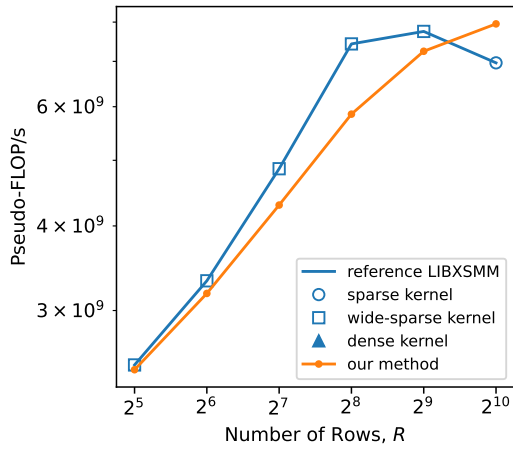


(e) Performance vs. number of columns, $U = 256$.

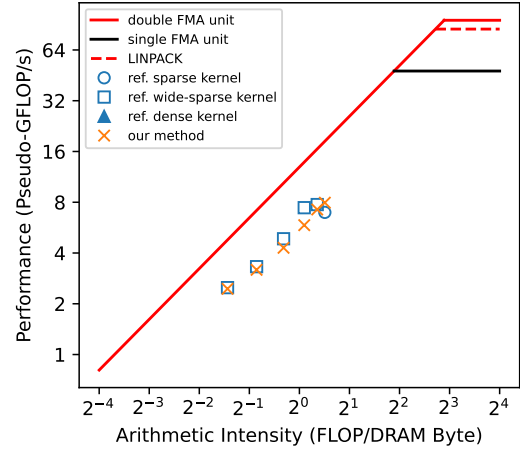


(f) Roofline plot, $U = 256$.

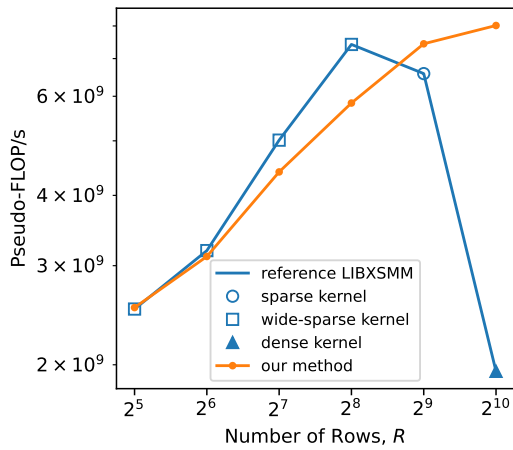
Figure E.9: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on c5n.xlarge machine



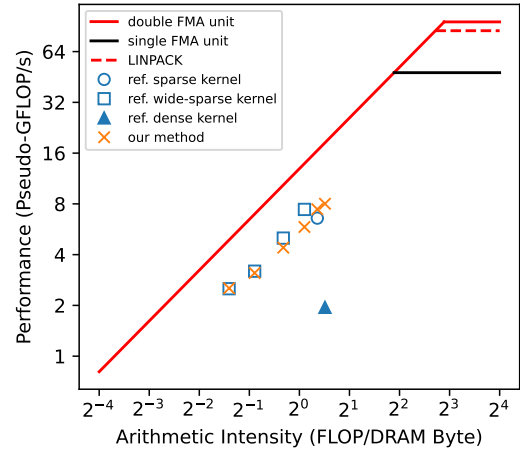
(a) Performance vs. number of rows, $U = 16$.



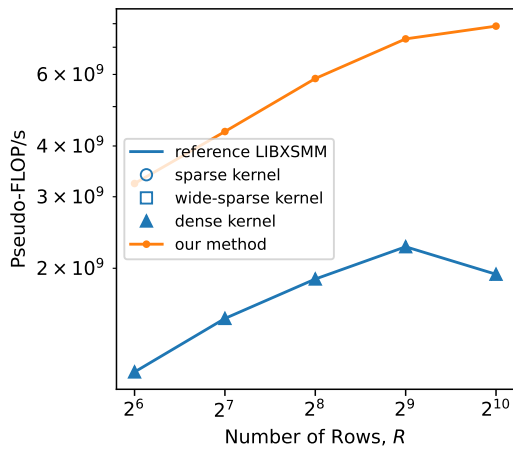
(b) Roofline plot, $U = 16$.



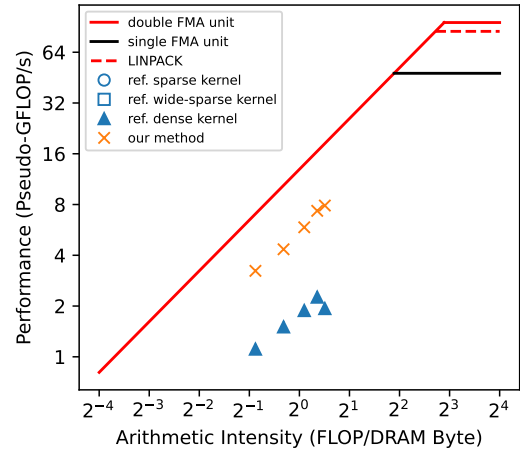
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

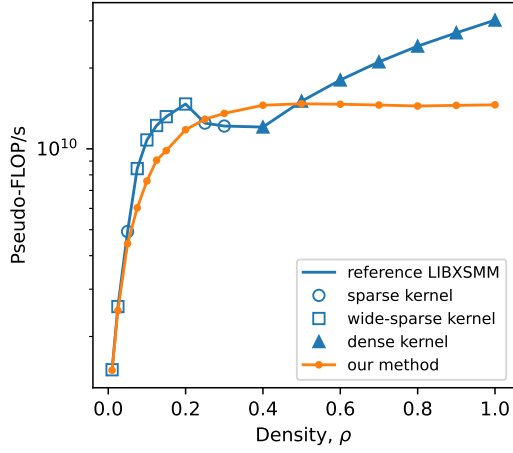


(e) Performance vs. number of rows, $U = 256$.

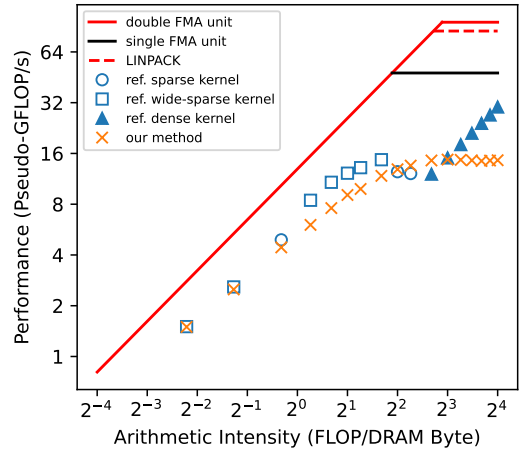


(f) Roofline plot, $U = 256$.

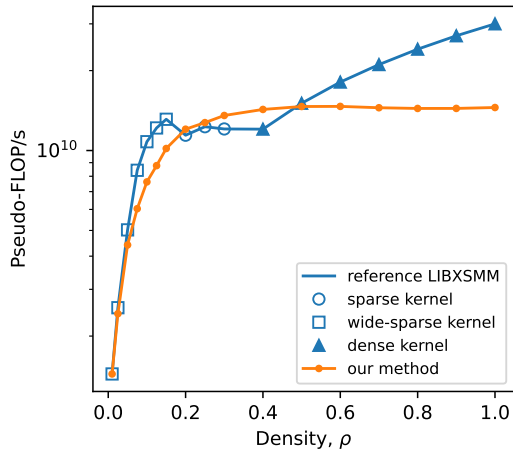
Figure E.10: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on c5n.xlarge machine



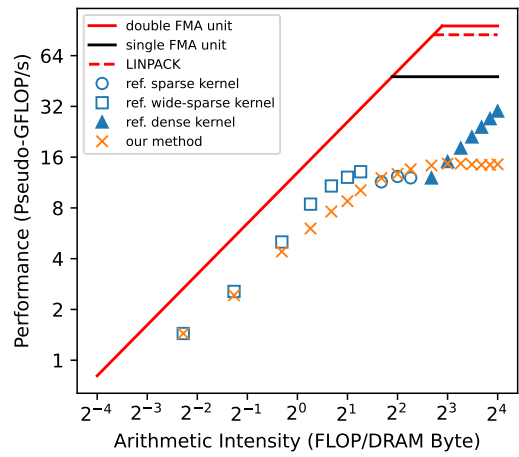
(a) Performance vs. density, $U = 16$.



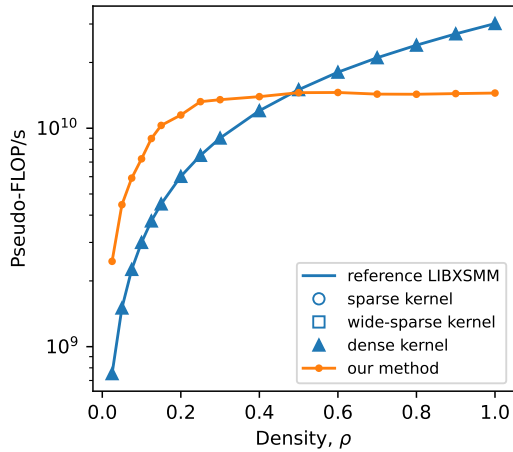
(b) Roofline plot, $U = 16$.



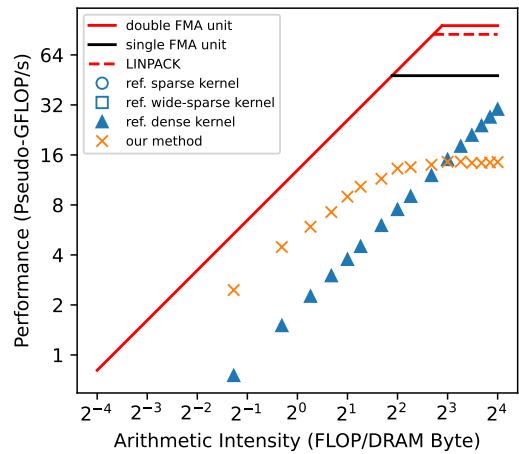
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

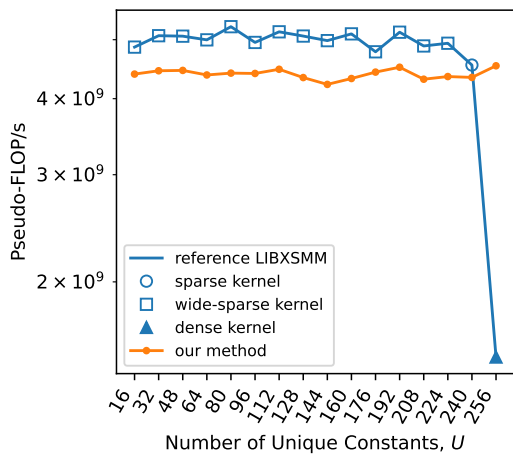


(e) Performance vs. density, $U = 256$.

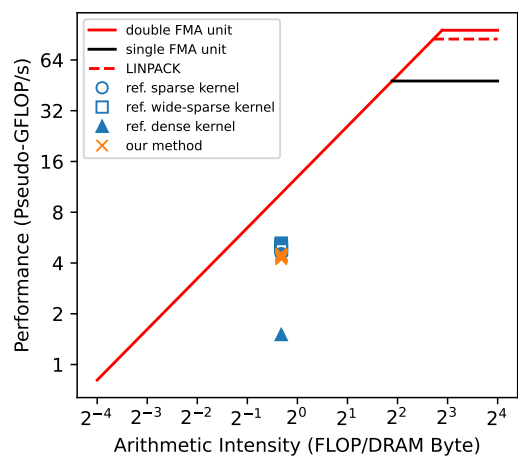


(f) Roofline plot, $U = 256$.

Figure E.11: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on c5n.xlarge machine



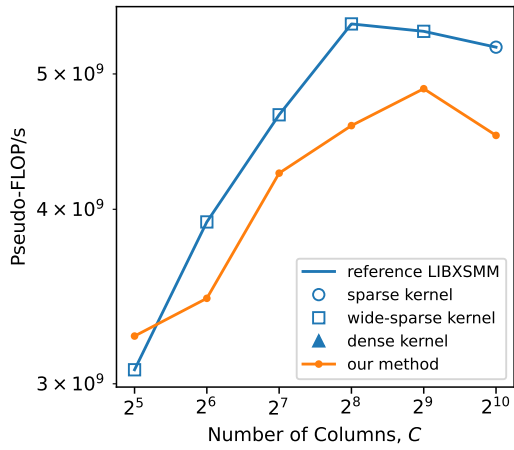
(a) Performance vs. number of unique absolute non-zero values.



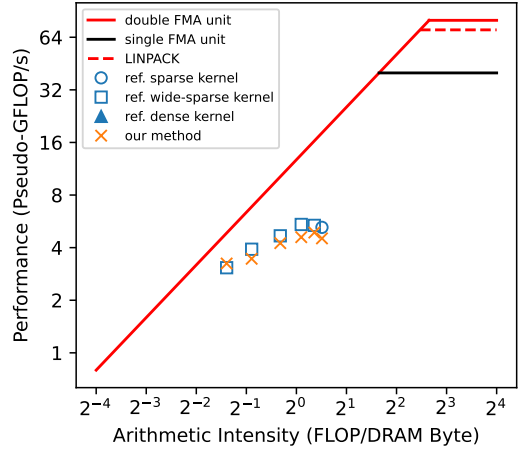
(b) Roofline plot.

Figure E.12: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} stride vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine

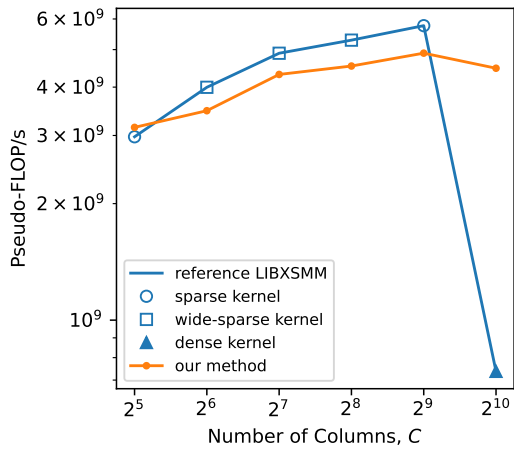
Benchmark Run on m5n.xlarge



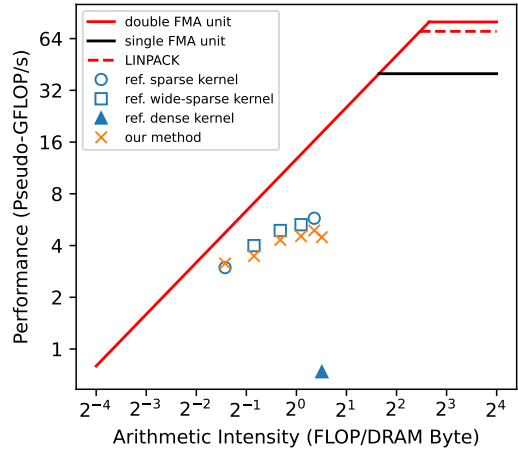
(a) Performance vs. number of columns, $U = 16$.



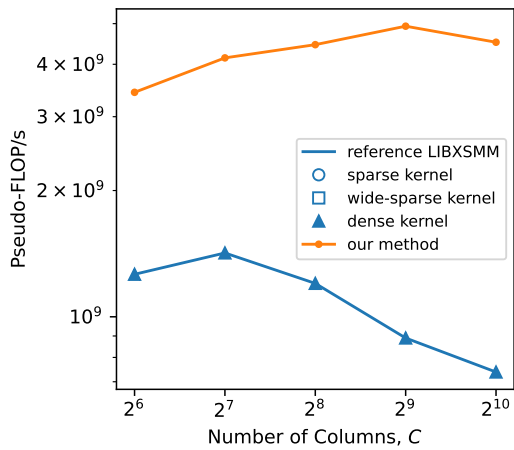
(b) Roofline plot, $U = 16$.



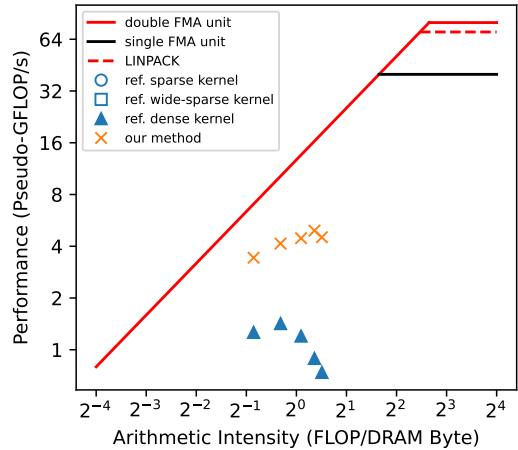
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

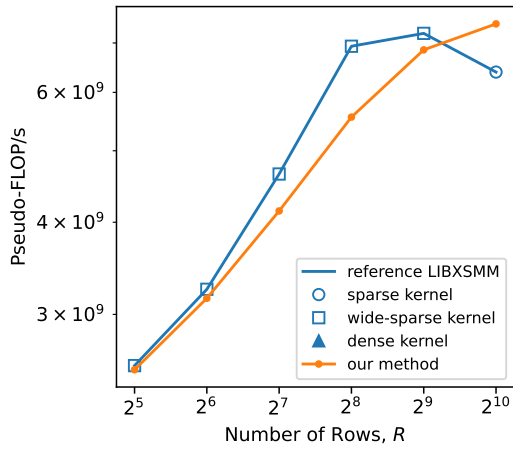


(e) Performance vs. number of columns, $U = 256$.

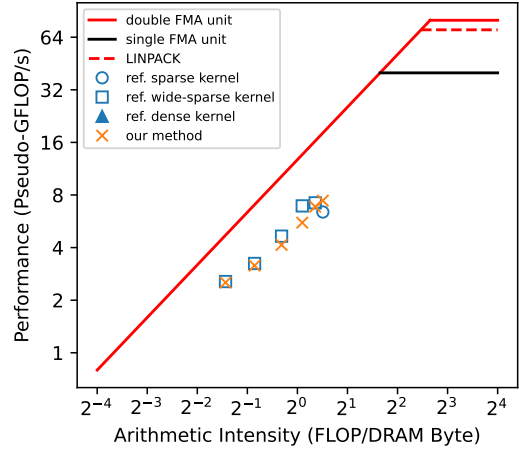


(f) Roofline plot, $U = 256$.

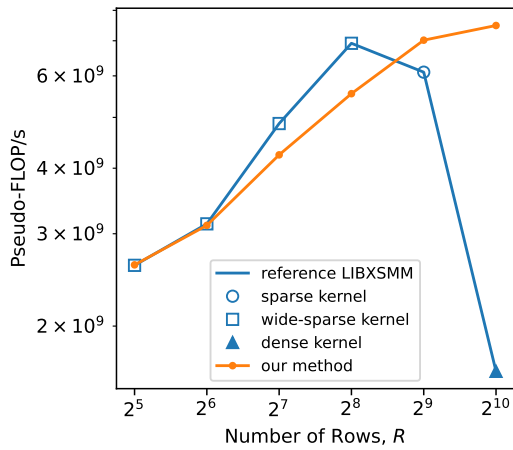
Figure E.13: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on m5n.xlarge machine



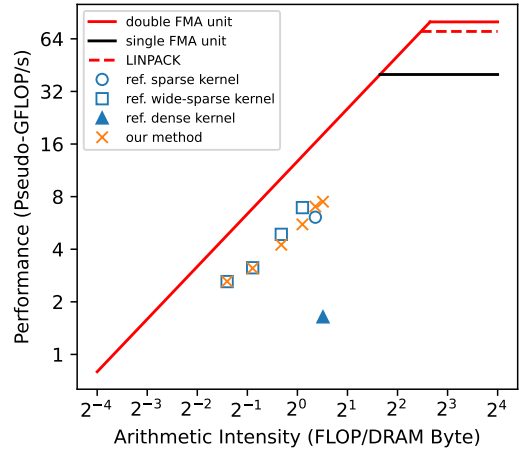
(a) Performance vs. number of rows, $U = 16$.



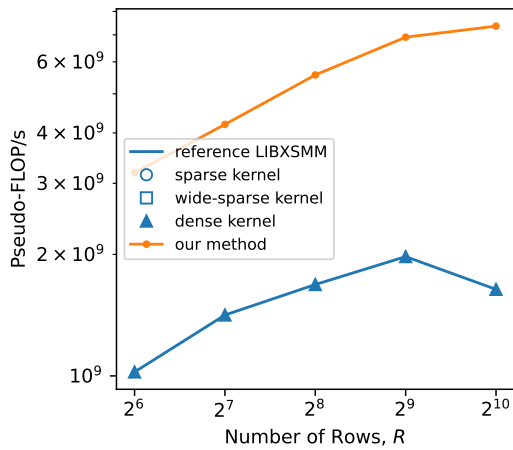
(b) Roofline plot, $U = 16$.



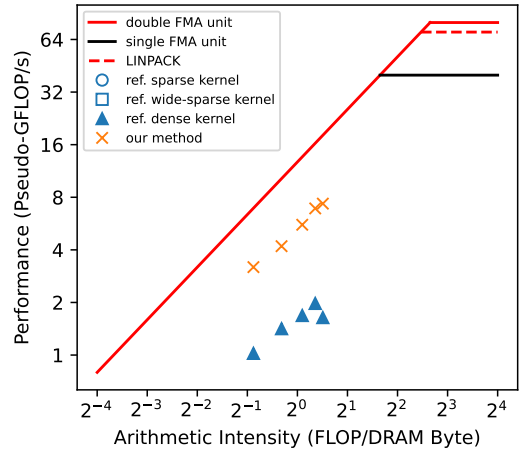
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

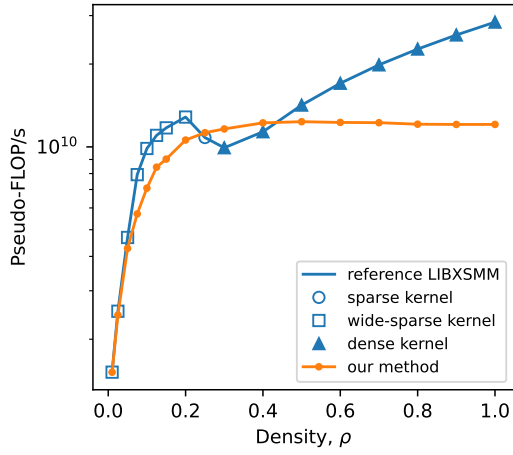


(e) Performance vs. number of rows, $U = 256$.

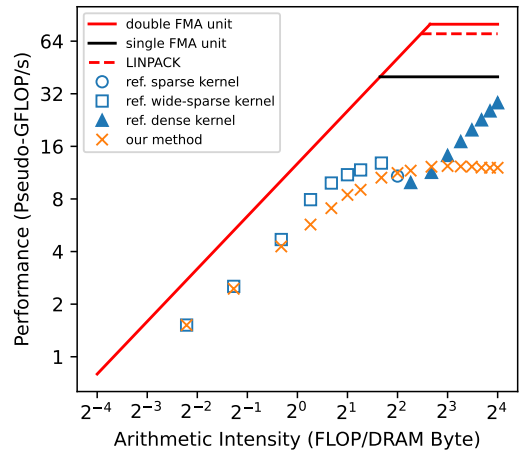


(f) Roofline plot, $U = 256$.

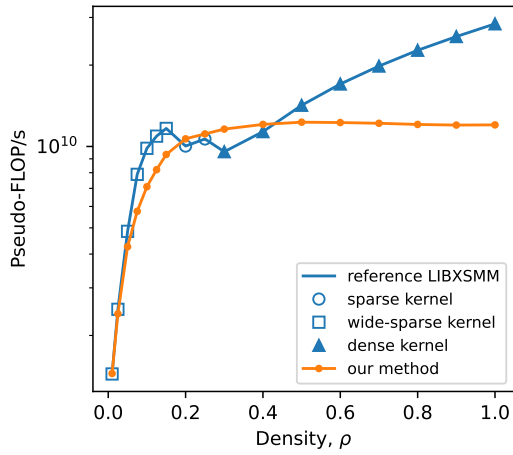
Figure E.14: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on m5n.xlarge machine



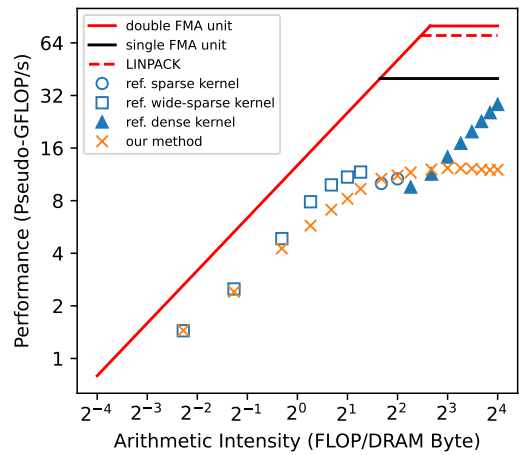
(a) Performance vs. density, $U = 16$.



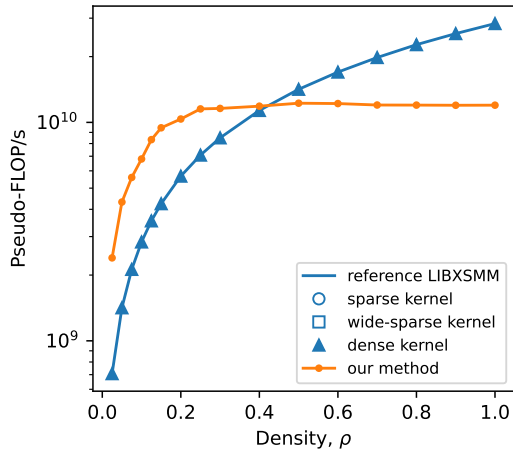
(b) Roofline plot, $U = 16$.



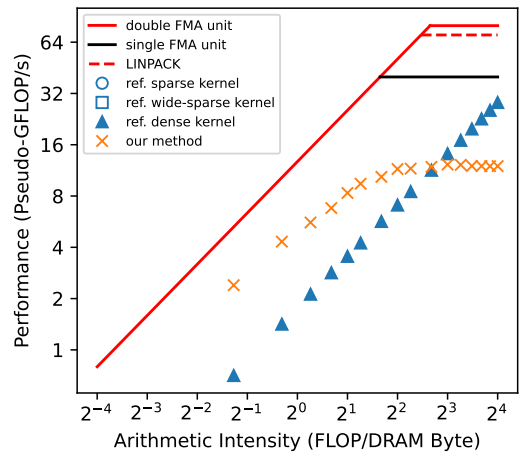
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

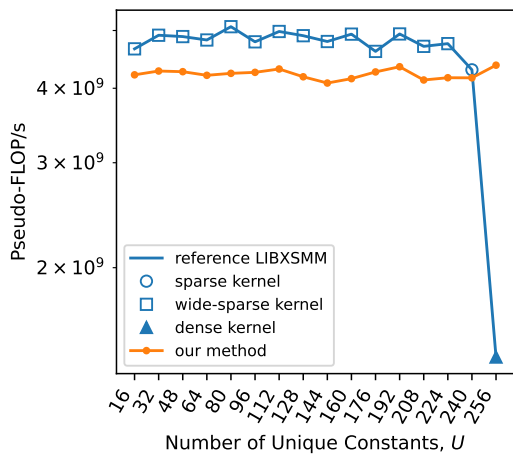


(e) Performance vs. density, $U = 256$.

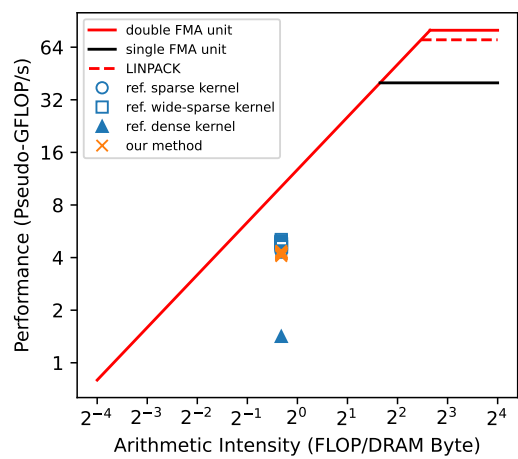


(f) Roofline plot, $U = 256$.

Figure E.15: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} strides vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on m5n.xlarge machine



(a) Performance vs. number of unique absolute non-zero values.



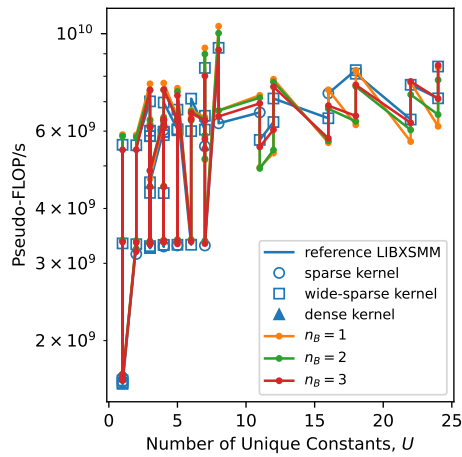
(b) Roofline plot.

Figure E.16: Runtime broadcasting with loading \mathbf{A} from memory and caching \mathbf{B} stride vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine

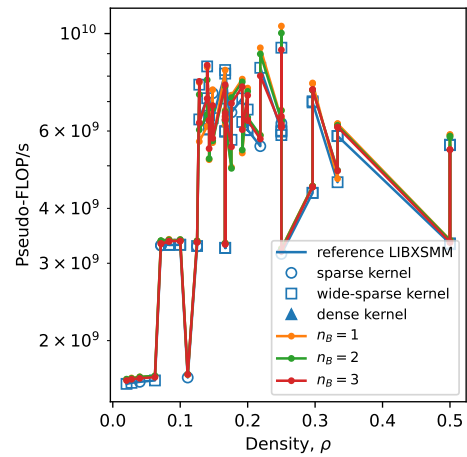
E.2 N Blocking

FyFR Operator Matrices

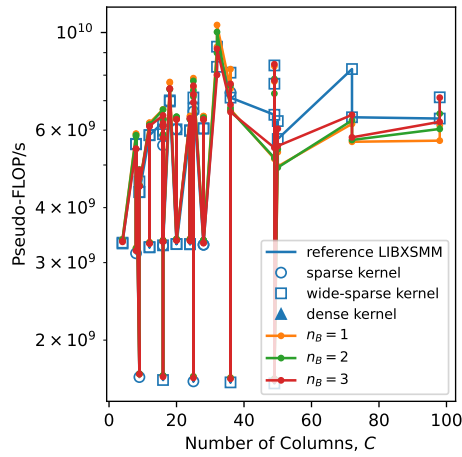
Benchmark Run on c5n.xlarge



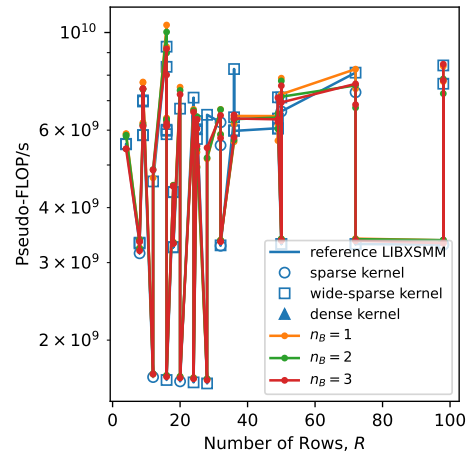
(a) Performance against number of unique \mathbf{A} constants.



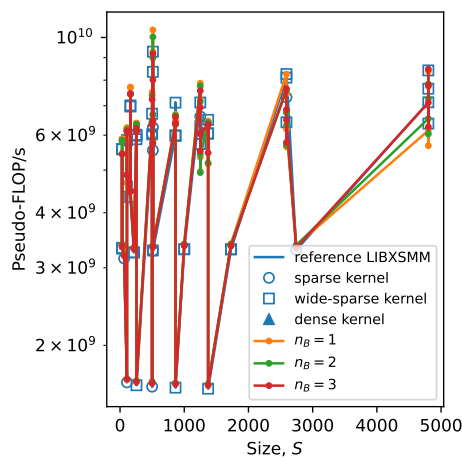
(b) Performance against \mathbf{A} density.



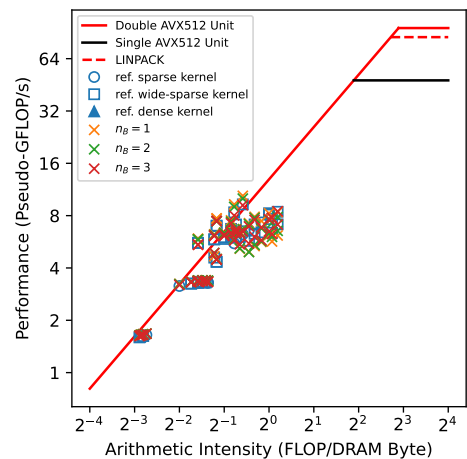
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

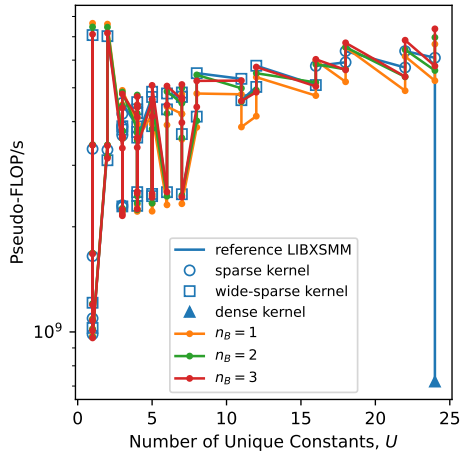


(e) Performance against number of \mathbf{A} size.

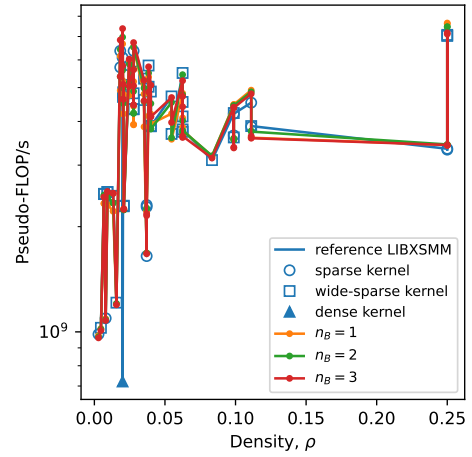


(f) Roofline plot.

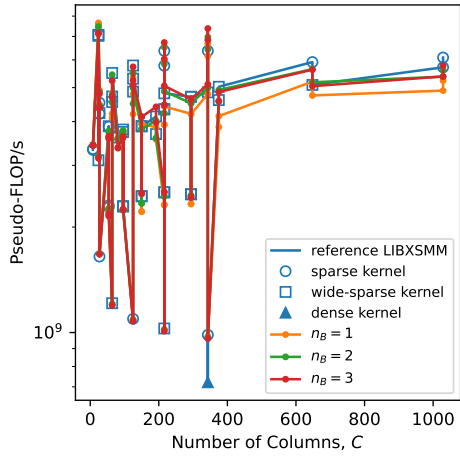
Figure E.17: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral element matrices. Benchmark run on c5n.xlarge machine.



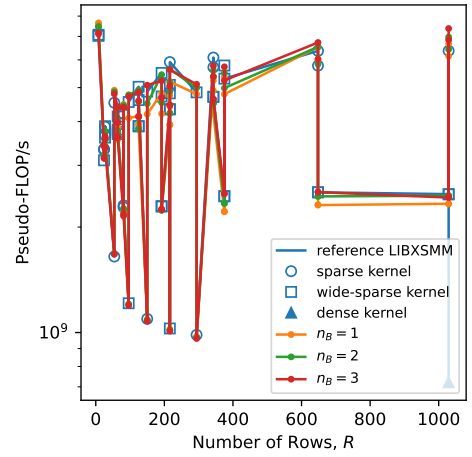
(a) Performance against number of unique \mathbf{A} constants.



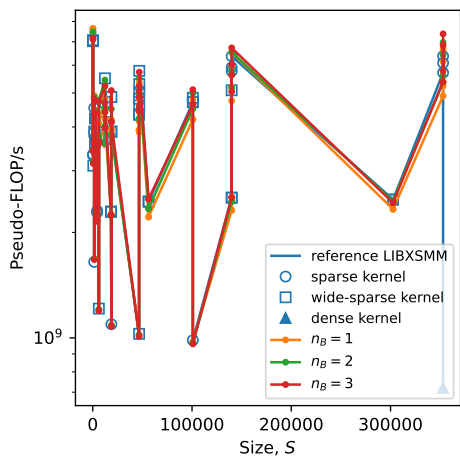
(b) Performance against \mathbf{A} density.



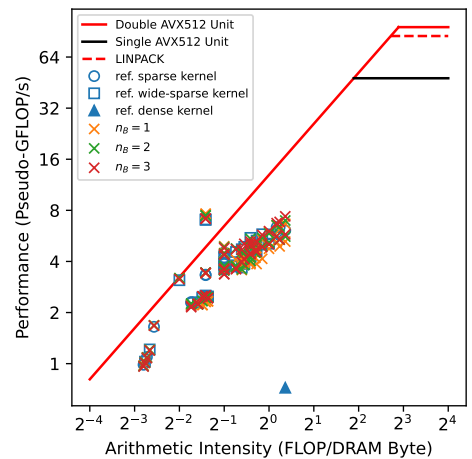
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

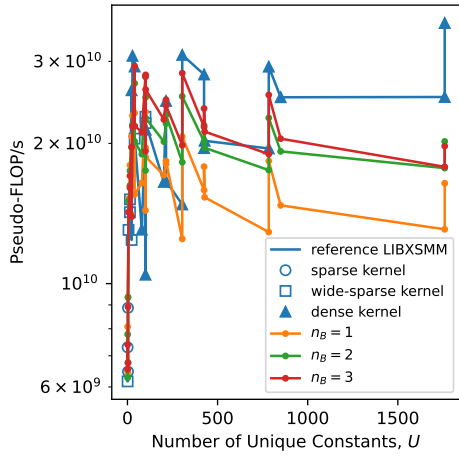


(e) Performance against number of \mathbf{A} size.

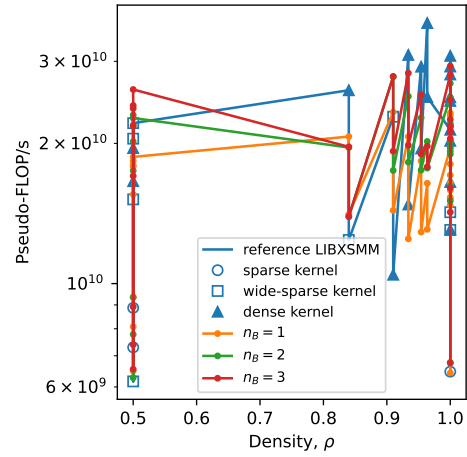


(f) Roofline plot.

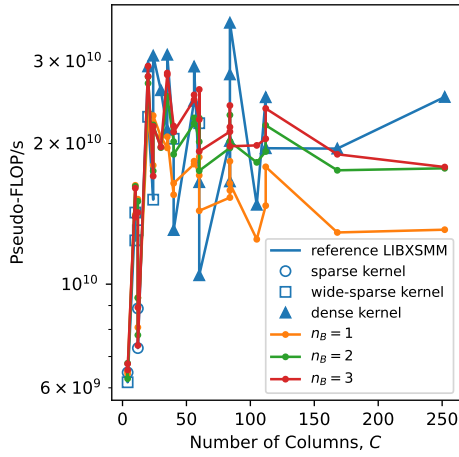
Figure E.18: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral element matrices. Benchmark run on c5n.xlarge machine.



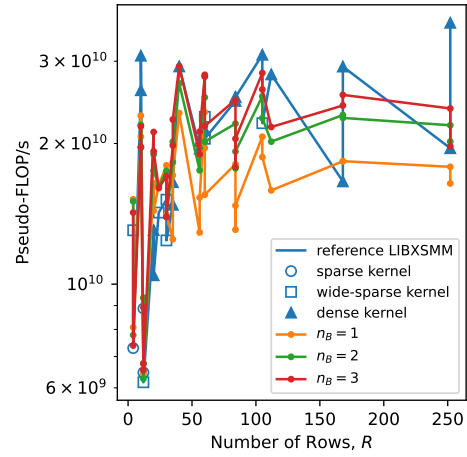
(a) Performance against number of unique \mathbf{A} constants.



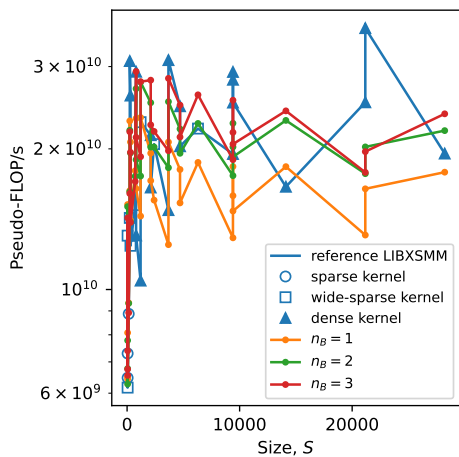
(b) Performance against \mathbf{A} density.



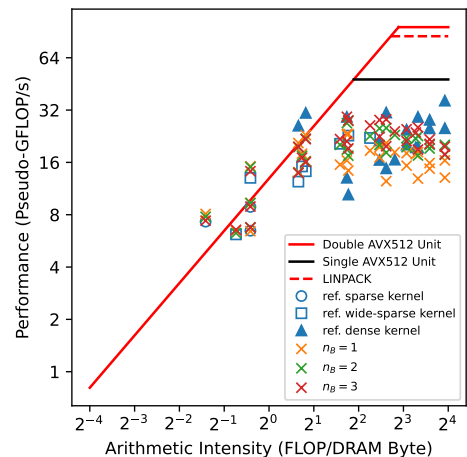
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

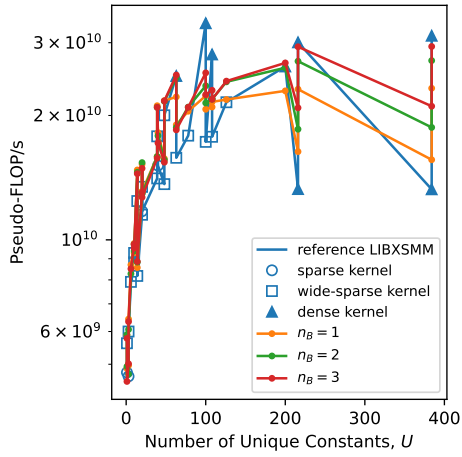


(e) Performance against number of \mathbf{A} size.

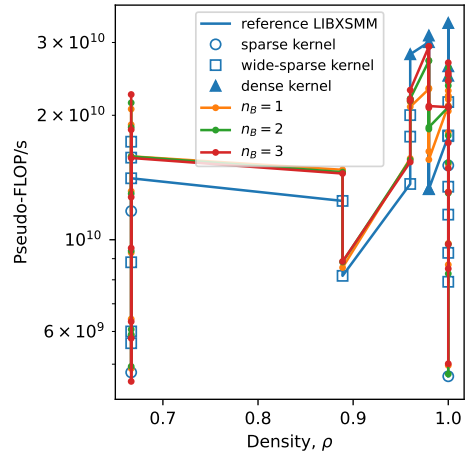


(f) Roofline plot.

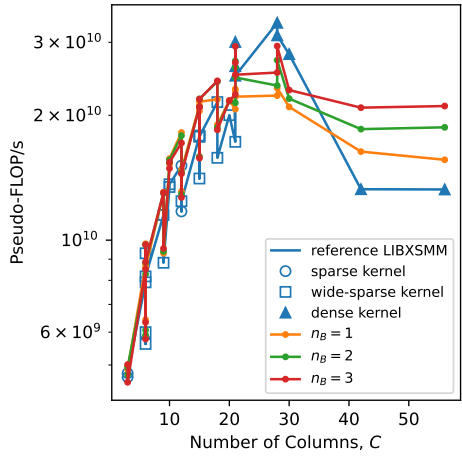
Figure E.19: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral element matrices. Benchmark run on c5n.xlarge machine.



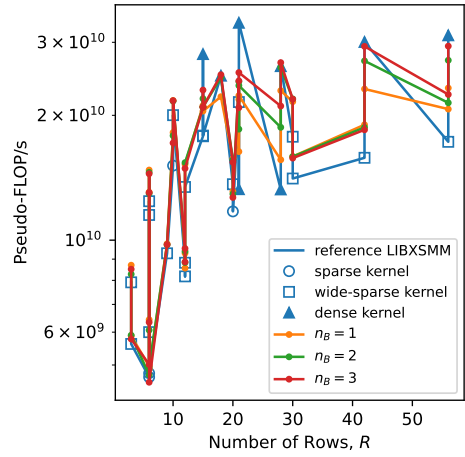
(a) Performance against number of unique \mathbf{A} constants.



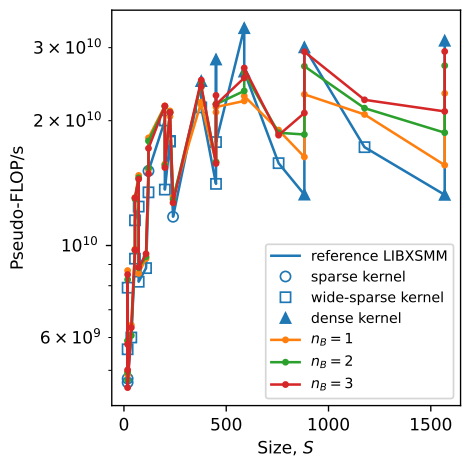
(b) Performance against \mathbf{A} density.



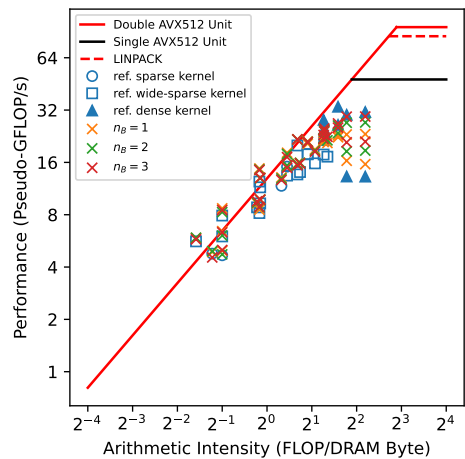
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.



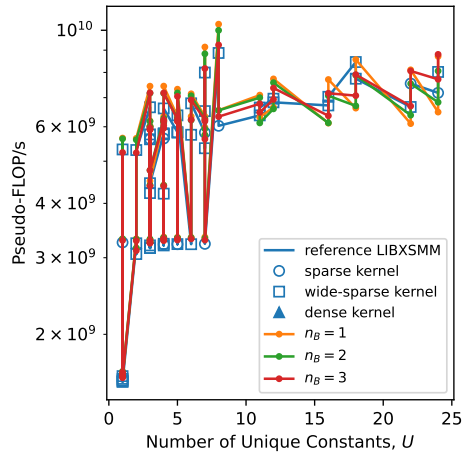
(f) Roofline plot.

Figure E.20: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR triangular element matrices. Benchmark run on c5n.xlarge machine.

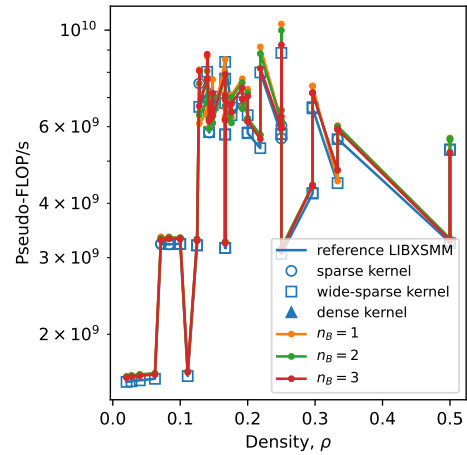
Benchmark Run on m5n.xlarge

Synthetic Matrices

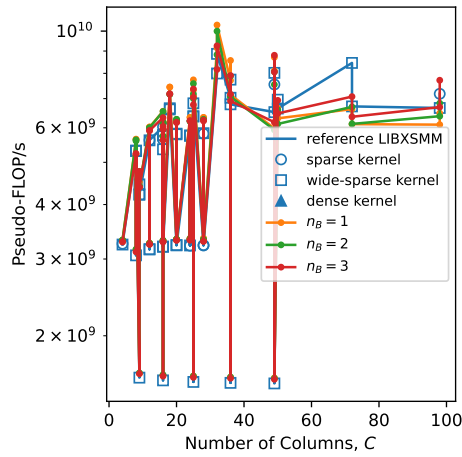
Benchmark Run on c5n.xlarge



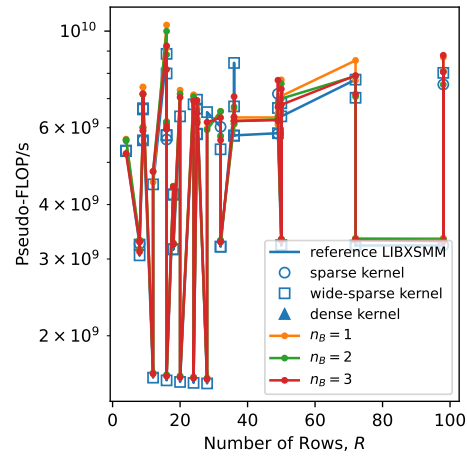
(a) Performance against number of unique \mathbf{A} constants.



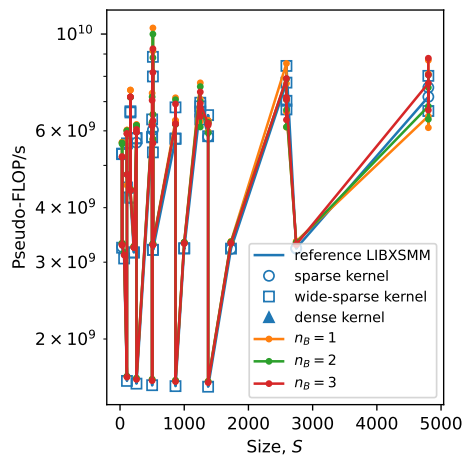
(b) Performance against \mathbf{A} density.



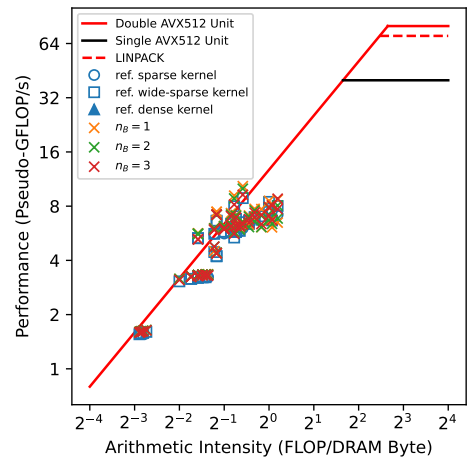
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

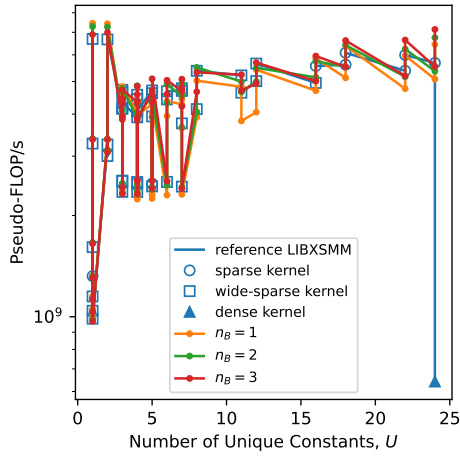


(e) Performance against number of \mathbf{A} size.

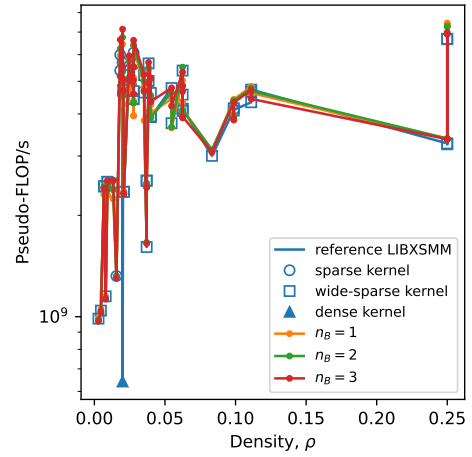


(f) Roofline plot.

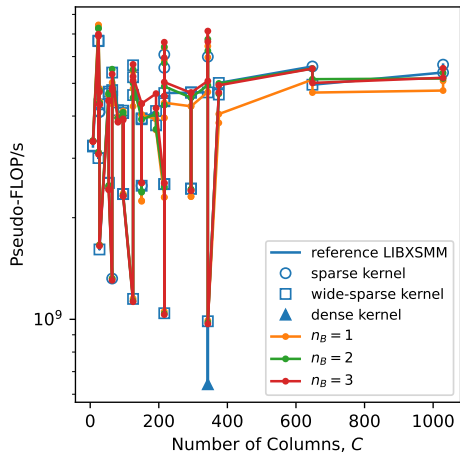
Figure E.21: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral element matrices. Benchmark run on m5n.xlarge machine.



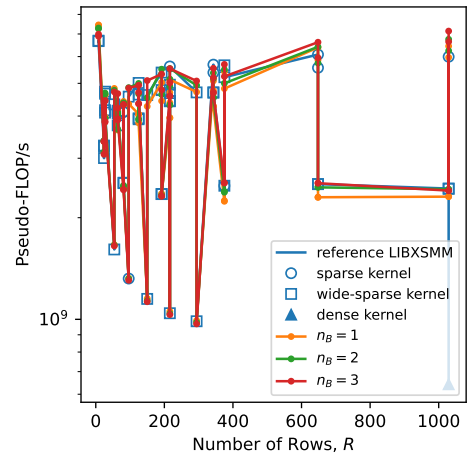
(a) Performance against number of unique \mathbf{A} constants.



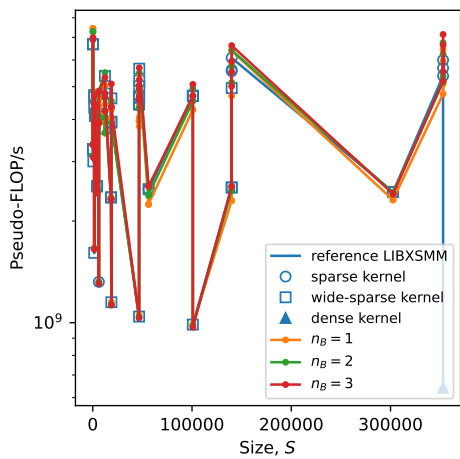
(b) Performance against \mathbf{A} density.



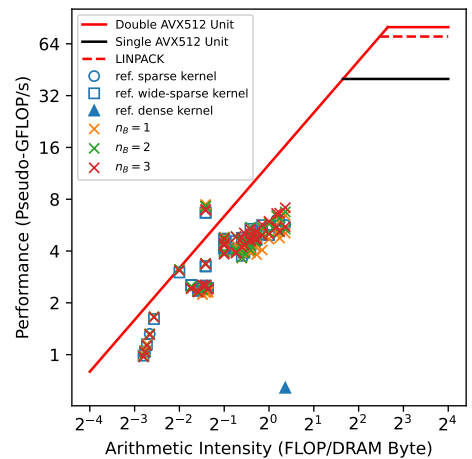
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

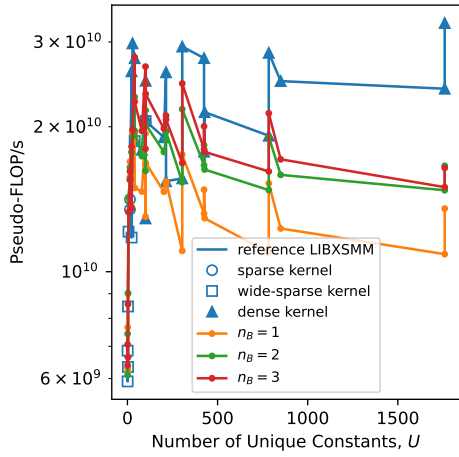


(e) Performance against number of \mathbf{A} size.

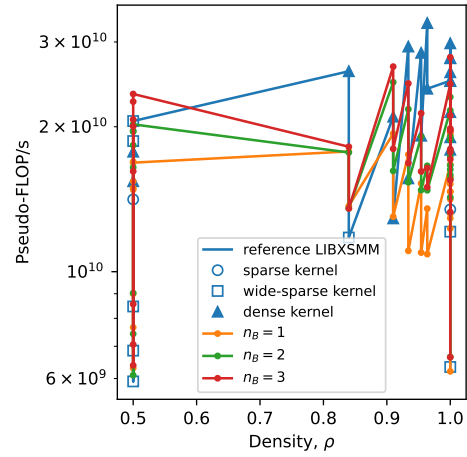


(f) Roofline plot.

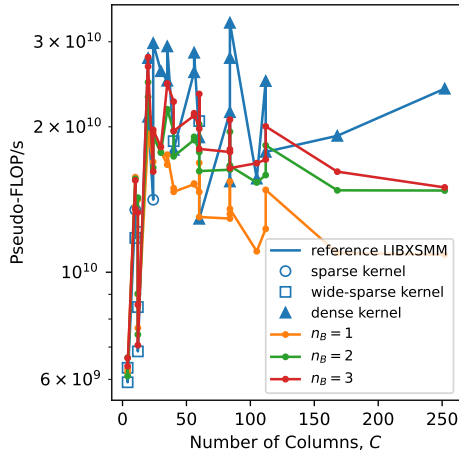
Figure E.22: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR hexahedral element matrices. Benchmark run on m5n.xlarge machine.



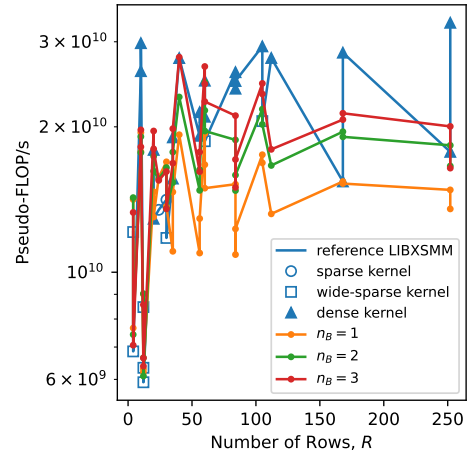
(a) Performance against number of unique A constants.



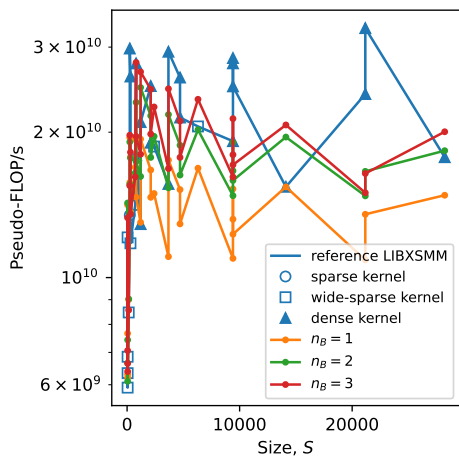
(b) Performance against A density.



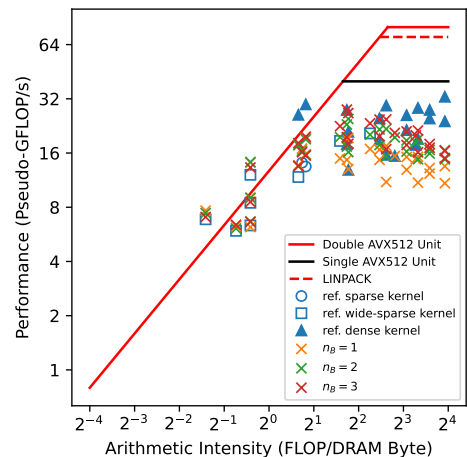
(c) Performance against number of A columns.



(d) Performance against number of A rows.

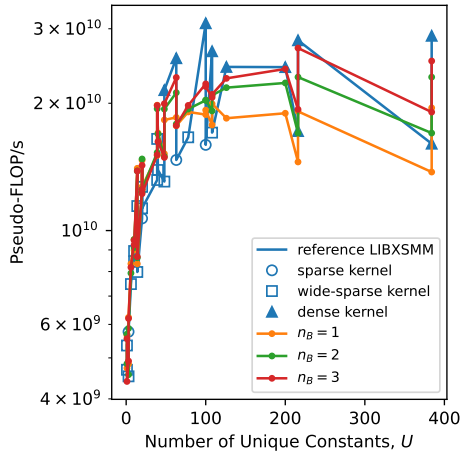


(e) Performance against number of A size.

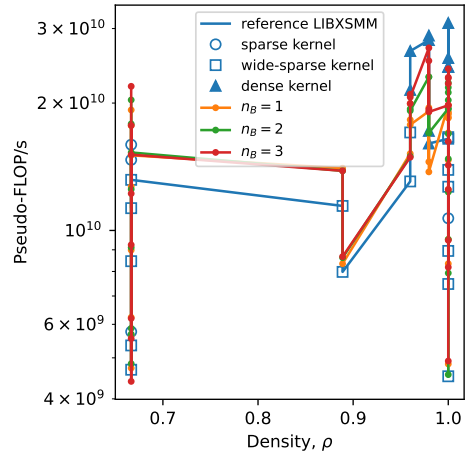


(f) Roofline plot.

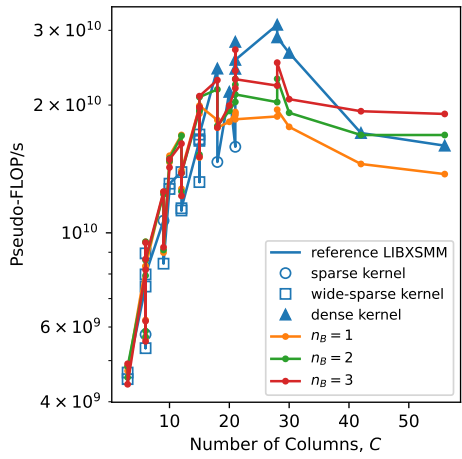
Figure E.23: Runtime broadcasting with loading A from memory, caching B strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral element matrices. Benchmark run on m5n.xlarge machine.



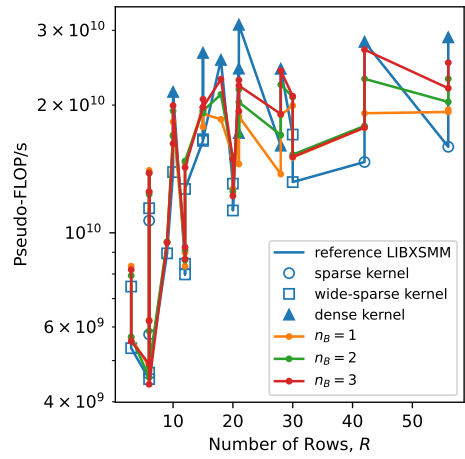
(a) Performance against number of unique \mathbf{A} constants.



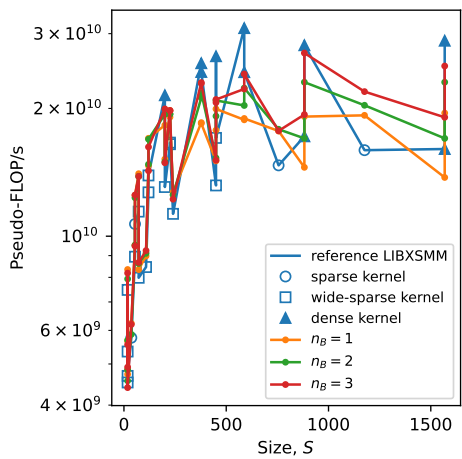
(b) Performance against \mathbf{A} density.



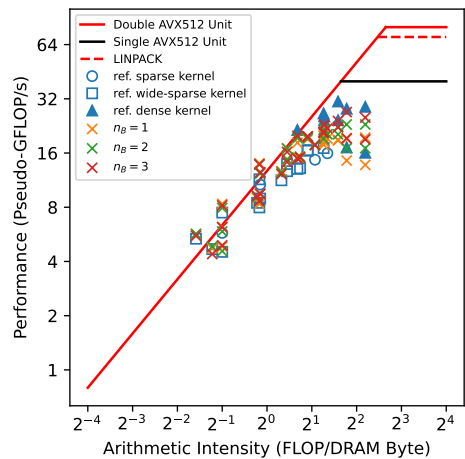
(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.

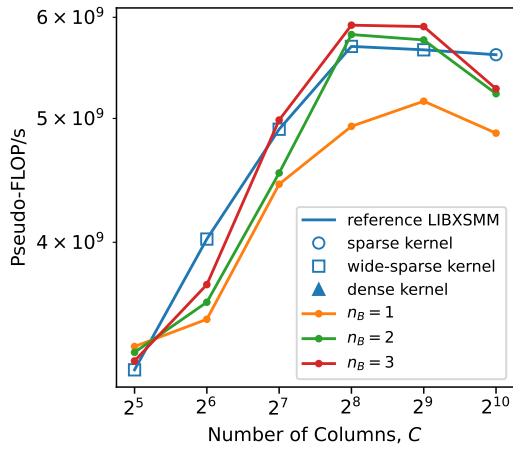


(e) Performance against number of \mathbf{A} size.

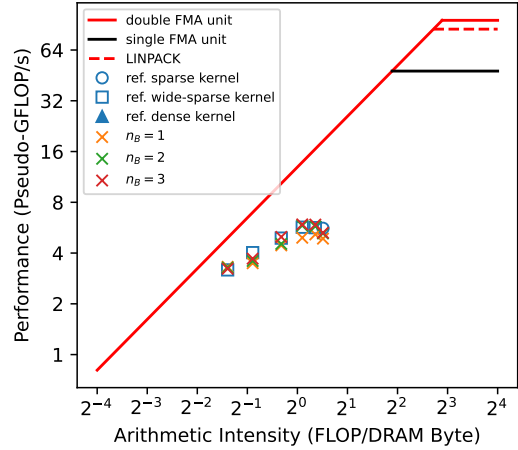


(f) Roofline plot.

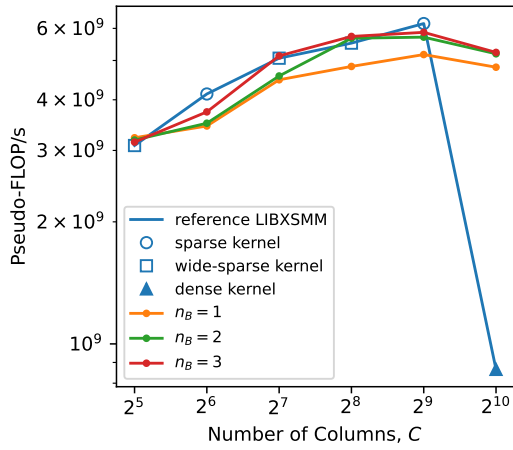
Figure E.24: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides in vector registers, and N blocking vs. reference LIBXSMM implementations, for PyFR triangular element matrices. Benchmark run on m5n.xlarge machine.



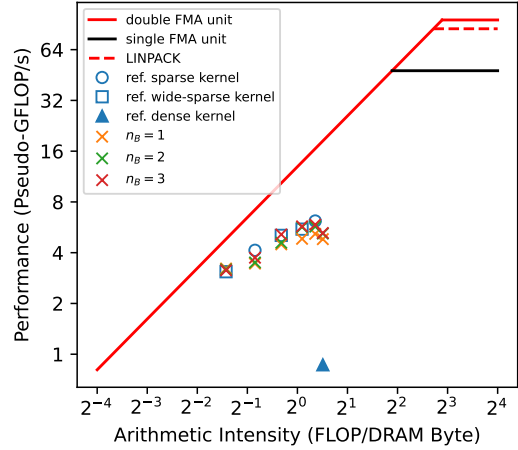
(a) Performance vs. number of columns, $U = 16$.



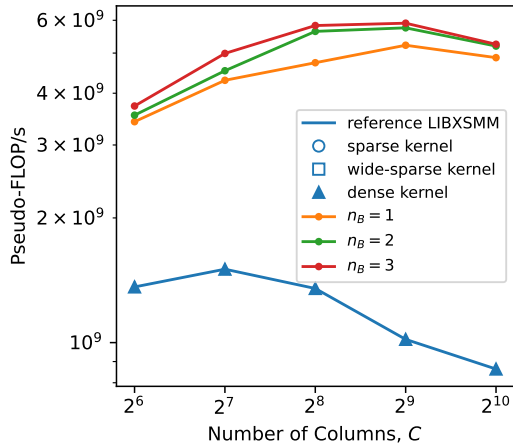
(b) Roofline plot, $U = 16$.



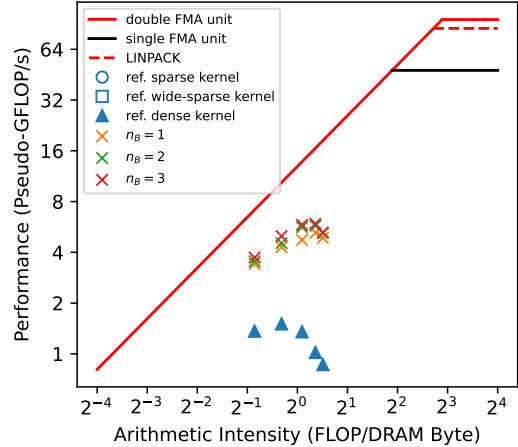
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

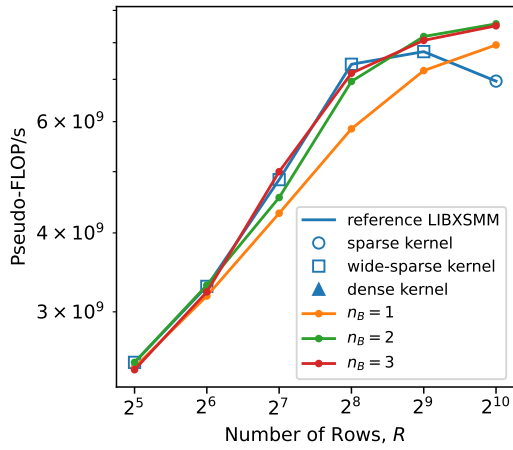


(e) Performance vs. number of columns, $U = 256$.

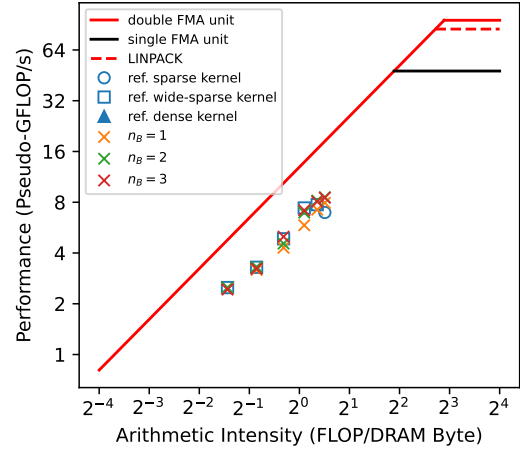


(f) Roofline plot, $U = 256$.

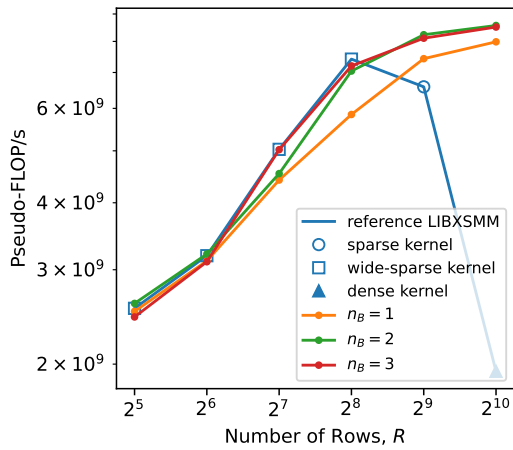
Figure E.25: Runtime broadcasting with loading A from memory, caching B strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of A columns. Benchmark run on c5n.xlarge machine



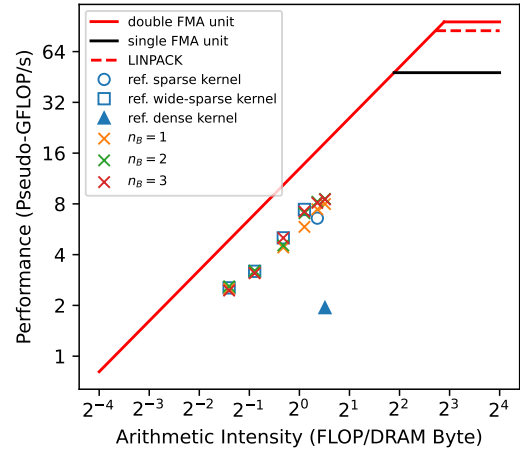
(a) Performance vs. number of rows, $U = 16$.



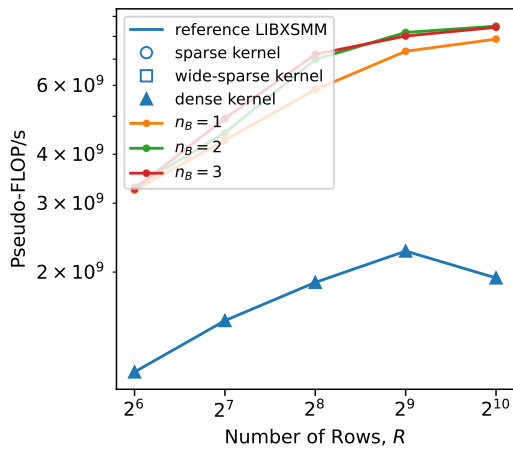
(b) Roofline plot, $U = 16$.



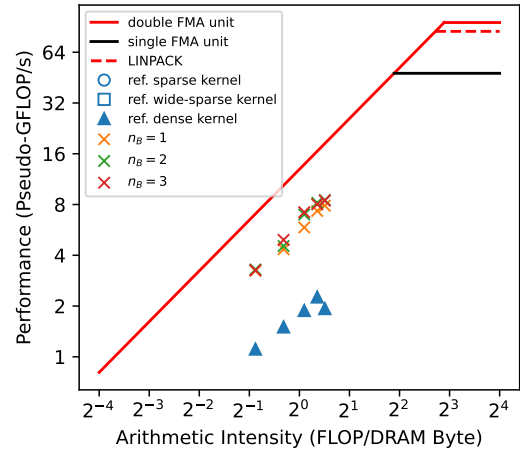
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

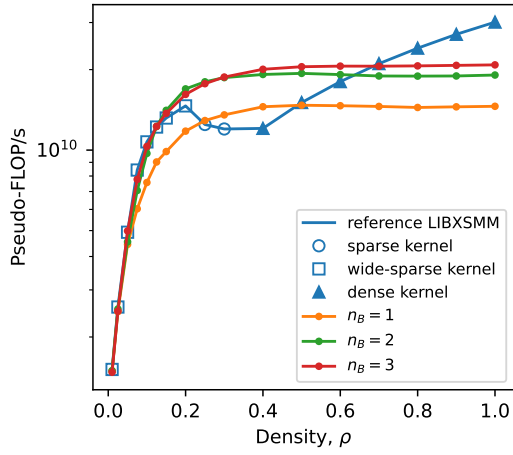


(e) Performance vs. number of rows, $U = 256$.

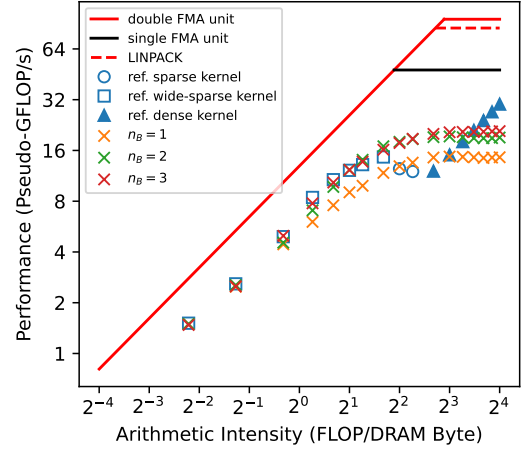


(f) Roofline plot, $U = 256$.

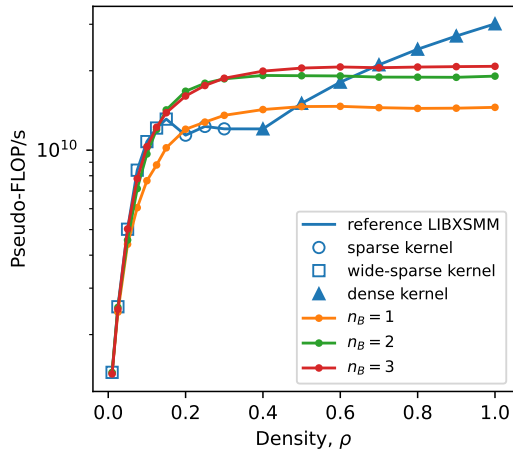
Figure E.26: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on c5n.xlarge machine



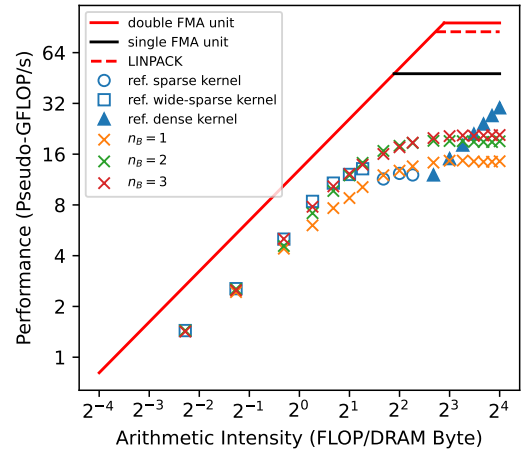
(a) Performance vs. density, $U = 16$.



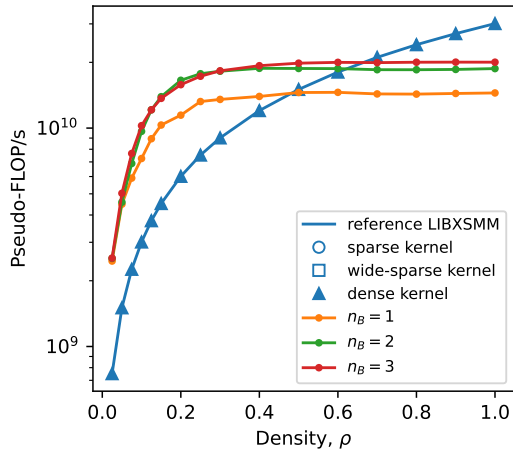
(b) Roofline plot, $U = 16$.



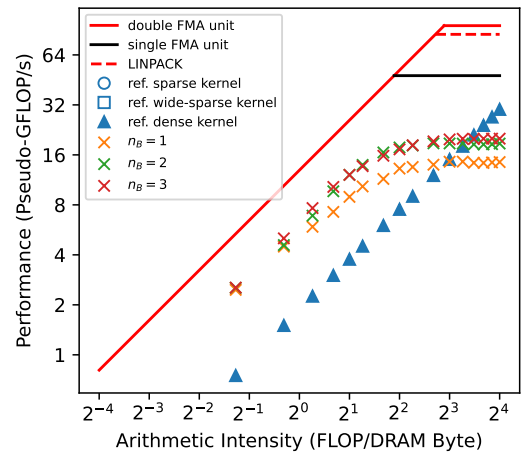
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

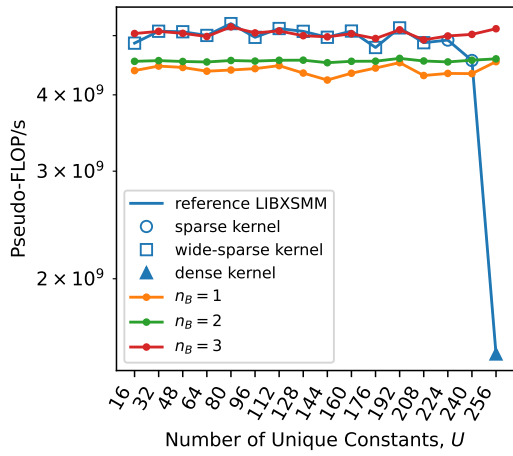


(e) Performance vs. density, $U = 256$.

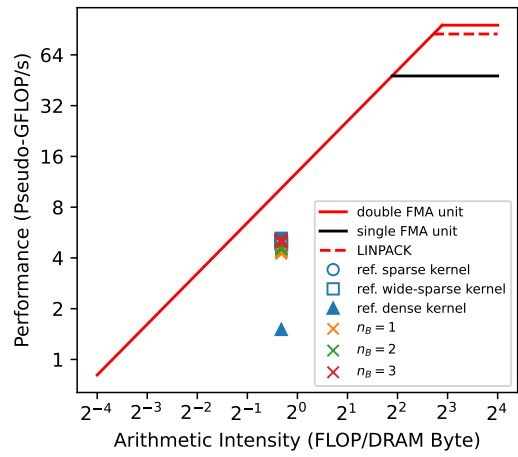


(f) Roofline plot, $U = 256$.

Figure E.27: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on c5n.xlarge machine



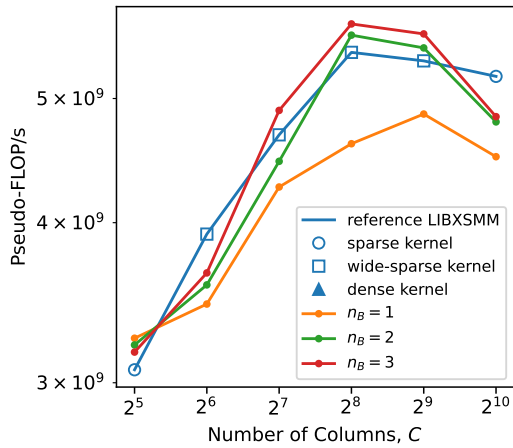
(a) Performance vs. number of unique absolute non-zero values.



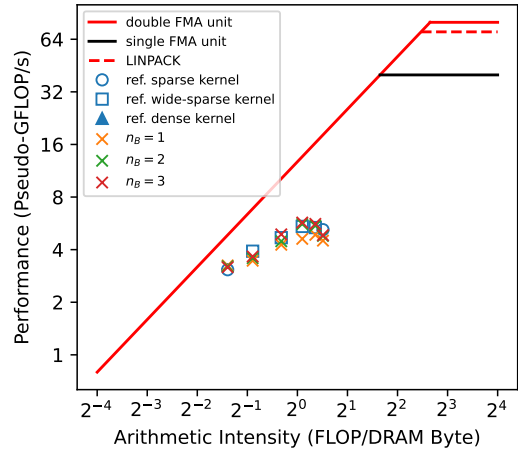
(b) Roofline plot.

Figure E.28: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine

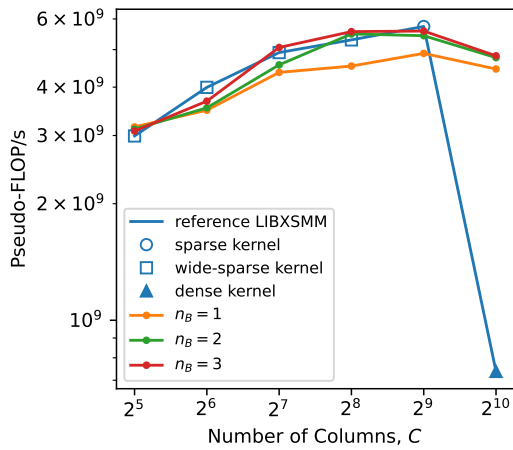
Benchmark Run on m5n.xlarge



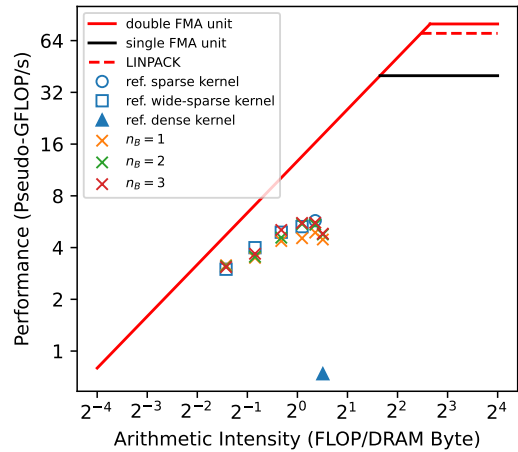
(a) Performance vs. number of columns, $U = 16$.



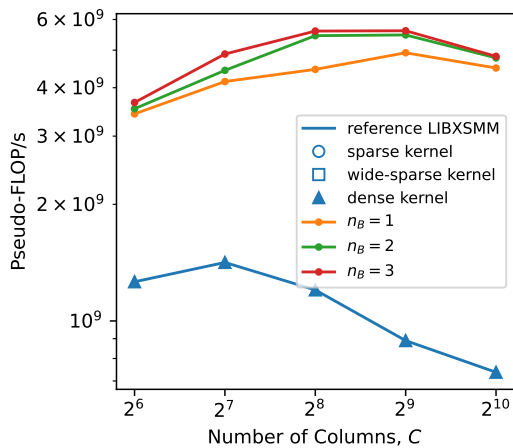
(b) Roofline plot, $U = 16$.



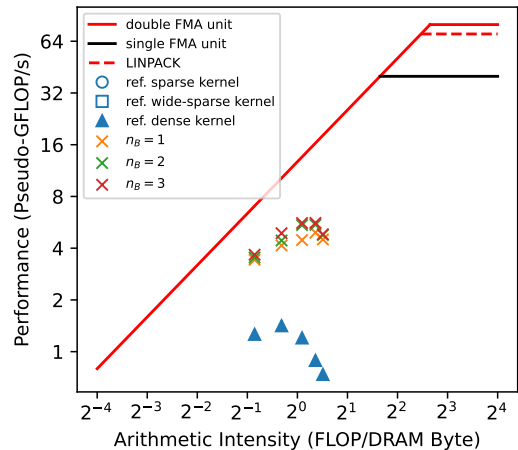
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

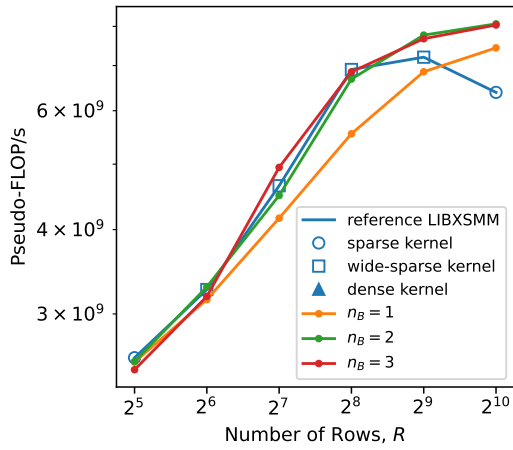


(e) Performance vs. number of columns, $U = 256$.

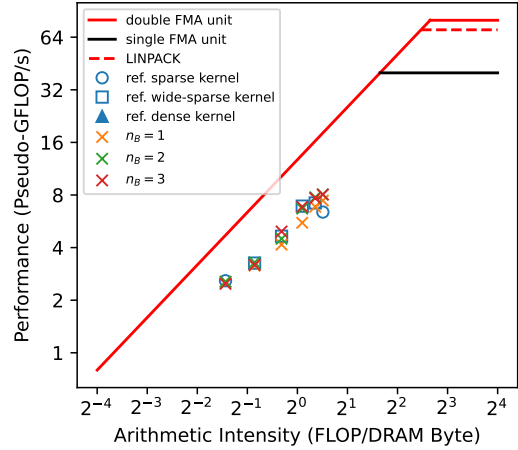


(f) Roofline plot, $U = 256$.

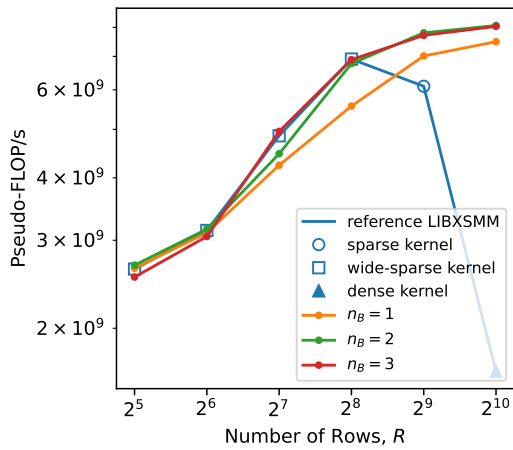
Figure E.29: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on m5n.xlarge machine



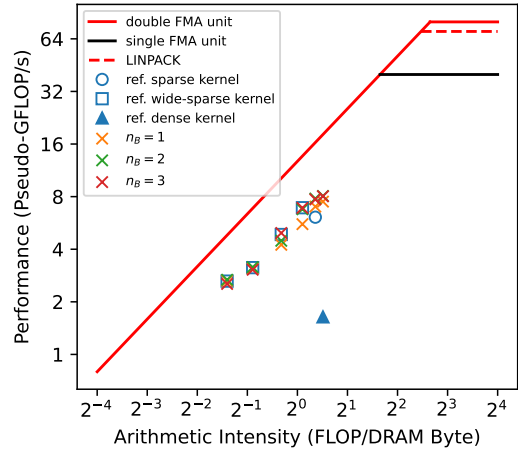
(a) Performance vs. number of rows, $U = 16$.



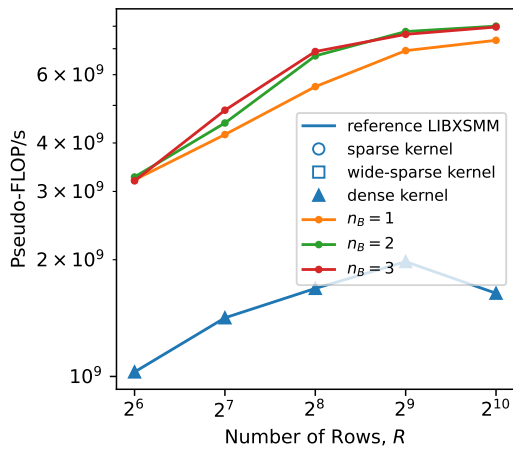
(b) Roofline plot, $U = 16$.



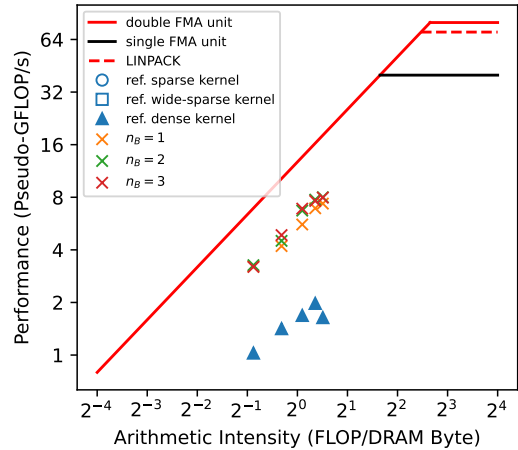
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

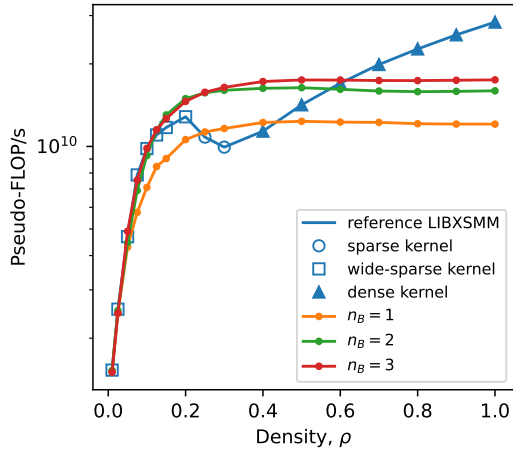


(e) Performance vs. number of rows, $U = 256$.

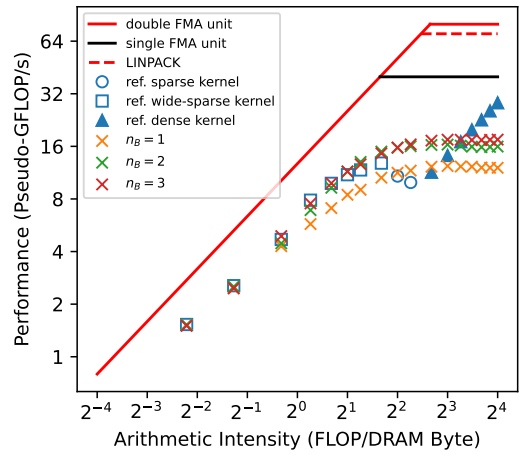


(f) Roofline plot, $U = 256$.

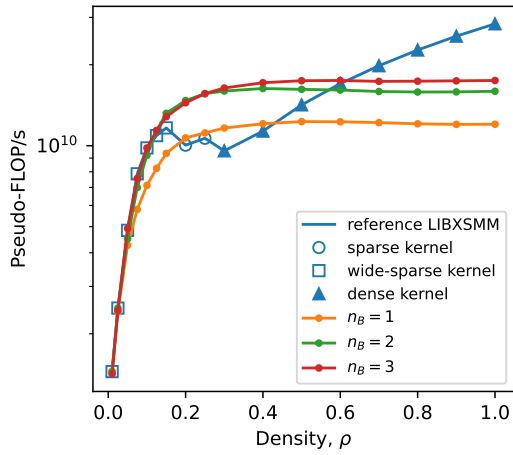
Figure E.30: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on m5n.xlarge machine



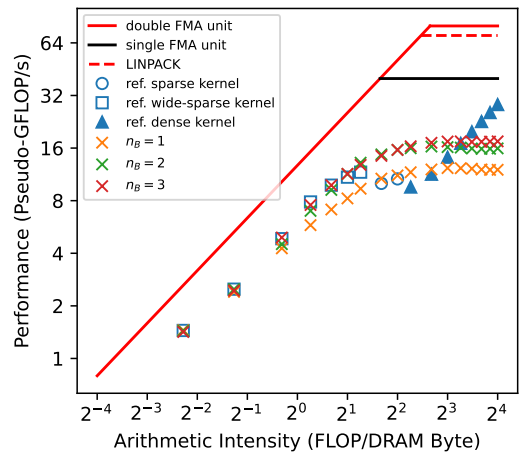
(a) Performance vs. density, $U = 16$.



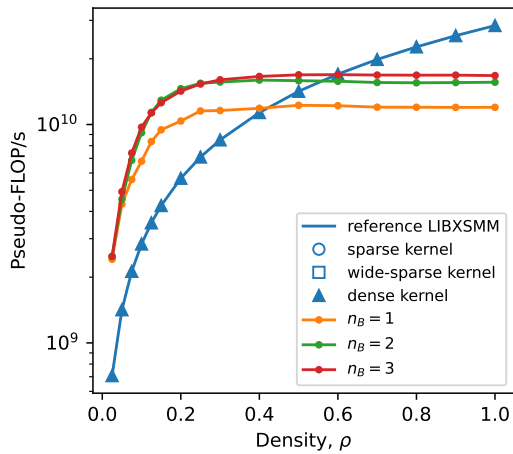
(b) Roofline plot, $U = 16$.



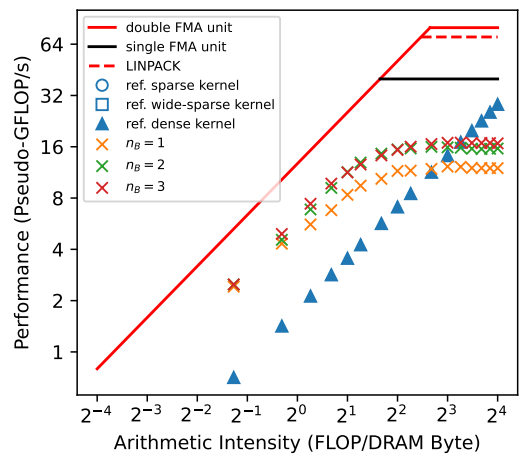
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

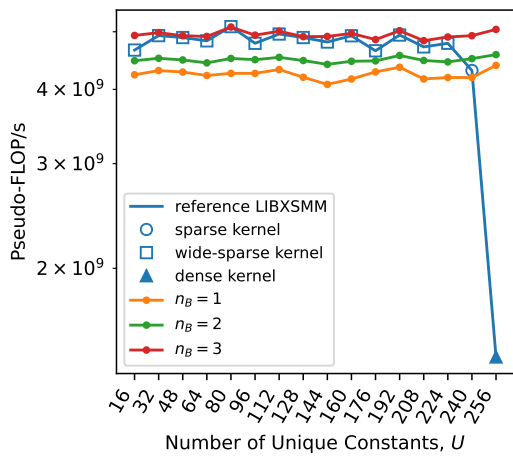


(e) Performance vs. density, $U = 256$.

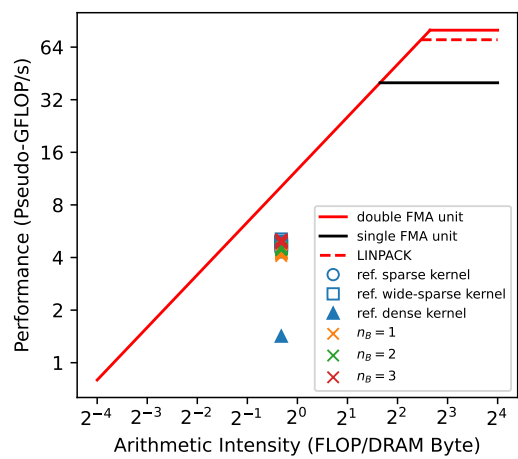


(f) Roofline plot, $U = 256$.

Figure E.31: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on m5n.xlarge machine



(a) Performance vs. number of unique absolute non-zero values.



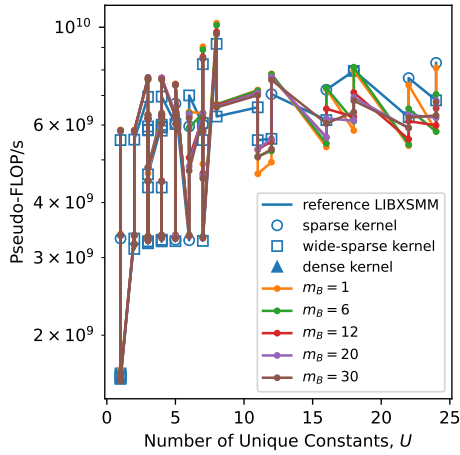
(b) Roofline plot.

Figure E.32: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and N blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on m5n.xlarge machine

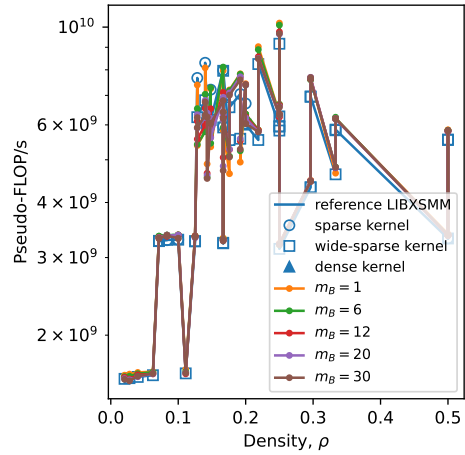
E.3 M Blocking

FyFR Operator Matrices

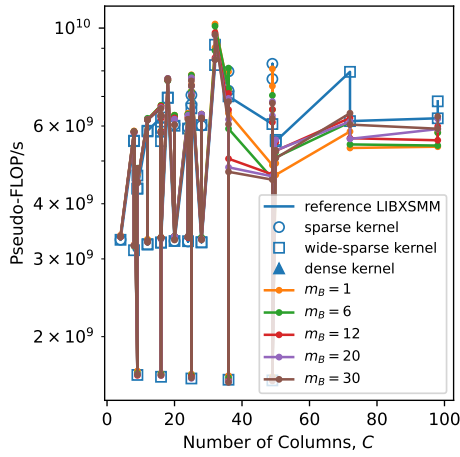
Benchmark Run on c5n.xlarge



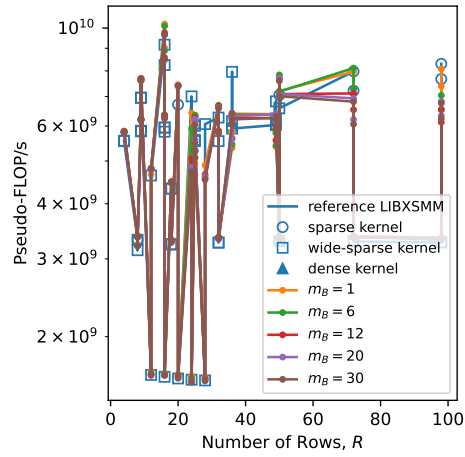
(a) Performance against number of unique \mathbf{A} constants.



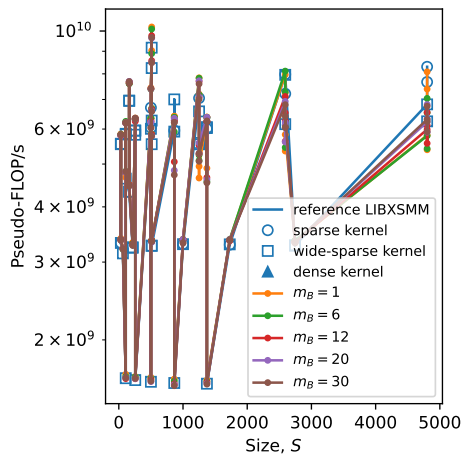
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure E.33: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral element operator matrices. Benchmark run on c5n.xlarge machine.

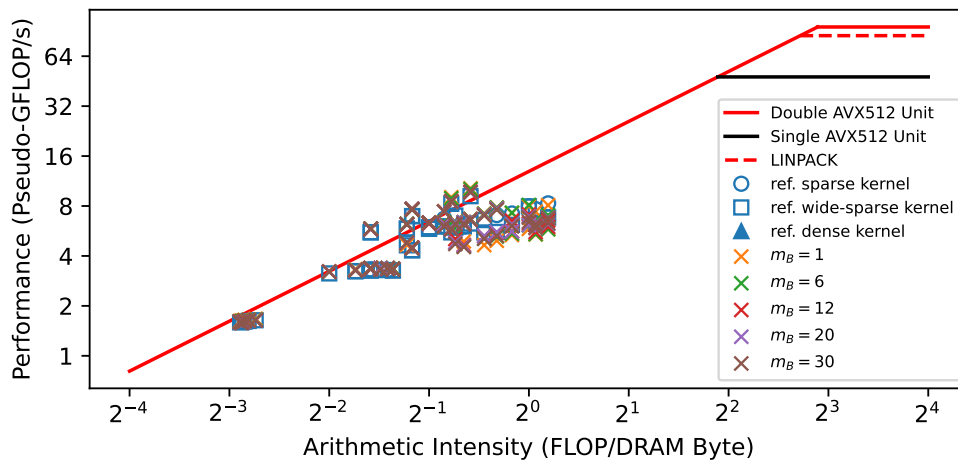
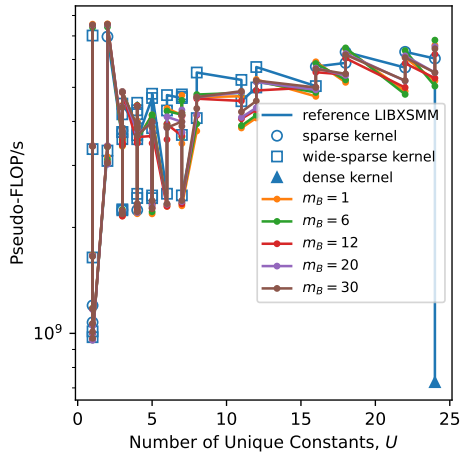
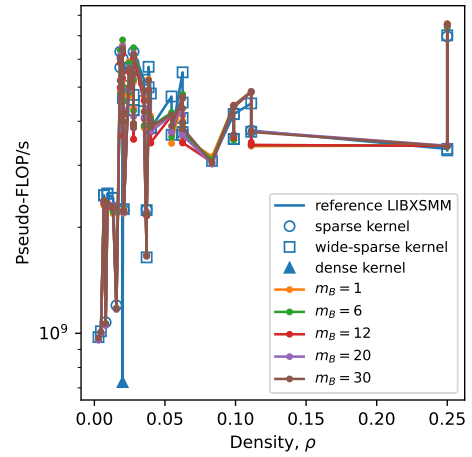


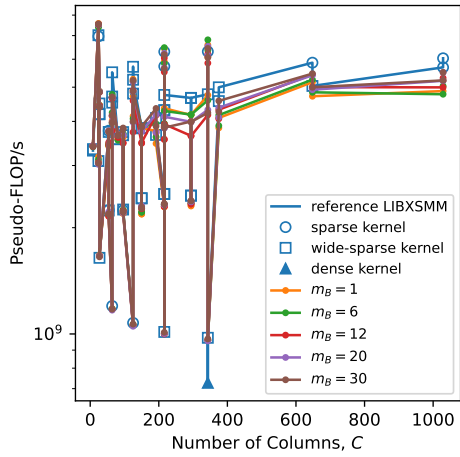
Figure E.34: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR quadrilateral element operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



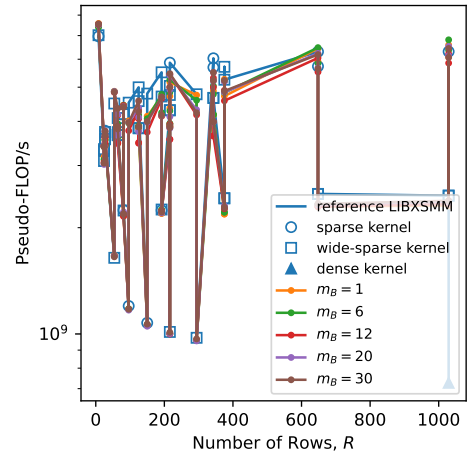
(a) Performance against number of unique \mathbf{A} constants.



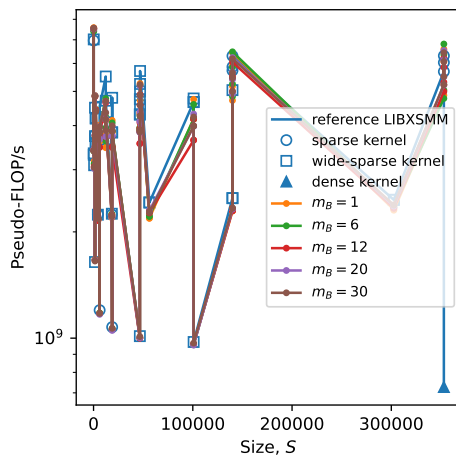
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure E.35: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for PyFR hexahedral element operator matrices. Benchmark run on c5n.xlarge machine.

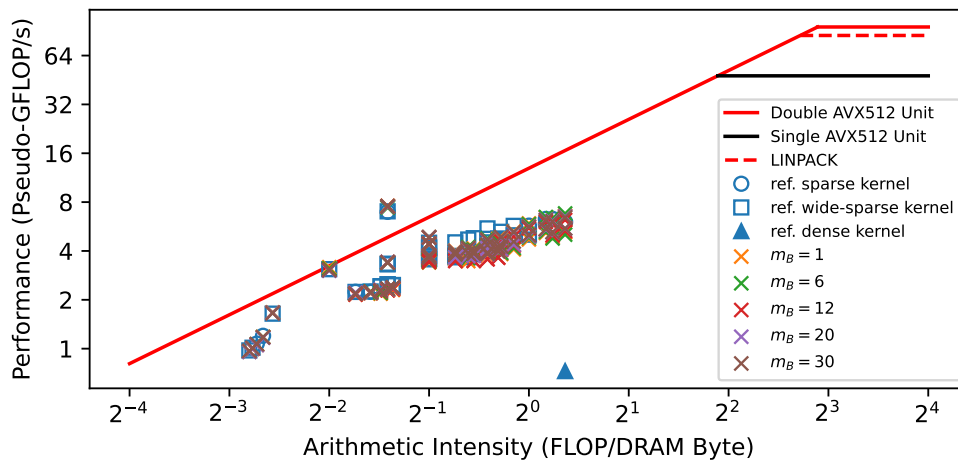
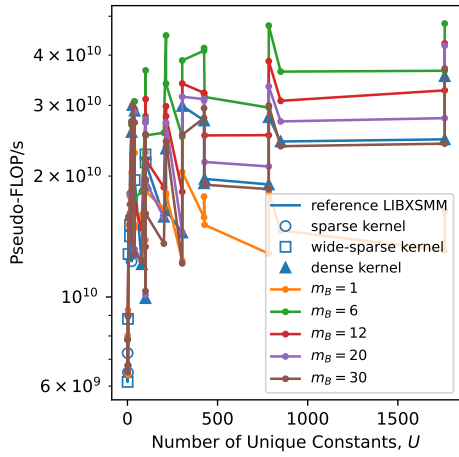
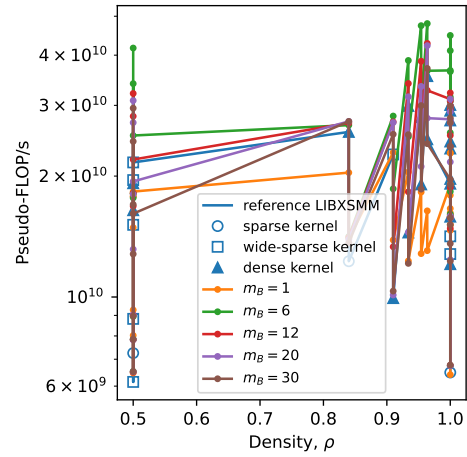


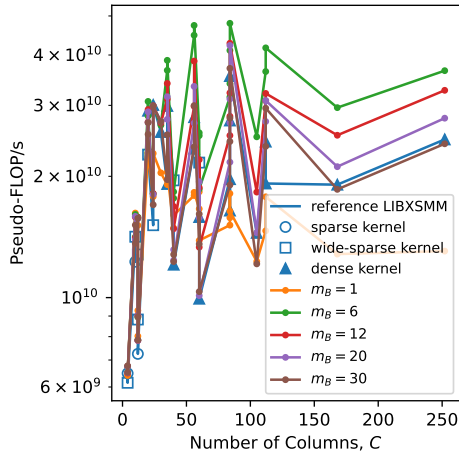
Figure E.36: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR hexahedral element operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



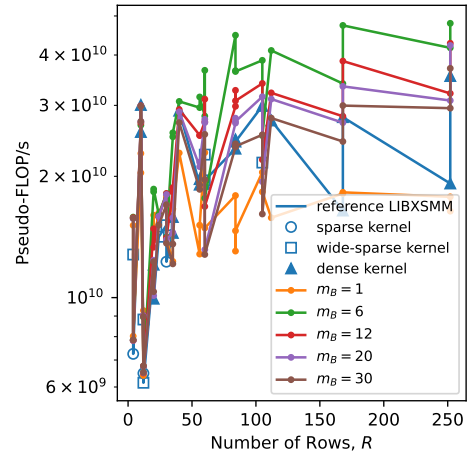
(a) Performance against number of unique A constants.



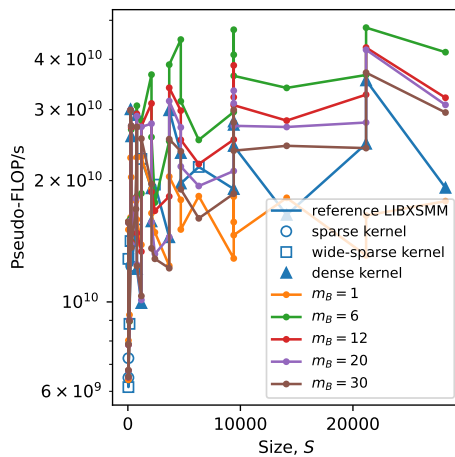
(b) Performance against A density.



(c) Performance against number of A columns.



(d) Performance against number of A rows.



(e) Performance against number of A size.

Figure E.37: Runtime broadcasting with loading A from memory, caching B strides and M blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral element operator matrices. Benchmark run on c5n.xlarge machine.

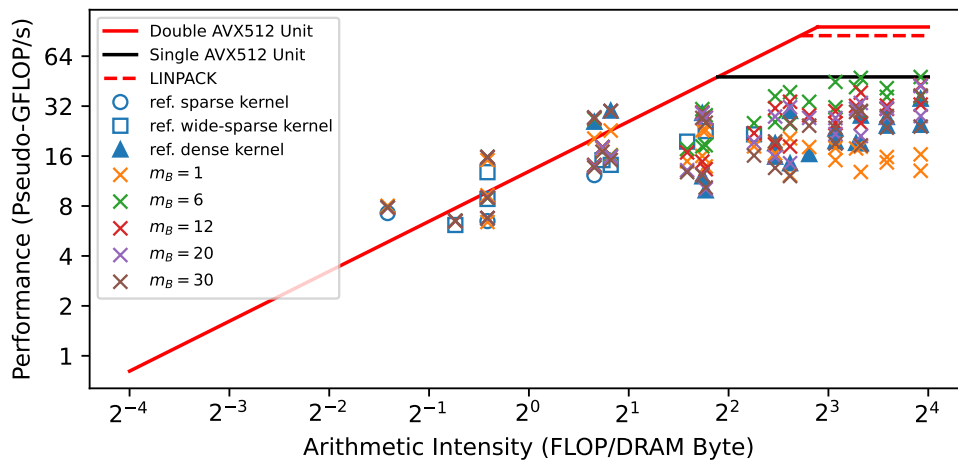
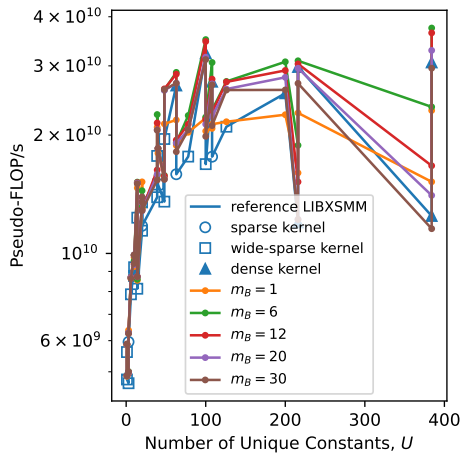
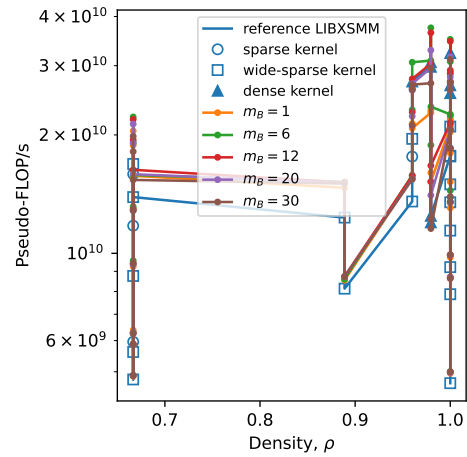


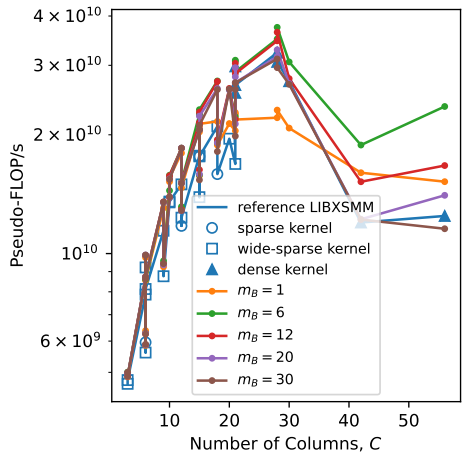
Figure E.38: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR tetrahedral element operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.



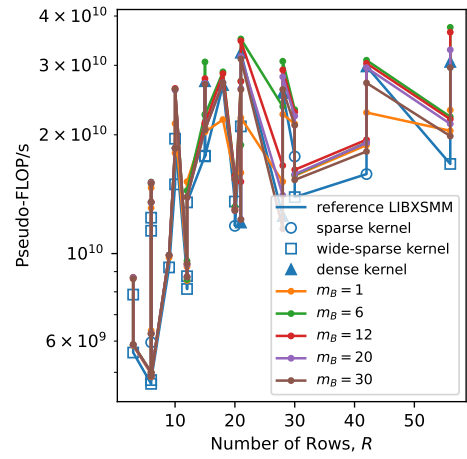
(a) Performance against number of unique A constants.



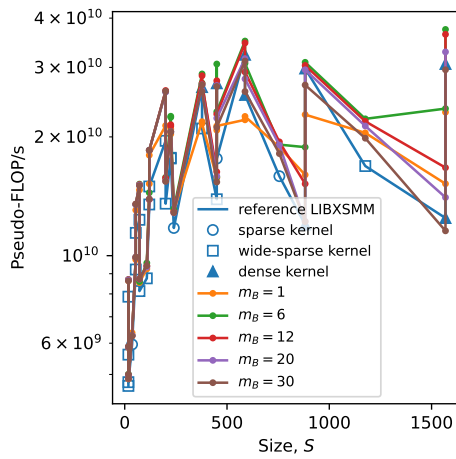
(b) Performance against A density.



(c) Performance against number of A columns.



(d) Performance against number of A rows.



(e) Performance against number of A size.

Figure E.39: Runtime broadcasting with loading A from memory, caching B strides and M blocking vs. reference LIBXSMM implementations, for PyFR triangular element operator matrices. Benchmark run on `c5n.xlarge` machine.

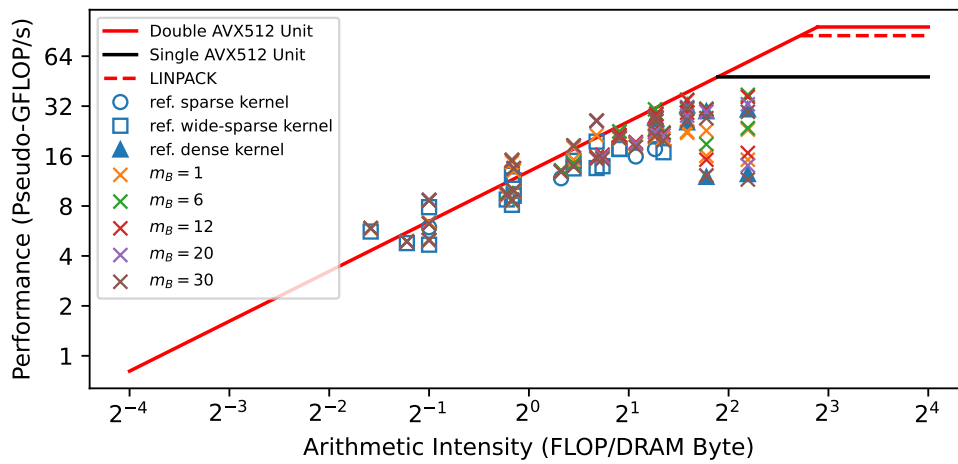
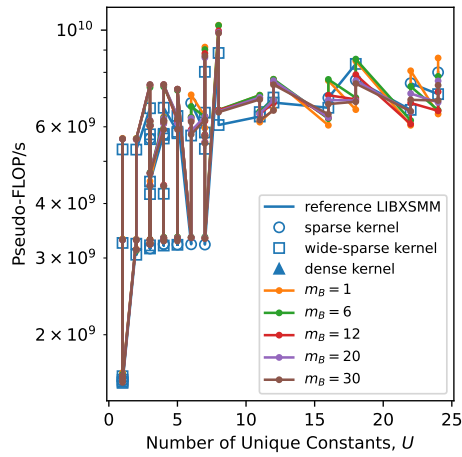
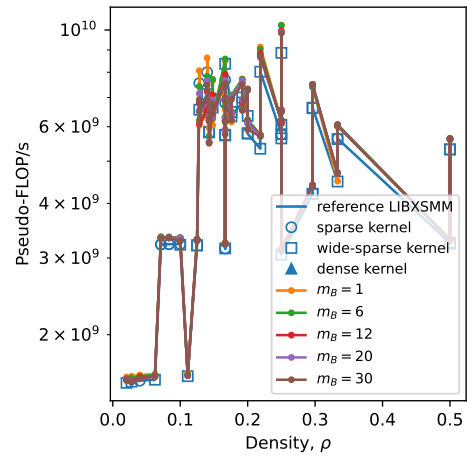


Figure E.40: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR triangular element operator matrices - roofline plots. Benchmark run on c5n.xlarge machine.

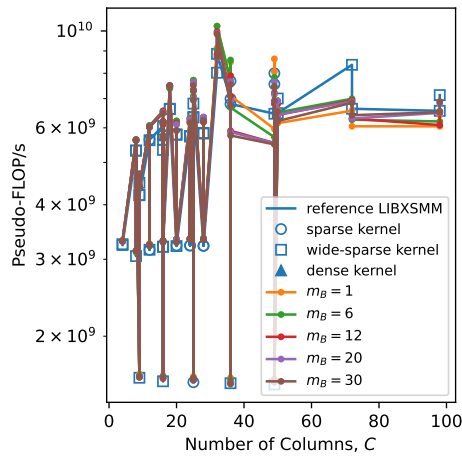
Benchmark Run on m5n.xlarge



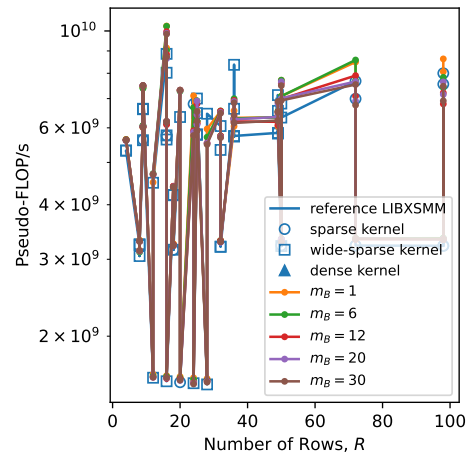
(a) Performance against number of unique \mathbf{A} constants.



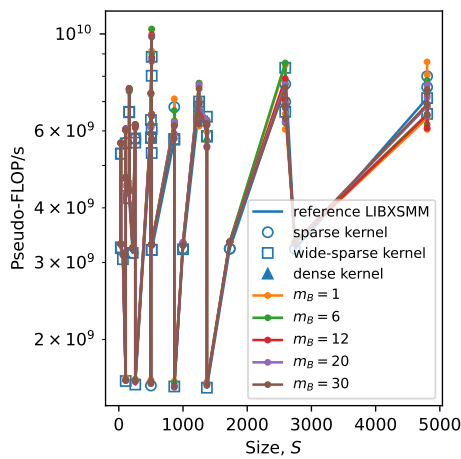
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure E.41: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for PyFR quadrilateral element operator matrices. Benchmark run on m5n.xlarge machine.

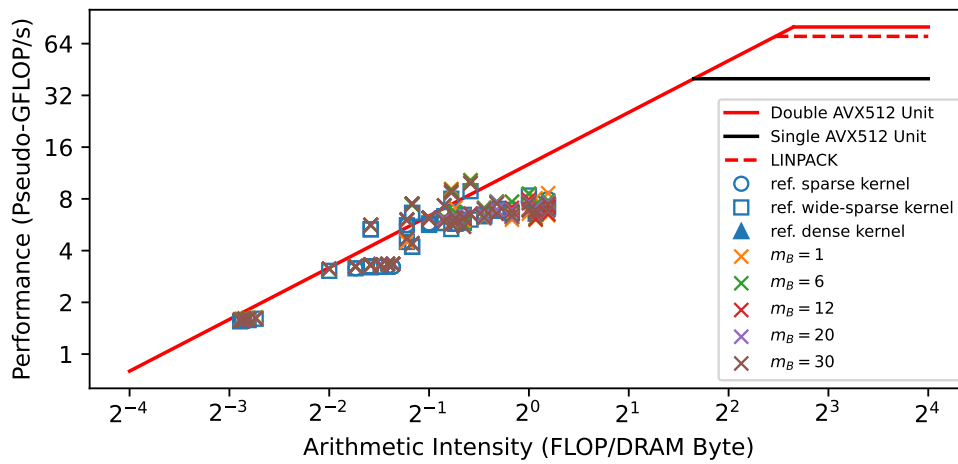
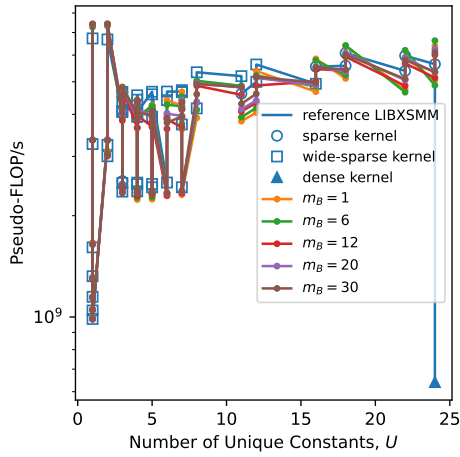
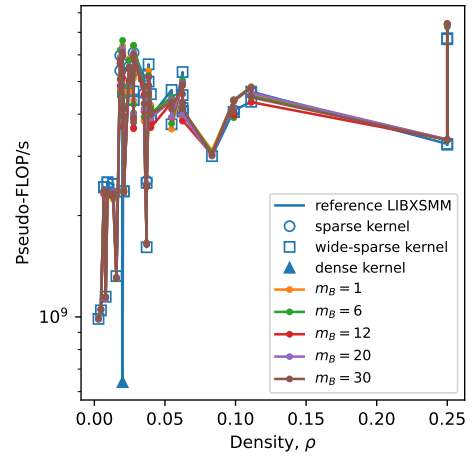


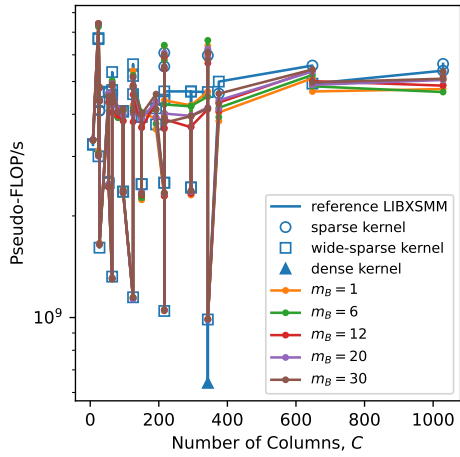
Figure E.42: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR quadrilateral element operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



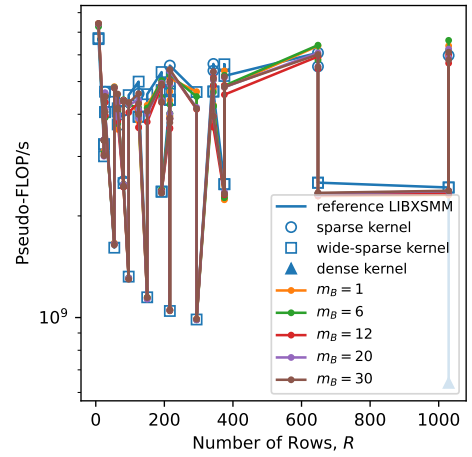
(a) Performance against number of unique \mathbf{A} constants.



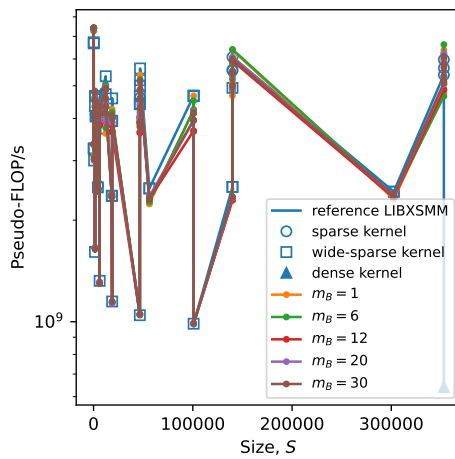
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure E.43: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for PyFR hexahedral element operator matrices. Benchmark run on m5n.xlarge machine.

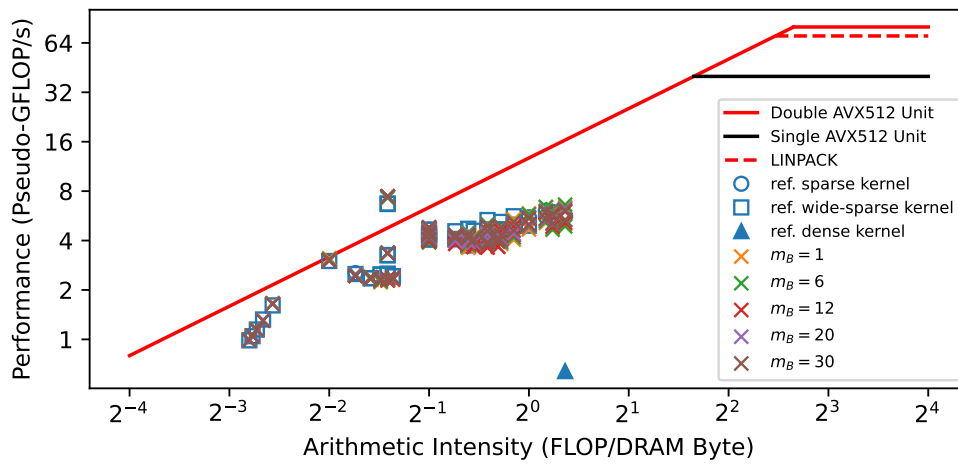
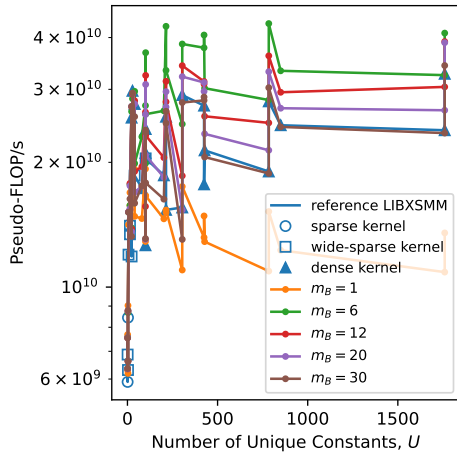
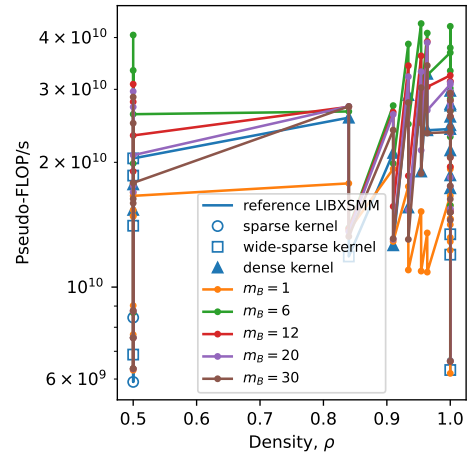


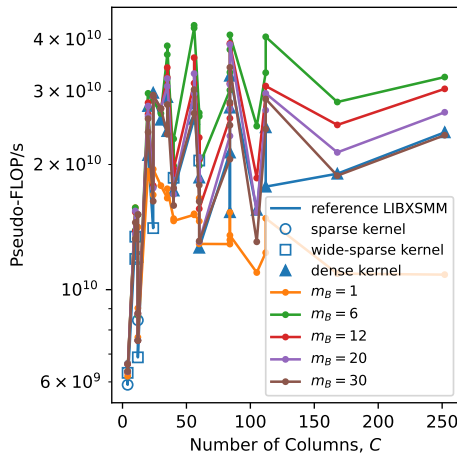
Figure E.44: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR hexahedral element operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



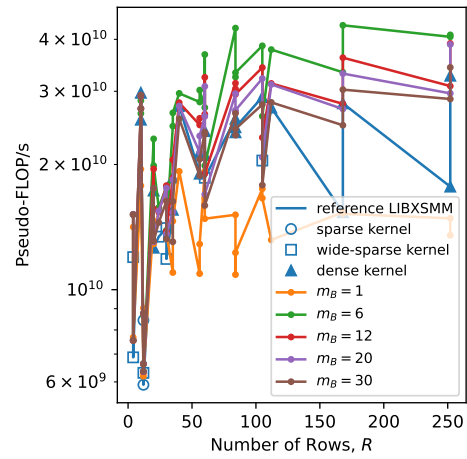
(a) Performance against number of unique A constants.



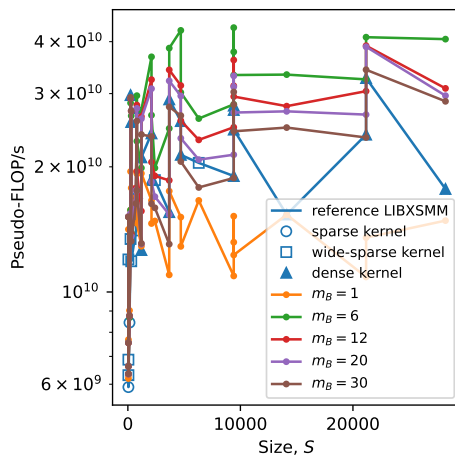
(b) Performance against A density.



(c) Performance against number of A columns.



(d) Performance against number of A rows.



(e) Performance against number of A size.

Figure E.45: Runtime broadcasting with loading A from memory, caching B strides and M blocking vs. reference LIBXSMM implementations, for PyFR tetrahedral element operator matrices. Benchmark run on m5n.xlarge machine.

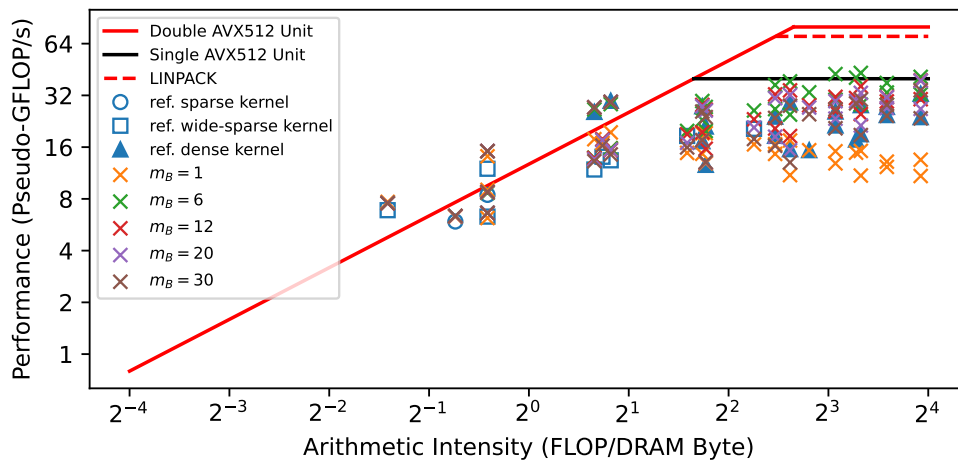
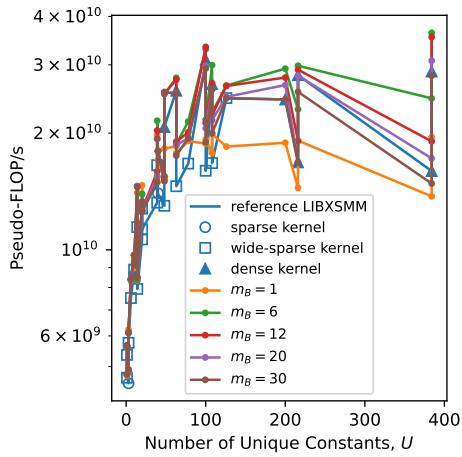
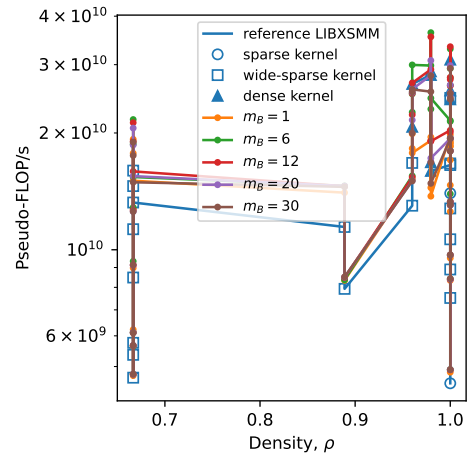


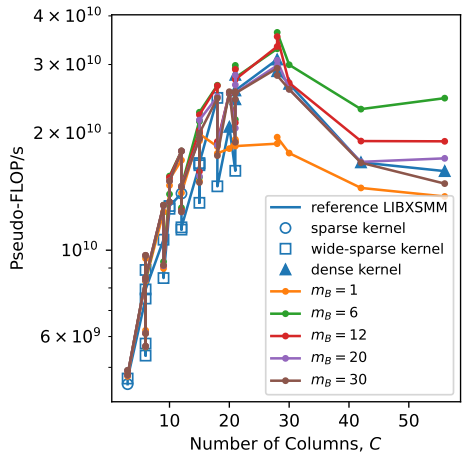
Figure E.46: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR tetrahedral element operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.



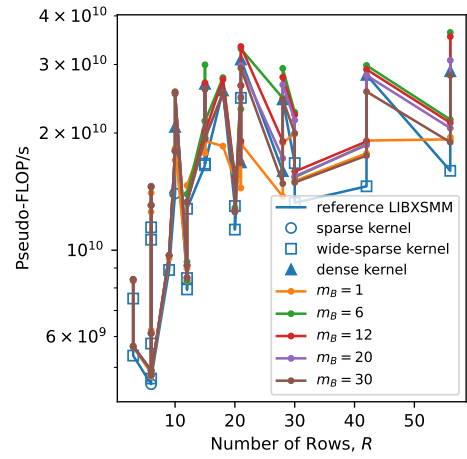
(a) Performance against number of unique \mathbf{A} constants.



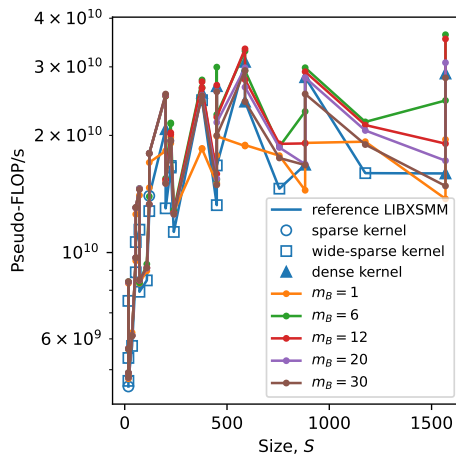
(b) Performance against \mathbf{A} density.



(c) Performance against number of \mathbf{A} columns.



(d) Performance against number of \mathbf{A} rows.



(e) Performance against number of \mathbf{A} size.

Figure E.47: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for PyFR triangular element operator matrices. Benchmark run on m5n.xlarge machine.

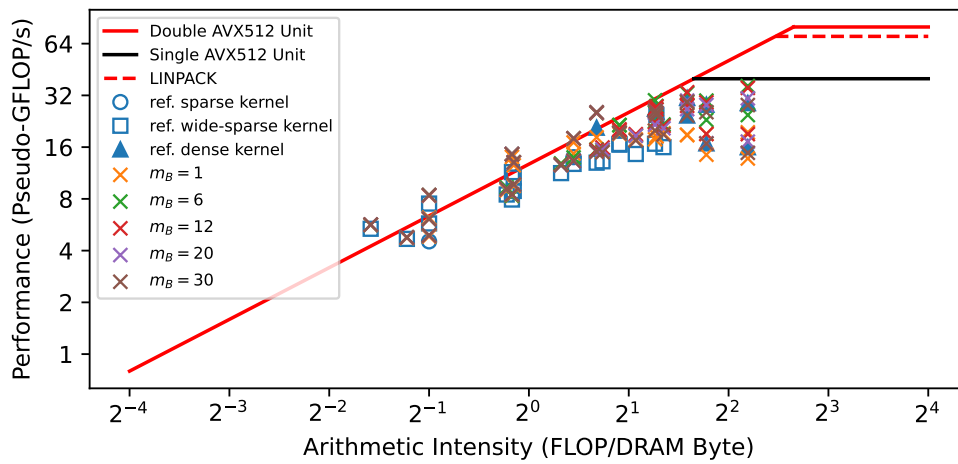
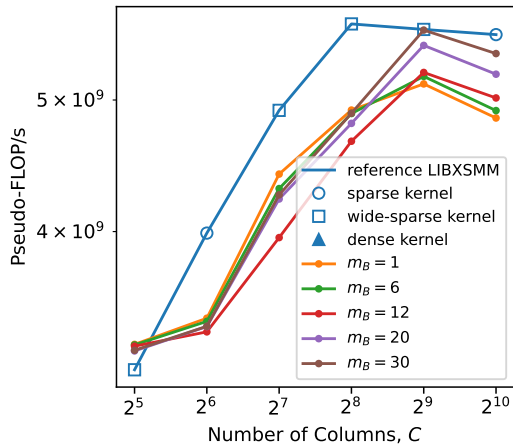


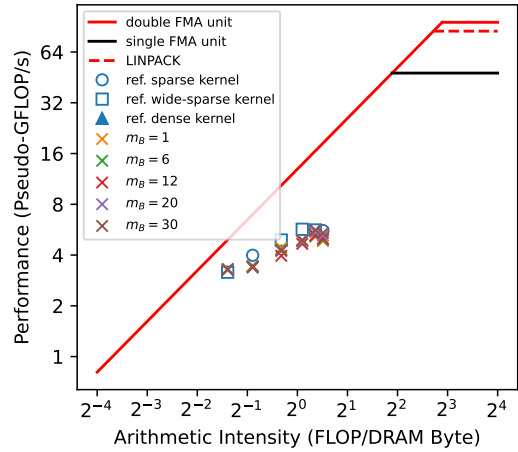
Figure E.48: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations for PyFR triangular element operator matrices - roofline plots. Benchmark run on m5n.xlarge machine.

Synthetic Matrices

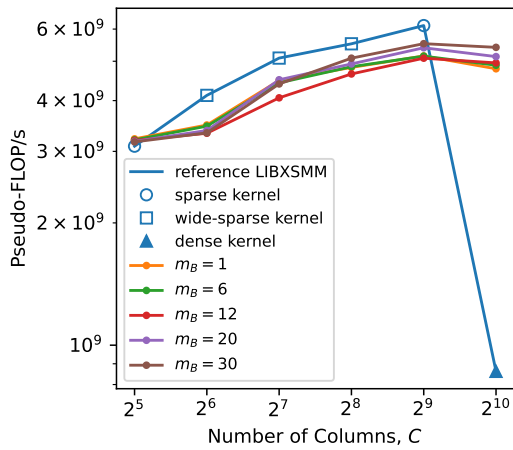
Benchmark Run on c5n.xlarge



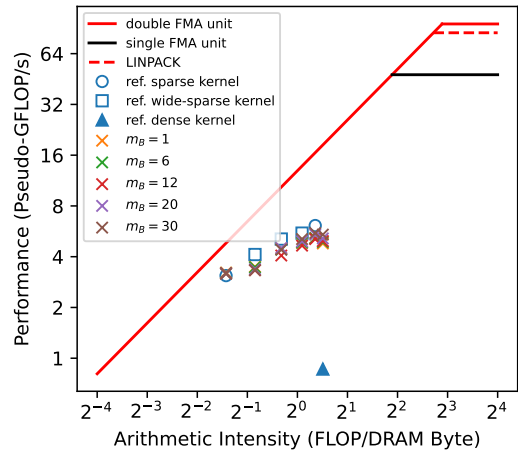
(a) Performance vs. number of columns, $U = 16$.



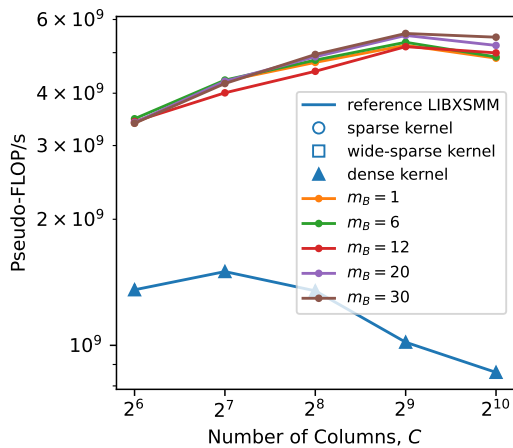
(b) Roofline plot, $U = 16$.



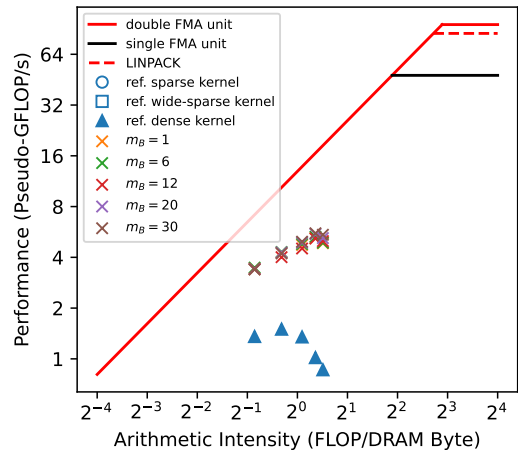
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

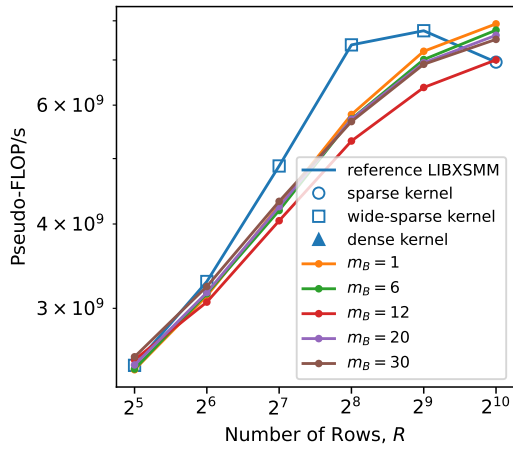


(e) Performance vs. number of columns, $U = 256$.

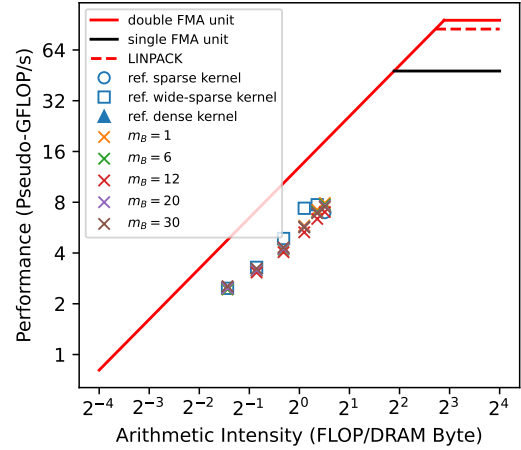


(f) Roofline plot, $U = 256$.

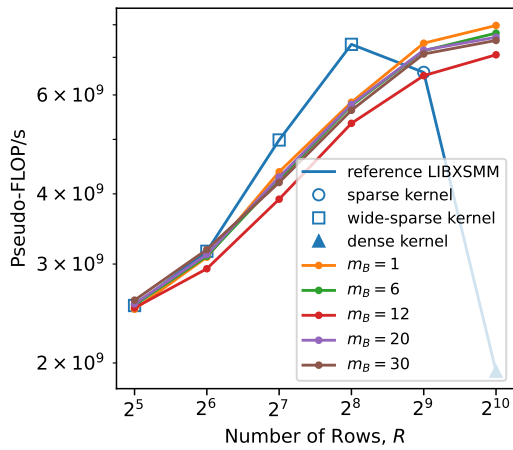
Figure E.49: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on c5n.xlarge machine



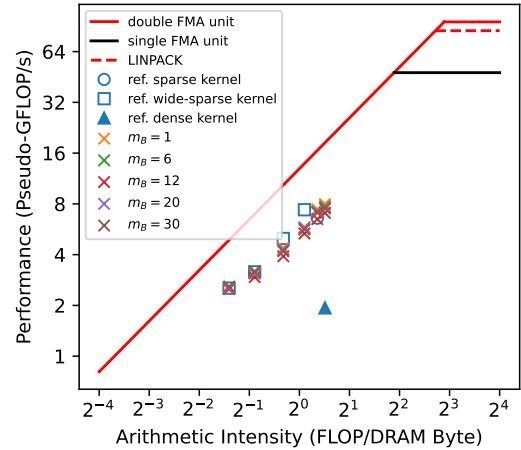
(a) Performance vs. number of rows, $U = 16$.



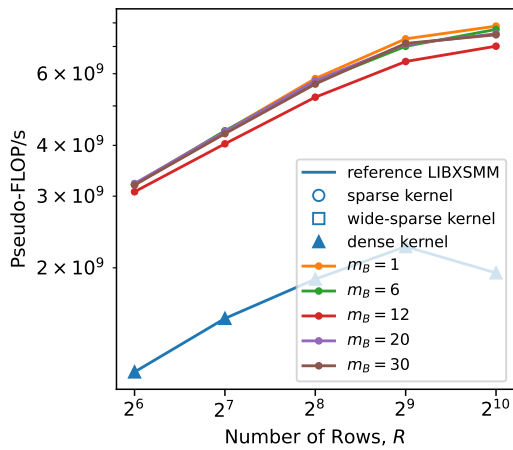
(b) Roofline plot, $U = 16$.



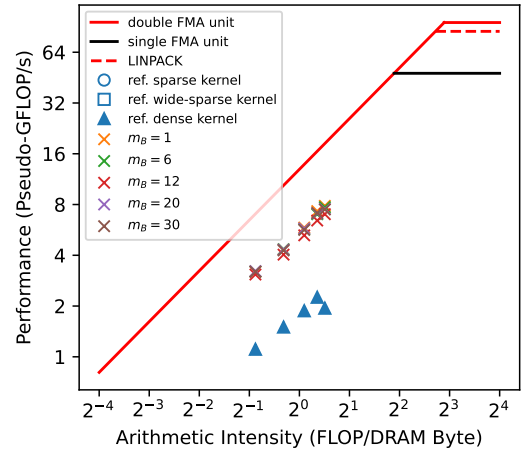
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

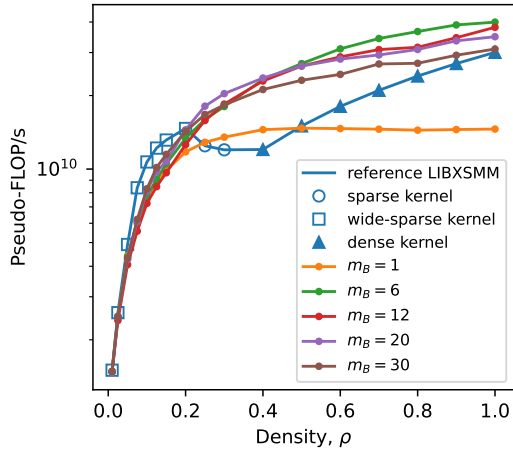


(e) Performance vs. number of rows, $U = 256$.

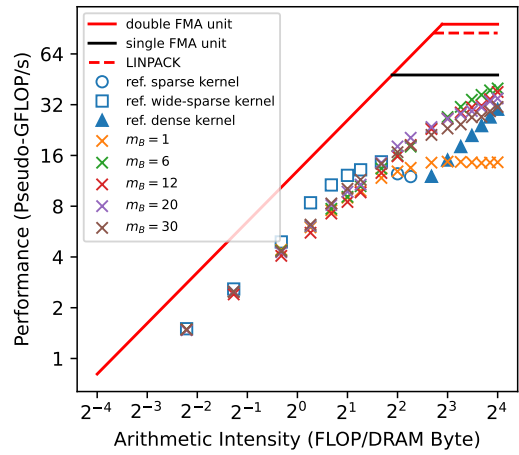


(f) Roofline plot, $U = 256$.

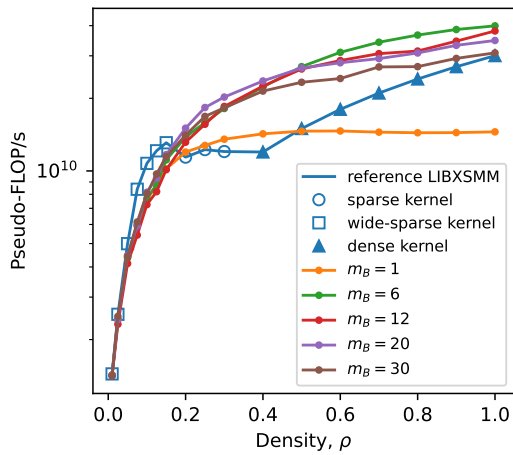
Figure E.50: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on c5n.xlarge machine



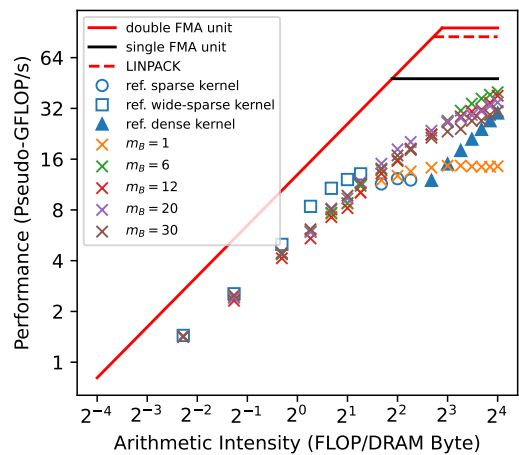
(a) Performance vs. density, $U = 16$.



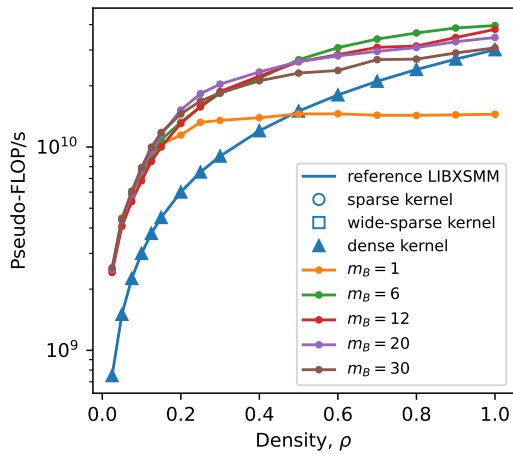
(b) Roofline plot, $U = 16$.



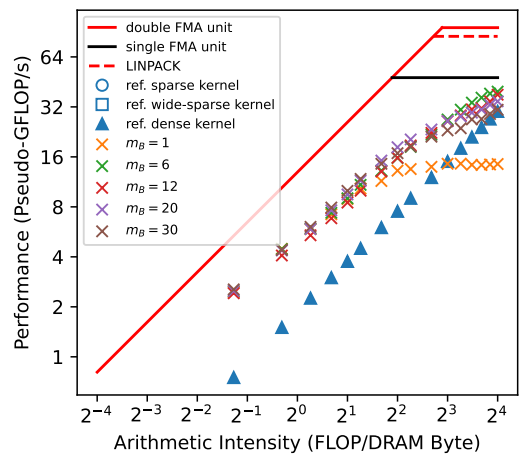
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

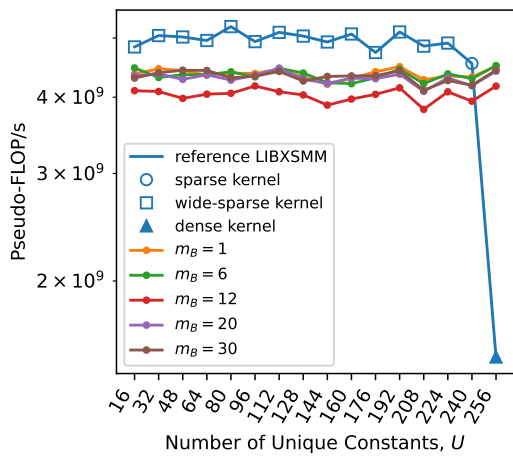


(e) Performance vs. density, $U = 256$.

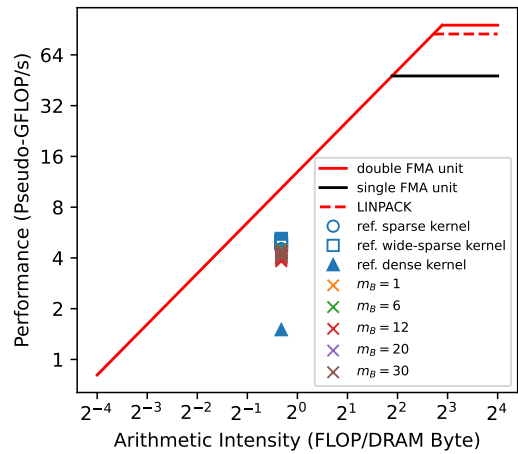


(f) Roofline plot, $U = 256$.

Figure E.51: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on c5n.xlarge machine



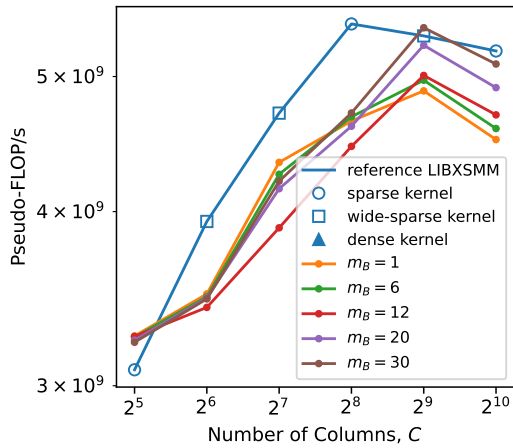
(a) Performance vs. number of unique absolute non-zero values.



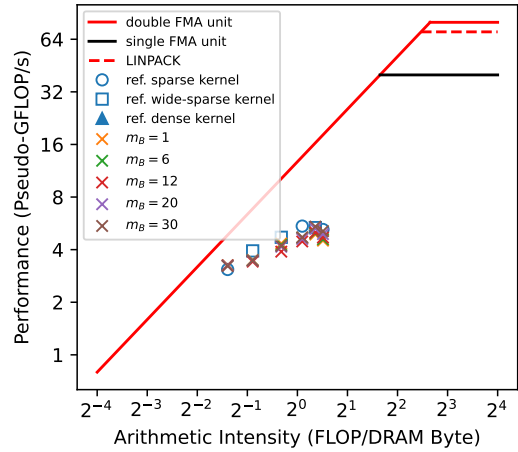
(b) Roofline plot.

Figure E.52: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on c5n.xlarge machine

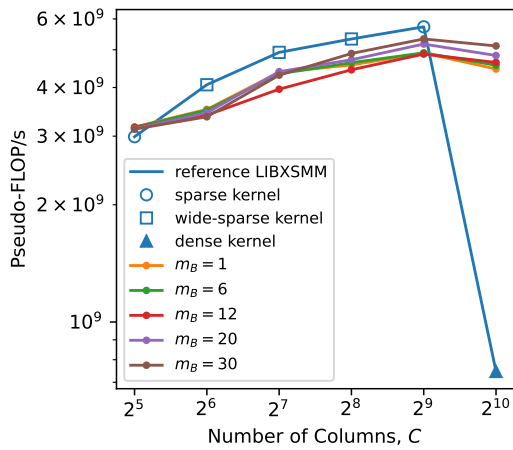
Benchmark Run on m5n.xlarge



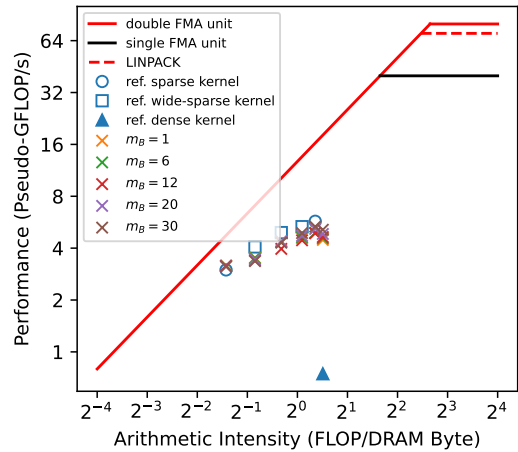
(a) Performance vs. number of columns, $U = 16$.



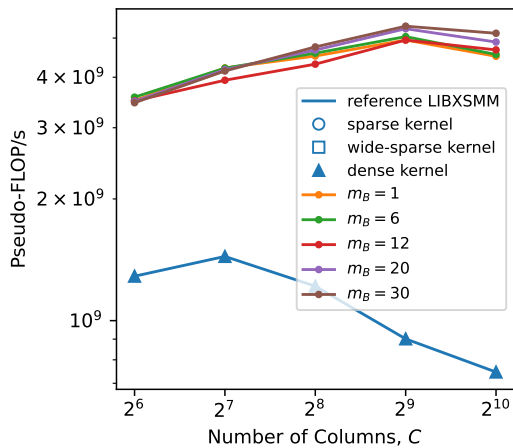
(b) Roofline plot, $U = 16$.



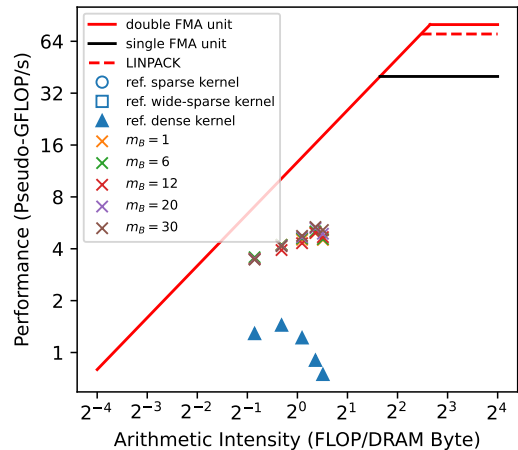
(c) Performance vs. number of columns, $U = 64$.



(d) Roofline plot, $U = 64$.

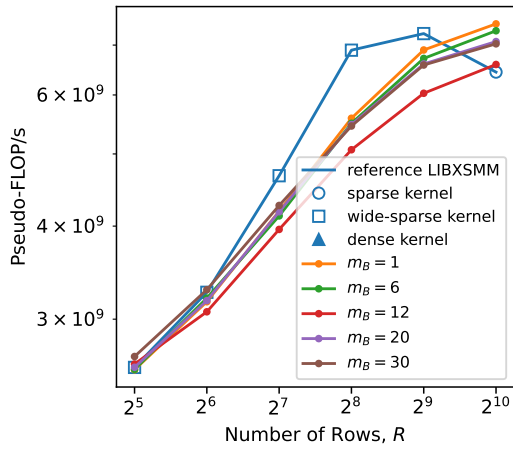


(e) Performance vs. number of columns, $U = 256$.

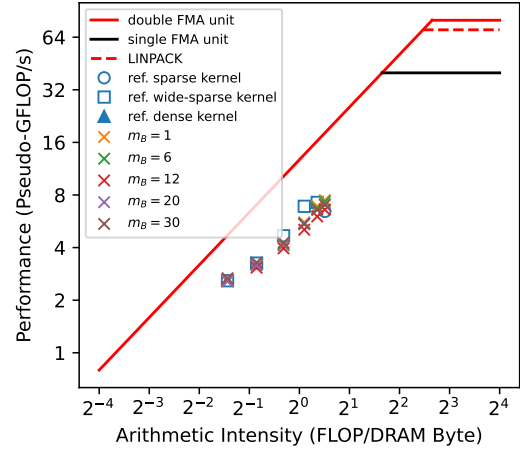


(f) Roofline plot, $U = 256$.

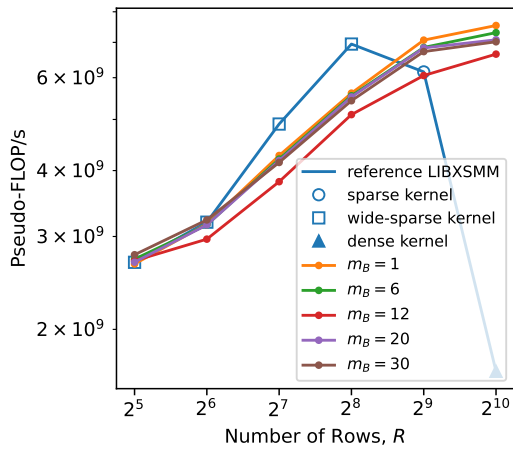
Figure E.53: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} columns. Benchmark run on m5n.xlarge machine



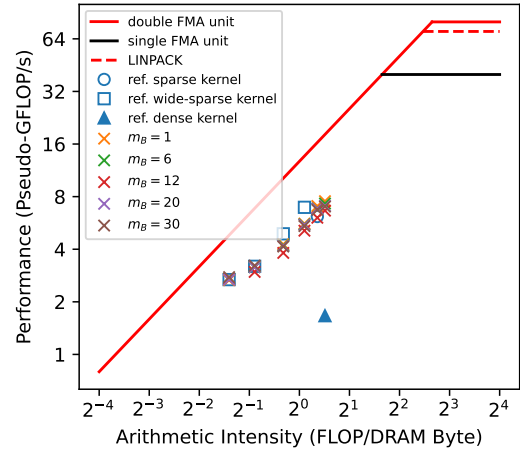
(a) Performance vs. number of rows, $U = 16$.



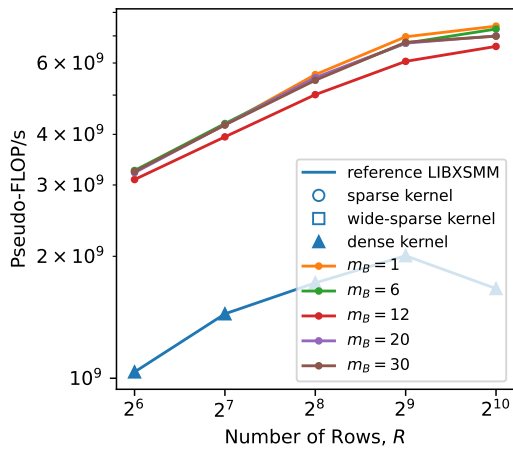
(b) Roofline plot, $U = 16$.



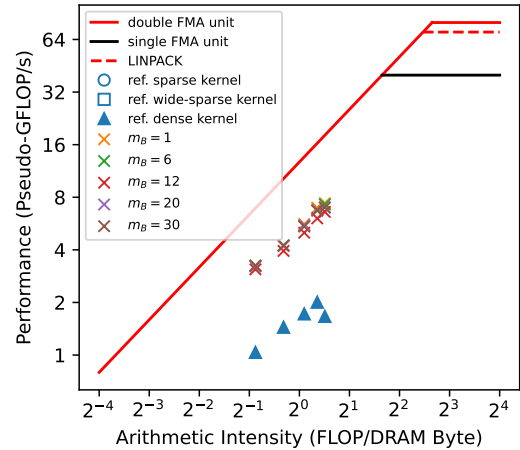
(c) Performance vs. number of rows, $U = 64$.



(d) Roofline plot, $U = 64$.

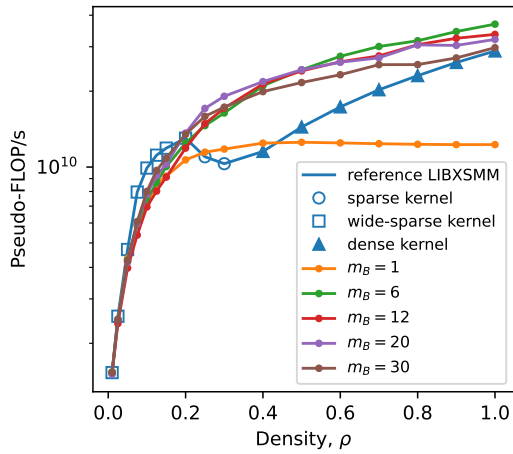


(e) Performance vs. number of rows, $U = 256$.

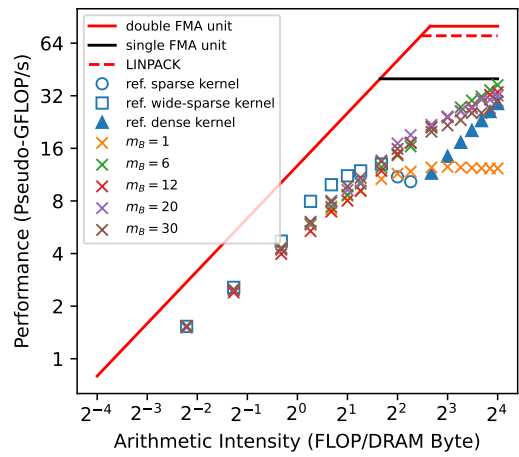


(f) Roofline plot, $U = 256$.

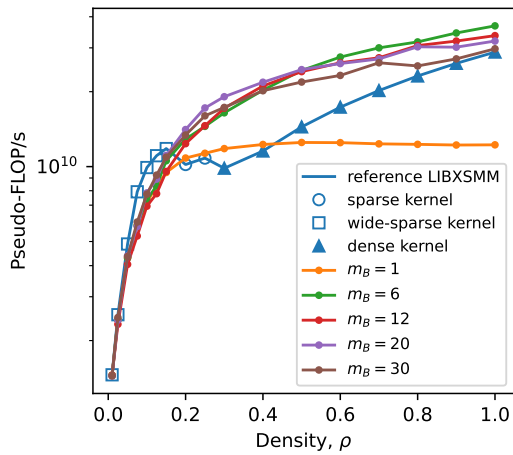
Figure E.54: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying number of \mathbf{A} rows. Benchmark run on m5n.xlarge machine



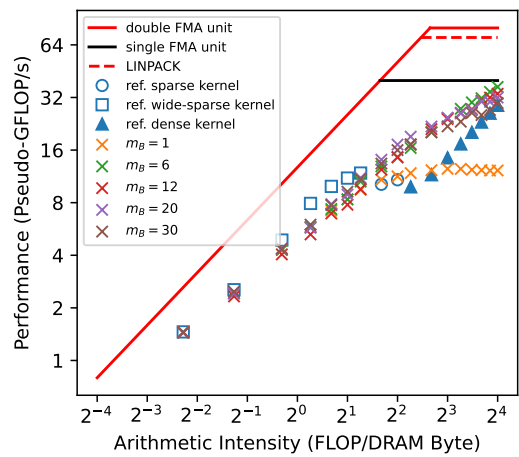
(a) Performance vs. density, $U = 16$.



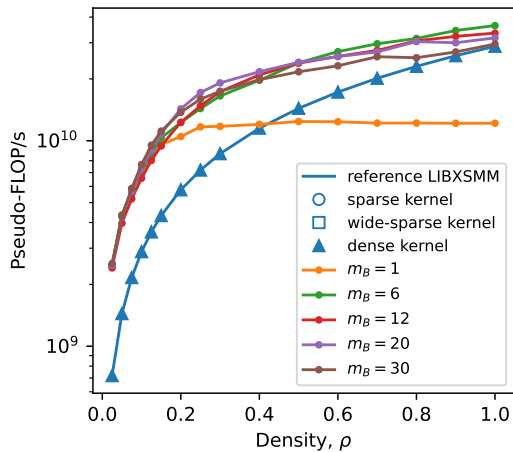
(b) Roofline plot, $U = 16$.



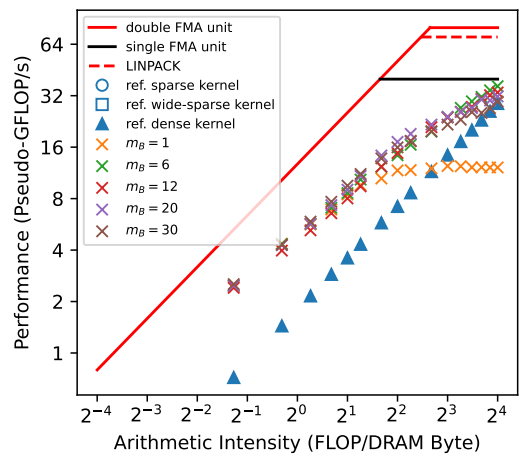
(c) Performance vs. density, $U = 64$.



(d) Roofline plot, $U = 64$.

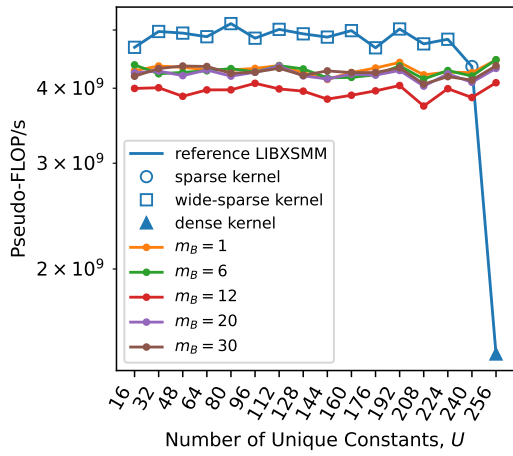


(e) Performance vs. density, $U = 256$.

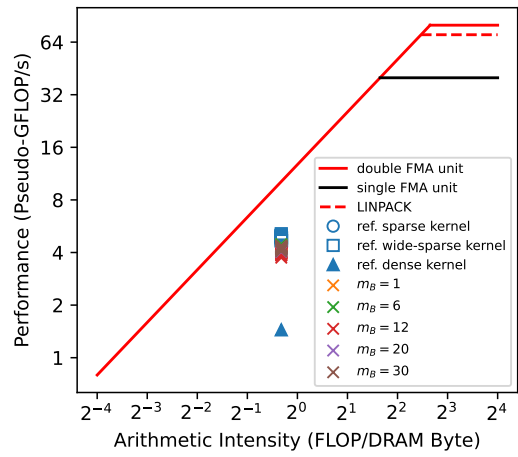


(f) Roofline plot, $U = 256$.

Figure E.55: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying \mathbf{A} density. Benchmark run on m5n.xlarge machine



(a) Performance vs. number of unique absolute non-zero values.



(b) Roofline plot.

Figure E.56: Runtime broadcasting with loading \mathbf{A} from memory, caching \mathbf{B} strides and M blocking vs. reference LIBXSMM implementations, for synthetic matrices with varying the number of unique absolute non-zero constants in \mathbf{A} . Benchmark run on m5n.xlarge machine