Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Using Register Packing for Small Sparse Matrix Multiplication

Author:
Mehedi Vin Paribartan

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Kin Leung

June 17, 2020

Abstract

Matrix multiplication (MM) is a well studied operation and is used heavily in multiple areas of science and engineering. This thesis is motivated by the MM that occurs in PyFR - an application that implements Flux Reconstruction schemes for higher-order fluid flow simulations on unstructured meshes. Most of the computation time within PyFR is spent carrying out MM with repeated uses of small operator matrices on multiple, large operand matrices, in a block-by-panel style. Some simulations can take days to weeks, so it is important to generate as optimal code as possible for the target hardware. Previous work resulted in GiMMiK - a library that can be used to accelerate MM on a range of hardware, including GPUs and CPUs. Intel has an open-source library, LIBXSMM, which is a Just-In-Time compiler that can also be used to accelerate the MM found within PyFR. LIBXSMM is also used to accelerate deep learning applications and is part of Intel's support to accelerate both the PyTorch and TensorFlow libraries on Intel hardware. LIBXSMM can generate code for the AVX-512 hardware found on Intel Skylake-SP CPUs, which is the target hardware in this thesis. The operator matrices encountered in PyFR applications contain repeated unique constant values. LIBXSMM contains a specialised small-sparse operator matrix routine that stores the unique non-zero values of the operator matrix in the the vector registers, supporting up to 31 unique non-zero constants. This routine is used by PyFR but many of the typical operator matrices encountered in its application contain more than 31 unique non-zero values and hence cannot be used for those matrices. We present methods to enhance the specialised routine to support operator matrices with up to 240 unique double precision values, an increase from the 31 supported by the reference LIBXSMM version. This is achieved by packing the unique values within the vector registers and efficiently broadcasting them at runtime for use within the MM routine. This method can free up registers which can then be used to implement further optimisations and provide speedups between 0.997 and 11.5 times over the reference LIBXSMM version, over a suite of 146 example operator matrices that can be encountered in PyFR applications. We show that the enhanced LIBXSMM can offer speedups between 0.484 and 5.457 times over GiMMiK over the same suite of example matrices and we suggest when PyFR users should chose one over the other.

As a result of this thesis, we made contributions to Intel's open-source LIBXSMM. This contribution enhanced the specialised small sparse operator matrix routine used by PyFR and increased the number of unique non-zero constants in the operator matrix from 31, to 240 for double precision and 480 for single precision.

Acknowledgements

I would like to thank Prof. Paul Kelly and Freddie Witherden of Texas A&M University for their guidance and incredible support on this project, and for taking the time for numerous meetings. I would like to thank Alex Heinecke, Senior Research Scientist at Intel Labs for also providing support and creative ideas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Matrix multiplication (MM) is a well studied operation due to demand for an efficient and performant implementation, arising from its heavy use in multiple areas of science and engineering. The are many libraries that conform to the Basic Linear Algebra Subprograms (BLAS) interface, which focus on how to optimally implement the BLAS routines on specific hardware architectures. The generic BLAS routines are not always optimal in cases where additional information is known before-hand. If the operator matrix is known to be sparse and relatively small (under 100 x 100), displayed in Figure 1.1, we can achieve a more efficient routine. The motivation for this thesis originates from the block-by-panel case of matrix multiplication that occurs in Flux Reconstruction implementations, which aims to simulate fluid flows [12]. Due to the popularity of matrix multiplication, we believe generating more optimal routines would help other areas of research as well. LIBXSMM is an open-source Intel library for specialised dense and sparse matrix multiplication [3]. The library is part of Intel's support to accelerate deep learning applications and is used by both the PyTorch and TensorFlow libraries to improve performance on Intel hardware. CP2K is a quantum chemistry and solid state physics software package that can use LIBXSMM to processes batches of small matrix multiplications that originate from distributed block-sparse matrix with problem-specific small matrices [3]. CP2K uses around 7% of the core-hours on ARCHER, the UK national supercomputer for science which can run with over $10,000$ CPU cores, and is the second highest ranked application by usage on the system [30]. Users of the software packages listed above and many more [3] will be able to benefit from contributions to LIBXSMM as a result of this work.

1.1 Context

1.1.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is a family of techniques that use numerical analysis to estimate the flow of fluids as well as the interaction between the fluids and surface

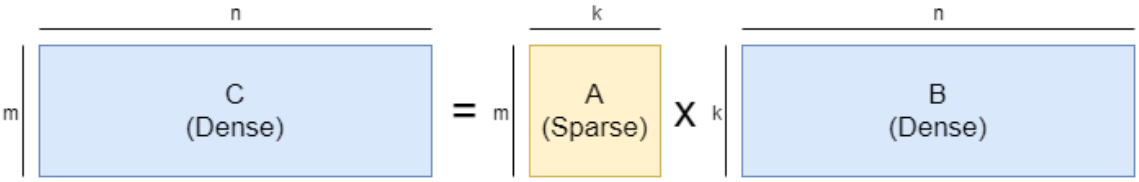


Figure 1.1: Visualisation of a block-by-panel type of matrix multiplication arising from Flux Reconstruction. The operator matrix is typically small, square and sparse. The operand and output matrices are typically fat and dense, implying $N >> M, K$ and $M \approx K$

areas. High-order (third and above) methods can potentially give more accurate results at the same computational cost as low-order methods. Traditionally, high-order methods were harder to implement and less robust, thus they had a lower adoption in both industry and academia.
In 2007 H.T. Huynh [12] presented the Flux Reconstruction (FR) approach, a single mathematical framework that unifies multiple high-order schemes. The framework made it simple to create economical solutions for high-order CFD problems. FR has good element locality lending itself to be efficiently run on streaming architectures, such as GPUs and CPUs with Single Instruction Multiple Data (SIMD) instruction sets.
PyFR [7] is an open-source Python library that implements the FR approach to solve advection-diffusion type problems on unstructured grids (known as meshes) and is designed to target a range of hardware architectures. In FR, partial differential equations are converted into block-by-panel style matrix multiplications where the operator matrix is sparse and heavily reused. The operator matrices also have a relatively *small amount of unique non-zero elements*.
In 2014, Wozniak [34] extended PyFR with GiMMiK to generate bespoke kernels to take advantage of the additional information known about the matrices, which provided performance speedups over generic BLAS libraries when running on GPUs. Section 2.2.3 covers GiMMiK in further detail. Other commercial libraries have more recently been released, such as cuSPARSE by NVIDIA [5] and MKL 'Inspector-Executor' routines by Intel [4], which also utilise additional information about the operator matrix. PyFR can target AVX-512 architectures via Intel's LIBXSMM [3], an open-source library for specialised sparse and dense matrix multiplications. In 2019 Price [23] evaluated VecReg, which also generates routines to run on AVX-512. Price compared VecReg against LIBXSMM with a suite of PyFR example operator matrices and obtained speedups in some cases.

1.1.2 Convolutional Neural Networks - An unrelated area that can benefit from this work

A type of deep learning class is Convolutional Neural Networks (CNN), typically used in the image processing domain. In a CNN, there are convolutional layers, where the input is convolved with a (learned) filter. The 'lowering method' [28] is a commonly applied technique to carry out the convolutions, where they are transformed into matrix multiplications. It can be common for the operator matrix arising from the filter to be sparse and have a small amount of unique non-zeros. As the convolution is applied repeatedly to many inputs during training, improving the speed of this operation can be beneficial.
In 2017 Park et. al. [13] designed Direct Sparse Convolutions (DSC) as an alternative approach to the lowering method. DSC leads to greater arithmetic intensity by reducing the amount of repeated information stored from the input. DSC can be considered as a 'virtual sparse-matrix-dense-matrix multiplication' [13] and so would benefit from more performant sparse-dense matrix multiplication routines.
Whilst not a direct motivator for this thesis, CNNs and other machine learning areas are an example of other scientific applications that could benefit from the work in this thesis.

1.2 Objectives

The current limitation with LIBXSMM's highly specialised sparse operator matrix multiplication routine that is used by PyFR, is that it only supports up to 31 unique non-zeros in the operator matrix. This is due to the method it uses to store the operator matrix within the vector register file (31 registers in AVX-512), where each value stored is repeated to fill the corresponding vector register. This is done to allow it to quickly access it in the

form required for the MM routine. If there are more unique non-zeros, LIBXSMM defaults to a less specialised routine. The first objective of this thesis will be to explore how we can adapt LIBXSMM to support more unique constants via packing for AVX-512 based CPUs.
If the number of unique non-zeros in the operator matrix is less than the total number we can pack within the vector register file, then packing can result in free, unused registers. Price used these free registers for column-based Common Sub-expression Elimination (CSE), but noted that there were many other CSE opportunities. We will explore other methods of utilising available hardware that do not require extensive analysis of the operator matrices, hopefully allowing the further optimisations to apply to the general small-sparse-matrix dense-matrix multiplication case. This thesis will primarily target AVX-512 on the Intel Skylake-SP architecture. The proposed techniques in conjunction with the register packing could be adapted for libraries other than LIBXSMM, to target other computer architectures.

1.3 Contributions

The contributions from this thesis are summarised as follows:

- We explore how to broadcast any selected vector lane, which leads to more unique non-zeros being able to be packed into the vector register file for the sparse-dense matrix multiplication routine within LIBXSMM. We evaluate the performance of kernels generated by this updated routine and report speedups between 0.976 and 10.915 times for some sparse PyFR operator matrices on the Intel Xeon 8175M.

- We compare with a suite of 170 matrices that arise in common PyFR applications, the performance of the new kernels against the characteristics of those operator matrices. We aim to provide an exhaustive evaluation with this diverse set of operator matrices, and to highlight when the new kernels perform faster.

- We present a synthetic suite of operator matrices that are used to further evaluate the new code generator. By controlling the other characteristics of an operator matrix, we are able to confidently draw conclusions about the effect on performance when one characteristic varies at a time.

- We experimentally explore the impact on performance when free registers are used to increase the parallelism of the kernel, by working on accumulating multiple strides of C concurrently. We report speedups between 0.997 and 11.500 times using these optimisations over the reference LIBXSMM for some sparse PyFR operator matrices.

- We present a hybrid strategy that uses heuristics to select between existing and newly proposed LIBXSMM routines to achieve good performance based on the characteristics of the operator matrix. The heuristics are formed using the results from the evaluation on the existing and new routines.

- We compare the hybrid strategy against GiMMiK and report speedups between 0.484 and 5.457 times on a suite of example operator matrices encountered in PyFR applications. For the sparse examples (quadrilateral and hexahedra based meshes) we report speedups of up to 5.457 times. When a solution accuracy of third-order and greater is used for dense examples (triangle and tetrahedra based meshes), we report speedups of up to 2.325 times.

1.3.1 Contributions to Intel's Open-Source LIBXSMM

The following contributions are made to Intel's open-source LIBXSMM as a result of this thesis:

- We submit code [20] to the LIBXSMM repository on GitHub, that increases the number of unique non-zeros supported by the sparse-dense routine from 31 to 176 (224) for DP (SP) data types.
- We submit code [21] to the LIBXSMM repository on GitHub, that further increases the number of unique non-zeros supported by the sparse-dense routine to 240 (480) for DP (SP) data types. This method builds on top of the first submission, but the former is preferred for fewer than 176 (224) DP (SP) unique non-zeros.

In Chapters 2 and 3, we cover various background information required to understand the remainder of the thesis, and a range of prior and related work. Chapter 4 outlines the evaluation environment and the test-suite, and covers the methodology used for evaluation. In Chapters 5 and 6, we describe and discuss our experiments on implementing register packing within LIBXSMM. In Chapter 7 we explore and evaluate a method to further increase the number of unique non-zeros compared to the register packing solution from Chapter 5. Register packing can result in free registers and in Chapter 8, we evaluate a method to utilise the free registers for potential performance speedups. Methods from Chapters 7 and 8 are combined and evaluated in Chapter 9. We then propose a new strategy that chooses between routines based on a few heuristics, and compare it against GiMMiK in Chapter 10. Finally, Chapter 11 will contain a summary of the results and discussion of future work.

Chapter 2

Background

In this chapter we first provide a brief overview of Computational Fluid Dynamics (CFD) and Flux Reconstruction (FR). In Section 2.2 the Python library PyFR is covered in more depth, including its approaches to heterogeneous computing. We cover examples from a suite of 170 matrices that arise in common PyFR applications. The evaluation will be based on this suite, and so we discuss the characteristics of the matrices. Lastly, we cover the Intel AVX-512 SIMD architecture and how the open-source library LIBXSMM targets this architecture for optimised sparse-dense matrix multiplication.

2.1 Flux Reconstruction Overview

In CFD there are three basic physical laws that state that mass, momentum and energy are conserved within a closed system. The Navier-Stokes equations are commonly used to describe the conservation of momentum in a fluid flow. The need to conserve other physical properties leads to continuity equations being solved in tandem. There are variants of the equation, such as if the flow is compressible or not. Regardless of the variation, they are given as a Partial Differential Equation. Analytical solutions, if possible in theory, are often too complex and so are computationally expensive to solve, so numerical methods are used to obtain estimated solutions.
In the numerical solutions, time and space are discretised. Three of the most popular spatial discretisations are; Finite Volume (FV) 'where the governing system is discretised onto a structured grid of points', Finite Difference (FD) 'where the domain is decomposed into cells and an integral form of the problem is solved within each cell', Finite Element (FE) 'where the domain is decomposed into elements inside of which sits a polynomial that is required to satisfy a variational form of the governing system' [33]. These methods are often implemented with first-order or second-order accuracy. The order of the accuracy determines how the error in the solution will respond to a change in the resolution of the grid [33]. It is possible to implement the schemes with higher-orders, but the computational cost increases rapidly. Spectral Difference (SD) is a more recent class of high-order methods [31], which involve decomposing within the frequency space. However, they have issues with geometrical flexibility.
Flux Reconstruction provides the superior accuracy of high-order spectral or finite difference methods with the geometrical flexibility of low-order finite volume or finite element schemes [7]. Detailed derivations of how the FR framework obtains numerical solutions to the governing equations, via a seven-stage process, are provided by Castonguay et. al. in [19] and Huynh in [12]. The remainder of this thesis should be accessible without having read the additional material.

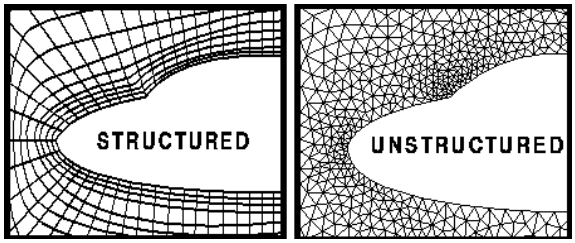


Figure 2.1: NASA FAST guide [18] shows the difference between Structured and Unstructured grids

2.2 PyFR

2.2.1 Overview

PyFR is an open-source Python library, first presented by Witherden et al. [7] that implements the FR approach to solve advection-diffusion type problems and is designed to target a range of hardware architectures [8]. PyFR is a high-order accurate solver. Traditionally, commercial solvers have been low-order accurate and can provide stable results relatively quickly. The downfall with low-order solvers is that they perform poorly when a higher accuracy is desired. PyFR satiates an increasing desire from both industry and academia for a higher accuracy solver to CFD problems, by implementing the FR scheme.
FR has good element locality and so PyFR is able to run the scheme efficiently on modern streaming architectures. Whilst most of the scheme is implemented in Python, PyFR uses external libraries to accelerate the matrix multiplication part of FR. PyFR uses different libraries when targeting different hardware [8].
In FR, partial differential equations can be converted into block-by-panel style matrix multiplications where the operator matrix $\mathbf{A}$ can be sparse, and is heavily reused. The resulting matrix is accumulated with a scalar β (usually either 0 or 1 - i.e. added or not).

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C} \tag{2.1}$$

This gives an opportunity to generate bespoke kernels by analysing the operator matrix. The kernel is then used on multiple, fat matrices ($\mathbf{B}$) over the FR process.

In an unstructured grid, the Euclidean space is discretised using a simple type of shape, where an irregular pattern is found among the shapes. The collection of the shapes in the space is often called the mesh. Figure 2.1 from NASA [18] shows an example of an unstructured grid and how it differs compared to a structured grid.

2.2.2 Operator Matrices

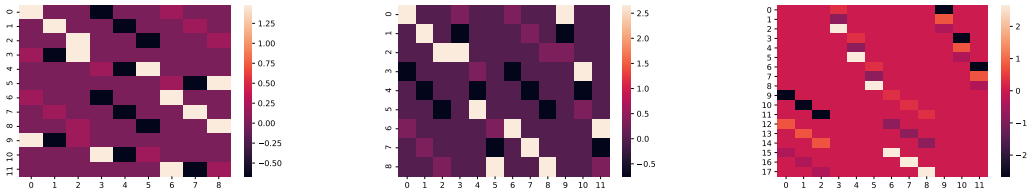
PyFR supports 2D and 3D grids and multiple types of shapes. The operator matrices that are repeatedly used originate from a process of storing *flux* and *solution* points into a file. This is an *arbitrary process* and could be changed to lead to *different patterns in the matrices*, but the process used is encoded into PyFR. This means that for a given PyFR version, every simulation uses the same process. It is also very unlikely, but possible, that the process would change between PyFR versions.

- Throughout this thesis, there will be 5 operator matrices for a given combination of shape, quadrature (numerical integration) and order of accuracy of the solution. Specifically, each set will have an **m0**, **m3**, **m6**, **m132** and **m460**.
- The characteristics (size, density and number of unique non-zeros) of the operator matrices highly depend on the shape used, as well as the quadrature method and also the order of the solution.
- Generally, as the order of the solution increases, so do the dimensions of the operator matrix.
- The pattern characteristic is **not** dependent on FR directly, but as mentioned above, the *arbitrary* storage layout chosen for flux and solutions points within PyFR.

The following sections showcase some properties of example operator matrices from PyFR. Appendix A provides more detail on every operator matrix from the suite of PyFR examples used in the evaluation. The matrices are directly plotted as heatmaps, where each cell represents an element of the matrix and the row and column numbers are labelled on the axes.
Note: the operator matrices are not a representation of the meshes, but represent operations carried out in PyFR.

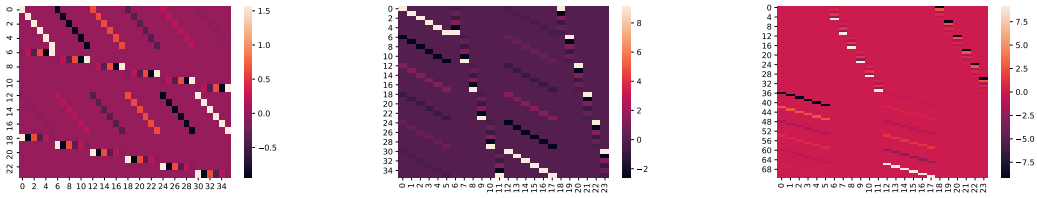
Quadrilaterals

Figures 2.2 and 2.3 show the operator matrices for Quadrilateral shapes using the Gauss-Legendre quadrature. The operator matrices increase in size as the order of the solution increases. Large sections of pink/purple are sections where the elements are all 0, highlighting that the the matrices are sparse. The pattern of the respective operator matrix for a given FR step (**m**_), holds as the order of the solution varies. Blocks of patterns can be seen better at the higher-orders, like in Figure 2.3 where interleaved diagonals span the matrix. The patterns arise from the storage layout of the points in memory and the neighbouring point being considered. For quadrilaterals, two sets of neighbours are considered, those within a stride of 1 and those within a stride of n. The stride of 1 leads to the sparser sections - for example the left half of Figure 2.4a.
Figure 2.4 shows the **m132** operator matrix for Quadrilateral fourth-order with slightly differing quadratures being used. The different stride lengths and their respective sections can be clearly seen. Tables A.1 and A.2 in Appendix A show that the dimensions stay the same, but the density and number of unique non-zeros varies when the quadrature used was varied. Figure 2.5 shows another example of how the operator matrix grows as the order increases. The top half of the matrices have a 'blocky' diagonal, associated with the neighbours within a stride of 1. The bottom half has a different diagonal pattern, associated with the neighbours within a stride of n.



(a) **m0** (b) **m3** (c) **m6**

Figure 2.2: Quadrilateral Gauss-Legendre Second-Order



(a) **m0** (b) **m3** (c) **m6**

Figure 2.3: Quadrilateral Gauss-Legendre Fifth-Order

(a) **m132** Gauss-Legendre (b) **m132** Gauss-Legendre-Lobatto

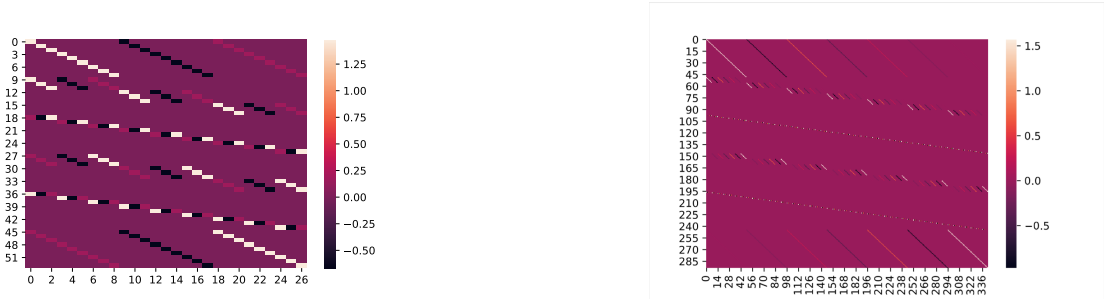
Figure 2.4: Quadrilateral Fourth-Order different quadratures

(a) **m460** Third-Order (b) **m460** Sixth-Order

Figure 2.5: Quadrilateral Third-Order vs Sixth Order

Hexahedra

Hexahedra meshes lead to even more sparse operator matrices as points within stride of n^2 in memory must also be considered, due to the increase in dimensionality over quadrilaterals. This results in the blocks connecting to points within a stride of 1 and n becoming sparser as the dimensions of the operator matrix increase, but those patterns for the blocks remain the same. Figure 2.6 illustrates how the size of the operator matrix increases greatly as the order of the solution increased from second to sixth, leading to a sparser matrix. The sections for strides of 1, n and n^2 are distinctly shown, from left to right respectively, in both matrices in Figure 2.7.



(a) **m0** Second-Order (b) **m0** Sixth-Order

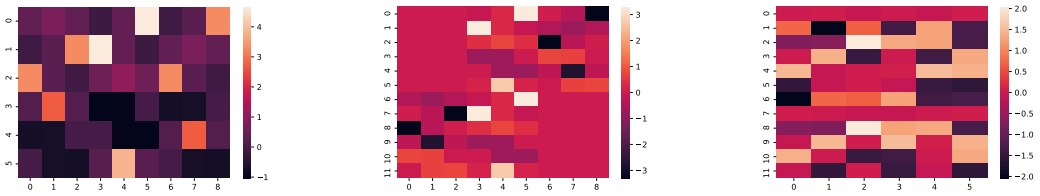
Figure 2.6: Hexahedra Second-Order vs Sixth Order

(a) **m132** Second-Order (b) **m132** Fourth-Order

Figure 2.7: Hexahedra Second-Order vs Fourth Order

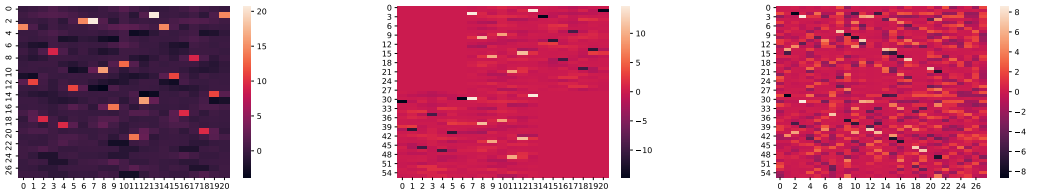
Triangles

Unlike the shapes above, triangles are a simplex element (1-simplex) and triangular meshes lead to dense operator matrices. Figures 2.8 and 2.9 show the operator matrices for a triangular mesh using the Williams-Shunn quadrature with second-order and sixth-order accuracy respectively. Again, as the order increases, the size of the matrices increases. However, the size of the matrices at sixth-order are relatively very small, compared to quadrilaterals and hexahedra. The patterns, whilst not very sparse, still hold for the respective operators as the order increases. Figures 2.8b and 2.9b show that operator **m6** has a density of just above 0.5, whilst the other operators are shown to be more dense.



(a) **m3** (b) **m6** (c) **m460**

Figure 2.8: Triangles Williams-Shunn Second-Order

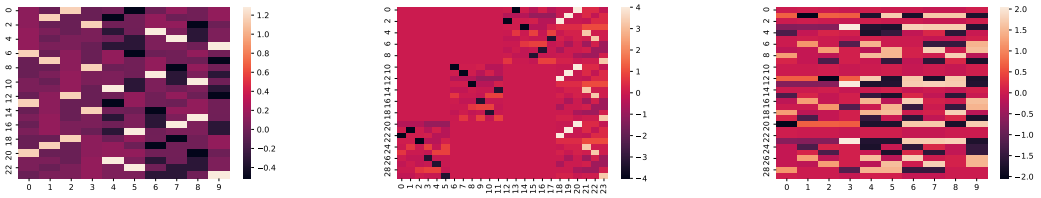


(a) **m3** (b) **m6** (c) **m460**

Figure 2.9: Triangles Williams-Shunn Sixth-Order

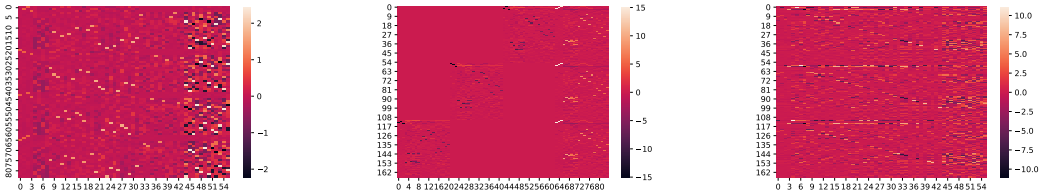
Tetrahedra

Figures 2.10 and 2.11 show some of the operator matrices for Tetrahedra (2-simplex) meshes using the Shunn-Ham quadrature. The operator **m6** for both second-order and fifth-order show distinct sparse regions, one for each different stride length. The other operators are shown to be denser than **m6**. The operator matrix size increases at a greater rate compared to triangular meshes, as the order increases, but they are still small compared to the operators for the other 3D shape, hexahedra.
In summary, the operator matrices are denser but smaller for triangles and tetrahedra, compared to sparser and potentially much larger matrices for quadrilaterals and hexahedra. The matrices for triangles and tetrahedra also have more number of unique non-zeros at higher orders, which is detailed in Appendix A.



(a) **m0** (b) **m6** (c) **m460**

Figure 2.10: Tetrahedra Shunn-Ham Second-Order



(a) **m0** (b) **m6** (c) **m460**

Figure 2.11: Tetrahedra Shunn-Ham Fifth-Order

2.2.3 GiMMiK

In 2014, Wozniak [34] extended PyFR with GiMMiK to generate bespoke kernels to take advantage of the additional information known about the matrices, which provided performance speedups over generic libraries when running on GPUs. The main optimisations made were to fully unroll the loops and to remove multiplications where the value from the operator matrix was 0. Another step was to embed the constants of the operator matrix in the GPU kernel code.
In 2016, GiMMiK v2.0 was released. The bespoke kernels targeting CPUs used OpenMP SIMD pragmas to take advantage of SIMD extensions on CPUs. The use of the OpenMP library gives less control to GiMMiK over the eventual assembly code, but provides a simple kernel code. The compiler is then required to support the pragma and then generate the final kernel assembly code. Roth [27] and Park [22] experimented with improving GiMMiK by exploring tiling schemes for kernels targeting CPU hardware.

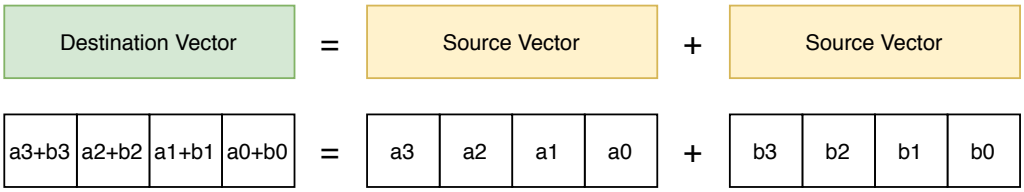


Figure 2.12: SIMD - Single Addition Instruction

2.3 Single Instruction Multiple Data Architectures

Single Instruction Multiple Data (SIMD) architectures operate on multiple sets of data with a single instruction. SIMD differs from vector-processing by operating on all elements of the vector simultaneously, as opposed to operating on the vector elements in a pipelined fashion. This means that multiple results are obtained from a single instruction operating identically on different source elements. Figure 2.12 illustrates how a parallel addition is performed with SIMD. Each element is stored in a 'lane' of the vector and the right most lane is numbered lane zero. The vectors in Figure 2.12 have 4 lanes and the left most lane in each vector is the third lane of that vector. The operation takes values from the same lane in each source vector, and places the result of the addition into the respective lane in the destination vector.

Historically SIMD was used for massively-parallel supercomputers. Modern SIMD is now more commonly found as Instruction Set Architecture (ISA) extensions. The Advanced Vector Extension (AVX) class of extensions have been heavily adopted among x86 class CPUs and Arm also has the Neon and Scalable Vector Extension (SVE) extensions. SVE is unique in that it does not specify the vector width, only a range of valid widths that the CPU vendor can implement.

2.3.1 AVX-512: A Single Instruction Multiple Data ISA

Intel AVX-512 is a SIMD ISA extension for x86 CPUs, and is an evolution of its predecessor AVX2, which it is backwards compatible with. AVX-512 registers are 512-bit wide, double that of AVX2. The registers can therefore store up to 8 double precision (DP) or 16 single precision (SP) floating point numbers. In the case of DP, each register has 8 vector lanes. AVX-512 also support signed and unsigned integers of differing lengths. The ISA provides the ability to perform arithmetic operations, bit manipulation, compression, shuffling and more between vectors.

The Intel microarchitecture Skylake Server Configuration has a sub-memory system that supports 2x64B loads and 1x64B stores in each cycle. This allows an entire 512-bit (64B) register to be loaded or stored during each cycle, using only one memory port. AVX-512 is included in Skylake-Scalable Performance (SP) server CPUs; the improved sub-memory system provides the memory bandwidth required by the doubling of the vector width.

The main instructions used during a matrix multiply are the Fused-Multiply-Add (FMA) and memory loads/stores. The FMA instruction multiplies two source values and adds them to the destination register. AVX-512 has an instruction that can FMA 8 DP values. In Skylake-SP, hardware support provides this instruction with the same cycle latency as an 8-wide DP vector addition.

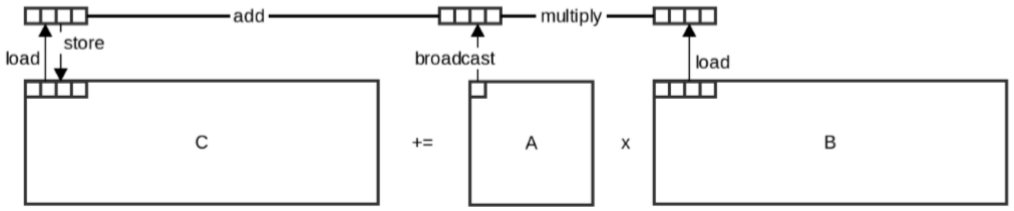


Figure 2.13: Roth illustrated the vectorisation of the matrix multiply routine[27]

2.3.2 Sparse MM: Vectorisation

As shown by Roth [27] in Figure 2.13, the vectorised matrix multiply is performed by the following procedure:

- For each row:
  - Load a stride of C equal to the width of the vector register. (only if adding αAB to βC where $\beta \neq 0$)
  - For each element in A on this row, broadcast the element into a vector register and multiply with the corresponding stride from B.
  - Store the accumulated value for the C stride.
- Repeat this for all 'columns' of strides in B.

With AVX-512 and DP, the strides of B and C are 8 values wide. A nice feature arises from this, which is that the strides fit exactly into a single 64 byte cache-line. Since Skylake-SP supports two 64 byte loads and one 64 byte store per cycle, each B-stride can be loaded in each cycle, even when storing the C-stride. Additionally, a column of B-strides takes up $\frac{8*K*64}{8}$ bytes for DP with dimension K. So for A with $K \leq 100$, a column of B-strides takes a max of 6.25 KiB, which would fit within the L1 data cache of a Skylake-SP core. This means that repeated access to B-strides for different rows of A should be read the stride from L1-cache after the first access for a previous C-stride calculation.
A couple of trivial optimisations arise from the above routine when A is sparse:

- If the element in A is 0, skip the load and resulting FMA of the corresponding B stride and continue to the next element of A.
- After eliminating the the above, fully unroll the loop to potentially improve performance.

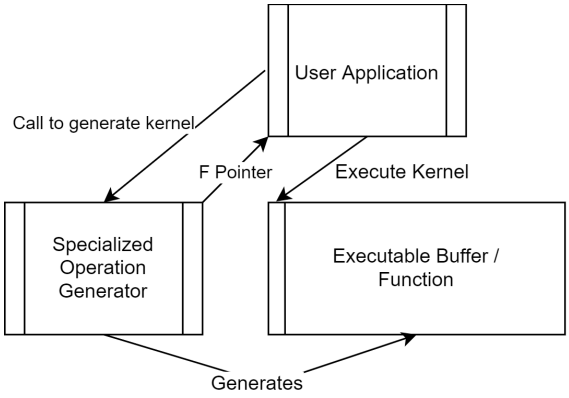


Figure 2.14: High-level view of Just-In-Time code generation in LIBXSMM

2.4 LIBXSMM

2.4.1 The Library

LIBXSMM is a library for specialised dense and sparse matrix multiplication as well as other related deep learning operations [3]. The library focuses on small matrix multiplication, which is approximated by problems with dimensions $MNK : (MNK)^{1/3} \leq 64$ [3]. The library is designed to target various Intel architectures such as Intel SSE, AVX, AV2 and AVX-512.
The library acts as a Just-In-Time (JIT) compiler, by generating code after analysing values discovered during runtime. The resulting byte-code is then placed into an executable buffer in memory. LIBXSMM is designed to be both compiler agnostic and threading-runtime agnostic. This means the same performance should be expected regardless of the compiler or threading library utilised by the user application.
Each specialised operation that LIBXSMM supports has its own dedicated function, that uses LIBXSMM's instruction API to 'write' assembly instructions into the executable buffer. LIBXSMM then casts this buffer to a suitable function pointer, allowing the user application to call the generated code. This process is visualised in Figure 2.14.

2.4.2 Use within PyFR

PyFR can optionally incorporate LIBXSMM as a matrix multiplication implementation whilst using the OpenMP library as its threading runtime. LIBXSMM is not required to provide a threading runtime and is threading runtime agnostic, so other threading libraries could be used. As well as running on CPU only clusters, it can use the library when heterogeneously computing [8]. In contrast to GiMMiK 2.0 where a third-party compiler writes the resulting kernel, LIBXSMM directly writes every line of assembly for the bespoke kernel, giving a finer grain of control, with the additional complexity that comes with this control.

2.4.3 Sparse MM: Storing A in the register file

LIBXSMM has a specific generator for when a sparse A has less than or equal to 31 unique non-zeros. As part of the JIT process, the number of unique non-zeros is counted. If above 31, then a slower alternative fallback strategy is used. As described in section 2.3.2, the elements of A have to be broadcast to fill a vector. If one vector register is used to accumulate the C-stride, then there are 31 free vector registers to store constants from

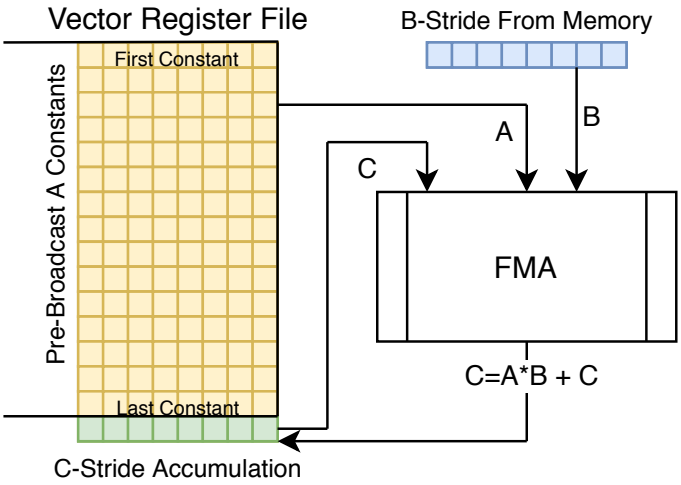


Figure 2.15: Pre-broadcasting constants from the operator matrix for use with the FMA operation - DP with AVX-512

A. The FMA operations can use memory locations as source operands as well as register sources. This means B-strides do not have to be stored in registers.
This generator pre-broadcasts (fills the vector with the same value) each unique element of A into separate vector registers. During the FMA, the broadcasted element of A is obtained from the corresponding register, as shown in Figure 2.15. This removes every load instruction for the operator matrix, leading to performance gains. The routine follows the strategy outlined in section 2.3.2.

Tiling Scheme

The sparse-dense MM routine in LIBXSMM applies a fixed tiling scheme. The user defines a 'chunk size' - number of columns for a 'chunk' of B. The chunk size must be a multiple of of the target SIMD vector width; in the case of AVX-512 using DP the width is 8, so the chunk size must be a multiple of 8. 48 was found to provide good performance for PyFR operator matrices.
Within the chunk of B, there are mini-chunks which have the same number of columns as the vector width. In the case of AVX-512 using a chunk size of 48, there are 6 mini-chunks each with a width of 8. The user successively calls the kernel on chunks of B. Internally, the kernels successively operate on each mini-chunk B. For AVX-512 this means that operator matrices with under 512 columns would lead to the mini-chunks of B remaining in L1-cache with minimal spilling. These mini-chunks are accessed up to an amount equal to the number of rows in the operator matrix, so it is important to keep them in L1 cache.

Summary

We have covered the motivation behind FR and PyFR - an implementation of it. The operator matrices found in PyFR have certain characteristics based on a number of factors which we discussed. We explained the basics of SIMD architectures and specifically, AVX-512 on Intel Skylake-SP. PyFR simulations contain sparse MM and we showed how SIMD can be used to accelerate this computation. Finally, we covered LIBXSMM and how it is used as a JIT compiler. We focused on the existing sparse-dense MM routine within LIBXSMM and how it makes use of the vector registers to store an operator matrix that has up to 31 unique non-zero elements.

Chapter 3

Related Work

In this chapter we will explore previous work which had the aim of speeding up PyFR's execution. We will then discuss a broad range of work that made use of SIMD architectures with a focus on the vector register use. Next, we analyse Common Subexpression Elimination (CSE) techniques from multiple research areas that have a common goal of FLOP (floating point operation) reduction.

3.1 GiMMiK

We briefly covered GiMMiK In Chapter 2 covering the main optimisations made by Wozniak in 2014 [34]. The main optimisations were simple; fully unroll loops and apply sparsity elimination. However, they led to significant speedups over other GEMM approaches on GPUs. The evaluation carried out was on a set of example operator matrices from PyFR, which is the same set we will use in our evaluation. In the evaluation, we will place a greater focus on the characteristics of the operator matrices and how they could be affecting performance.
GiMMiK v2.0, released in 2016, added support for generating kernels that target CPUs. In 2015, Roth explored adding new tiling schemes to GiMMiK [27], targeting CPU SIMD architectures. The tiling greatly benefited performance for the large and the dense operator matrices found within the set of example PyFR operator matrices. In 2016, Park presented a 'tailored tiling' scheme for GiMMiK when targeting CPUs [13]. The main contribution was to implement and evaluate a tiling strategy that had dynamic tiles, based on the number of sequential non-zero elements in the operator matrix. Suppose a threshold T was set. Then the tile on A would be made by traversing the first column of A until T non-zero elements have been found. If required, the tile can continue over to the next columns(s), until T is reached. Tiles are continuously made until all of A has been tiled. Park reported speedups of up to 1.95 times for large and dense matrices compared to fix-sized tiling schemes. However, Park also reported speedups of 0.9 times (slowdowns) for some small and sparse matrices. The tailored tiling scheme does not take into consideration the uniqueness of the non-zero elements within the tiles it forms.
The sparse-dense MM routine in LIBXSMM applies a more fixed tiling scheme, which is explained in Section 2.4.3. The difference in the approach taken by LIBXSMM is that part of the tiling scheme is exposed to a higher layer of abstraction - the user must be aware of it. This design choice allows the user to use any threading runtime library to operate on tiles in parallel, making LIBXSMM threading runtime agnostic.

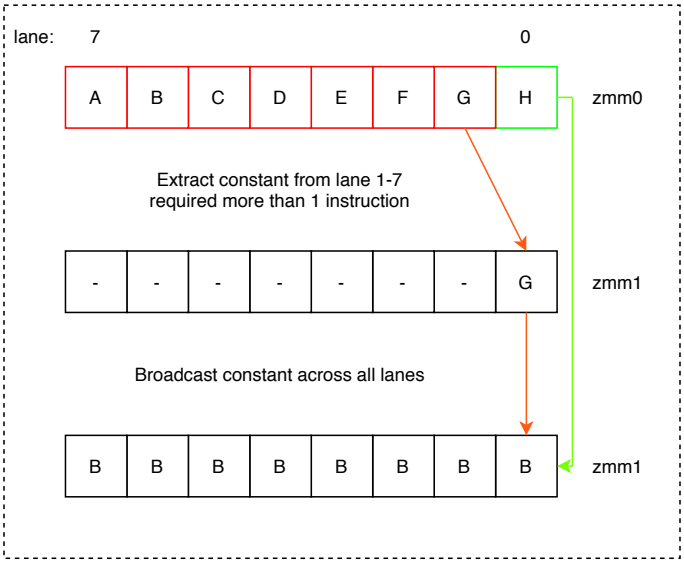


Figure 3.1: VecReg runtime constant unpacking/broadcasting from lane zero is 'cheaper' in terms of instructions required than unpacking from the other lanes in the register [23]

3.2 VecReg

Overview

In 2019, Price presented VecReg [23], a Just-In-Time (JIT) code generator targeting AVX-512 hardware to perform sparse matrix multiplication. Price used the Python Mako templating library to generate C code that makes use of AVX-512 intrinsics. The C code would then be compiled using Intel's C/C++ compiler, ICC.

The main difference between VecReg and LIBXSMM's sparse register routine is how the values from the operator matrix are stored in the register file. As mentioned in Chapter 2, LIBXSMM pre-broadcasts each of the unique non-zeros into a vector register. VecReg took a different approach, by compactly packing up to 8 (double precision) unique non-zeros into a 512-bit array, as displayed in Figure 3.1. VecReg followed the same higher-level routine of matrix multiplication that LIBXSMM uses, and so still required the values of A to be repeated across an entire, single vector register. During runtime, VecReg kernels broadcast the values of A into a vector register, by extracting them from the compact format. VecReg relies on ICC to perform this broadcasting.

Evaluation

The compact storage of unique non-zeros led to spare registers that could be used to apply CSE to reduce the FLOP count of the kernels. Price found up to 1.6x speedups over LIBXSMM, which was attributed to being able to perform CSE, and in general found speedups over LIBXSMM for operator matrices with more than 31 unique non-zeros (when LIBXSMM defaults to a less specialised sparse kernel).

Price notes that LIBXSMM was loading C when not required, which has been corrected in a newer version of the library. Price continues his evaluation when A is dense and compares against LIBXSMM, to see how ICC, and therefore how VecReg, handles operator matrices with more than the storage limit of 240 unique unique non-zero. Price approximated 256, but one register is always required for storing the broadcasted version of a value of A and another to store a cumulative sum for matrix C.

Although Price does not explicitly state the following, we believe the assumption that the runtime broadcasting would not be on the critical-path was essential for performance to

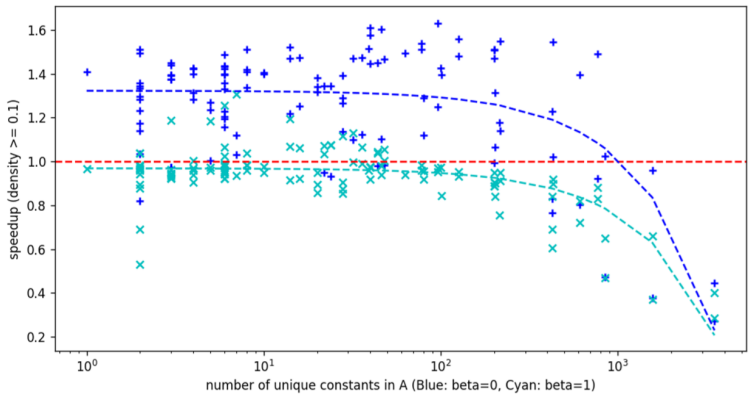


Figure 3.2: VecReg Speedup vs unique non-zeros in A (density of A $\geq$ 0.1) [23]

not degrade for operator matrices that do fit within LIBXSMM sparse A limit of 31 unique non-zeros.

Investigating VecReg in practice

If the value is stored in lane zero, then ICC can use the 'VBROADCAST' instruction to achieve the desired broadcast with one instruction, which is represented by the green step in Figure 3.1. However, for other lanes, Price's assumption was that ICC would generate code that is longer than a single assembly instruction to broadcast the constant, represented by the red step in Figure 3.1.
A couple of sample kernels generated by VecReg are provided by Price. We investigated the assembly output of ICC, whilst using the same compiler options as Price (importantly, '-O3 -march=skylake-avx512 -qopt-zmm-usage=high' were used). ICC did not store any of the 512-bit arrays directly into the AVX-512 registers. Instead, packed values of A were loaded from memory into lane zero of a register. This was then broadcast using 'VBROADCAST'. By not utilising the register file to store A, the ICC compiler misses out on potential performance improvements. This investigation reveals VecReg does not implement the theoretical method proposed by Price.
Figure 3.2 shows the speedups of VecReg over LIBXSMM versus the number of unique non-zeros in A. The points for $\beta = 0$ should be ignored due to the previously mentioned issue in LIBXSMM with loading C being fixed. For $\beta = 1$ and the number of unique non-zeros ≤ 31, most of the data points show that VecReg was noticeably slower. This is explained by VecReg/ICC kernels loading A from memory (most likely L1 data cache). Therefore results of VecReg cannot be used to say that runtime broadcasting does not add to the critical-path, as we believe the performance penalties arise from loading A from memory.

Key Takeaways

We believe any runtime broadcasting operations should be tuned to use minimal instructions to avoid reaching the critical-path. To achieve this, alternative storage layouts and code generation would have to be explored. It should be noted that although there are a couple of issues with Price's work, the idea of packing values seems to be an excellent approach to being able to support more operator matrices and to allow for CSE.

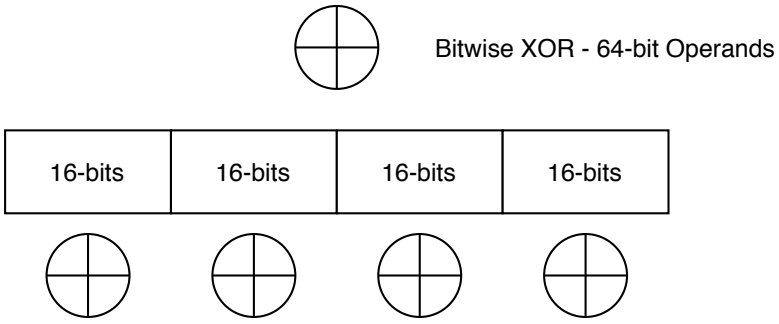


Figure 3.3: Bitwise XOR Operation on Multiple Data

3.3 Compiling for SIMD Within A Register

SIMD Within A Register (SWAR) integrates SIMD within conventional microprocessor instructions and memory layouts. The primary goal was to improve the speed of certain multimedia operations, by computing on multiple 8/16-bit data values within 32/64-bit scalar registers. In 1998, Fisher et al. [24] presented a language and compiler to allow users to 'write portable SIMD programs that could be compiled into efficient SWAR modules'. Some operations, like bitwise XOR, are called *polymorphic*, as the same instruction can be used regardless of the data-size vs the register-size. Figure 3.3 shows how the single machine operation can trivially morph into multiple operations. These operations are easy to encode into SWAR.

Another class of operations are the *communication* operations, where the operations "logically transmits data across processing elements in an arbitrary pattern" [24]. In the SWAR model, if a register is logically partitioned, then shift and rotate instructions can be used to transmit data between partitions. Fisher et al. note that the available ISA extensions at the time, such as MultiMedia eXtension (MMX), did not provide for efficient complex communication patterns, and so SWAR programs should avoid the use of them. So even when SIMD extensions were available on the target machine, the SWAR program would be slowed down by inter-partition communication operations. However, with new extensions such as AVX2 and AVX-512, the logical idea of inter-partition communication can be efficiently performed as direct hardware support is provided for it.

3.4 Intel SPMD Program Compiler

Single-Program Multiple-Data (SPMD) is a programming model that runs the same serial program in parallel, on multiple sets of data. A key difference to a single program utilising SIMD, is that the control-flow for each set of data can diverge in SPMD. In 2012, Pharr et al. [16] developed a compiler, Intel SPMD Program Compiler (ISPC), that generates code for SPMD on CPUs with SIMD hardware.

In the single core case, when calling an ISPC function, a group of *program instances* start running concurrently, and are referred to as a '*gang*'. Each program instance has its own vector lane in the SIMD registers. The number of program instances, or the *gang size*, is in practice no larger than twice the number of lanes in the SIMD register [26]. It is relatively easy to extend the SPMD program to work across multiple cores for further parallelisation. Pharr et al. outline how divergent control-flow can be managed in Section 3.3 in [16]. The main idea is to use runtime write-masks that ignore new results if control has diverged. This is good for basic *for* loops and simple *if* statements, as the control-flow is converted into partially predicated instructions.

Deeply-nested *if* statements would lead to less energy-efficient code, as all possible branches

would be executed. However, the user of ISPC should be aware of this, and one of the 'non-goals' of ISPC was to not protect the programmer, but the goal was to allow them to achieve higher performance.
ISPC targeted hardware that provided cross-lane SIMD operations. ISPC exposes the capability via built-in functions that allow program instances to exchange data between the gang. Two useful operations are; broadcast a value to all other gang members and arbitrarily share a value to other members. So, ISPC has schemes to share data from any program instance, to any other instance(s) in the gang, by utilising the rich-set of instructions on modern SIMD that provide cross-lane operations. We can also make use of these cross-lane operations to unpack a packed version of the sparse A matrix.

3.5 Pixel Interlacing to Trade off the Resolution of a Cellular Processor Array against More Registers

Cellular Processor Array (CPA) vision-chips are compact and low power-budget chips that embed a Processing Element (PE) at each pixel. The PE consists of a processing unit (PU) and registers. The number of registers and general complexity of the circuity of each PE is constrained by many physical and cost constraints. These chips capture, store and process the light intensity at each pixel. By having a massively parallel computation on the chip via a SIMD approach, meaningful pre-processed data can be output instead of transferring entire frames of data, which is much slower.
In 2015, Martel et al. [17] proposed a method of creating a virtual 'super pixel'. This logically groups together neighbouring pixels and pools together the registers within the PEs. The greater the number of pixels in a super-pixel, the greater the number of registers per super-pixel, but also the lower the resolution of the chip. This is a trade-off made to allow for more complex on-chip computation for algorithms that require the use of more registers and/or more temporary registers.
The super-pixels are arranged on a 2D grid. For some algorithms, data is required to be shifted around the grid. A shift can be performed either vertically or horizontally. However, when grouping pixels together into a super-pixel, greater care has to be taken to shift only the correct register to the immediate neighbouring super-pixel, and not every register value. Martel et al. achieve this using two components, an activity flag and masking registers. Detail on how the correct shifting is achieved can be found in Section 2.B from [17]. Essentially, the registers in PEs that shouldn't be operated on during the shift are masked to not be overwritten, which is decided by the activity flag. The goal of the routine can be considered to transfer data between lanes of a SIMD vector, where the vector lanes/elements are super-pixels.
This method of shifting with write-masks could be used to unpack a packed form of the sparse matrix A for our case of sparse matrix multiplication. However, without fast hardware support for this style of operation being in a loop, alternative cross-lane data transfer could be more efficient.

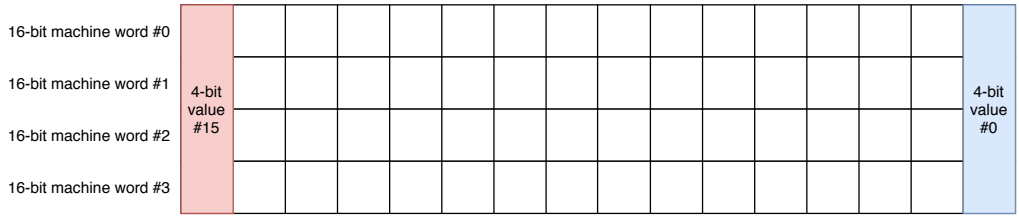


Figure 3.4: Bitslice array of sixteen 4-bit values

3.6 Customizable Precision of Floating-Point Arithmetic with Bitslice Vector Types

The work carried out in 1998 by Fisher et al. [24] on SWAR was strictly for integer data types. In 2016, Xu et al. [29] proposed methods to compute on arrays of *custom-precision* floating point data types. An area where custom-precision can be useful is *approximate* computing, where the input and output are approximations, and a small additional imprecision is unlikely to impact the final results. Approximate computing can be implemented by using fewer binary bits. When designing custom hardware solutions either via FPGAs or ASICs, these custom-precision data types can be configured precisely. However, on general-purpose processors, this is not the case. The drawback of using larger than required data types, such as 32-bit floating-point (FP) when the application may only require 13-bit FP, is that more memory space and memory bandwidth is required, especially when dealing with large arrays of the data. There is also a potential benefit to power consumption when using lower-precision [15]

A software bitslice representation was used, taken from the implementation of some cryptography algorithms [2]. In this representation, each machine word contains a single-bit from each array element. For example, if the machine word size is 16-bits, then for a data precision of 4-bits, four 16-bit machine words are used. This gives a 16 element array of 4-bit values and is shown in Figure 3.4. The machine size in bits determines the width of the vector. To carry out arithmetic operations, integer bitwise operations can be carried out on each machine word, and so the individual bits of the array elements. This allows for software equivalent algorithms to replace dedicated hardware units, such as adders. Xu et al. provide software intrinsics that carry out these arithmetic operations on the users array of custom length data types.

Performance was evaluated against various integer types, (as the CPU did not have native support for custom-precision FP). Only FP8 saw speedups; 2x over 32-bit integer and 4x over 64-bit integer for division, and around a 1.1x speedup for multiplication compared to 64-bit integers. Custom FP16 and FP32 resulted in significant slowdowns. Xu et al. concluded that for smaller data types, typically used in approximate computing, the bitwise parallelism outweighed the cost of the bitwise arithmetic.

The scheme presented showcased a form of packing values across machine words, to enable a novel SIMD computation within general-purpose microprocessors. Whilst this work directly operated on the packed form, the SIMD sparse matrix multiplication routine cannot trivially do so. However, packing data within vector registers, as opposed to reading from cache, should reduce memory bandwidth requirements as well.

3.7 Topological Optimisation of the Evaluation of Finite Element Matrices

Kirby et al. [25] presented a framework in 2006 that aimed to reduce FLOP count over the evaluation of finite element matrices. The matrix being evaluated is known as the 'stiffness' matrix, which represents the system of linear equations to be solved in order to obtain an approximation for the original differential equation.
Kirby et al. make the abstraction to only consider the vector products when multiplying a R^{nm} matrix with a vector in R^m in the the optimisation problem. The objective was to minimise the number of multiply-add pairs, or FMA operations. The total number of FMAs is bounded by nm when calculating

$$\left[(y^i)^t g\right]_{i=1}^{n} \quad y, g \in R^m \tag{3.1}$$

A discrete distance was calculated between $y, z \in R^m$, and if they are close, then the product of $y^t g$ should be easy to compute once $z^t g$ is known.
Kirby et al. formed a graph where a node represents a vector product $y^t g$. This node $i \in m$ was connected to every other node $j \in m, i \neq j$. The edges were weighed by how close y^i was to y^j using a chosen distance metric. A lower distance represented a lower amount of FMAs that were required to go from the vector product of the node i to the vector product of node j. Kirby et al. further expand on the distance metric, which they define as a complexity-reducing relation [25]. An example metric that satisfies the formal definition is the Hamming distance.
By calculating a directed minimum spanning tree (MST), and calculating the products in the traversal order, the minimum number of FMAs should have been used to calculate all the vector products.
As noted by Kirby et al. there were a couple of downfalls with the approach. The true optimal number of FMAs was not guaranteed to be found, as the complexity-reducing relation only considered two vectors at a time. By considering more than two, it is possible to find routines that would use even fever FMAs.
After removing multiplications by zero, the additional optimisations only provided a 'modest' speedup [25]. This can be attributed to the routine now being more memory bound. A modification to the modelling of the cost to include memory access could have potentially improved the speedup provided by the additional optimisations.
By forming a fully connected graph, a trivial algorithm to find the MST would have quadratic complexity. Kirby et al. propose various methods to reduce the search space, including only considering edges that have a minimum FMA reduction payoff.
We can utilise the notion of using a weighted-graph to decide which CSE optimisations decisions to make if some clash, on a per operator matrix basis, as part of the JIT process. An important takeaway from this related piece of work, is that we should consider the memory access patterns and costs when weighting the edges.

3.8 Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation

In 2013, Russell and Kelly [10] presented EXCAFE, a code generator that used symbolic integration to perform finite element local assembly, as opposed to the popular approaches of quadrature and tensor contractions. This allowed the authors to perform a CSE pass, which was crucial to achieving at least similar performance levels as quadrature and tensor contraction implementations.
Russell and Kelly extended the factorizer from the work of Hosangadi et al. [1]. They had

to adapt it to work with the representations they used, but also improved the work to be able to detect more types of factorisations.
Russell and Kelly achieved a reduction in operation count by over a factor of 4 in some cases, across their selected benchmark. Our work will require a factorizer if implementing CSE and so the work of Russell and Kelly would be a good starting point. However, as noted by the authors, the code-generator does not scale well with large problems. While the symbolic integration part of EXCAFE suffers with large sizes, the factorisation is both computationally and memory intensive. Since the operator matrix in PyFR is small, this should not be too much of an issue.

3.9 Compiler-Level Matrix Multiplication Optimization for Deep Learning

In 2019, Zhang et al. [11] proposed two algorithms, a greedy best first search and a novel Neighborhood Actor Advantage Critic (N-A2C) algorithm, designed to optimise the GEMM routine by choosing the optimal tiling configurations for the target hardware.
The nodes in the graphs were the tiling configurations. Tiling configurations that are close, are likely to have similar performance. Both algorithms dynamically searched new nodes by exploring configurations that were likely to not have similar performance as the current configuration. For the edges in the graphs used in both algorithms, the weights are calculated by measuring runtime differences between the nodes.
Since LIBXSMM is a JIT compiler, we can make the assumption that AVX-512 would be present on the CPU to collect measurements, if we are targeting AVX-512 hardware during the JIT process. This has the potential to simplify the weight calculation on any graph model, as ideally our model would consider memory access, and not just FMA uses. The major disadvantage of this approach, as acknowledged by Zhang et al, is that the measurement is subject to large noise. This greatly impacts the decisions taken by the greedy BFS algorithm [11].
It is important to highlight that this paper looked into GEMM, and not sparse-dense MM. So the random, dynamic searching of configurations could be justified, as there was no prior information to take advantage of, other than the dimensions.

Summary

We covered two previous pieces of work which focused on improving PyFR's execution speed. The first, GiMMiK, was made to work on CPUs as well as GPUs and we compared some of its key differences to LIBXSMM. The second, VecReg, was covered in detail and we highlighted a key flaw in its implementation - it does not utilise the vector registers in the way it was previously thought to have done so. The related work that made use of SIMD architectures had a common feature which was moving data between lanes of vector registers. Through our analyses, we find that hardware support for cross-lane communication operations allows for some algorithms to become feasible in practice, by providing a fast enough method to move data. The pieces of work we discussed that make use of CSE showed that it has been a well studied matter and that there are various solutions to find optimal combinations of CSE.

Chapter 4

Evaluation Methodology

In this chapter we first outline both the hardware and software environment that the evaluation will be performed on. We then detail the benchmark used to measure performance on example operator matrices encountered in PyFR applications and operator matrices from a new synthetic suite. Next, we explain the performance metric that will be used for the evaluation and also describe the process of generating roofline plots for the evaluation. Finally, we outline any threats to the validity of the results and discuss the measures taken to attempt to neutralise them.
We describe and explain the benchmarking process in detail to help to make the results repeatable and so that if desired, others can validate the measurements we report. A script is provided in the accompanying files to make it easier to run the benchmark and obtain results.

4.1 Environment

Software Version

All testing will be run on Ubuntu 18.04 LTS. The benchmark will be compiled using GCC version 7.5.0, but the exact compiler used should not effect performance as the LIBXSMM kernels are compiler agnostic. The updates to LIBXSMM were made to the parent commit '51e64904fc53c19de79ec8e66414233f4fc4130e', which itself builds upon version 1.14. The maximum code buffer size was doubled in LIBXSMM for all versions tested (including reference) in Chapter 6. In Chapters 7, 8, 9 and 10, the code buffer size was increased by a factor of 8. Appendix B details the reasons why this change was required.

Benchmark Controls

To minimise the gap from the theoretical peak kernel performance and the best execution time obtained, we can control a few things. Firstly, we can pin the benchmark to a single core using "taskset -c 0" in Linux. This prevents any repeat compulsory low level cache misses from the process being moved to a different core. The benchmark should be run with real-time priority, which can be easily set within the Linux operating system using "nice -20". This important measure prevents the OS from scheduling other processes to run on the core the benchmark is running on. Compared to the execution times of the kernels, the scheduling overhead may not be that significant, but the CPU time taken by other process can be very significant. Since we will be measuring the time from when the kernel is called to when it returns, any time taken by other processes will increase the measured time.

Hardware

Originally we intended to run the benchmark on a private compute cluster. However, there would often be other work being executed on the same node that the benchmark was assigned to. The system in place did not allow for the benchmark to be given exclusive access to a core. This led to a large variance in performance between runs and so an alternative platform had to be used.
Instead, the benchmarks will be run on Amazon Web Services (AWS) Elastic Compute (EC2) nodes, that have Intel CPUs with AVX-512. The exact instance type is 'm5.xlarge' which uses a single core from the Intel Xeon Platinum 8175M, a Skylake-SP based CPU, along with 8GB of RAM. AWS provides exclusive access to the physical core, even though the entire CPU is shared. However, as the benchmark aims to keep reused data within at least L2 cache, sharing L3 cache should not be detrimental. Disabling Hyper-Threading should not be necessary due to setting CPU affinity and process priority.

4.2 Benchmark Suite

We will be testing the kernel performance for different operator matrices ($\mathbf{A}$) in the matrix multiplication from equation 4.1 with **DP**. $\alpha = 1$ and $\beta = 0$ will be the only versions tested, $\beta = 1$ would have the same effect on performance for the reference and updated versions of LIBXSMM. This is because all versions tested would load the stride of $\mathbf{C}$ in the same way, before continuing with their strategies to carry out the matrix multiplication.

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C} \tag{4.1}$$

Emulating PyFR's use of LIBXSMM

The strategy found to be optimal with LIBXSMM for PyFR splits the $\mathbf{B}$ matrix every 48 columns. Splitting $\mathbf{B}$ takes advantage of the massive multi-threading available in a typical PyFR simulation setup, and 48 columns was found to be optimal by the LIBXSMM and PyFR maintainers. The setup effectively calls a 'hot' kernel with new data. The kernel is very likely to keep data in L1, and only spill to L2 cache if the $\mathbf{B}$ matrix has a very, very large amount of rows ($\mathbf{A}$ has a very large amount of columns). The C program PyFR uses to call the kernels, is linked to the LIBXSMM library. If dynamic linking was to be used, once the kernel is 'hot', runtime linker look-ups should be resolved and cached, so function call expenses should be minimal over the entire simulation.
To provide performance metrics that translate well into PyFR's real-world use case, we propose a method that aims measure the performance of the kernel when passed 48 columns of $\mathbf{B}$. For each operator matrix kernel, we successively pass 48 column 'chunks' of $\mathbf{B}$, where $\mathbf{B}$ is random with a total of $192,000$ columns of data. The time measured is just *before* calling the kernel with the first chunk of $\mathbf{B}$, to when the kernel *returns* from calling it with the last chunk of $\mathbf{B}$. The C '*gettimeofday*' function is used to measure time.
Figure 4.1 shows the results of an experiment to determine if using a total of X columns in $\mathbf{B}$ can emulate a PyFR simulation. The kernel is repeatedly called called for successive 48 column chunks of $\mathbf{B}$. After operating on all of $\mathbf{B}$, the process is repeated 60 times, as we intend to do in the benchmark. The test operator matrix was taken from the PyFR examples (quad-p4-gauss-legendre-m132). If X is too low, then chunks of $\mathbf{B}$ remain in the last-level cache (LLC) between calls to the kernel for the same chunk of $\mathbf{B}$. So the best performance measured would not be realistic for unseen $\mathbf{B}$. $X = 192,000$ was chosen as it was large enough to not fit in LLC, leading to the performance within the flat range in Figure 4.1 for the average execution time per 48 columns of $\mathbf{B}$.

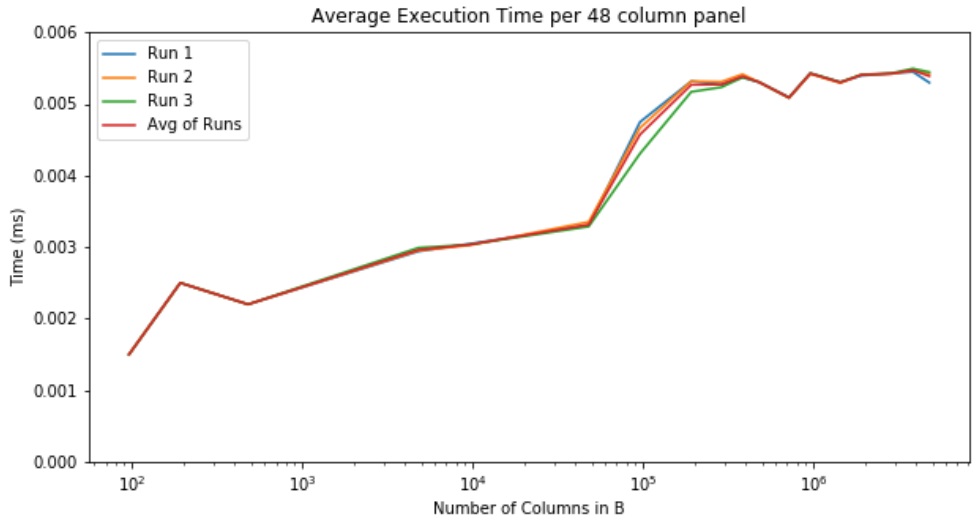


Figure 4.1: Investigate Number of B columns - Average Best Time per 48 columns

Repeating Experiments

During a single benchmark run, each kernel is called 60 times with the 192,000 column **B**. The *best* time is recorded. For the evaluation, the entire benchmark is run on 3 separate AWS VM instances and the *average of the best times* across the 3 runs is used. The benchmark is statically linked to LIBXSMM, eliminating runtime linking overheads.
The benchmark consists of two separate sets of operator matrices; the first a collection of example operator matrices encountered in PyFR applications and the second a set of synthetic matrices.

4.2.1 PyFR Example Operator Matrices

A subset of examples were taken from a set provided by PyFR, listed in Table 4.1. From the whole set (detailed in Appendix A), any matrix with less than or equal to **176** unique non-zeros were used. Matrices with over 176 unique non-zeros would have used the same kernel generation strategy in our updated LIBXSMM as the in the reference version. In Table 4.1

Order	1	2	3	4	5	6
Quadrilateral - GL	*all*	*all*	*all*	*all*	*all*	*all*
Quadrilateral - GLL	-	*all*	*all*	*all*	*all*	*all*
Hexahedra - GL	*all*	*all*	*all*	*all*	*all*	*all*
Hexahedra - GLL	-	*all*	*all*	*all*	*all*	*all*
Triangle - WS	*all*	*all*	*all*	$m0, m3, m6$	$m0, m3, m6$	$m0$
Tetrahedra - SH	*all*	*all*	$m0, m3, m6$	$m0$	-	-

Table 4.1: PyFR Example Operator Matrices Used in Benchmark

'GL' is Gauss-Legendre quadrature, 'GLL' is Gauss-Legendre-Lobatto, 'WS' is Williams-Shunn and 'SH' is Shunn-Ham. 'all' is equivalent to having $m0, m3, m6, m132, m460$ operator matrices.

4.2.2 Synthetic Operator Matrices

As the operator matrices vary in more than one characteristic between them in the PyFR examples, a suite was synthesised to only vary one characteristic at a time. The base configuration was 128 rows and columns, a density of 0.05 and either 16 or 64 number of unique non-zeros (U). The data was uniformly, randomly sampled from the available unique constants. The data was also uniformly, randomly placed in the matrix. Two base configurations were made to be able to compare against two routines LIBXSMM used, one for $U \leq 31$ (LIBXSMM sparse-dense register routine) and one for $U > 31$ (LIBXSMM dense routine).

- Vary number of rows in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 16$
- Vary number of rows in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 64$
- Vary number of columns in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 16$
- Vary number of columns in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 64$
- Vary density $\in [0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]$. $U = 16$
- Vary density $\in [0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]$. $U = 64$
- Vary number of unique non-zeros in A
  $\in [8, 16, 24, 31, 40, 48, 56, 64, 78, 96, 112, 128, 144, 160, 176, 192]$

4.3 Performance Metric

An alternative performance metric to execution time will be used for the evaluation. The aim is to provide insight into how well the hardware capabilities are utilised *effectively* and to also better showcase performance speedups.
We call the metric used ***pseudo-FLOP/s***. For a given operator matrix, the pseudo-FLOPS (Floating Points Operations) is first counted. The algorithm to count the pseudo-FLOPS is a basic 3 loop matrix multiply, with the exception that when an element is 0 in **A**, the FLOPS are not counted. The algorithm assumes an FMA is available, and each FMA counts as two FLOPS. The count is then divided by the measured execution time of the kernel from the benchmark, resulting in a performance metric of 'pseudo-FLOP/s'.

The 'pseudo' prefix is used as the same FLOP count will be used across the different kernels/strategies implemented, even though the actual FLOPS required by an optimised kernel might be less. This leads to a different 'pseudo-FLOP/s' as the optimised kernel should have lower execution times, thus displaying the performance improvement in a hopefully more contextual way. Similarly, a less optimised kernel could have a lower pseudo-FLOP/s than the actual FLOP/s. Figure 4.2 transforms the time measurements from Figure 4.1, using the new 'pseudo-FLOP/s' measurement. The data being displayed is the same, but shown in a different context.

4.4 Roofline Plots

The evaluation includes plotting roofline plots. The compute bound lines are calculated using the AVX-512 boost frequency of the Intel Xeon 8175M, 2.4GHz. Two compute bound lines are shown; the first is for when the CPU core has a single AVX-512 FMA unit, and the second for when the core has two AVX-512 FMA units. This is due to lower-end Skylake-SP CPUs only having one FMA unit per core. The Intel Xeon 8175M used in testing has *two* FMA units per core.

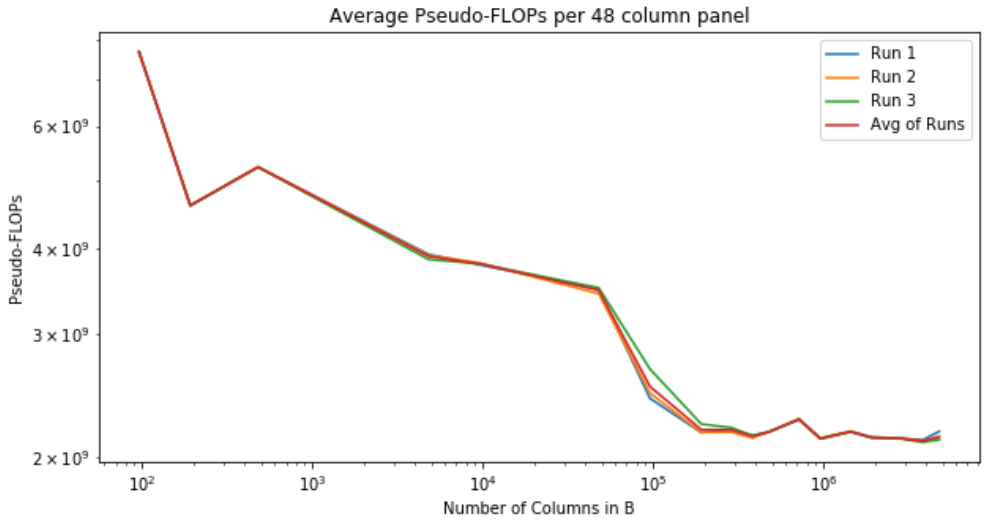


Figure 4.2: Investigate Number of B columns - Average pseudo-FLOP/s per 48 columns

The memory bandwidth used for the memory bound line was obtained using a modified STREAM benchmark. The modification was to instead measure the operation of $a[i] = a[i] + b[i]$. This gave a $1:1$ ratio of load-store, as when writing to a, the cache-line must also be loaded from memory. Intel's C/C++ compiler ICC was used to compile the benchmark, with the compiler flags "-Ofast -ipo -static -xskylake-avx512 -qopenmp -DSTREAM_ARRAY_SIZE=80000000" used. The measured peak bandwidth for a single-core was 13.516GB/s.
Some data points in the following evaluation chapters lie above the memory bound line. This is possibly due to LIBXSMM gaining from better hand-written pre-fetching versus ICC's best efforts to add pre-fetching when compiling the tuned STREAM benchmark. LIBXSMM will issue SW pre-fetch instructions that go ahead of the (automatic) HW pre-fetching, helping to improve BW.

4.5 Validity of Results

Using AWS would at first suggest that the other VMs running on the same physical system would lead to a large variance in performance. To investigate this, we ran a small test with results shown in Figure 4.2, where each run was on a *different* AWS VM instance. Most of the data points are very tightly grouped, showing that the variation due to other workloads was not as large as first thought to be. This assumes a fair distribution of VM instances and workloads across the physical nodes in AWS.
Another potential issue is that AWS does not allow us to control the CPU frequency unless we rent enough cores or the entire CPU. This means any timing measurements will be affected by the turbo frequency of the CPU, which is likely to not remain constant due to varying CPU temperatures and other uncontrollable factors when using an AWS VM. After the first round of testing, a reliable, stable average recording of 2.4GHz was found for the frequency, which is one of the AVX-512 heavy-use boost frequencies of the CPU.
Another potential pitfall of using AWS is having to share memory. However, on AWS it is highly unlikely all other users will simultaneously use a lot of memory bandwidth. The bandwidth obtained (repeatable) on a single core VM using the modified STREAM benchmark is around the peak single core memory bandwidth measured in an experiment by AnandTech [14]. So sharing the memory does not appear to impact the bandwidth we

can achieve.
In PyFR simulations, kernels are run on all available cores with memory bandwidth being heavily used by all cores. AnandTech also showed that as more cores utilise memory bandwidth, the bandwidth available per core decreases [14]. So the threat to the validity of our results is that for kernels that are fully memory bound, they may be running faster than if that kernel was being ran on all cores. We will consider this limitation in the evaluation, especially when discussing the roofline plots. One point to note is that not all CPU architectures have as large a decreasing bandwidth per core as more cores are utilised, as Skylake-SP does.
Additionally, it is possible that the stressed core running the benchmark could have access to more L3 cache, if other AWS users on the same machine have low memory use. So any results that rely on access to a larger L3 share may not show typical performance that would occur in a PyFR simulation. However, the kernels are intended to operate using L1 cache and have minimal spilling. So the availability of more L3 cache should not impact the performance obtained especially since we use $192,000$ columns for $\mathbf{B}$ (which ensures $\mathbf{B}$ does not fit on the Intel Xeon Platinum 8175M's L3 cache).

Summary

We provided detail on both the software and hardware environment used for the evaluation and described the process of how we settled on the chosen platform. For the benchmark process, we showed the results of an initial experiment to tune some benchmark parameters in order to better simulate a PyFR workload. We also described the synthetic suite that forms part of the benchmark. The performance metric '*pseudo-FLOP/s*' will be used in the evaluation and we explained how it is calculated and the motivations behind using it. Next, we described how the boundaries of the roofline plots were obtained. Finally, we addressed the major potential threats to the validity of the measured results based on the chosen benchmark environment and methodology.

Chapter 5

Register Packing Solutions

In this chapter we introduce two schemes to pack the matrix **A** in an AVX-512 register file, that enables single instruction broadcasting, regardless of the vector lane that the value is stored in. The value is required in a broadcasted format (repeated across all of the lanes in the vector) for the MM routine. The first solution uses a shuffle instruction with an intricate storage layout and the second uses a permute instruction. Both solutions are compared in terms of the number of unique non-zero values they could pack into a register file.

5.1 Solution with Shuffles

AVX-512 Shuffle Instruction

The following storage layout was designed to be able to use the *VSHUFF64X2* AVX-512 instruction. On Skylake-SP it has a 3 cycle latency [9]. The instruction has two source vector register operands, an 8-bit operand and one destination vector register operand. The 8-bit operand is encoded at compilation time via embedding into the instruction.
In this instruction, the vector registers are split into sections of 128-bits, so each source and destination register has 4 sections. The *VSHUFF64X2* instruction copies 128-bit sections from the source registers to the destination.
The 8-bit integer, called the *selector*, selects which source sections to copy for a destination register. The selector is split into 4 groups of 2-bits. The lowest 2-bits select for the lowest 128-bits in the destination, and the highest 2-bits select for the highest 128-bits in the destination.
The lowest 4-bits of the selector, select for the first two 128-bit sections in the destination register, which are chosen from the first source register. A 2-bit selector can choose any of the four 128-bit sections from the source register.
So the lowest half (256-bits) of the destination register is given two 128-bit sections from the *first* source register. The 128-bit sections chosen can be the same or different. Similarly, the highest 4-bits of the selector choose 128-bit sections from the second source register. The highest half (256-bits) of the destination register is given two 128-bit sections from the *second* source register. The entire operation is illustrated in Figure 5.1 [6].

Single Instruction Broadcasting

To make use of the *VSHUFF64X2* instruction for a single-instruction broadcast the 128-bit sections have to contain *only* the desired value to be broadcast. This means that the value must be repeated for 128-bits. For double precision, this means storing the value twice. For single precision, the value must be stored four times. The logical register layout is shown in Figure 5.2, which shows that there are four 128-bits sections in a register. A

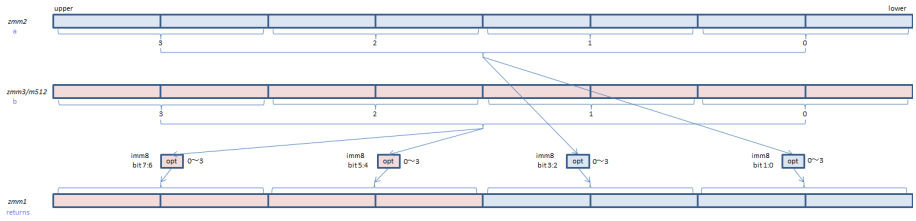
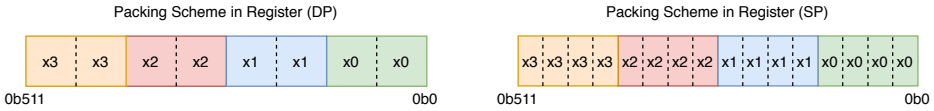


Figure 5.1: Illustration of VSHUFF64X2 [6]

trade-off is made by repeating values to obtain a single-instruction broadcast, that uses only one temporary register to hold the broadcasted value.



(a) Double Precision (b) Single Precision

Figure 5.2: Layout within Register for DP and SP values

Figure 5.3 shows how the *VSHUFF64X2* instruction can be used with the DP logical storage layout from Figure 5.2 to achieve a single instruction broadcast, regardless of which lane the value is stored in (groups of sequential lanes due to repetition). The same register is used as **both** source operands, which can be seen in Figure 5.3 by both the source registers being $zmm0$. This allows the selector to choose the same 128-bit source section, for all four 128-bit sections in the destination register. The end result is that the selected source section is broadcast to all four sections - a $1-4$ broadcast operation. The same process would apply to SP and would also be a $1-4$ broadcast, as the SP value must be repeated four times to fill the 128-bit section.

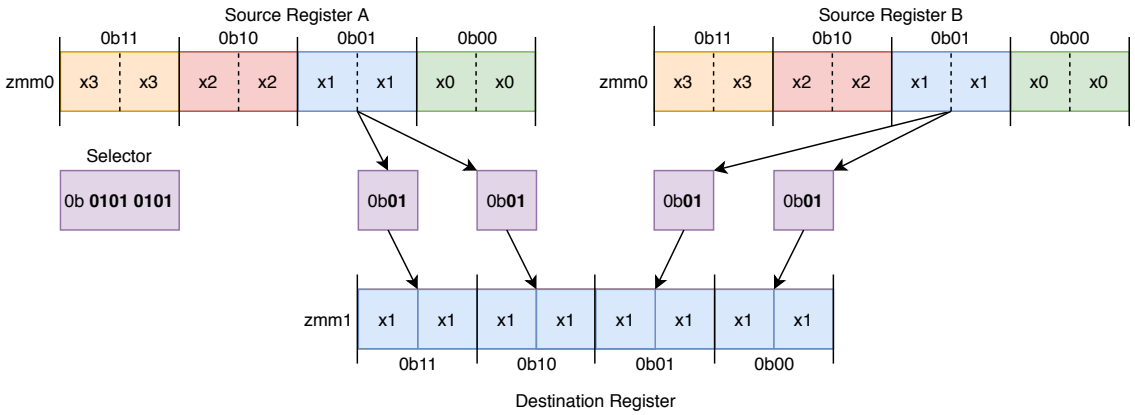


Figure 5.3: Using VSHUFF64X2 to Broadcast

Figure 5.4 showcases the four possible broadcasts and the corresponding selector values required to achieve them. The binary values show that a 2-bit value is repeated, which is due to the same source 128-bit section being chosen for all four destination sections.

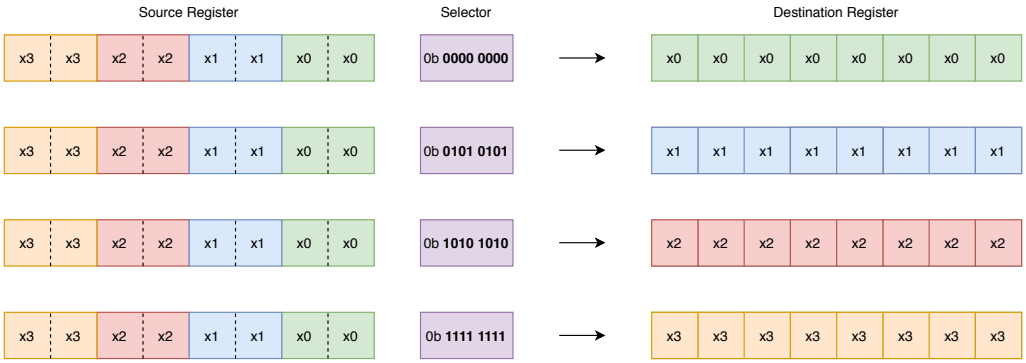


Figure 5.4: The four possible effective broadcasts

Register Packing for the MM routine

By fully storing **A** in the register file with the packing scheme, no memory access is required for **A** during the MM routine. The adaptation made to the routine from Section 2.3.2 is to broadcast the required value of **A** from the *register file* using the *VSHUFF64X2* instruction. Figure 5.5 shows that the 31^{st} register, *zmm30*, is used as the temporary register to hold the broadcasted value. This is then multiplied with the stride of **B** (loaded from memory) in the FMA to calculate (part of the accumulation of) the stride of **C**.

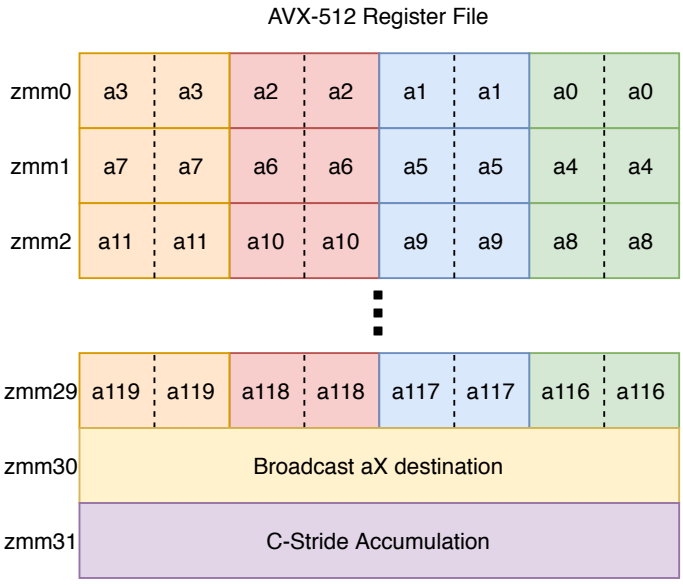


Figure 5.5: The Vector Register File logical layout when using register packing for the SpMM routine

5.2 Solution with Permutes

AVX-512 Permute Instruction

Permute instructions provide a method to rearrange the order of elements in a vector. The *VPERMD* and *VPERMQ* instructions allow for arbitrary positioning of elements. Both instructions have a 3 cycle latency on Skylake-SP [9]. A source element can be sent to any destination lane, as shown in Figures 5.6 and 5.7. *VPERMD* has 16 elements, splitting the vector register into 32-bit values. *VPERMQ* has 8 elements, splitting the vector register into 64-bit values.
For both instructions, one source operand, either a vector register or memory location, is used as the source of data. The other source operand is used as a selector. The selector has the same number of values as the number of elements in the destination register. This means that each destination value can be filled with any of the source elements. By choosing the same source element for every destination element, a single instruction broadcast is achieved.

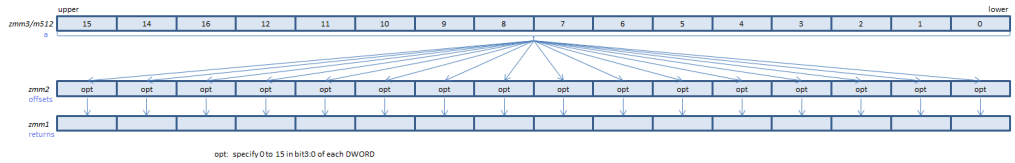


Figure 5.6: Illustration of VPERMD/PS [6]

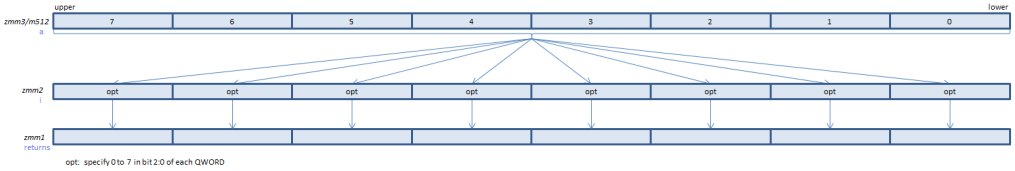


Figure 5.7: Illustration of VPERMQ/PD [6]

32-bit Permute for 64-bit values

LIBXSMM has the *VPERMD* instruction available, but not the *VPERMQ*. Whilst the 64-bit permute instruction could have been added, it is possible to perform the exact same operation with the 32-bit permute instruction.
Figure 5.8 shows the logical layout within a register packed values for both DP and SP. The *VPERMD* treats the register as the SP layout. However, a DP value (64-bits) can be treated as the concatenation of two consecutive 32-bit (SP) values.
Figure 5.9 shows eight 512-bit values for the selector operand of *VPERMD*. Each operand

(a) Double Precision (b) Single Precision

Figure 5.8: Layout within Register for DP and SP values

corresponds to broadcasting a certain 64-bit value from the data source operand. For example, broadcasting bits 63-0, the selector alternates between an index of 0 and 1. As

this is selecting 32-bit values, the first 64-bits of the destination is composed of the first and second 32-bits from the source. These 2 32-bits join to form the DP value. The pair of 32-bits is repeatedly selected for every 64-bit destination as well. The linked white boxes in Figure 5.9 detail which source lane (DP/64-bit value) is broadcasted by the corresponding permute selector operands.

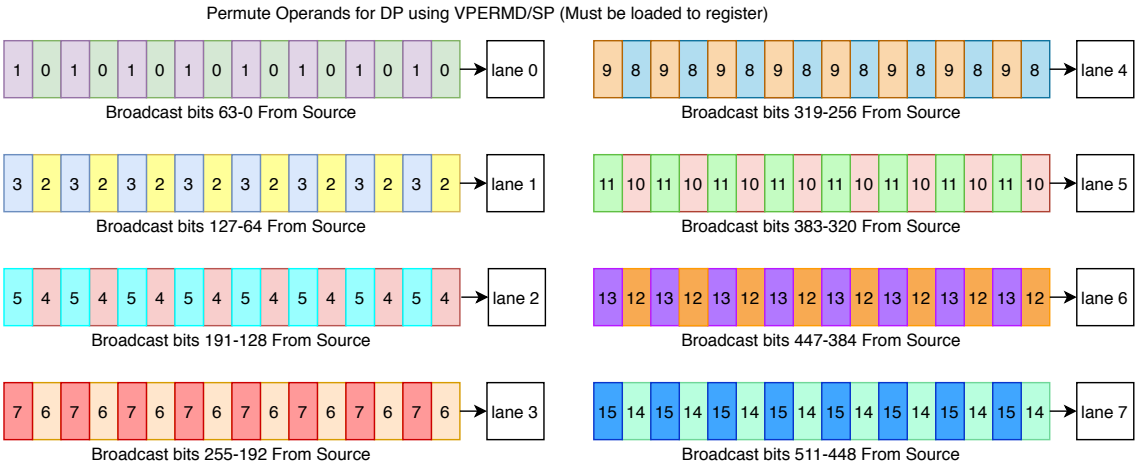


Figure 5.9: Using VPERMD to Broadcast DP

Register Packing for the MM routine

VPERMD can be used to broadcast either SP values (trivially) or DP values as shown. This solution, like the Shuffle solution, fully stores A in the register file, so no memory access is required for A during the MM routine. The adaptation made to the routine from Section 2.3.2 is also the same, apart from the broadcasting instruction and the layout in the register file. Figure 5.10 shows that for DP, 8 vector registers are used to store the permute operands. This leaves 22 registers to store data, meaning 176 DP values can be packed. The other 2 registers are used to hold the broadcasted value and the C-stride being accumulated.
For SP, 16 registers are used to hold permute operands, as there as 16 of them. This leaves 14 registers for data with a total of 224 SP values that can be packed.

5.3 Summary: Comparing the Solutions

Table 5.1 shows that the solution using permutes supports a greater number of unique non-zeros to be packed, for both DP and SP. Both instructions also have a 3 cycle latency on Skylake-SP and both issue *one* micro-operation [9].
However, the permute solution requires 8/16 permute operands to be loaded into the register file for DP/SP respectively. The trade-off for supporting more unique non-zeross is that space is taken up by storing the permute operands in the register file. Whilst the *maximum* number of supported non-zero values is higher, the permute solution is not always the most *space* efficient. Figure 5.11 shows the amount of registers required by each solution (for both DP and SP) as the number of unique non-zeros to be packed increases from 32 (when below 32, the current LIBXSMM pre-broadcast solution is more optimal). The solution using *VSHUFF64X2* is more space efficient for DP until 64 values are to be packed, and for SP until 85 values are to be packed. The greater the number of free registers available, the greater the potential number of further optimisations that could be made.

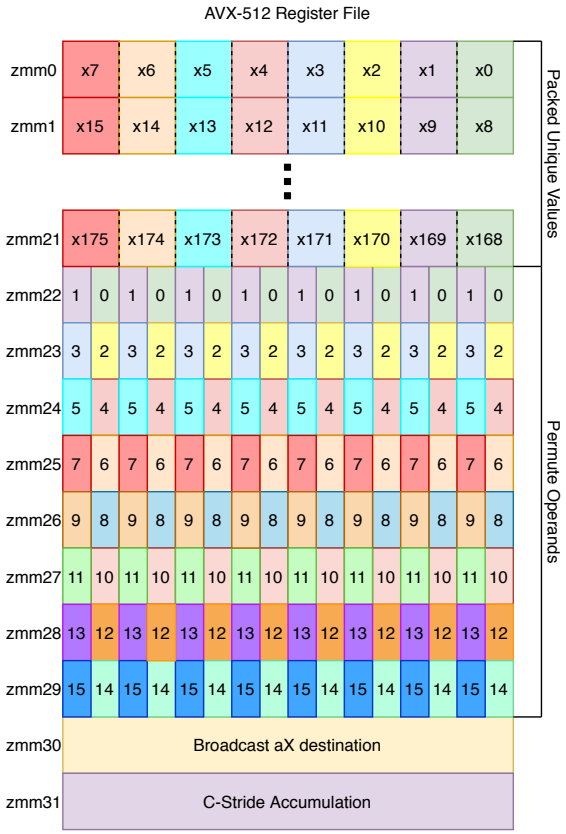


Figure 5.10: The Vector Register File logical layout when using register packing for the SpMM routine (DP)

Method	SP	DP
Shuffle	120	120
Permute	224	176

Table 5.1: Number of Unique Non-Zeros that can be packed with the Solutions

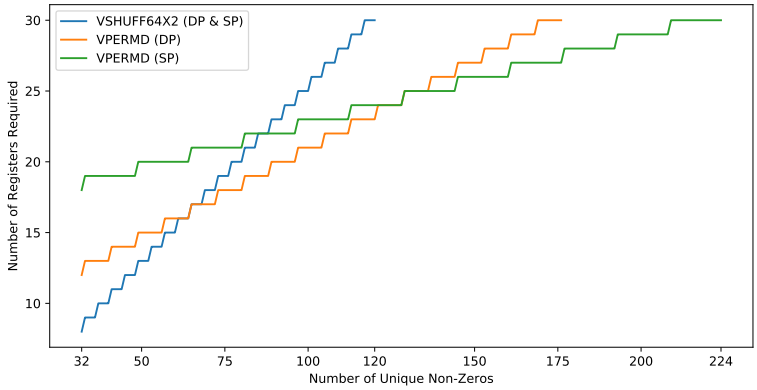


Figure 5.11: Comparing the number of registers required by the packing schemes against the number of non-zero values required to be packed

Chapter 6

Register Packing Evaluation

For the evaluation of register packing, we will show results using the **permute** solution, as the the maximum number of unique non-zeros that can be packed with it is higher. As no further optimisations are used in this chapter, the number of free registers is not of concern. Initial evaluation has shown that the performance difference between both solutions is within margin of error for matrices with a number of unique non-zeros supported by both. This was expected as both are 3 cycle latency, single instruction broadcasts, with all the data of **A** remaining in the register file.

The register packing using permutes is evaluated against the reference LIBXSMM on the PyFR suite and synthetic suite in this chapter. We separately explore the PyFR suite based on the shape used for the mesh, comparing the performance against the number of unique non-zeros, as well as the density of the operator matrices. Additionally, accompanying roofline plots are shown and discussed. When evaluating the synthetic suite, we plot performance against the (independent) varying characteristic and discuss the experimental results obtained with the help of roofline plots. Throughout the evaluation, we refer to the following proposed hypotheses and test them against the results.

Hypotheses

1: Kernels for operator matrices where the number of unique non-zeros (U), is bound by $U \leq 31$, should not have decreased performance when using register packing compared to other *sparse* strategies deployed by LIBXSMM. The runtime broadcasting should *not* add to the critical path of execution.
2: Kernels for operator matrices where the number of unique non-zeros (U), is bound by $31 < U \leq 176$, should have increased performance compared to other strategies deployed by LIBXSMM.

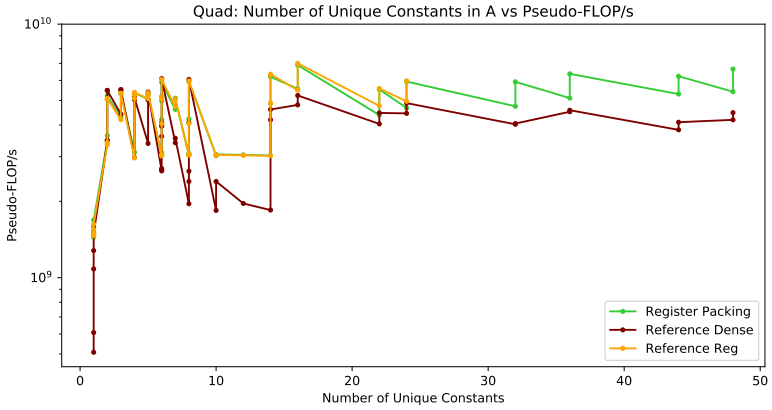
6.1 Evaluation on PyFR Suite

The following plots compare the performance, measured by *pseudo-FLOP/s*, of the register packing strategy (green line) versus two existing LIBXSMM strategies; the register routine from Section 2.4.3 (orange line) and a *dense* MM routine (maroon line). When the number of unique non-zeros (U) is greater than 31, LIBXSMM falls back to this dense MM routine. For this evaluation we also measured the performance of the dense routine when $U < 31$ as well as the existing sparse register routine.
The dense routine used by LIBXSMM is a well optimised dense GEMM routine for small matrices, that implements various tiling strategies. In the following roofline plots, the Arithmetic Intensity (AI) is calculated differently for the sparse routines compared to the

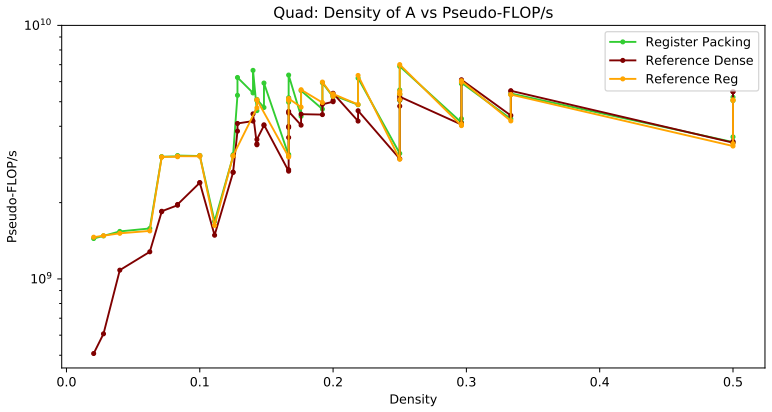
dense routine. For the sparse routines, strides of B and C are only counted as memory loads/stores if the element in A that causes the load/store is non-zero. This (correctly) reduces the count of number of bytes transferred for sparse matrices. The AI for the dense routine counts every byte of B and C for the amount of bytes transferred to/from memory. Appendix C contains further plots from the benchmark, plotting performance vs the number of rows in A, the number of columns in A and the size of A.

Quadrilaterals

Figure 6.1a shows the performance of the three strategies for the quadrilateral operator matrices vs each matrices U. For $U \leq 31$, it can be seen that the dense strategy performs slower than both sparse routines for some matrices, and the same for others, but never faster. Figure 6.1b plots the performance vs the density of the same operator matrices. This shows that when the dense routine performs the same as the other strategies, the operator matrices are relatively dense. So although $U \leq 31$, the increased density allows the dense routine to reach the same performance. When the matrices are very sparse, the sparse routines can perform over 2x faster.



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 6.1: Register Packing vs Reference Performance - PyFR Quadrilateral Examples

Comparing the existing register routine and the register packing routine, for $U \leq 31$, the performance is nearly identical apart from two data points where the existing routine performs slightly, yet insignificantly, faster. This data generally supports hypothesis **1**, that there is *no* performance penalty for runtime broadcasting.
For $U > 31$, the register packing strategy *always* performs faster than the dense routine. Figure 6.1b shows that these matrices have a sparsity between 0.1 and 0.2, which explains why the *dense* routine cannot achieve similar performance. This part of the data supports hypothesis **2**, that register packing would provide increased performance for $31 < U \leq 176$. Speedups of up to around 1.5x can be observed in Figure 6.1a for $U > 31$.
The results are also shown as a roofline plot in Figure 6.2. For kernels with an Arithmetic Intensity (AI) of < 0.5, the sparse, register based routines are close to reaching the roofline based on memory bandwidth (BW). For these same matrices, the dense routine is shown to not come close to reaching the roofline as shown by the few, lower maroon points. This is due to using the pseudo-FLOP/s metric, where the dense routine is doing a lot of computation that is not considered useful (multiplying when the value of the operator matrix is 0). If using a traditional FLOP/s metric, we would expect these points to be higher on the plot. For $AI > 0.5$, the sparse routines also do not reach the roofline, but are still above the dense points. This suggest these kernels can reach a higher performance with more intricate optimisations.
There is one point, in Figure 6.2 where the dense routine is plotted above the sparse routines (the group of points above the roofline). Figure 6.1b indicates that this kernel was for a matrix with a density of 0.5, and so the dense routine performing faster is not too surprising in this specific case.

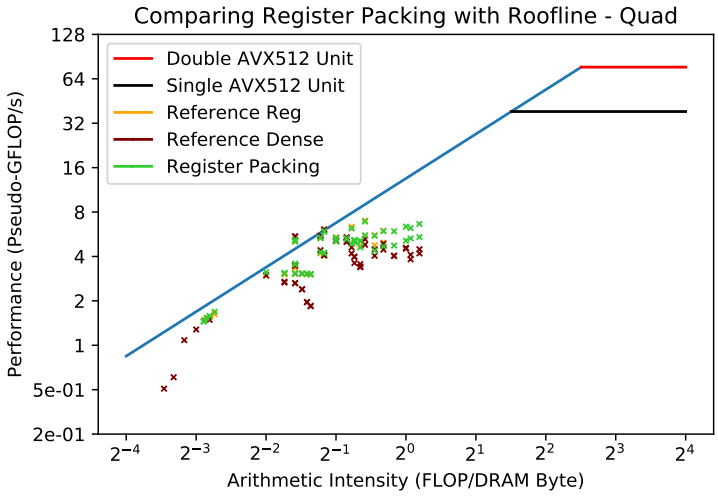
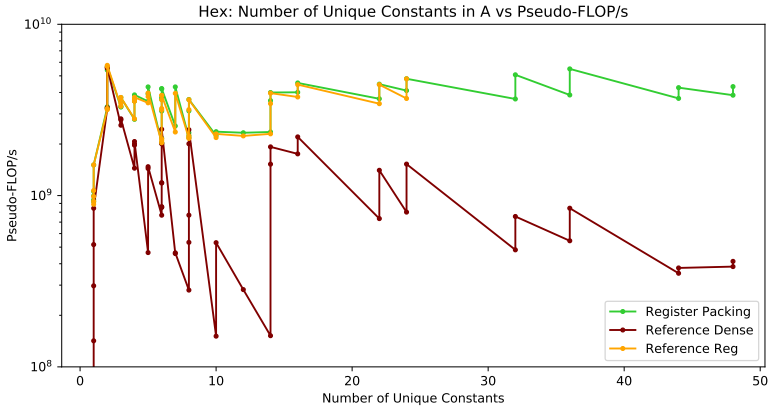


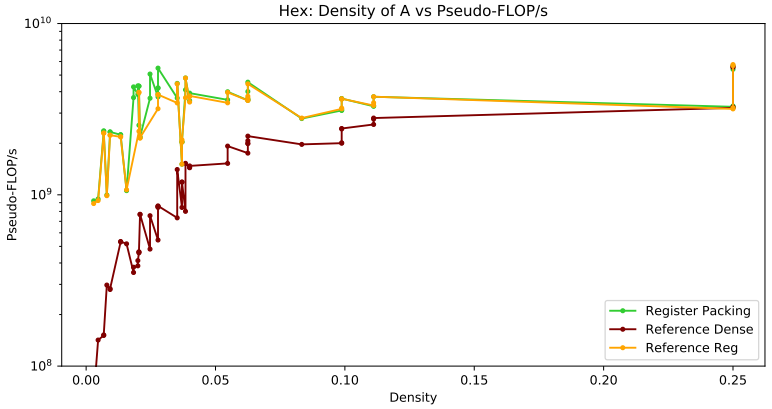
Figure 6.2: Register Packing vs Reference: Roofline Plot - PyFR Quadrilateral Examples

Hexahedra

The hexahedra operator matrices are more sparse relative to quadrilaterals. Figure 6.3b shows the performance of the three strategies vs the density of the matrices. Both sparse routines perform up to 10x faster over the dense routine for density under 0.05.
Figure 6.3a shows that for $U \leq 31$, the register packing routine performs near identical to the reference sparse register routine. This supports hypothesis **1**, that the runtime broadcasting does *not* add to the critical-path. Figure 6.4 shows that not all of the register packing kernels reach the memory BW roofline. This could be due to the memory access pattern of the kennels not achieving the peak we obtained in the modified STREAM benchmark. However, in a real PyFR simulation, we would expect a lower peak BW when all cores are used on Skylake-SP. We would then expect the few sparse kernels that reach the memory bound in the single core case to not be as fast in a real simulation. However, they would still be faster than the kernels from the dense routine.
For $31 < U \leq 176$, the register packing is shown to perform to perform around $8 - 10$x



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 6.3: Register Packing vs Reference Performance - PyFR Hexahedra Examples

faster than the dense routine. However, it should be noted these kernels were for very sparse matrices, with a density of under 0.05. This demonstrates the advantage of being able to still use a sparse routine, where the loops are fully unrolled and multiplications by 0 in A are skipped. By supporting a higher U, the updated LIBXSMM is prevented from falling back on the dense routine.
Figure 6.4 highlights how much the dense routine is not suited for hexahedra operator

matrices, with many of the data points for dense being well below the roofline due to multiplying by 0s from A, leading to low pseudo-FLOP/s. It should be noted, if traditional FLOP/s was to be plotted, the dense kernels would be much closer to the roofline.
As mentioned above, the sparse routines do not reach the roofline, and are further away when compared to the quadrilateral points in Figure 6.2. This suggests further optimisations could potentially have a greater impact on the kernel performance for hexahedra or that the an alternative memory access pattern could provide higher BW.

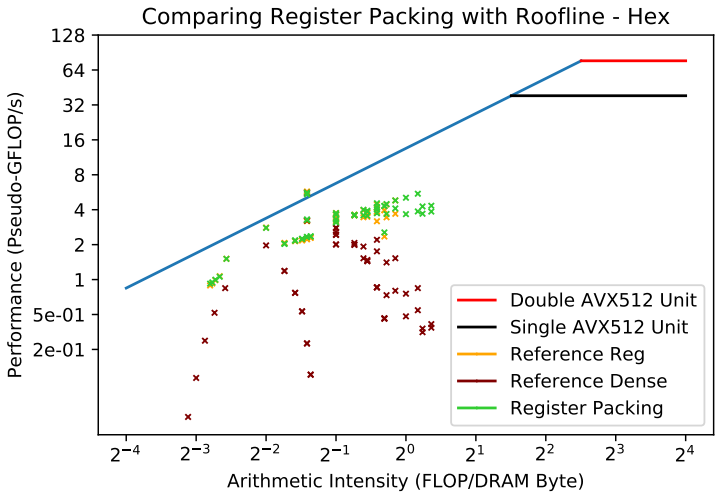
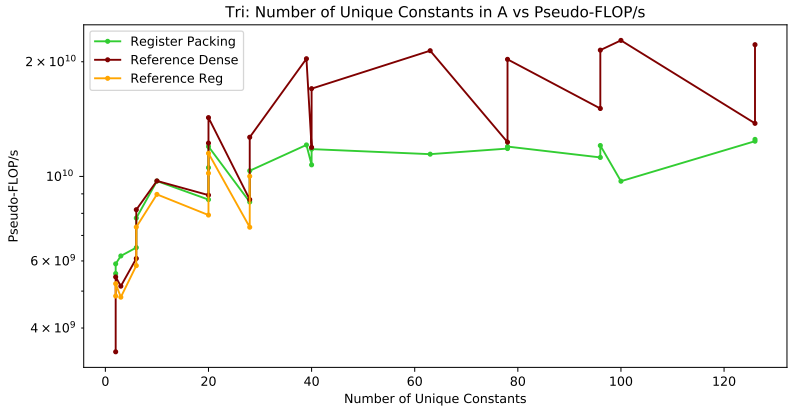


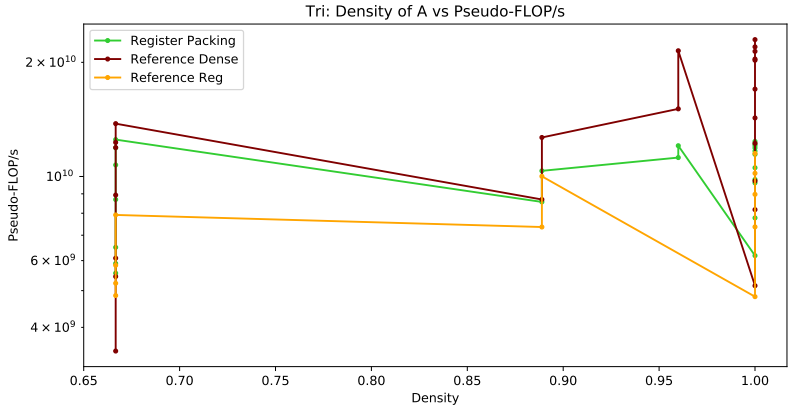
Figure 6.4: Register Packing vs Reference: Roofline Plot - PyFR Hexahedra Examples

Triangles

Operator matrices for triangles are more dense than the above matrices, as shown in Figure 6.5b. So using a sparse routine may not be the most optimal due to the many methods applied by advanced dense GEMM routines to increase performance, such as LIBXSMM's dense routine. However, for $U \leq 31$ the performance of all three strategies are close, but the dense appears to be the best, and the reference sparse register routine the worst. It is hard to draw conclusions as to why the register packing performed faster than the reference register routine, as the operator matrices vary in multiple ways other than density and U. For $U > 31$, the register packing either performs around the same, or around 0.5x as fast as the dense routine. Whilst not explicitly shown, this is most likely due to the density varying as well as U. So the 2x faster dense kernels are for *more* dense operator matrices. This data then suggests that hypothesis **2** is incorrect, as LIBXSMM has better alternatives for $U > 31$ than the sparse register packing routine.
For the higher AIs (> 1) in Figure 6.6, the dense routine points are shown to be close to the roofline, suggesting that the matrices are dense (which is the case) as the pseudo-FLOP/s plot appears to show expected performance for a traditional FLOP/s plot. For lower AIs, ≤ 1 all three strategies are very close to the roofline. The plot also shows that the register packing performs faster than the reference register routine. The kernels are close to being fully memory bound, and the runtime broadcasting did *not* make performance slower. This supports hypothesis **1**, that the additional work does not add to the critical-path.



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 6.5: Register Packing vs Reference Performance - PyFR Triangles Examples

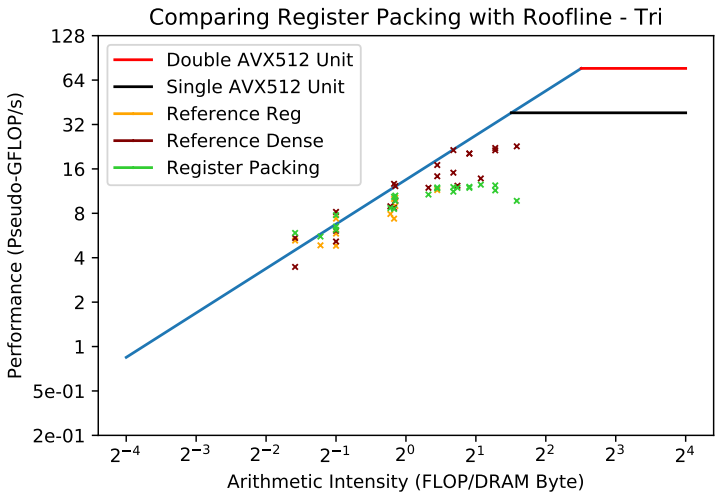
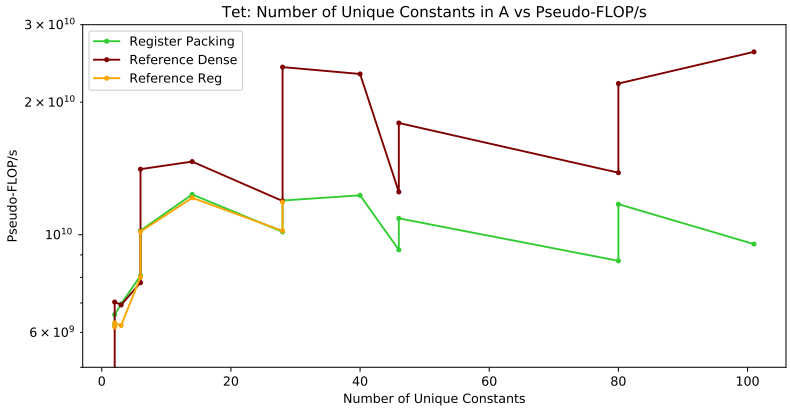


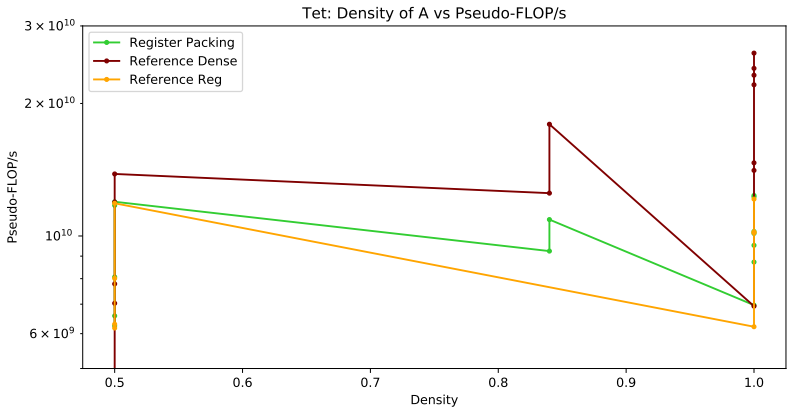
Figure 6.6: Register Packing vs Reference: Roofline Plot - PyFR Triangles Examples

Tetrahedra

Like triangles, tetrahedra operator matrices are dense. For $U \leq 31$, the register packing routine performs near identically to the reference sparse routine, shown in Figure 6.7a. However, the dense routine was again the best for $U \leq 31$. This supports hypothesis **1**, that the runtime broadcasting did not add to the critical path of the sparse routine. For $U > 31$, the dense routine kernels once again outperformed the register packing kernels. Figure 6.7b shows that these speedups came when the density was over 0.8. Regardless, the data shows that hypothesis **2** does not hold, and that LIBXSMM does have better alternatives to register packing for $31 < U \leq 176$ in certain scenarios.
Figure 6.8 shows a similar story as the roofline plot for the triangle operator matrices showed.



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 6.7: Register Packing vs Reference Performance - PyFR Tetrahedra Examples

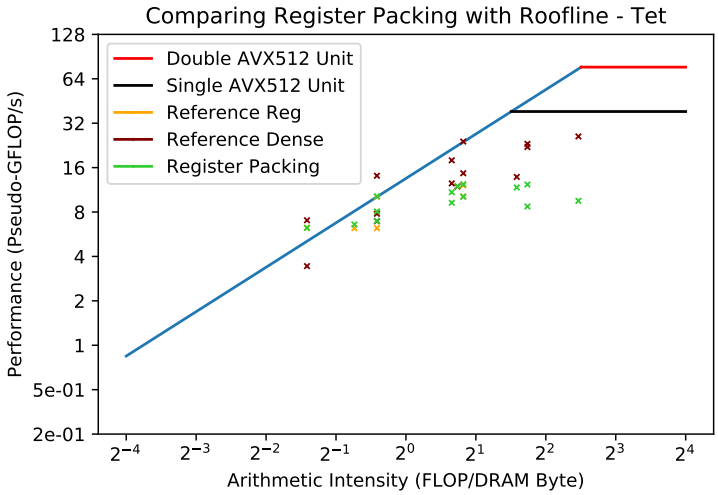


Figure 6.8: Register Packing vs Reference: Roofline Plot - PyFR Tetrahedra Examples

6.2 Evaluation on Synthetic Suite

In the synthetic suite one of the main characteristics of the operator matrices is varied whilst the rest are held constant. The base configuration was 128 rows and columns, a density of 0.05 and either 16 or 64 number of unique non-zeros (U). The reference version of LIBXSMM uses the sparse register MM routine for the set of matrices where $U = 16$ (orange), and uses the dense MM routine for the set where $U = 64$ (maroon). In the following plots, the results for the sparse register packing MM routine are shown for $U = 16$ (green) and $U = 64$ (cyan).

Vary Number of Rows

Figure 6.9 shows that the sparse routines at least maintain around a 2x speedup over the dense routine as the number of rows in A is increased. At 1024 rows, the *dense* routine performs *slower* than for 512 rows. The dense routine in LIBXSMM picks between many different sub-strategies depending on the dimensions of the matrices. However, the available options are designed for *small* operator matrices. So as A gets too large, the dense routine has issues to pick an optimal sub-routine/strategy. The sparse routines, of course, do not have this issue as they store A efficiently within the registers. Hypothesis **2** is supported by this data, as LIBXSMM does not have an alternative that performs *faster* in this case.
The register packing for $U = 16$ shows no performance penalty for broadcasting at runtime, strongly supporting hypothesis **1**. Register packing has very similar performance when $U = 64$ compared to when $U = 16$. The register packing routine broadcasts the element of A on every use, even if it was the most recently used (so already available). This leads to the similar performance. An improvement to the routine could be to not issue a broadcast if already available. The overhead of doing this check would come at the JIT compile time, not at runtime. However, as shown by Figures 6.10a, there may not be any gain from doing this. The register packing points are below the roofline and at the same locations for the reference sparse routine for $U = 16$, suggesting that the runtime broadcasting does not add to the critical-path when not memory BW bound. So, removing unnecessary broadcasting should *not* lead to any significant performance gain.

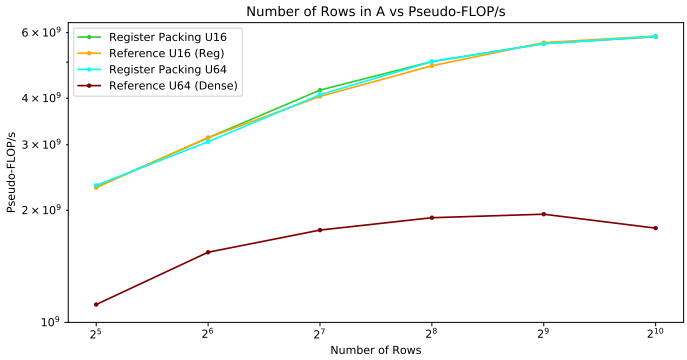
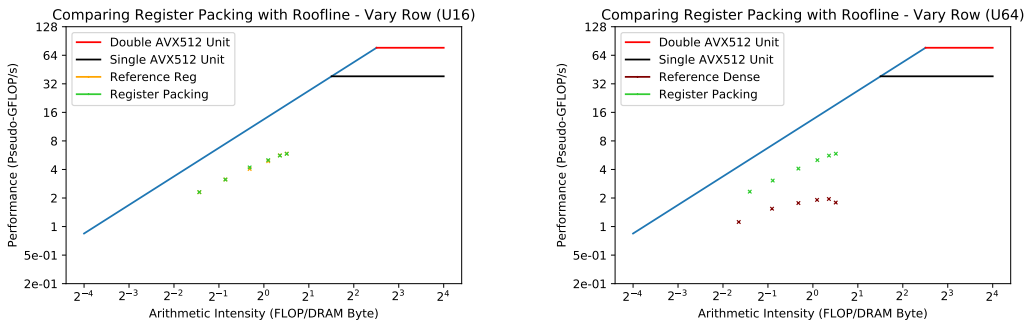


Figure 6.9: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Rows



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 6.10: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Rows

Vary Number of Columns

As the number of columns (C_A) in A increased beyond 128, the performance speedup of the sparse routines over the dense routine increased, shown in Figure 6.11. As C_A increases, the number of *rows* in B increases. At $C_A = 256$ and beyond, the dense routine struggles to manage the larger dimensions of **both** A and B. For the sparse routines, at $C_A = 512$, the 8 columns of B being repeatedly accessed in the routine, take up $32KB$. Skylake-SPs L1 data cache size is also $32KB$, leading to the routine spilling to L2 cache. However, the drop in performance due to using L2 for A is smaller than the drop due to the dense routine using sub-optimal small strategies for large matrices. This led to an increase in the speedup, to around 3x, whilst the performance of the sparse routines decreased.
For $U = 16$, the register packing routine does not perform slower than the reference sparse routine. In fact for both $U = 16$ and $U = 64$, register packing can perform faster, for example when $C = 512$. The data supports both of the original hypotheses. Figure 6.12b shows how the inability of dense routine to manage larger matrices moves the data points further from the roofline. We also see how bad cache use leads to lower data points on the roofline plots.

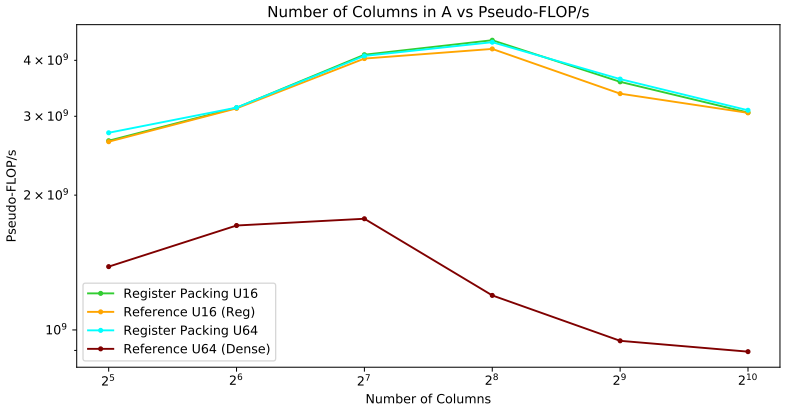
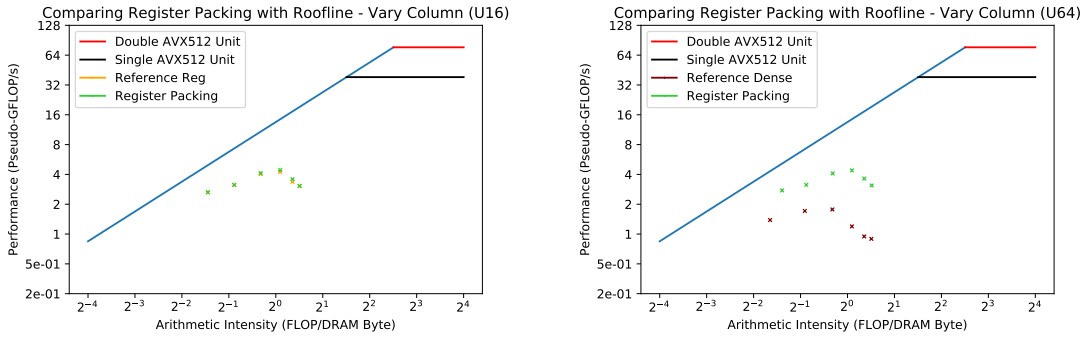


Figure 6.11: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Columns



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 6.12: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Columns

Vary Density

So far, the dense MM routine has always performed slower in the synthetic suite, which was expected as all the previous matrices had a density of 0.05. Figure 6.13 uncovers a critical point, where the dense routine stops being slower. At a density of 0.25, the dense routine performs around the same as the sparse routines, and performs faster as the density increases further. This suggests hypothesis **2**, that LIBXSMM doesn't have a faster performing strategy than register packing, is only true up until a critical density. This information can be used to add a density heuristic to LIBXSMM to decide to fall back to the dense routine, even if the sparse routine supports the U in the operator matrix.
Again, the data supports hypothesis **1**, that the runtime broadcasting is not detrimental to performance. For both $U = 16$ and $U = 64$, register packing performs at least the same as the reference sparse routine at $U = 16$.
Figure 6.14b shows that the performance of the reference dense routine kernels increase linearly with AI. However, performance of register packing kernels level off as AI increases. The levelling off is not due to the runtime broadcasting, as Figure 6.14a shows that the reference sparse routine kernels also level off. The cause for the plateau is more likely due to the under-utilisation of the available execution units, and is shown not to be due to memory BW limitations. This is where further optimisations could greatly improve performance for the sparse routines.

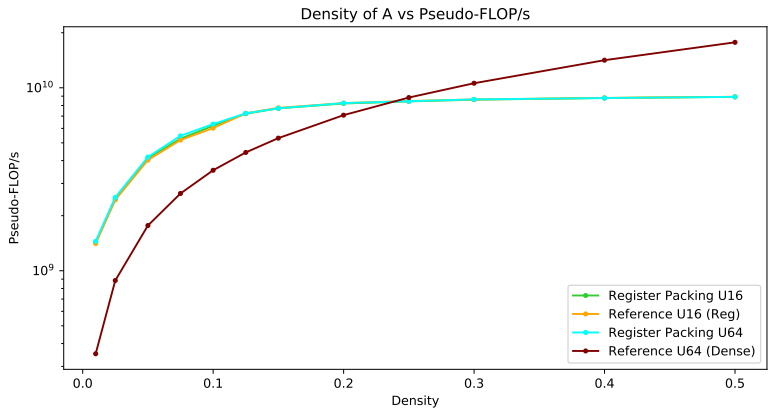
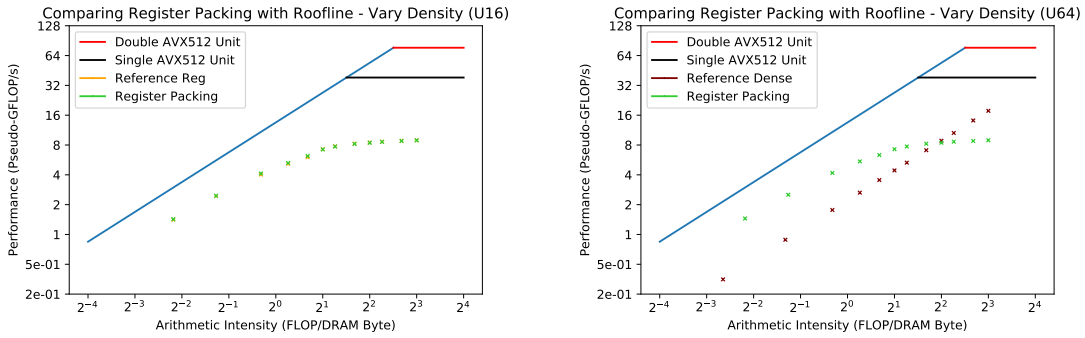


Figure 6.13: Register Packing vs Reference: Performance - Synthetic Suite Vary Density



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 6.14: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Density

Vary Number of Unique Non-Zeros

Finally, we vary U with the other properties held constant. Figure 6.15 shows that register packing had over a 2x speedup over the dense routine for $U \leq 176$. This shows that for a *small* A, that is relatively sparse ($\rho = 0.05$), register packing consistently delivers a speedup for matrices that it can support. At $U = 192$, the LIBXSSM with register packing defaulted to the same dense routine as found in the reference version.
Surprisingly, register packing consistently performs faster than the reference sparse routine for $U \leq 31$. We believe this is due to the out-of-order (OOO) scheduling within the CPU core, or possibly something else as intricate. Whilst hard to find evidence for what exactly gives the speedup, this speedup appeared in the experiments above, notably when the number of columns was varied.
Figure 6.16 shows that the register packing kernels did not reach the roofline. This is possibly due to matrices tested not being sparse enough, like some of the hexahedra examples from PyFR. However, this again suggests that further optimisations could help the kernels for these synthetic matrices reach the roofline.

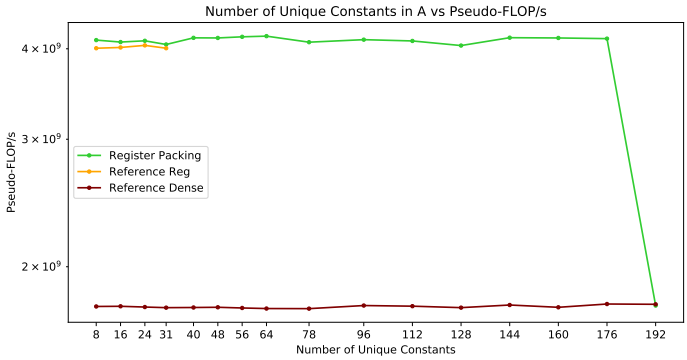


Figure 6.15: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Unique Non-Zeros

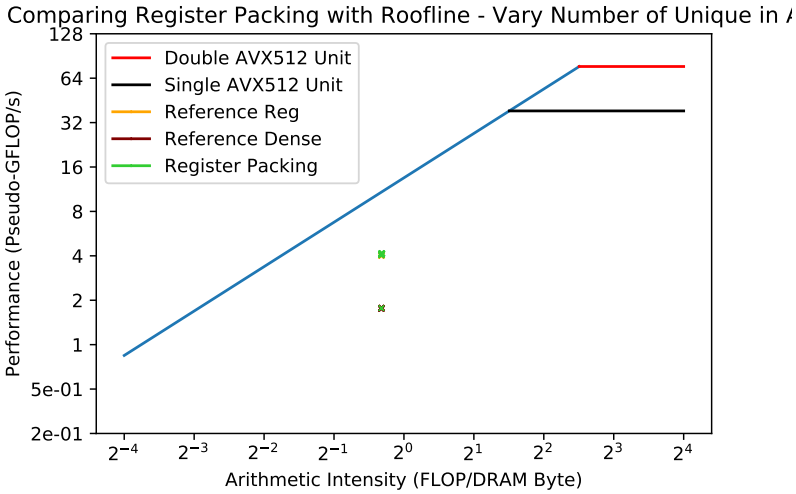


Figure 6.16: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Unique Non-Zeros

6.3 Summary

The results from all the experiments supported hypothesis **1**, that stated that there should be no performance penalty for runtime broadcasting for $U \leq 31$, compared to other LIBXSMM sparse strategies. Hypothesis **2** was disproved by the experiment with the denser matrices found in PyFR, and the synthetic experiment where the density was varied. We found that at around a density of **0.25**, the dense MM routine in LIBXSMM performs faster than all the sparse MM routines tested. This value is further supported by the hexahedra experiment, where all matrices had a density of under 0.25, and the dense MM routine never performed faster.
We also found that that using an **A** with too many rows (1024) or too many columns (256) led to the dense MM routine performance decreasing, showing that LIBXSMM's dense MM routine is suited for smaller operator matrices.
Finally, it was shown that register packing with runtime broadcasting can lead to slightly faster performance over the pre-broadcast, sparse MM register based routine from the reference LIBXSMM. We believe this is due to the OOO scheduling and execution in Skylake-SP working better for the code generated by register packing and its runtime broadcasting.

Chapter 7

Register Packing using L1 Cache to store Selector Operands

In this chapter we will iterate on the register packing solution that uses permute instructions for the broadcasting, showing how even more unique non-zeros can be packed by making a trade-off. The first section describes the steps taken to achieve this. We will then evaluate the modified version using the benchmark against the version from the previous two chapters, which we will refer to as the *base* register packing version. Due to supporting more non-unique zeros, we will also evaluate against the reference LIBXSMM for operator matrices that the *base* register packing version does not support.

7.1 Solution

7.1.1 Layout of Operands

In the *base* register packing version, vector registers are used to store the *selector* operands for the permute instructions, which are required for broadcasting the packed values. Instead, these can be stored in data memory, as shown in Figure 7.1. In total, the operands (for DP values) would take up 512 bytes ($8 * 64B$), or 1.57% ($\frac{1}{64}$) of the L1 data cache in a Skylake-SP CPU core (for SP they would take up 3.13%). This would free up 8 (16) vector register when using DP (SP). Table 7.1 compares the amount of unique non-zeros that can be packed when using 'L1 selector operands' compared to the *base* register packing. A total of 240 unique non-zeros can be packed for DP, and 480 for SP.

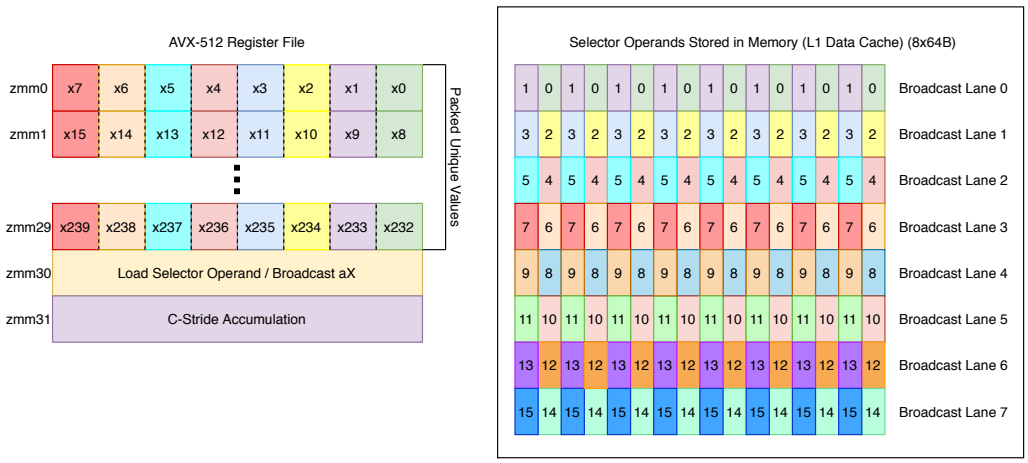


Figure 7.1: Logical storage layout for unique non-zeros and selector operands

Method	SP	DP
Base	224	176
L1	480	240

Table 7.1: Comparing the Number of Unique Non-Zeros that can be packed against the base register packing version

7.1.2 Broadcasting Process

However, this increase comes at a cost. An additional memory load instruction has to be issued for every permute instruction issued. Figure 7.2 displays the steps to broadcast the packed value. First, the selector operand is loaded into the *single* temporary register (zmm30). This is used as the second source register for the permute instruction (first source contains the packed value). The result of that permute instruction is written in the same register, zmm30, overwriting the selector operand with the broadcasted value. This can then be used for the FMA in the matrix multiplication routine.

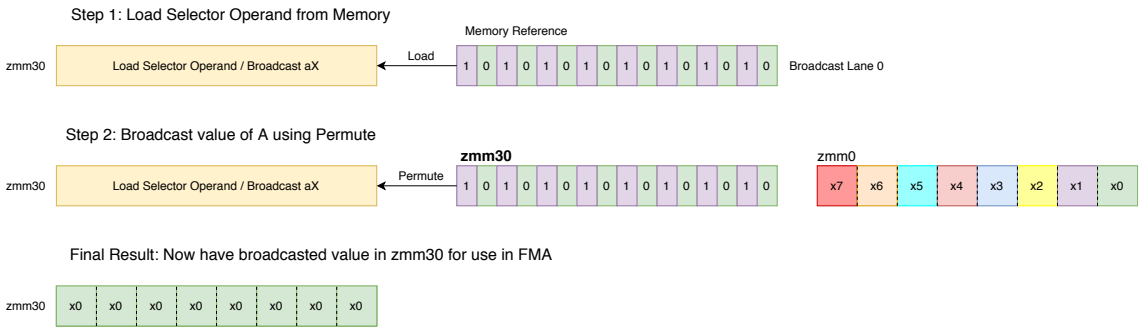


Figure 7.2: Updated broadcasting process

7.1.3 Alternative Strategy

The selector operand for the permute instruction *must* be a vector register and not a memory reference. The data operand (one that contains the packed value) can however be a memory reference. This would breakdown into multiple micro-operations.
An alternative strategy would be to continue to store the selector operands in the register file, and to selectively store packed values in memory, once the number of unique non-zeros is above 176. This would lead to fewer additional memory loads. However, the risk in deploying this strategy is that the packed values of A can be evicted from L1-cache. If this occurs often, the stalls would lead to slower performance.
In the case of storing the selector operands in L1-cache, they are much less likely to be evicted, as they are likely to be loaded more frequently. There is also a bound on the amount of space taken by them, so there is less of a concern on the contention for L1 between them and B.

7.2 Evaluation

As in the previous evaluation on the *base* register packing solution, we will show and discuss performance, measured by *pesudo-FLOP/s*, against the number of unique non-zeros (U) and the density of the operator matrices. Roofline plots will also be provided.
In the following sections we will first evaluate the modified register packing, referred to as 'Register Packing L1' (RP L1) against the *base* version, referred to as 'Register Packing Base' (RP Base) using the benchmark.
Additionally, we evaluate against a few more PyFR operator matrices that have $176 \leq U \leq 240$. The performance is compared against the *dense* routine from the reference LIBXSMM version. The focus throughout the evaluation will be to determine if the following hypothesis holds.

Hypothesis

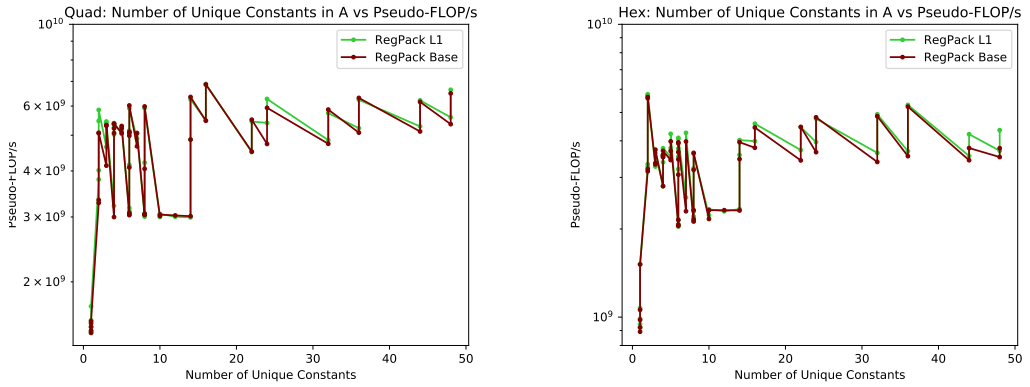
3: Kernels loading selector operands from memory (L1 data cache) should perform about the same as kernels using the *base* register packing solution. The reason for this is that there should be spare read bandwidth to the L1 data cache in the kernels using the *base* solution, which can be used to load the selector operands.

7.2.1 Evaluation on PyFR suite

The following plots show the performance of RP L1 (green lines) versus RP Base (maroon lines). From the PyFR suite of operator matrices, the examples from quadrilaterals and hexahedra have been placed into a 'sparse' group, and the examples from triangles and tetrahedra form a 'dense' group. All examples tested have a number of unique non-zeros supported by both strategies. The evaluation will focus on determining if there is a performance difference due to reading selector operands from memory. Appendix D.1 contains further plots from the benchmark, plotting performance vs the number of rows in A, the number of columns in A and the size of A.

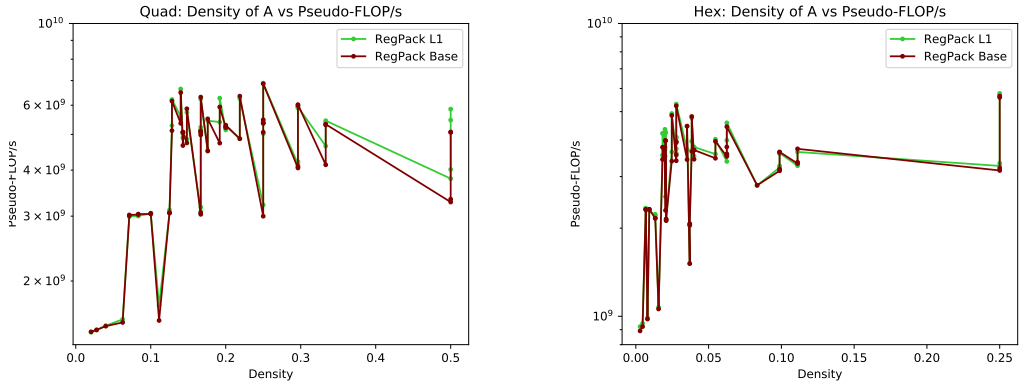
Sparse Operator Matrices

Figures 7.3 and 7.4 show that there was no significant performance difference between RP L1 and RP Base for sparse operator matrix examples from PyFR. There are a couple of points for both quadrilaterals and hexahedra kernels where the the RP L1 performed up to around 1.1x faster. Figure 7.5 shows that the *majority* of these speedups are for kernels that do not reach the (memory) bound. These smaller speedups can then be attributed to the micro-architecture of Skylake-SP favouring the instruction sequence of RP L1, leading to a small pseudo-FLOP/s increase. If RP L1 displayed the same speedups for the same kernels across a variety of micro-architectures, then a different conclusion could be made. However, given that the evaluation is only carried out on one architecture, and that the *majority* of kernels perform the same, the results from the 'sparse' group of operator matrices support hypothesis **3**.



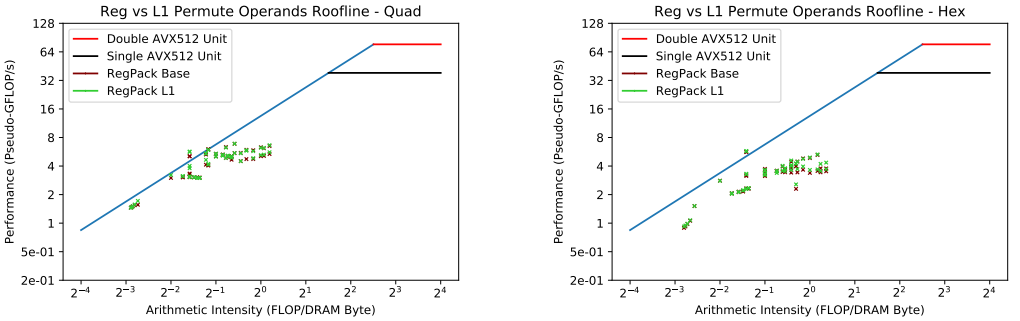
(a) Quadrilaterals (b) Hexahedra

Figure 7.3: Register Packing L1 vs Base - PyFR Sparse Examples (Number of Unique Non-Zeros)



(a) Quadrilaterals (b) Hexahedra

Figure 7.4: Register Packing L1 vs Base - PyFR Sparse Examples (Density)

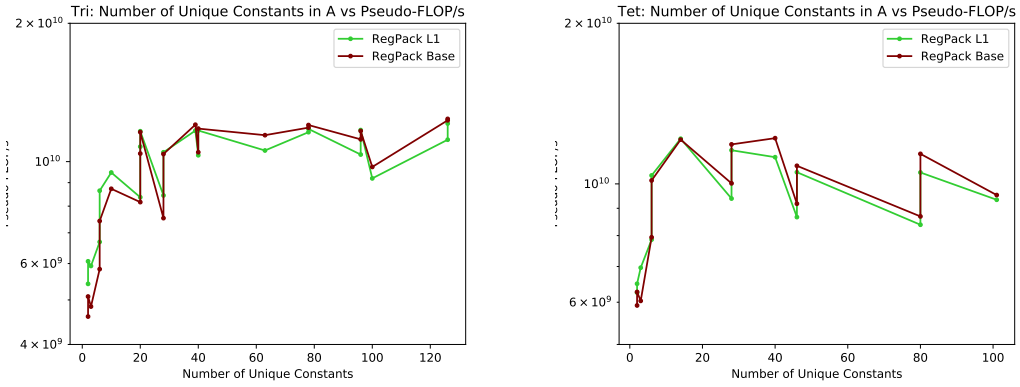


(a) Quadrilaterals (b) Hexahedra

Figure 7.5: Register Packing L1 vs Base - PyFR Sparse Examples (Roofline Plots)

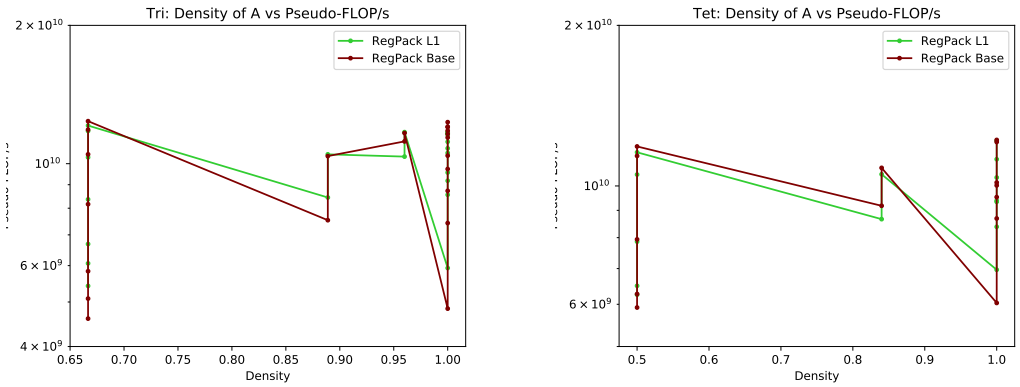
Dense Operator Matrices

Figures 7.6 and 7.7 show that generally, kernels using RP Base perform *slightly faster* than kernels using RP L1, up to around 1.1x, disproving hypothesis **3**. For the denser operator matrices, more broadcasts are required and so more reads from memory for selector operands are required. Interestingly, there are a couple of data points at a density of 1.0 where RP L1 performed around 1.2x faster. Figure 7.8 shows that this occurs at extremely small matrices, where there are fewer than 5 columns. Figure 7.9 reveals that the slowdowns occur at high AI, which correspond to the larger and denser operator matrices. Slowdowns mean that the more complex broadcasting was adding to the critical-path of execution.
However, at low AI, the results show performance speedups for RP L1 over RP Base, especially for triangles, shown in Figure 7.9a. These points correspond to operator matrices that are relatively small. These kernels were memory bound when using RP Base, and so the additional broadcasting could not negatively impact performance as there was 'free' time whilst waiting on main memory.



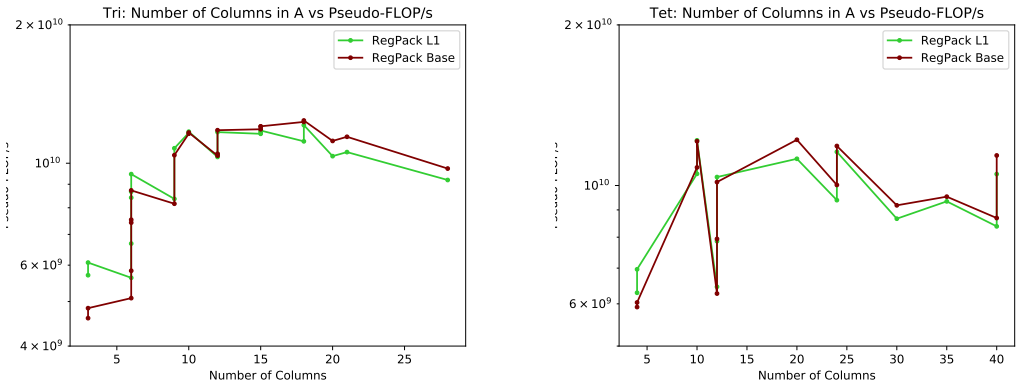
(a) Triangles (b) Tetrahedra

Figure 7.6: Register Packing L1 vs Base - PyFR Dense Examples (Number of Unique Non-Zeros)



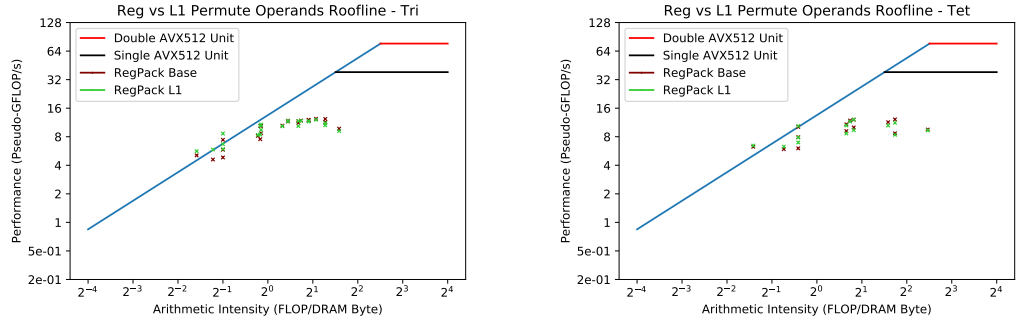
(a) Triangles (b) Tetrahedra

Figure 7.7: Register Packing L1 vs Base - PyFR Dense Examples (Density)



(a) Triangles (b) Tetrahedra

Figure 7.8: Register Packing L1 vs Base - PyFR Dense Examples (Number of Columns)



(a) Triangles (b) Tetrahedra

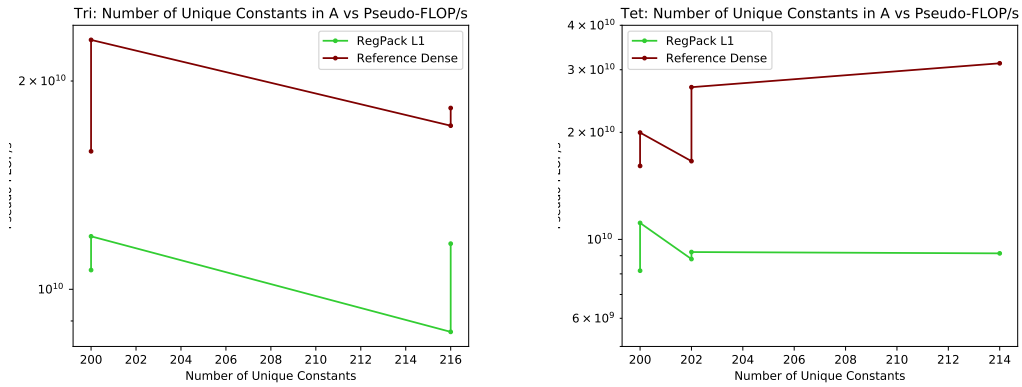
Figure 7.9: Register Packing L1 vs Base - PyFR Dense Examples (Roofline Plots)

7.2.2 Evaluation on Large PyFR Operator Matrices

The following evaluation is performed on example PyFR operator matrices that have $176 \leq U \leq 240$. RP Base does not support these, and so RP L1 is compared against the dense routine from the reference LIBXSMM. Instead of testing for hypothesis **3** in this specific evaluation, we are mainly interested in seeing if the results from Chapter 6 hold, where the dense routine beat RP Base for the dense group of PyFR matrices. Appendix D.2 contains further plots from the benchmark, plotting performance vs the number of rows in A, the number of columns in A and the size of A.

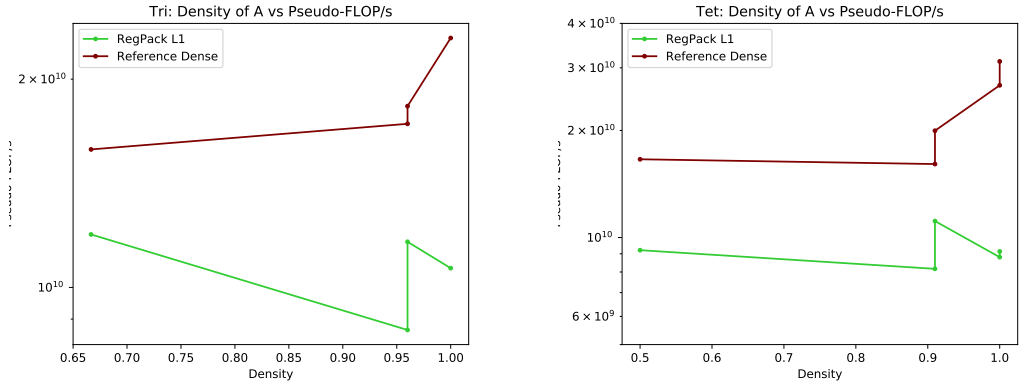
Dense Operator Matrices

Figures 7.10 and 7.11 clearly show that RP L1 is *slower* than the dense routine for these operator matrices, by a factor of up to 3x in some cases. Figure 7.12 shows that none of these kernels were memory bound. From this, we cannot determine if it was either the use of L1 selector operands, or the general sparse MM routine, that was the main contributing factor to the slower performance. Results from Chapter 6 on RP Base with similar matrices would suggest that the main reason would be the general sparse MM routine.



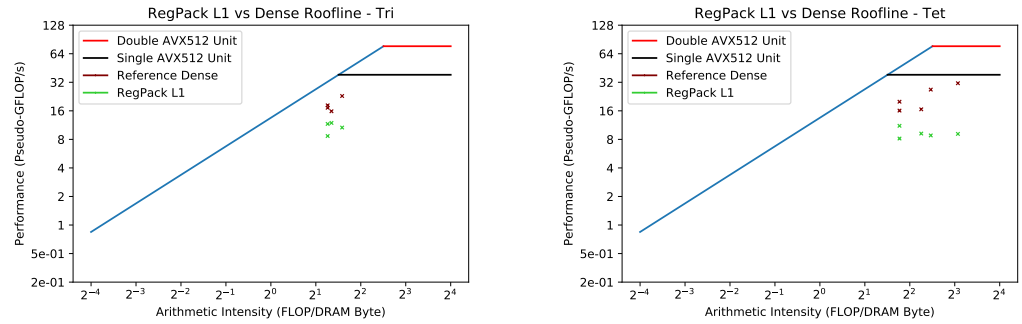
(a) Triangles (b) Tetrahedra

Figure 7.10: Register Packing L1 vs Reference - PyFR Large Dense Examples (Number of Unique Non-Zeros)



(a) Triangles (b) Tetrahedra

Figure 7.11: Register Packing L1 vs Reference - PyFR Large Dense Examples (Density)



(a) Triangles (b) Tetrahedra

Figure 7.12: Register Packing L1 vs Reference - PyFR Large Dense Examples (Roofline Plots)

7.2.3 Evaluation on Synthetic Suite

We will now evaluate RP L1 against RP Base using the synthetic suite of operator matrices. In the following plots, the results for RP L1 kernels are shown for $U = 16$ (green) and $U = 64$ (cyan). The results for RP Base kernels are shown for $U = 16$ (orange) and $U = 64$ (maroon). The results will be used to determine if hypothesis **3** holds or not, which is repeated below for convenience.

Hypothesis

3: Kernels loading selector operands from memory (L1 data cache) should perform about the same as kernels using the *base* register packing solution. The reason for this is that there should be spare read bandwidth to the L1 data cache in the kernels using the *base* solution, which can be used to load the selector operands.

Vary Number of Rows

As the number of rows increases, so does the size of A, and so does the total number of broadcasts that are issued in the kernel. The strides of B are reused more often and so the broadcasting can have a higher chance of impacting performance as less time is spent waiting on main memory loads. A very small speedup (less than 1.03x) when using RP Base over RP L1 is shown in Figure 7.13 at very large number of rows. However, for a more typical number of rows (≤ 128) RP L1 has a speedup over RP Base. Overall, the data provides support for hypothesis **3**, as it shows that there is no significant difference in performance.
Figure 7.14 shows that all of the kernels were not memory bound. Perhaps the small observed differences in performance would not occur if they were memory bound. In a scenario where the kernel was being run on all cores, the memory bandwidth per core would be reduced on Skylake-SP, and so these differences may not persist.

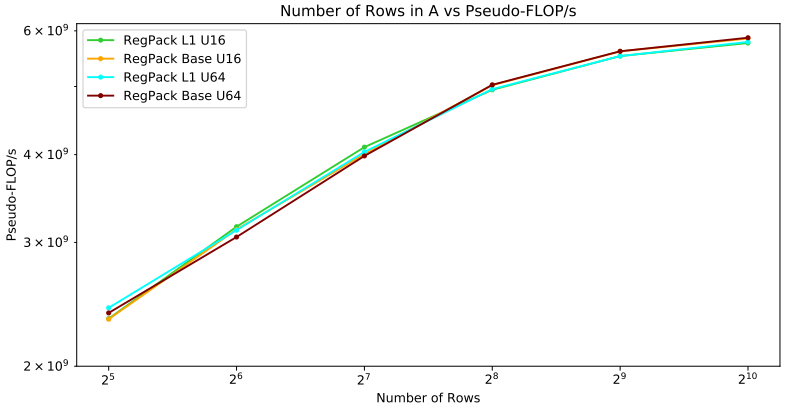
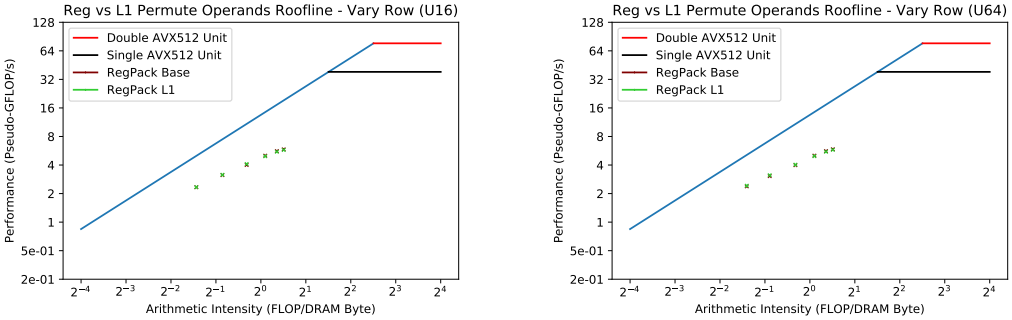


Figure 7.13: Register Packing L1 vs Base: Performance vs Number of Rows



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 7.14: Register Packing L1 vs Base: Roofline Plot - Vary Number of Rows

Vary Number of Columns

Similarly to the results seen in Chapter 6, Figure 7.15 shows that increasing the number of columns to 512 and beyond leads to a decrease in performance for both RP L1 and RP Base. This is due to the 512+ strides of B no longer fitting within the L1 data cache. However, around a 1.1x speedup is observed when using RP L1 for some of the kernels. The roofline plot in Figure 7.16 shows us that these kernels are further away from the memory bound. In these cases, the latency of loading the selector operands from L1 is masked by the latency of loading strides of B from L2. So the relatively small speedups can be attributed to micro-architecture specifics (possibly various OOO mechanisms) of Skylake-SP working better with the instruction sequence of RP L1 kernels. Again, investigating on other micro-architectures would help to confirm or reject that last conclusion.
For typical sizes of operator matrices, $NumCol \leq 128$, hypothesis **3** holds. However, the results show the hypothesis does not hold for large matrices when running on Skylake-SP.

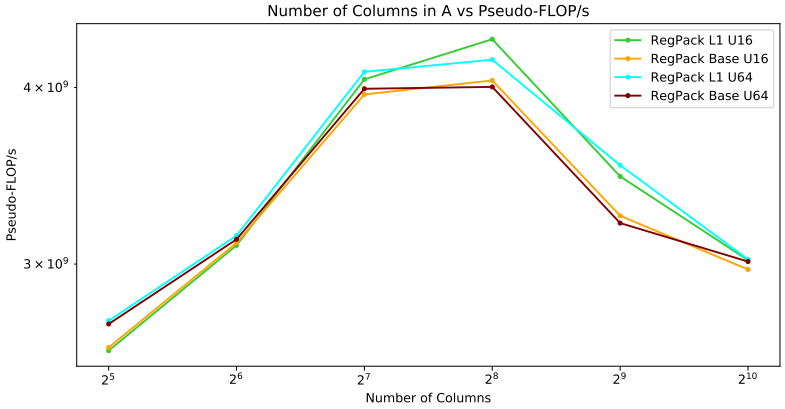
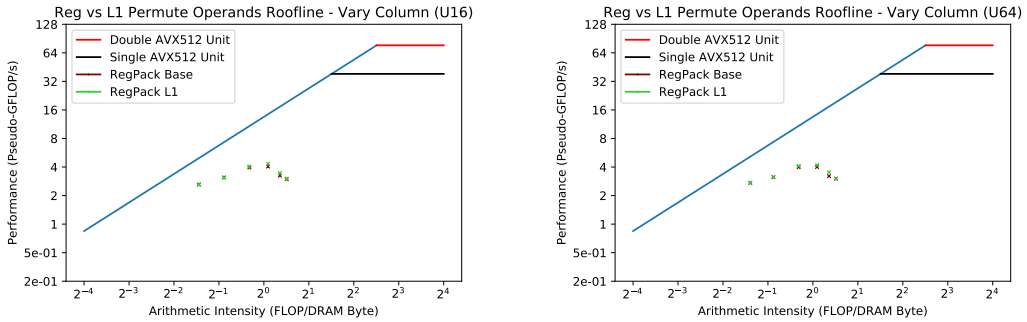


Figure 7.15: Register Packing L1 vs Base: Performance vs Number of Columns



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 7.16: Register Packing L1 vs Base: Roofline Plot - Vary Number of Columns

Vary Density

Figure 7.17 shows that there was no significant performance difference between RP L1 and RP Base as the density increased. At the higher densities, the current strides of B are reused more often. The additional step of loading selector operands from L1 is shown to not add to the critical path. The CPU is most likely being bound by the FMA operations. It appears that the OOO execution of the Skylake-SP core cannot squeeze out further instruction-level parallelism (ILP) with the RP L1 kernels (and RP Base). Figure 7.18 reveals that the kernels for denser matrices do not reach the roofline, suggesting they are not using the hardware as effectively as they could be. This led to slower than possible performance, which could have been masking the L1 selector operand loads. The results support hypothesis **3**, and show that loading the selector operands from L1 did not lead to performance differences.

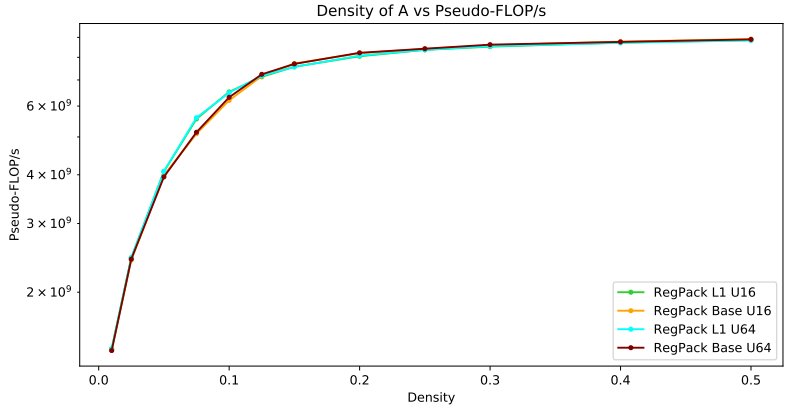
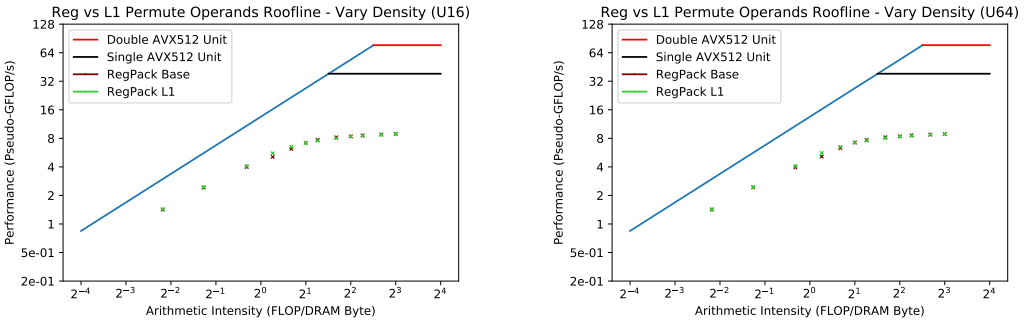


Figure 7.17: Register Packing L1 vs Base: Performance vs Density



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 7.18: Register Packing L1 vs Base: Roofline Plot - Vary Density

Vary Number of Unique Non-Zeros

The original aim of using L1 selector operands was to support more unique non-zeros. Figure 7.19 shows that RP L1 maintains a similar level of performance for an operator matrix with $U = 192$ as it does for $U \leq 176$. In comparison, RP Base defaults to the dense routine for $U = 192$. As only U is varied, we can conclude that RP L1 continues to provide the sparse routine level of performance for small sparse matrices (up to a size of $128x128$ and a density of around 0.05) that have a larger U.
The results do support hypothesis **3**, but we should note that the performance for RP L1 kernels are *consistently slightly* faster than RP Base, but the difference is not noticeable on larger scales, such as in Figure 7.20. Again, no concrete conclusions can be drawn to explain this without testing on more micro-architectures.

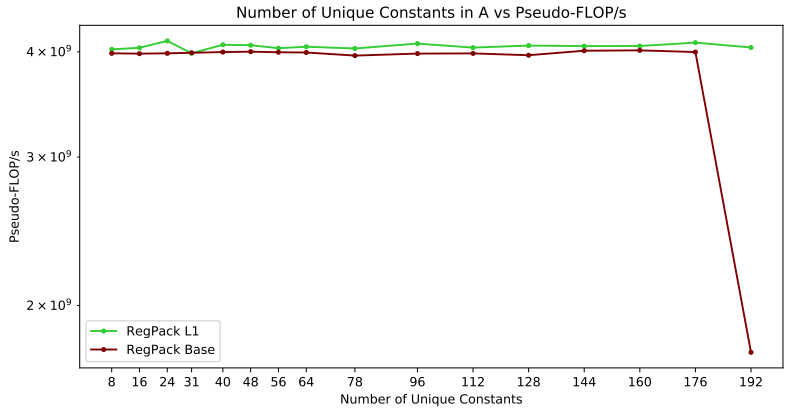


Figure 7.19: Register Packing L1 vs Base: Performance vs Number of Unique Non-Zeros

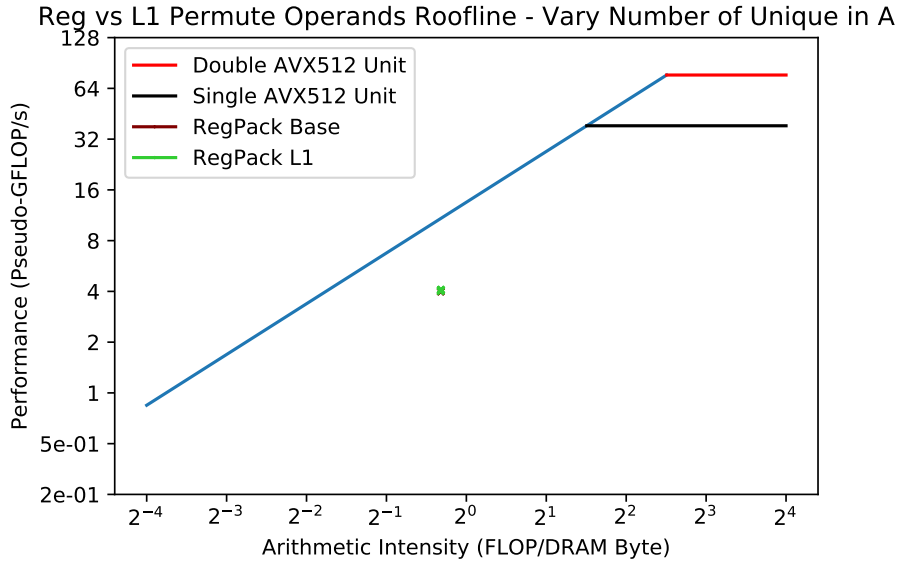


Figure 7.20: Register Packing L1 vs Base: Roofline Plot - Vary Number of Unique Non-Zeros

7.3 Summary

We evaluated RP L1 against RP Base on both the PyFR suite and synthetic suite. The results showed that hypothesis **3** does not strictly hold. RP L1 appears to be suited for small sparse matrices where the kernels are either memory bound or under utilise the hardware when not memory bound. In those cases any potential L1 selector operand load penalties would be hidden. However, the 'sparse' group of PyFR operator matrices suggested that for kernels which are *not* memory bound, will also not have a penalty for using L1 operands either. The 'dense' group of PyFR operator matrices revealed that RP Base could often have up to a 1.1x speedup over RP L1 for denser matrices, suggesting that hypothesis **3** only holds for small sparse matrices.
Evaluation on even larger and denser PyFR operator matrices, (that RP Base could not support), showed that RP L1 was not as performant as the optimised dense routine in LIBXSMM for those matrices.
We observed a few instances were RP L1 would have up to a 1.1x speedup over RP base, where the reason for the speedup was not obvious. It is possible that the Skylake-SP micro-architecture could better execute (via OOO/ILP) the instruction sequence of RP L1 kernels.
Ultimately, for the target use case of this LIBXSMM sparse routine - small sparse operator matrices - RP L1 is a solution that provides support for an increased U (240 DP / 480 SP) in the operator matrix without adding a performance penalty.

Chapter 8

Multiple Accumulations

When using register packing to store the operator matrix in the vector register file, if the number of unique non-zeros is less than the maximum number supported, then there are free, unused vector registers. In this chapter, we cover two methods that make use of the unused registers. After outlining how they work, we will investigate if runtime broadcasting impacts performance when using either method. Then we will evaluate the methods using the benchmark, finally coming to a conclusion if either (or both) lead to an increase in performance.

8.1 Solutions

8.1.1 N Blocking: Operating on multiple mini-chunks of B

In 2.4.3 we describe how the kernels are passed 'chunks' of **B**, where each column of strides of **B** are called 'mini-chunks'. In the reference sparse-dense MM routine, only one 'mini-chunk' is operated on, used when calculating the strides of **C** (corresponding to the rows of **A**). This can be referred to as N Blocking, where $N = 1$. The advantage of using $N = 1$, is that each 'mini-chunk' / block of **B** can be larger and still fit within L1 cache.
However, for small operator matrices, where the number of columns is less than 256(171), the L1 data cache in a Skylake-SP CPU core can fit 2(3) 'mini-chunks' of B. This enables the kernel to quickly (L1 cache hit) access different parts of **B** to calculate multiple strides of **C**, by making use of data parallelism. The CPU hardware will itself carry out the instruction level parallelism.
Figure 8.1 demonstrates the order that a kernel would issue FMAs using $N = 2$ blocking. For a row in **A**, after broadcasting the unique value, issue the first (#1) FMA with the corresponding stride of **B** from the first mini-chunk, accumulating into the first stride of **C**. Then, issue the second (#2) FMA with the corresponding stride of **B** from the second mini-chunk, accumulating into the second stride of **C**. This is repeated for each unique value in the row of **A** to calculate the $N = 2$ strides of **C**. Some (if not most) of the strides of **B** from both mini-chunks will be in L1 data cache, to use with the next row of **A**, to calculate the next $N = 2$ strides of **C**.

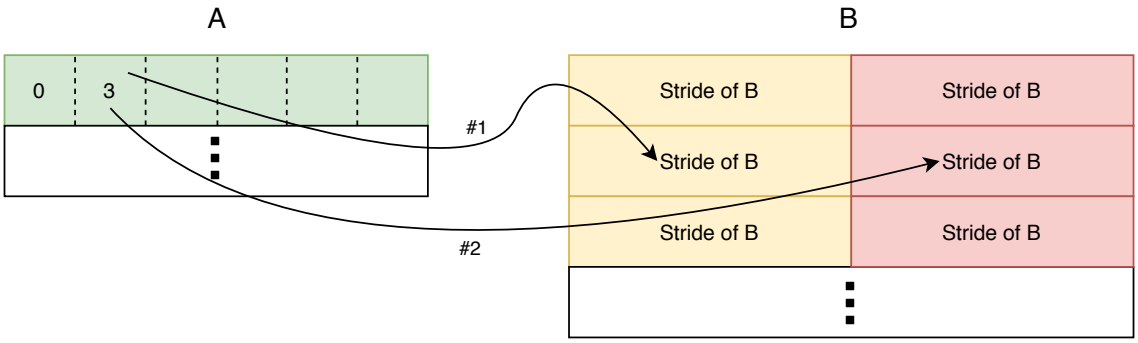


Figure 8.1: Multiple Accumulation - N Blocking Visualised

Table 8.1 shows that increasing N by 1, reduces the number of free registers that can be used to pack **A** by 1. This is due to each additional accumulation of a stride of **C** requiring an additional vector register to hold the (partial) result. Going from $N = 1$ to $N = 3$, the number of unique non-zeros that can be packed decreases from 176 to 160 for DP.

N	Free Reg (DP/SP)	U (SP)	U (DP)
1	22(14)	224	176
2	21(13)	208	168
3	20(12)	192	160

Table 8.1: Comparing the Number of Unique Non-Zeros (U) that can be packed for different N Blocking

8.1.2 M Blocking: Interleaving rows of A

Unlike N blocking, this method operates on the same 'mini-chunk' of **B**. Different rows of **A** are accessed in an 'interleaved' fashion, to work on accumulating multiple strides of **C**. Figure 8.2 shows an example of this 'M Blocking', where $M = 2$.
Each row of **A** has its own dedicated register to hold the broadcasted value. The two rows are iterated together, with FMAs (and preceding broadcasts) being issued for whichever row has a non-zero value in the current column. For this example, the first FMA (#1) is issued due to the second row having the value 3 in the first column. The first row has a zero value, and so no FMA is issued. The next column, it is the opposite case, with the second FMA (#2) issued due to the first row. On the third column, both rows have non-zeros, and so two FMAs (#3 and #4) are issued respectively.
This is repeated until all of rows of **A** have been dealt with. If the number of rows is not exactly divisible by M, the final remaining group of rows ($< M$) are interleaved.

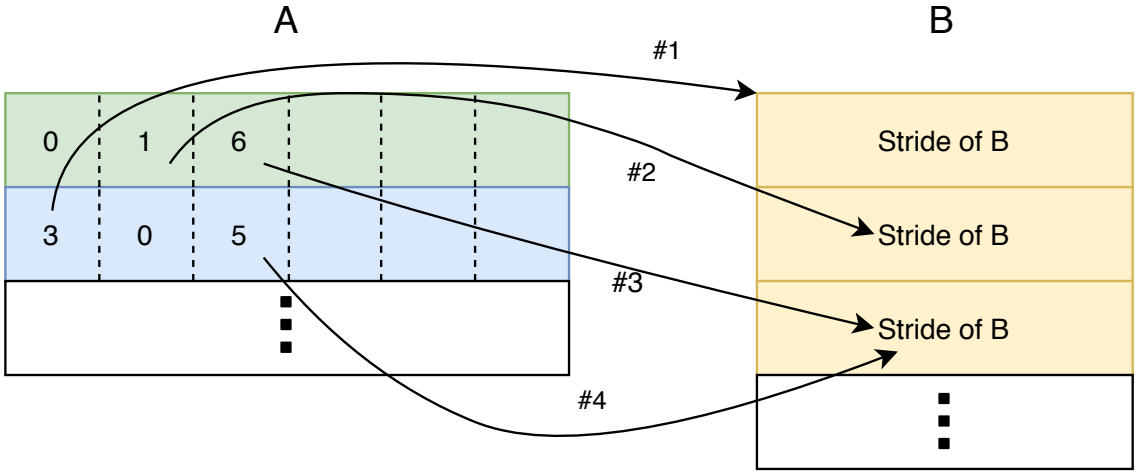


Figure 8.2: Multiple Accumulation - M Blocking Visualised

Table 8.2 shows that increasing M by 1, reduces the number of free registers that can be used to pack **A** by 2. This is due to each additional accumulation of a stride of **C** requiring both; a vector register to hold the broadcasted value from the corresponding row and an additional vector register to hold the (partial) result. Going from $M = 1$ to $M = 3$, the number of unique non-zeros that can be packed decreases from 176 to 144 for DP.

M	Free Reg (DP/SP)	U (SP)	U (DP)
1	22(14)	224	176
2	20(12)	192	160
3	18(10)	160	144

Table 8.2: Comparing the Number of Unique Non-Zeros (U) that can be packed for different M Blocking

8.2 Investigating the effect of runtime broadcasting

The first hypothesis (**1**) in Chapter 6 that was evaluated to be true, stated that 'the runtime broadcasting should not add to the critical path of execution'. When issuing more FMAs to carry out multiple accumulations, this cannot be assumed to still hold. So, we will evaluate both N and M blocking on a subset of the operator matrices from the benchmark, when runtime broadcasting is and isn't used. The subset is formed by testing only on operator matrices with a number of unique non-zeros ≤ 26. This allows the values to be pre-broadcast into the register file whilst using additional blocking. The proposed hypothesis to be evaluated is as follows.

Hypothesis

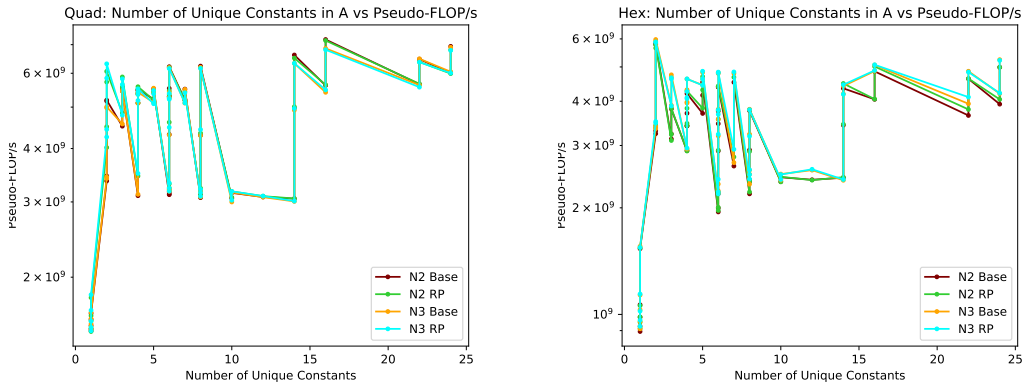
4: Kernels carrying out multiple accumulations in 'parallel' and using runtime broadcasting should not perform slower than kernels carrying out multiple accumulations in 'parallel' and using pre-broadcast values. The runtime broadcasting should not add the critical path of execution, even when more FMAs are being issued, as in the case of multiple accumulations.

8.2.1 N Blocking

Using the subset of operator matrices from the benchmark, we show the performance for both $N = 2$ and $N = 3$ blocking, with and without runtime broadcasting. In the following figures, NX *Base* represents $N = X$ blocking *without* runtime broadcasting, and NX *RP* represents $N = X$ blocking *with* runtime broadcasting.

Sparse Operator Matrices

Figure 8.3 shows that for most of the sparse PyFR operator matrices, there is no difference in performance when using and not using runtime broadcasting, for both $N = 2$ and $N = 3$. However, there are a few kernels where using runtime broadcasting can lead to performance gains of up to 3%, which isn't a large enough and common enough of a difference to delve further into. The results strongly support hypothesis **4**.



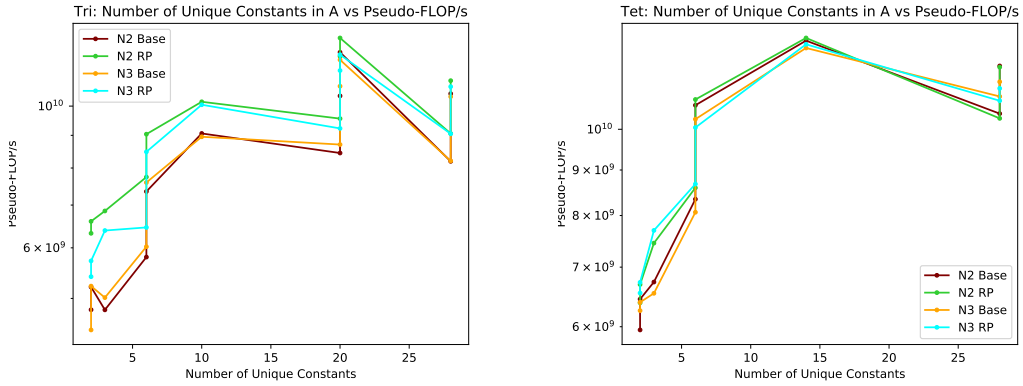
(a) Quadrilaterals

(b) Hexahedra

Figure 8.3: Register Packing impact on N Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)

Dense Operator Matrices

Figure 8.4 shows that for the dense PyFR operator matrices, the kernels using runtime broadcasting either perform around the same or faster than kernels using pre-broadcast values. In the case of triangular meshes and $N = 2$ blocking, speedups around 1.3x can be observed when using runtime broadcasting. We can attribute this to the Skylake-SP micro-architecture favouring runtime broadcasting, as seen in Chapter 6. Again, the results support hypothesis **4**.



(a) Triangles

(b) Tetrahedra

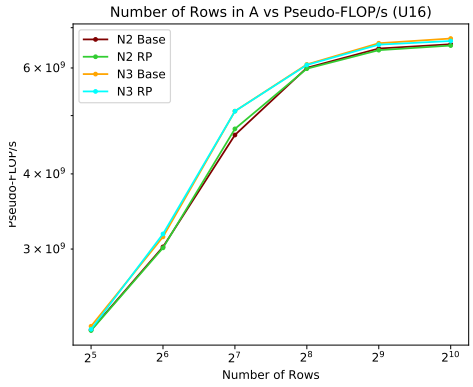
Figure 8.4: Register Packing impact on N Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)

Synthetic Operator Matrices

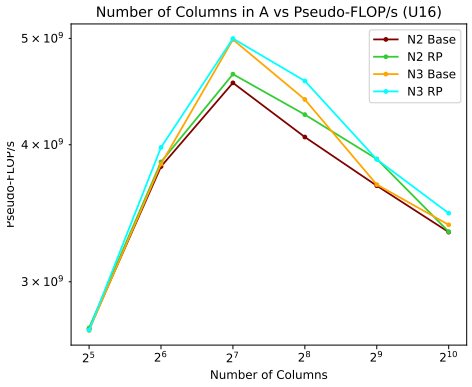
The results from the subset of synthetic operator matrices are shown in Figure 8.5. All of the data shows that the kernels with runtime broadcasting perform either the same or faster than kernels using pre-broadcast values, for both $N = 2$ and $N = 3$.

Summary

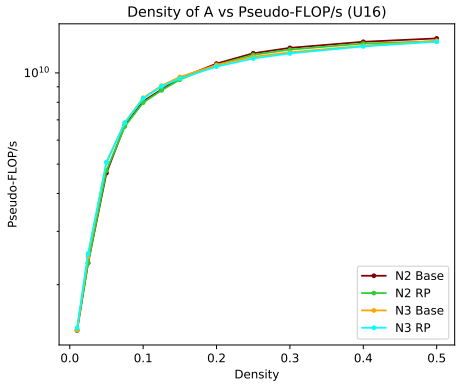
Results from the subset of both PyFR and synthetic operator matrices show that hypothesis **4** holds for additional N blocking.



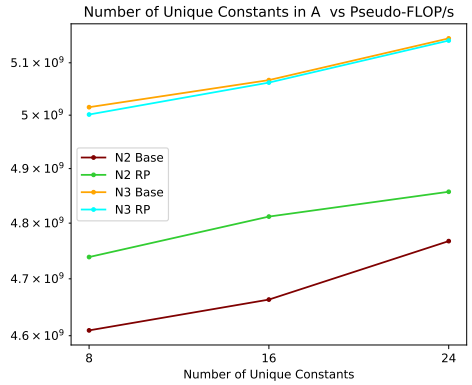
(a) Vary Number of Rows in A



(b) Vary Number of Columns in A



(c) Vary Density of A



(d) Vary Number of Unique Non-Zeros in A

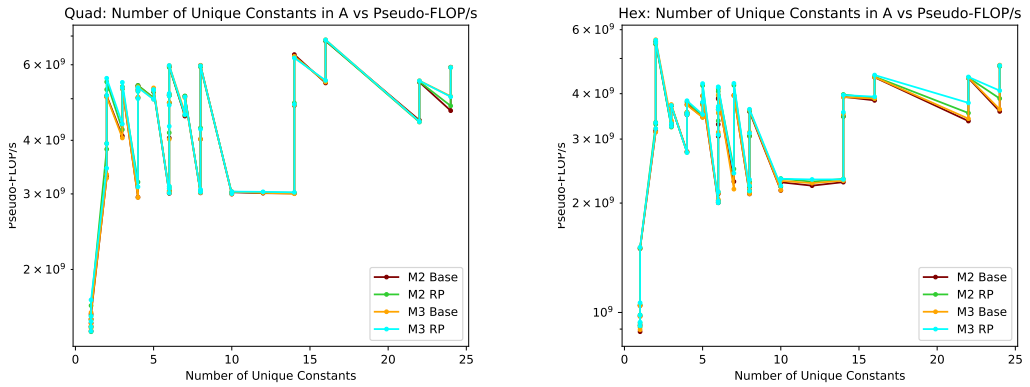
Figure 8.5: Register Packing impact on N Blocking - Synthetic Operator Matrices

8.2.2 M Blocking

Using the subset of operator matrices from the benchmark, we show the performance for both $M = 2$ and $M = 3$ blocking, with and without runtime broadcasting. In the following figures, MX *Base* represents $M = X$ blocking *without* runtime broadcasting, and MX *RP* represents $M = X$ blocking *with* runtime broadcasting.

Sparse Operator Matrices

Figure 8.6 shows that for the sparse PyFR operator matrices, the kernels using runtime broadcasting either perform around the same, or slightly (1.1x) faster than kernels using pre-broadcast values. Again, this difference can be explained by the Skylake-SP micro-architecture favouring runtime broadcasting. Hypothesis **4** is supported by the results.



(a) Quadrilaterals

(b) Hexahedra

Figure 8.6: Register Packing impact on M Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)

Dense Operator Matrices

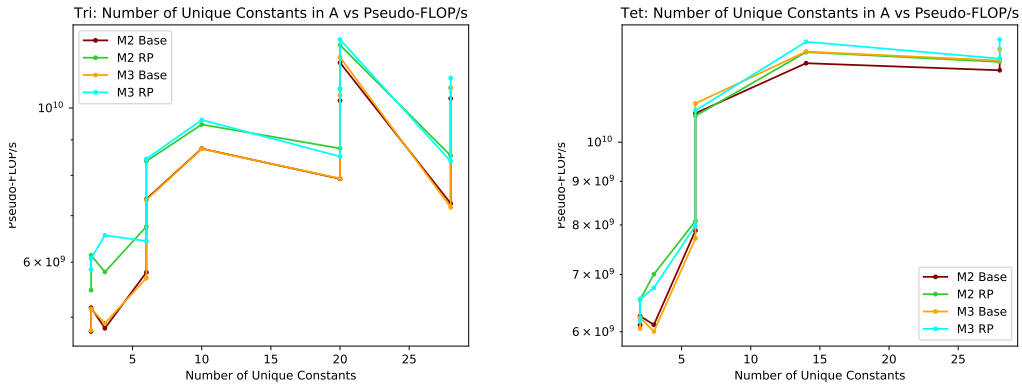
Figure 8.7 shows that for the dense PyFR operator matrices, the kernels using runtime broadcasting either perform around the same or up to around 1.3x faster than kernels using pre-broadcast values. Importantly, no kernels using runtime broadcasting perform noticeably slower when using runtime broadcasting. The results show that Hypothesis **4** holds for the dense PyFR matrices as well.

Synthetic Operator Matrices

The results from the subset of synthetic operator matrices are shown in Figure 8.8. All of the data shows that the kernels with runtime broadcasting perform either the same or faster than kernels using pre-broadcast values, for both $M = 2$ and $M = 3$.

Summary

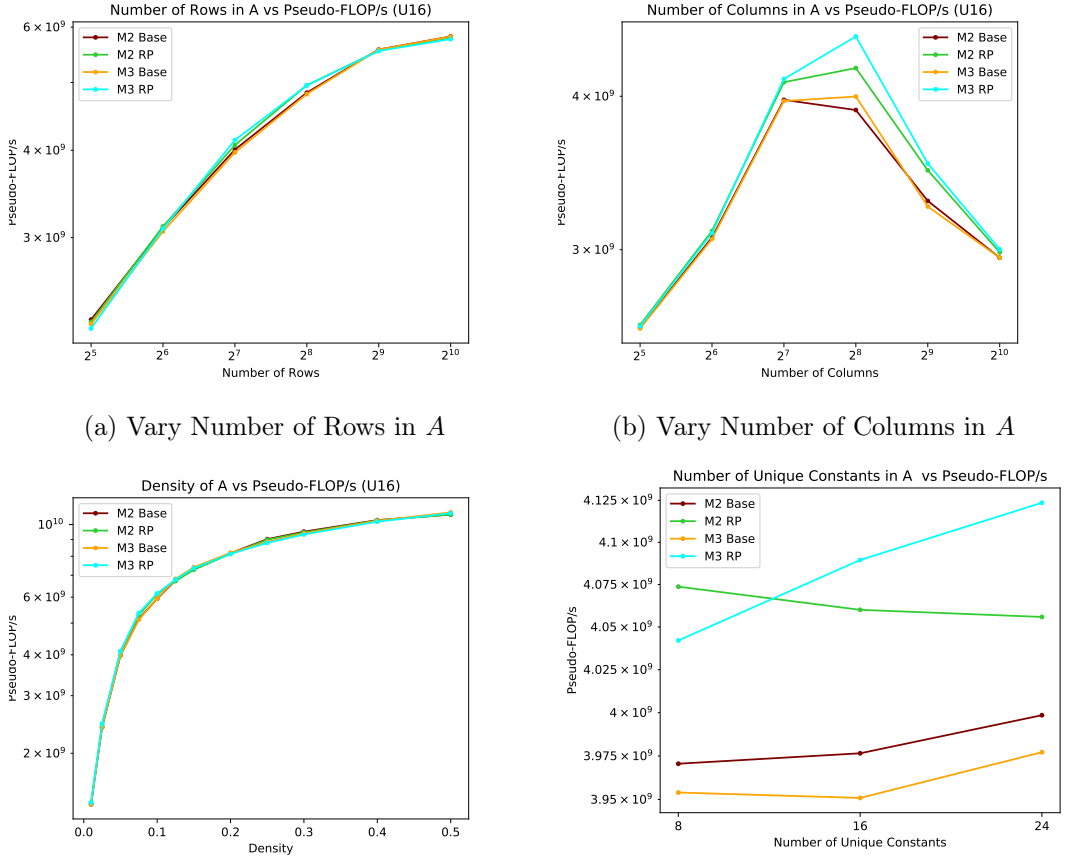
Results from the subset of both PyFR and synthetic operator matrices show that hypothesis **4** holds for M blocking.



(a) Triangles

(b) Tetrahedra

Figure 8.7: Register Packing impact on M Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)



(a) Vary Number of Rows in A

(b) Vary Number of Columns in A

(c) Vary Density of A

(d) Vary Number of Unique Non-Zeros in A

Figure 8.8: Register Packing impact on M Blocking - Synthetic Operator Matrices

8.2.3 Summary

We evaluated both N and M blocking on a subset of the operator matrices from the benchmark, with and without the kernels using runtime broadcasting. Looking at the results, we showed that hypothesis **4** is true for both N and M blocking - runtime broadcasting does not add to the critical path of execution, even when carrying out multiple accumulations.

8.3 N Blocking Evaluation

In this section we will evaluate $N = 2$ and $N = 3$ blocking being used in conjunction with register packing. We will compare performance against the reference LIBXSMM (maroon lines) as well as the base register packing (RP Base) from Chapter 6 (orange lines). Throughout the evaluation, we will discuss the following hypothesis.

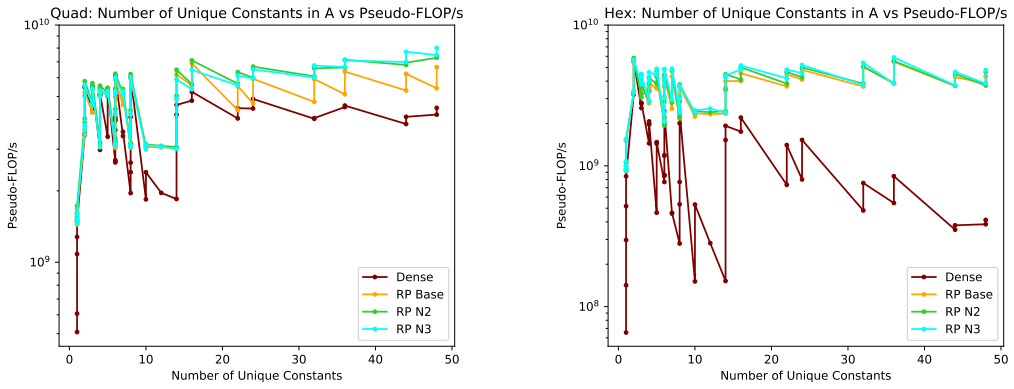
Hypothesis

5: Using additional N blocking with register packing leads to faster performance over register packing without additional N blocking.

8.3.1 PyFR Suite

Sparse Operator Matrices

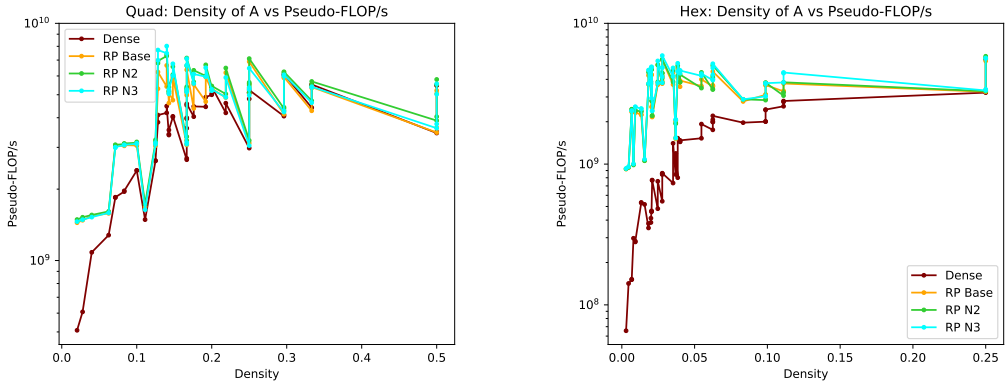
When plotting performance against the number of unique non-zeros or the density, as in Figures 8.9 and 8.10, N blocking is shown to lead to faster performance for some of the sparse PyFR operator matrices. For hexahedra, the performance is either the same or marginally faster. For quadrilaterals, Figure 8.9a shows speedups of up to 1.1x when using N blocking. When plotting against the number of columns, as in Figure 8.11, it is more clear that N blocking performed faster when there are around 50 to 100 columns at a density between 0.1 and 0.2 for quadrilaterals. The hexahedra operator matrices do not have examples that fit those features, and so we do not see similar speedups.
Interestingly, $N = 3$ appears to give the same performance as $N = 2$. Figure 8.12 shows that this is most likely due to the kernels saturating memory bandwidth, as they are memory bound. This implies that the execution of FMAs is not the limiting factor. The results do not support hypothesis **5**, as additional N blocking does not *always* lead to faster performance than without it being used.



(a) Quadrilaterals

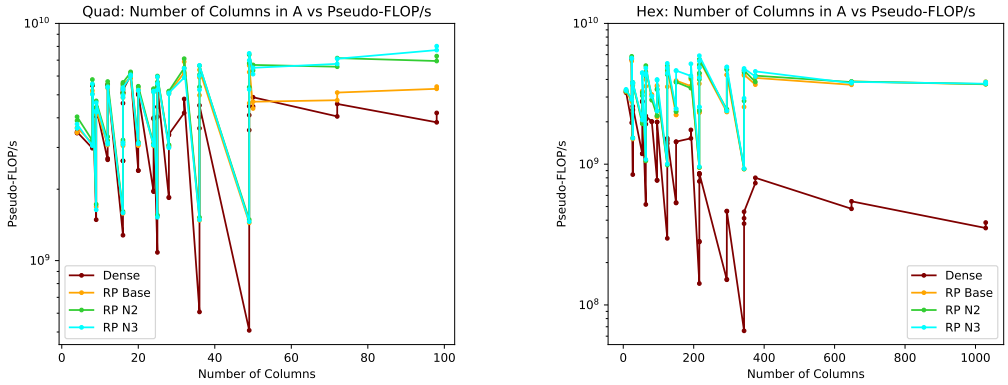
(b) Hexahedra

Figure 8.9: N Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)



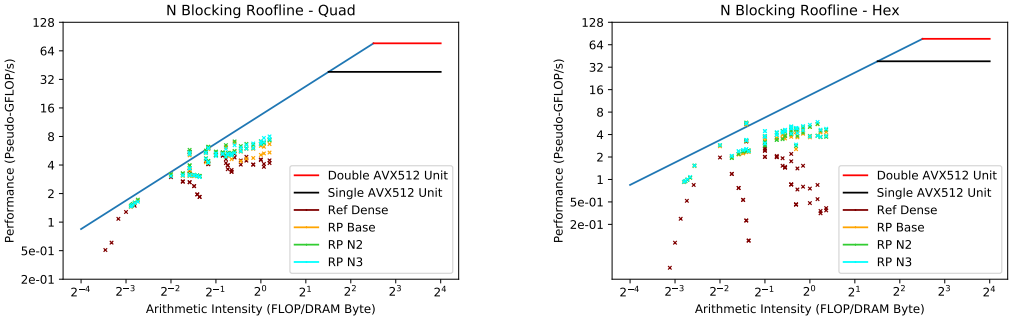
(a) Quadrilaterals (b) Hexahedra

Figure 8.10: N Blocking - PyFR Sparse Examples (Density)



(a) Quadrilaterals (b) Hexahedra

Figure 8.11: N Blocking - PyFR Sparse Examples (Number of Columns)

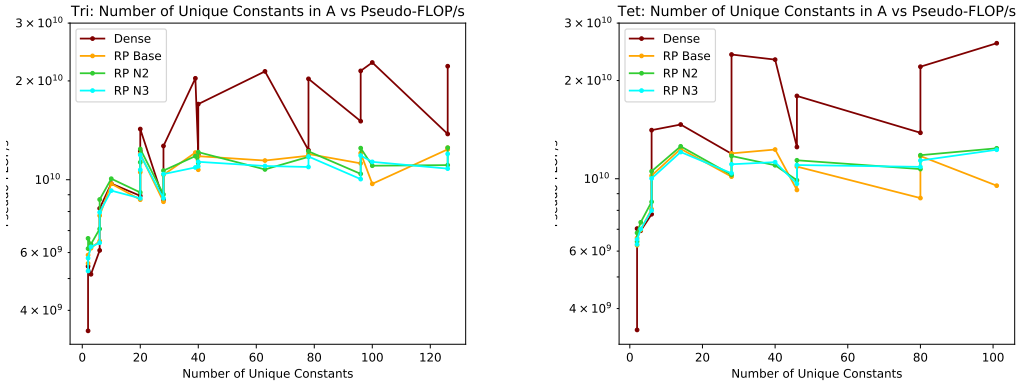


(a) Quadrilaterals (b) Hexahedra

Figure 8.12: N Blocking - PyFR Sparse Examples (Roofline Plots)

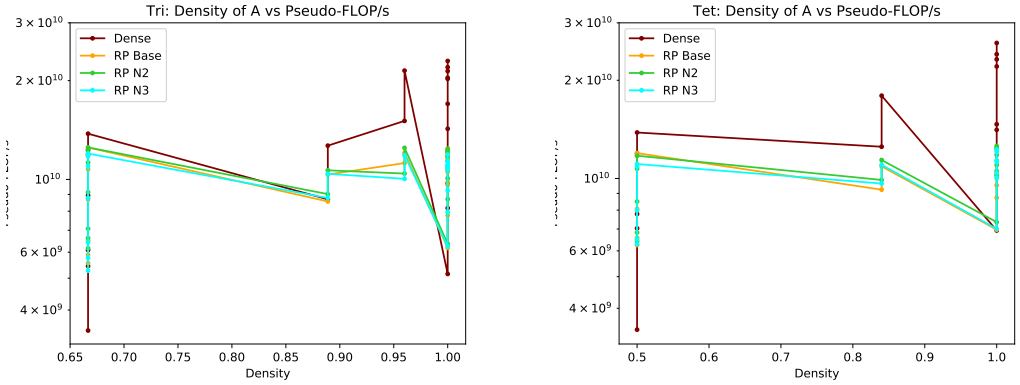
Dense Operator Matrices

Additional N blocking was not as effective for the set of dense PyFR operator matrices, as it was for the sparse examples. Figures 8.13a and 8.14a shows that for the majority of kernels for triangles, performance was the same as when not using additional N blocking. In the case of tetrahedra, there were a few cases with around a 1.2x speedup when using additional N blocking, but the performance was the same for most kernels. Figure 8.15b reveals that the speedups for hexahedra kernels where when the kernels were not near the memory bound. It is possible that the additional FMA instructions being issued lead to better utilisation of the execution hardware in the CPU core. Figure 8.15a shows that the kernels without additional N blocking for triangles, were already achieving performance levels for respective AIs that tetrahedra achieved once using N blocking of either 2 or 3. Overall, the results on the dense PyFR operator matrices disprove hypothesis **5**.



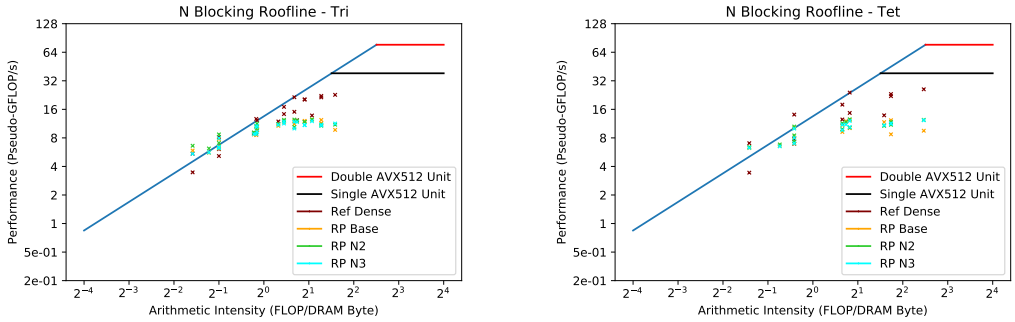
(a) Triangles (b) Tetrahedra

Figure 8.13: N Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)



(a) Triangles (b) Tetrahedra

Figure 8.14: N Blocking - PyFR Dense Examples (Density)



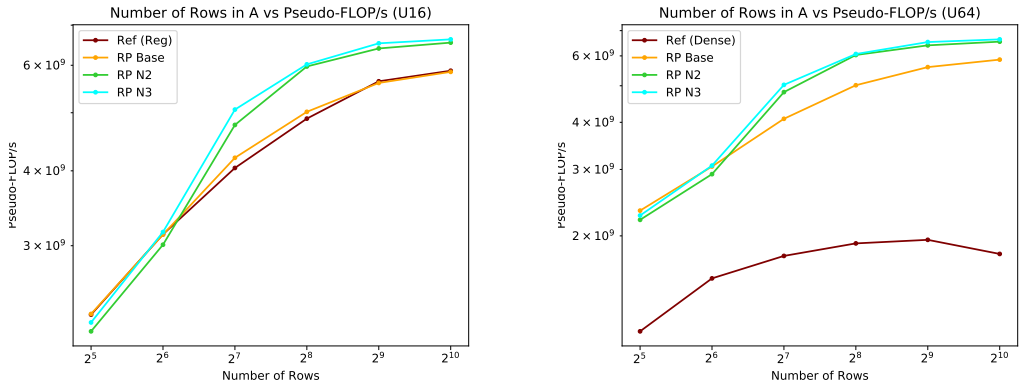
(a) Triangles (b) Tetrahedra

Figure 8.15: N Blocking - PyFR Dense Examples (Roofline Plots)

8.3.2 Synthetic Suite

Vary Number of Rows

Figure 8.16 shows that with 128 rows or more in **A**, additional N blocking leads to up to 1.2x speedups over $N = 1$. $N = 3$ had similar performance to $N = 1$ for fewer rows than 128, whereas $N = 2$ performed slightly slower. Figure 8.17 highlights that from 128 rows and beyond, the arithmetic intensity increases. This is due to the strides of B being reused more often as the number of rows increases. It appears that, up to 64 rows, additional N blocking does not increase performance. In this case, the strides are not reused often enough to greatly benefit from having them in L1 data cache. When they are reused more often, performance can increase by up to 1.2x. We could expect that at a greater density, N blocking would start to provide speedups with an even fewer amount of rows in **A**. The results show that hypothesis **5** does not hold.

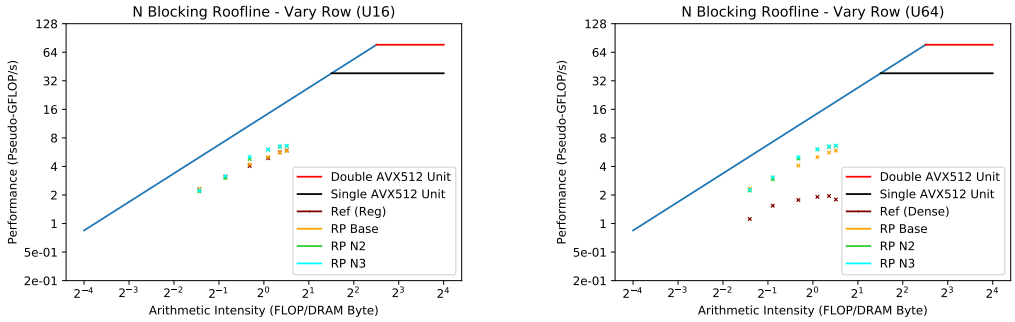


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.16: N Blocking: Performance vs Number of Rows

Vary Number of Columns

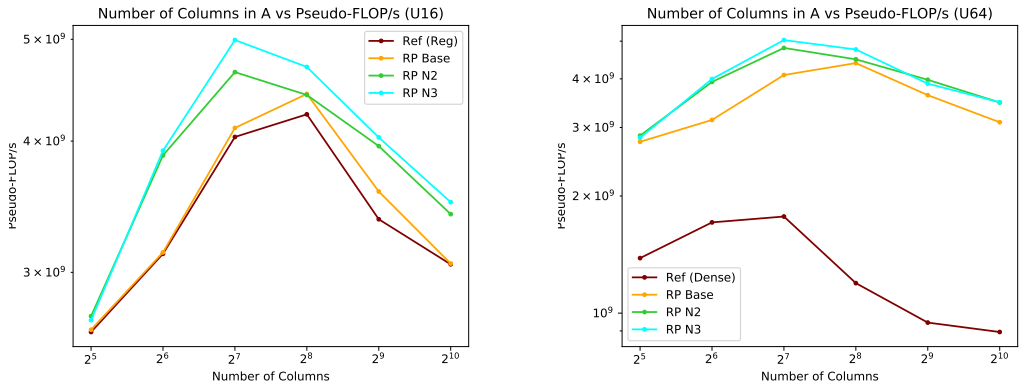
Figure 8.18 shows that additional N blocking provides an increase in performance when the number of columns is ≥ 64. Increasing the number of columns, for a given density, increases the total number of strides of B that need to be loaded from memory. Additional N blocking increases the data parallelism by loading more strides in parallel, which can help to reduce the total time spent waiting for data to load into the L1 cache.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.17: N Blocking: Roofline Plot - Vary Number of Rows

Additional N blocking has slower performance at 256 columns than at 128 columns. At 256 columns, the L1 data cache gets filled and so spilling to L2 cache leads to slower performance. Whereas for RP Base, performance increased at this step as the L1 data cache does not get filled until 512 columns.
However, performance with additional N blocking was *not slower* than RP Base with 256+ columns. In the case of $N = 3$, performance was continuously faster as the number of columns increased further. This suggests that the benefits to performance of using additional N blocking when data is stored in L1 cache, translates to performance improvements when data is stored in L2 cache. Figure 8.19 illustrates the impact on performance when accessing L2. The kernels are now impacted by L2 hit times. Unlike previous results, the experiment on varying columns shows support for hypothesis **5**.

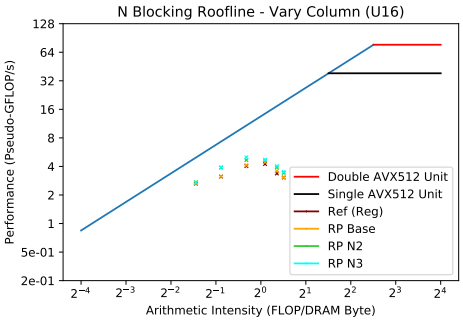


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

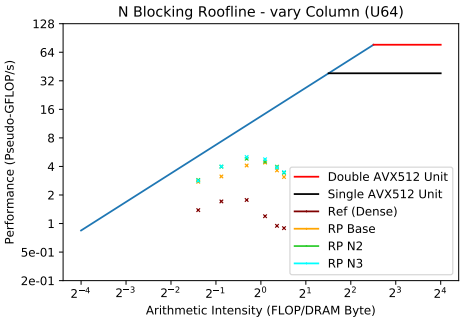
Figure 8.18: N Blocking: Performance vs Number of Columns

Vary Density

As the density increased beyond 0.05 (density in the above synthetic experiments), the performance speedup from using additional N blocking increased, as shown in Figure 8.20. However, from a density of 0.4 and above, the dense routine from LIBXSMM was faster by up to 1.5x. The strides of **B** are reused more often as a direct result of the density increasing. Figure 8.21 reveals that additional N blocking helps the sparse routine to achieve performance closer to the peak pseudo-FLOP/s for a given AI. This suggests that the kernels using it are making better use of the available hardware.

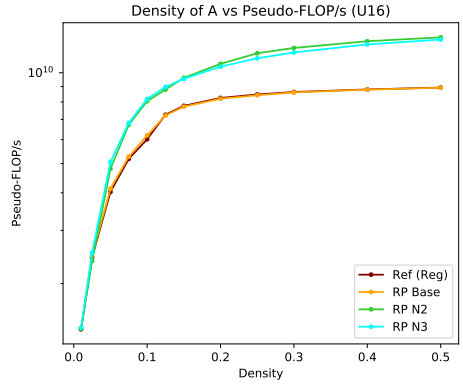


(a) 16 Unique Non-Zeros

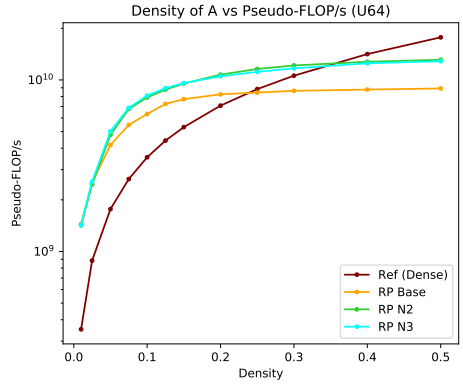


(b) 64 Unique Non-Zeros

Figure 8.19: N Blocking: Roofline Plot - Vary Number of Columns

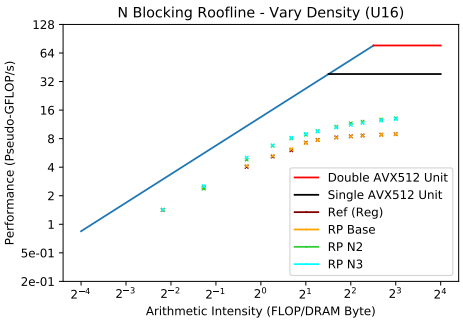


(a) 16 Unique Non-Zeros

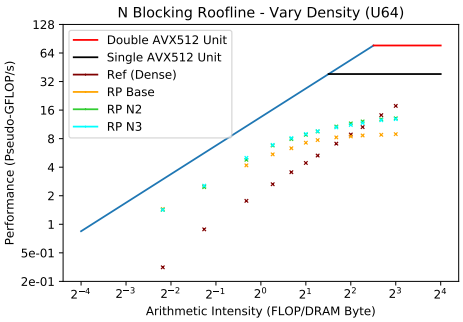


(b) 64 Unique Non-Zeros

Figure 8.20: N Blocking: Performance vs Density



(a) 16 Unique Non-Zeros



(b) 64 Unique Non-Zeros

Figure 8.21: N Blocking: Roofline Plot - Vary Density

Vary Number of Unique Non-Zeros

Since we showed that the runtime broadcasting does not impact the performance when using additional N blocking, it is not surprising to see relatively flat performance in Figure 8.22, as the number of unique constants is increased. We can however more clearly see the around 1.25x speedup $N = 3$ blocking provides over RP Base at the configuration of size and density being tested. The additional N blocking strategies support fewer unique non-zeros, and so default to the dense routine sooner. Hypothesis **5** is supported by the results.

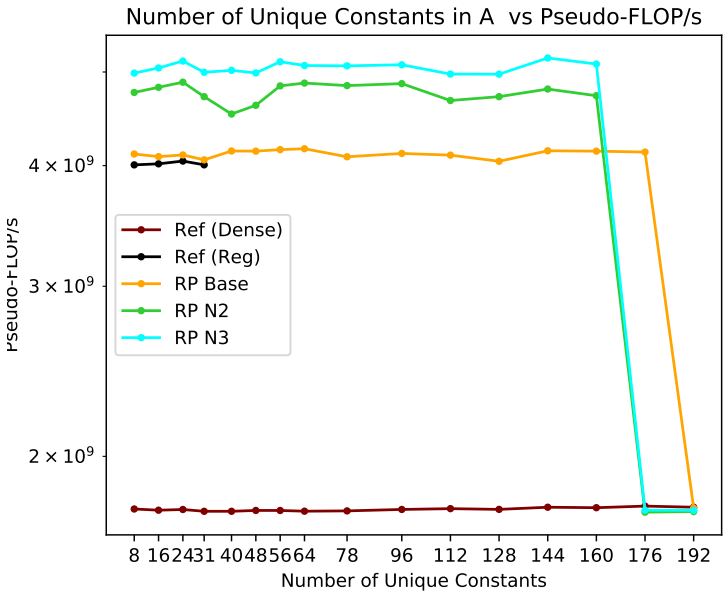


Figure 8.22: N Blocking: Performance vs Number of Unique Non-Zeros

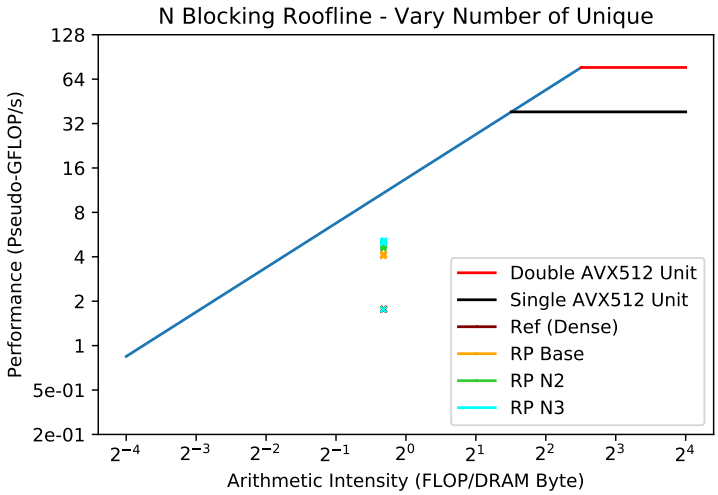


Figure 8.23: N Blocking: Roofline Plot - Vary Number of Unique Non-Zeros

8.3.3 Summary

We evaluated using additional N blocking with the register packing strategy on both the PyFR and synthetic operator matrices. Specifically, we tested $N = 2$ and $N = 3$. Through the evaluation, we conclude that hypothesis **5** is false, and that additional N blocking does not *always* lead to faster performance over RP Base. However, through evaluation on the synthetic suite, we uncovered that additional N blocking can be very advantageous for slightly less sparse (but density still ≤ 0.5) matrices and for matrices that have a slightly higher number of rows and/or columns.

8.4 M Blocking Evaluation

In this section we will evaluate $M = 2$ and $M = 3$ blocking being used in conjunction with register packing. We will compare performance against the reference LIBXSMM (maroon lines) as well as the base register packing (RP Base) from Chapter 6 (orange lines). Throughout the evaluation, we will discuss the following hypothesis.

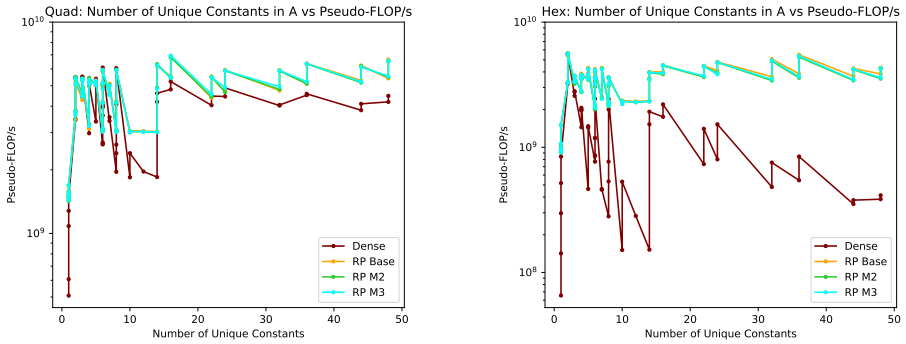
Hypothesis

6: Using additional M blocking with register packing leads to faster performance over register packing without additional M blocking.

8.4.1 PyFR Suite

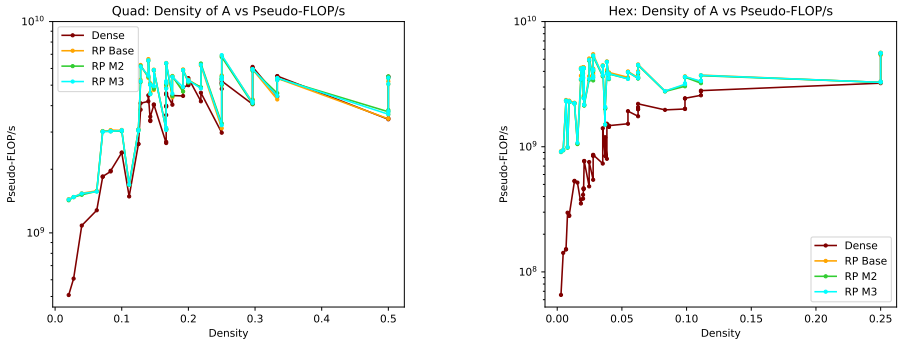
Sparse Operator Matrices

Both Figures 8.24 and 8.25 reveal that additional M blocking leads to the same performance as RP Base for sparse PyFR matrices. Figure 8.26 shows that these kernels are primarily memory bound. Smaller and sparse matrices have a lower AI, and so attempts to increase execution throughout will be hindered by memory bandwidth. Although performance did not decrease, the results show that hypothesis **6** is not true.



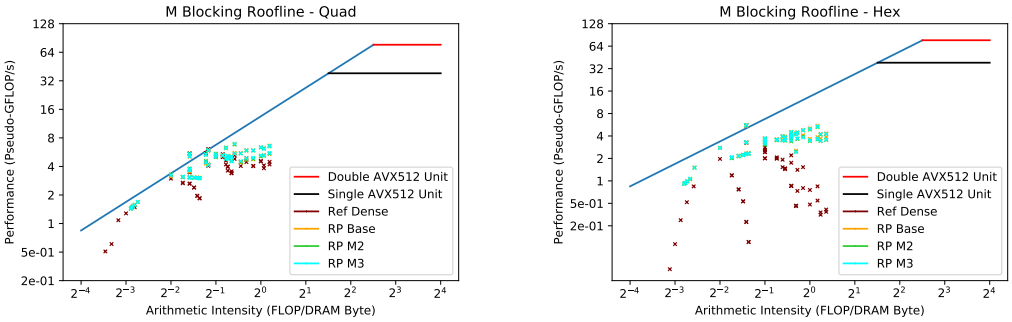
(a) Quadrilaterals (b) Hexahedra

Figure 8.24: M Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)



(a) Quadrilaterals (b) Hexahedra

Figure 8.25: M Blocking - PyFR Sparse Examples (Density)

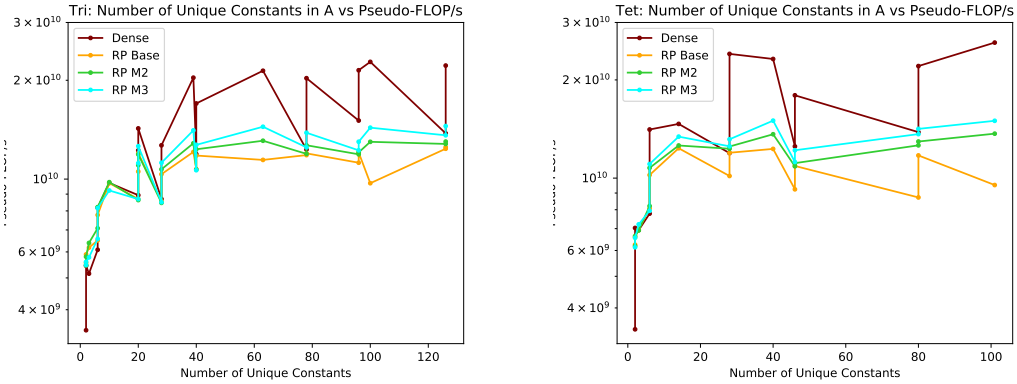


(a) Quadrilaterals (b) Hexahedra

Figure 8.26: M Blocking - PyFR Sparse Examples (Roofline Plots)

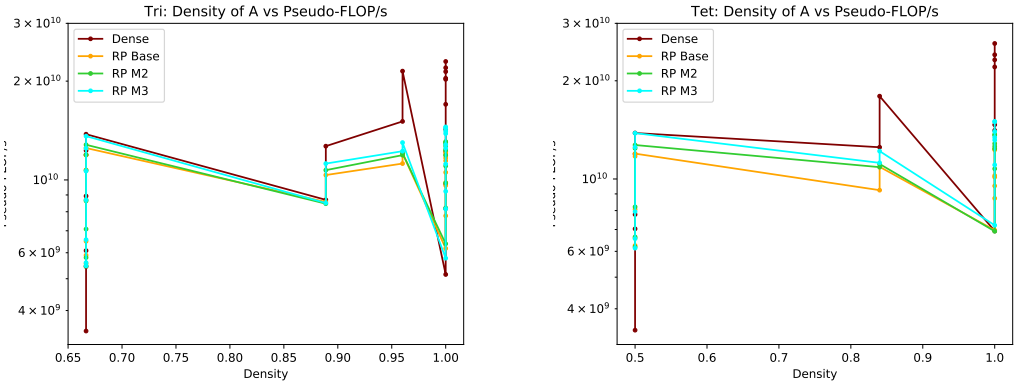
Dense Operator Matrices

Figure 8.27 shows that additional M blocking provided up to 1.5x speedup over RP Base in the case of $M = 3$. Generally, $M = 3$ performed faster than $M = 2$. Figure 8.28 does not show a clear trend in speedups when the density changes. However, Figure 8.29a shows that as the number of columns increases, the additional M blocking provides speedups. Larger and denser matrices have a higher AI and so the kernels for them will be less likely to be memory bound. Figure 8.30 shows that the speedups occur at higher AIs where there is room to make better use of the hardware by issuing more FMAs. However, not all kernels benefited, as some dense PyFR operator matrices are still memory bound. Thus we cannot say hypothesis **6** is fully supported by the results. We should note that the dense routine still provided faster performance than any of the kernels from the various RP routines.



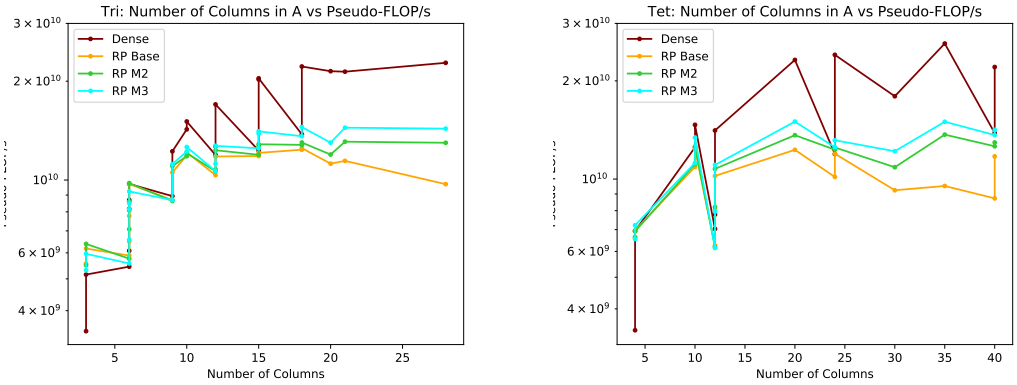
(a) Triangles (b) Tetrahedra

Figure 8.27: M Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)



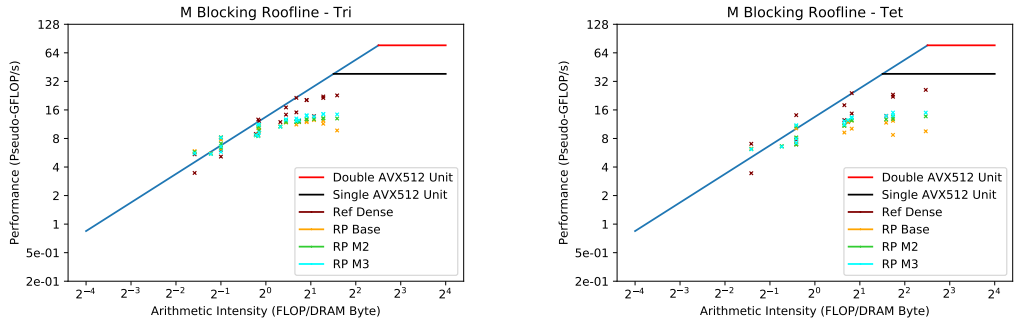
(a) Triangles (b) Tetrahedra

Figure 8.28: M Blocking - PyFR Dense Examples (Density)



(a) Triangles (b) Tetrahedra

Figure 8.29: M Blocking - PyFR Dense Examples (Number of Columns)



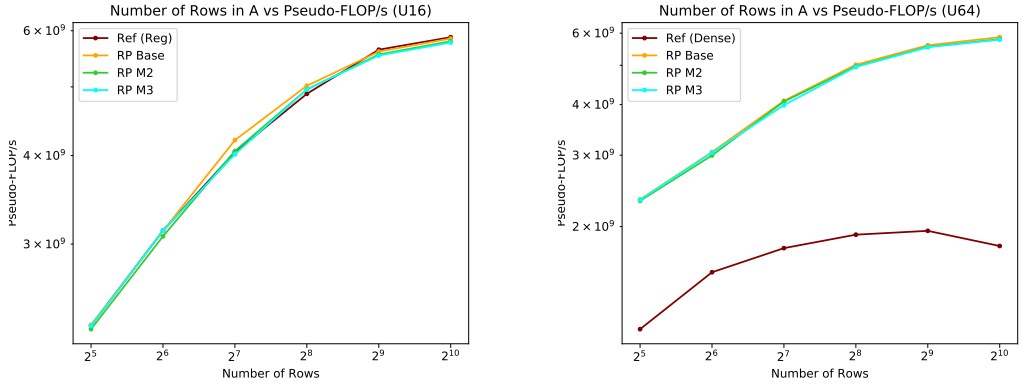
(a) Triangles (b) Tetrahedra

Figure 8.30: M Blocking - PyFR Dense Examples (Roofline Plots)

8.4.2 Synthetic Suite

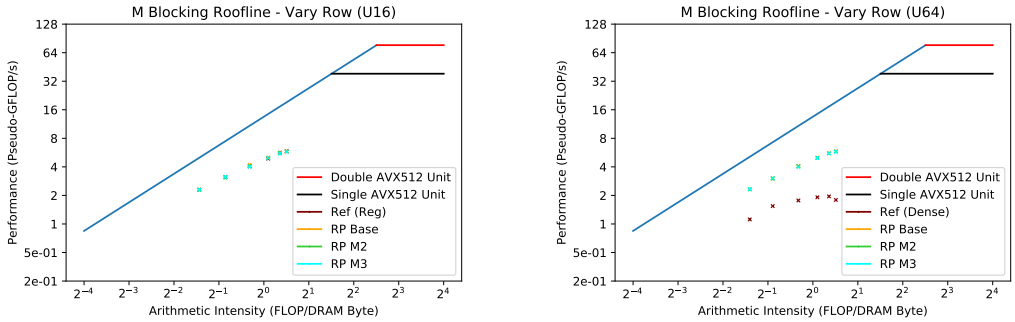
Vary Number of Rows

Figure 8.31 shows that as the number of rows varied, there was no noticeable performance difference between RP Base and RP with additional M blocking. Figure 8.32 shows that these kernels were not fully memory bound. So it was surprising to see that issuing additional FMAs did not lead to an increase in performance. One explanation for this is that the density of these matrices, 0.05, is too small to see benefits from additional M blocking. For low densities, each additional row is likely to access strides of **B** *not* already loaded into L1 data cache, and so the memory bandwidth limits the performance. Hypothesis **6** is disproved by the experiment.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.31: M Blocking: Performance vs Number of Rows

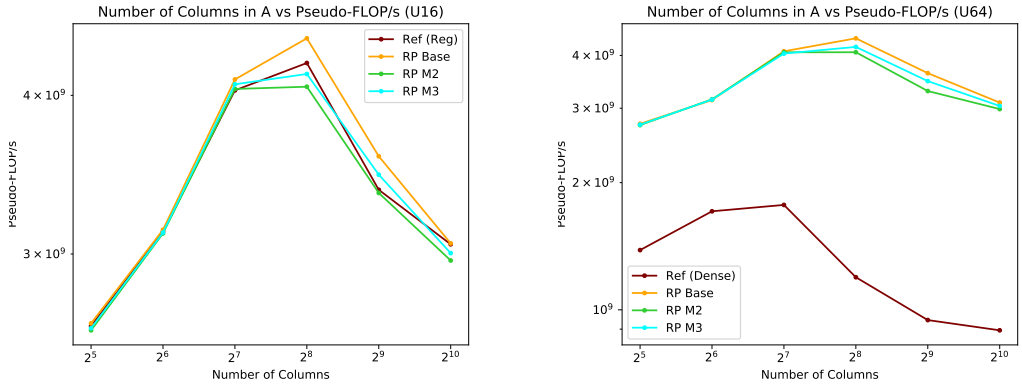


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.32: M Blocking: Roofline Plot - Vary Number of Rows

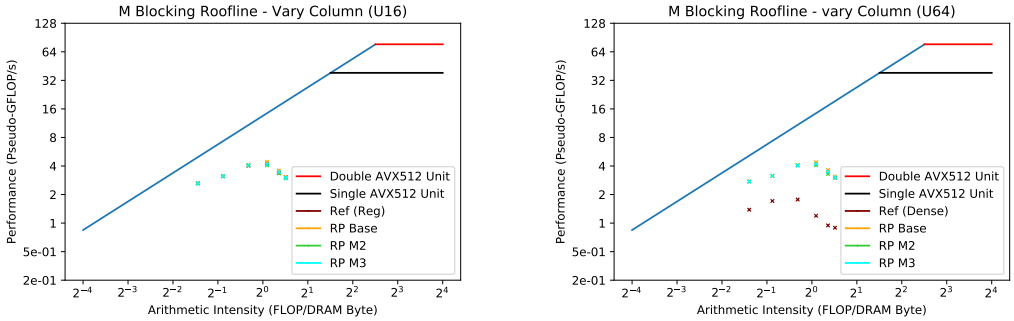
Vary Number of Columns

Interestingly, Figure 8.33a reveals that beyond 256 columns, additional M blocking performed slower (around 0.9x) than RP Base. Increasing the number of columns increases the total number of strides of **B** that could be loaded from memory. Additional M blocking increases the number of parallel loads, and at the density of 0.05, these loads would be more likely to be from memory, and not L1. Combining these two together, the kernels using additional M blocking place a greater load on the memory controller to read values from memory. Beyond 256 columns, it is possible that this increase in load is causing the decrease in performance.
We should note the large drop off in performance for all kernels beyond 512 columns is due to spilling into L2 cache. The performance penalty when using additional M blocking occurs before this, at 256 columns. Figure 8.34 shows that the kernels are not fully memory bound but do not benefit from issuing more FMAs. The experiment disproves hypothesis **6**.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.33: M Blocking: Performance vs Number of Columns

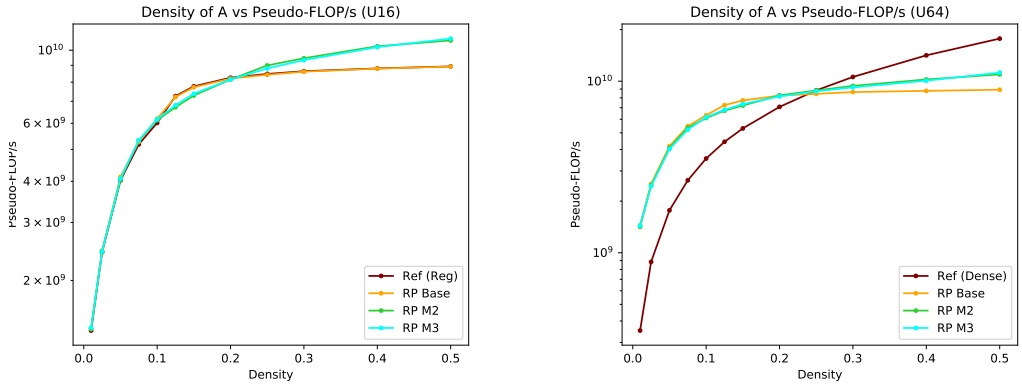


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.34: M Blocking: Roofline Plot - Vary Number of Columns

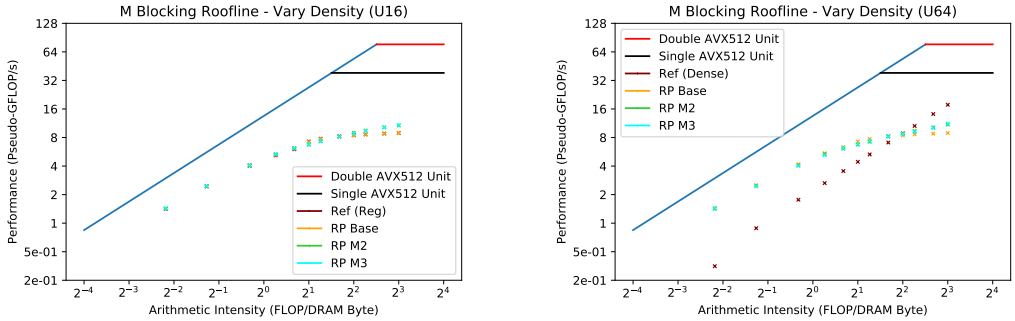
Vary Density

From around a density of 0.25, additional M blocking starts to provide a speedup over RP Base, peaking at around a 1.25x speedup at a density of 0.5, as shown in Figure 8.35a. However, the dense routine is shown to provide even faster performance over additional M blocking in Figure 8.35b. Figure 8.36 shows that only at very high AIs, additional M blocking can provide faster performance over RP Base. Issuing additional FMAs to access the same mini-chunk of B only helps when the kernels are in the compute bound region of the roofline plot. Whilst performance speedups are shown, hypothesis **6** does not hold as the speedups do not occur for all matrices.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.35: M Blocking: Performance vs Density



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 8.36: M Blocking: Roofline Plot - Vary Density

Vary Number of Unique Non-Zeros

Varying the number of unique constants does not effect the performance difference between RP Base and additional M blocking. Figure 8.37 stresses how using additional M blocking leads to defaulting to the dense routine sooner, leading to significant slowdowns. If tested at a higher density, and thus a higher AI, we would expect additional M blocking to provide a speedup, getting closer to the roofline in Figure 8.38.

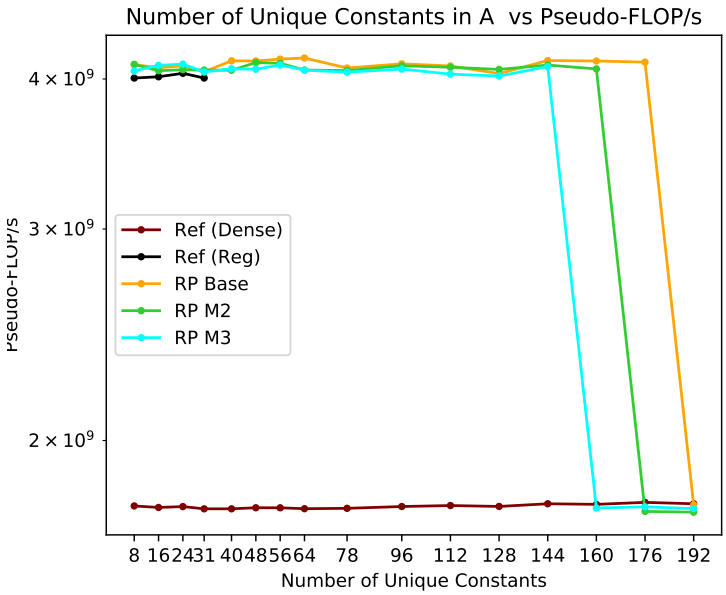


Figure 8.37: M Blocking: Performance vs Number of Unique Non-Zeros

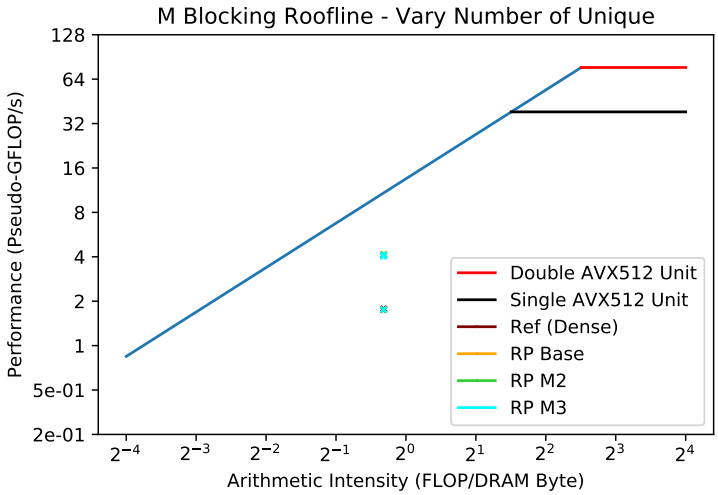


Figure 8.38: M Blocking: Roofline Plot - Vary Number of Unique Non-Zeros

8.4.3 Summary

We evaluated using additional M blocking with the register packing strategy on both the PyFR and synthetic operator matrices. Specifically, we tested $M = 2$ and $M = 3$. Through the evaluation, we conclude that hypothesis **6** is false, and that additional M blocking does not *always* lead to faster performance over RP Base. For denser matrices (≥ 0.5) that have high AIs, additional M blocking can provide up to 1.5x speedups over RP Base for PyFR matrices. However, the dense LIBXSMM routine still performed faster in those scenarios.

8.5 Summary

We outlined two methods that make use of the unused registers when register packing is used; additional N blocking and additional M blocking. Runtime broadcasting was found to *not* impact performance when using either method. We concluded that both hypothesis **5** and **6** were false, and that additional N/M blocking does not always provide speedups over RP Base.
However, for additional N blocking, we found that for sparse PyFR matrices, it can provide speedups of up to 1.1x over RP Base. For additional M blocking, we observed speedups of up to 1.5x over RP Base for denser PyFR matrices, but it was still slower than the dense LIBXSMM routine in those cases.

Chapter 9

Register Packing with Multiple Accumulation and L1 Operands

In Chapter 7 we showed that using L1 selector operands for register packing was suitable for small and sparse operator matrices. In Chapter 8 we showed that additional N blocking was also advantageous for those matrices. For denser matrices, additional M blocking provided speedups. These methods can be mixed together and combined to provide speedups whilst also supporting a greater number of unique non-zeros.
In this chapter we will combine additional N/M blocking with L1 selector operands to see if the performance speedups over RP Base remain, whilst being able to support more unique non-zeros. If there is spare bandwidth to L1 data cache, then the performance should be the same. If there are stalls when loading new strides of **B**, then there is an opportunity to load selector operands without penalty, whilst waiting on main memory loads.
Figures in Appendix E show the performance when mixing additional N *and* M blocking with L1 operands. The combinations did not lead to faster performance than using only one of N or M blocking for most matrices. In the cases that they did (some dense PyFR examples), they still performed slower the dense routine from LIBXSMM.

9.1 Solutions

By using L1 selector operands, a further 8/16 vector registers (DP/SP) out of 32 are available for use. In Chapter 7 they were only used to pack more unique values. However, they can also be utilised for additional N/M blocking. Table 9.1 summarises the combinations tested, which chapters detail the performance of them, and how many DP/SP values can be packed using them. For comparison, RP Base is also detailed. All of the other combinations support more unique non-zeros whilst using some form of additional blocking, compared to RP Base.

Combination	Chapter	Num of Free Registers	Max Num Values Packed (DP/SP)
RP Base	Ch. 6	22	176 / 352
L1 N2	Ch. 9	29	232 / 464
L1 N3	Ch. 9	28	224 / 448
L1 M2	Ch. 9	28	224 / 448
L1 M3	Ch. 9	26	208 / 416
L1 N2 M3	App. E	23	184 / 368
L1 N3 M2	App. E	24	192 / 384

Table 9.1: Comparing the Number of Unique Non-Zeros (U) that can be packed for different combinations of method

9.2 L1 N Blocking Evaluation

In this section we will evaluate $N = 2$ and $N = 3$ blocking being used in conjunction with register packing that also uses L1 selector operands. We will compare performance against the base register packing (RP Base) from Chapter 6 (black lines) and the register packing with L1 selector operands (RP L1) from Chapter 7 (pink lines). Roofline plots are not provided for this evaluation as previous results have shown that the kernels would be appear at around the same areas on a roofline plot. Throughout the evaluation, we will discuss the following hypothesis.

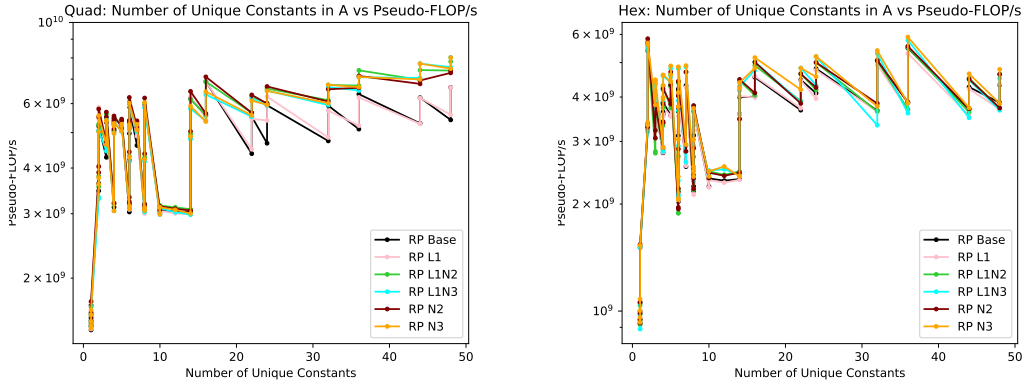
Hypothesis

7: Using L1 selector operands does not decrease performance when also using additional N blocking with register packing. The reason for this is that there should be enough bandwidth to L1 data cache and opportunities to load selector operands without penalty if there are stalls when loading strides of **B**.

9.2.1 PyFR Suite

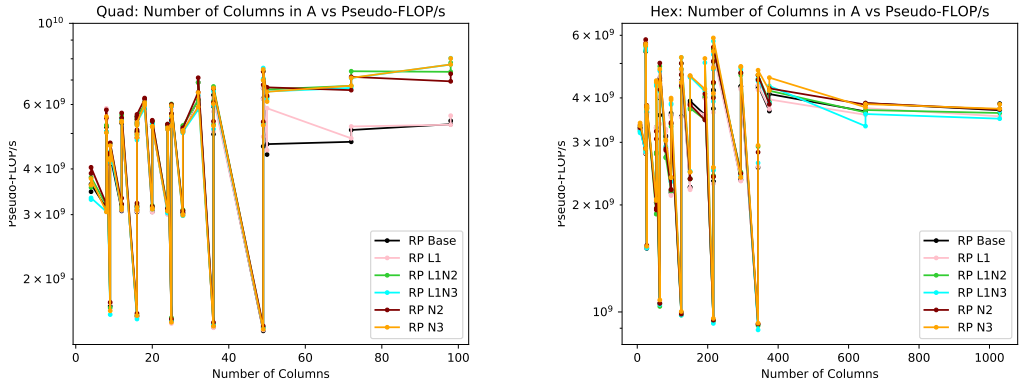
Sparse Operator Matrices

Figures 9.1a and 9.2a shows that using L1 selector operands does not impact performance when also using additional N blocking, for quadrilateral kernels. However, for hexahedra, as the number of columns increases in Figure 9.2b, the performance decreases to around 0.9x when using L1 selector operands. As the number of columns increases, the number of strides of **B** that are loaded increases. The slowdown could be due to the L1 selector operands taking up L1 data cache space, causing spilling to L2 cache for the increased amount of strides of **B**. Due to those few kernels, hypothesis **7** is shown to not be true.



(a) Quadrilaterals (b) Hexahedra

Figure 9.1: Register Packing L1 N Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)

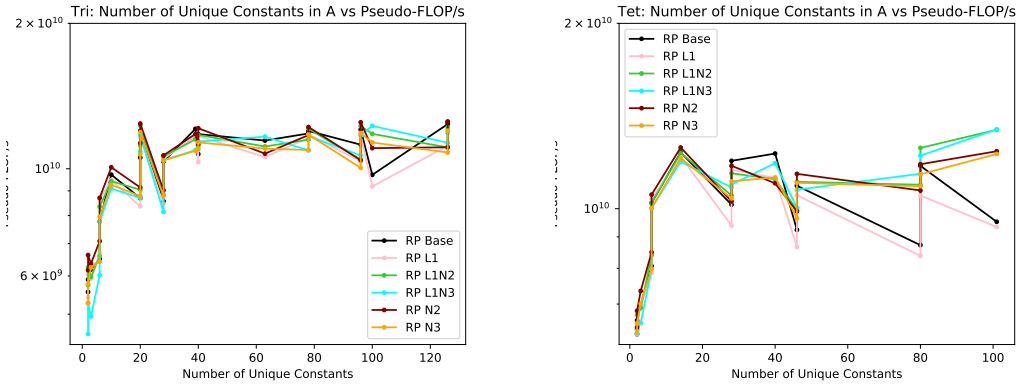


(a) Quadrilaterals (b) Hexahedra

Figure 9.2: Register Packing L1 N Blocking - PyFR Sparse Examples (Number of Columns)

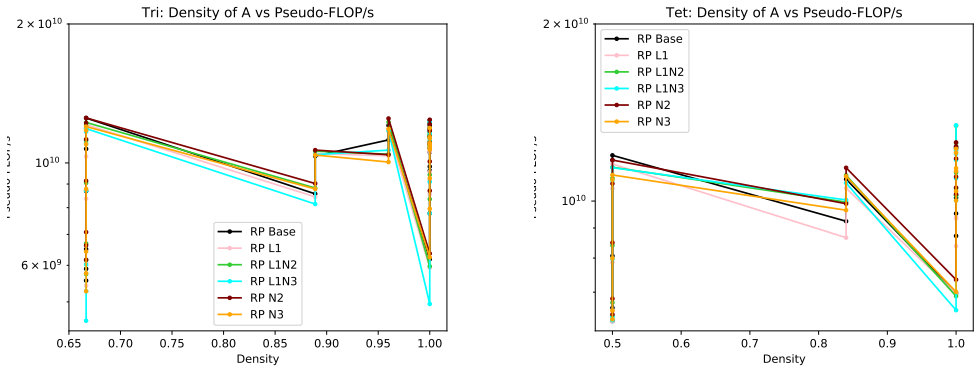
Dense Operator Matrices

For the dense PyFR operator matrices, Figure 9.3 shows that using L1 operands leads to mixed results when compared to just using additional N blocking. Figure 9.4 reveals that most of the larger slowdowns occur at very high densities. At higher densities, strides of $\mathbf{B}$ are reused more, and so have more broadcasts associated with them. This suggests that the kernels are over-saturating bandwidth to the L1 data cache, introducing a new bottleneck. The results show hypothesis **7** does not hold, especially for $N = 3$.



(a) Triangles (b) Tetrahedra

Figure 9.3: Register Packing L1 N Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)



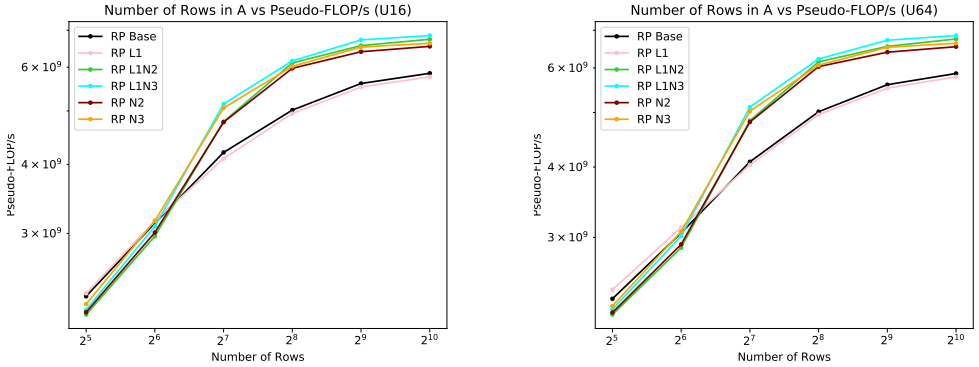
(a) Triangles (b) Tetrahedra

Figure 9.4: Register Packing L1 N Blocking - PyFR Dense Examples (Density)

9.2.2 Synthetic Suite

Vary Number of Rows

When using L1 selector operands, speedups of around 1.1x can be observed at very large number of rows in Figure 9.5 when compared to not using L1 selector operands. As in Chapter 7, without testing on a range of micro-architecture, no solid conclusions can be drawn from this behaviour. However, this specific experiment supports hypothesis **7**.

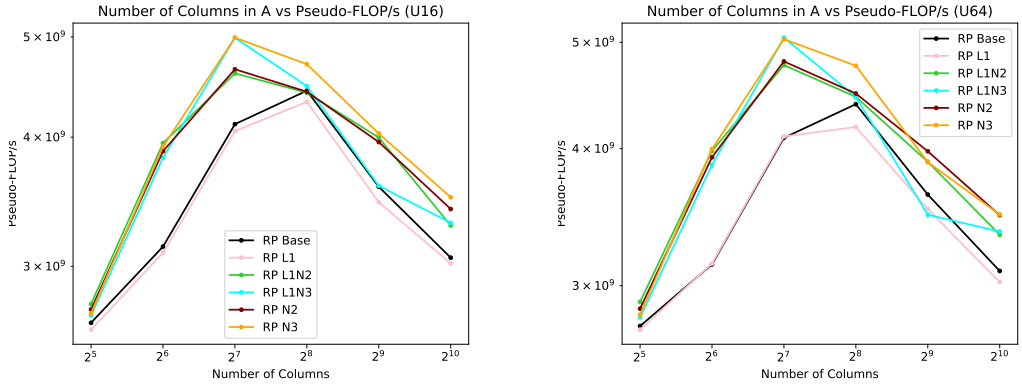


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 9.5: Register Packing L1 N Blocking: Performance vs Number of Rows

Vary Number of Columns

For up to 128 columns, the kernels using L1 selector operands perform the same as the kernels that don't use them. However, Figure 9.6 shows that from 256 columns and beyond, using L1 selector operands with $N = 3$ blocking leads to around 0.85x the performance compared to when not using them. For $N = 2$, a smaller drop occurs, with around 0.95x the performance compared to when not using L1 selector operands, but at 1024 columns. This is explained by the L1 selector operands taking up space in the L1 data cache, causing the strides of **B** to be evicted to a higher level. As the number of columns increases, the number of strides of **B** that are loaded increases, so those matrices are affected more by L1 cache evictions. This is clear evidence that hypothesis **7** is false, and that not considering the space taken by selector operands was a mistake when forming the hypothesis.

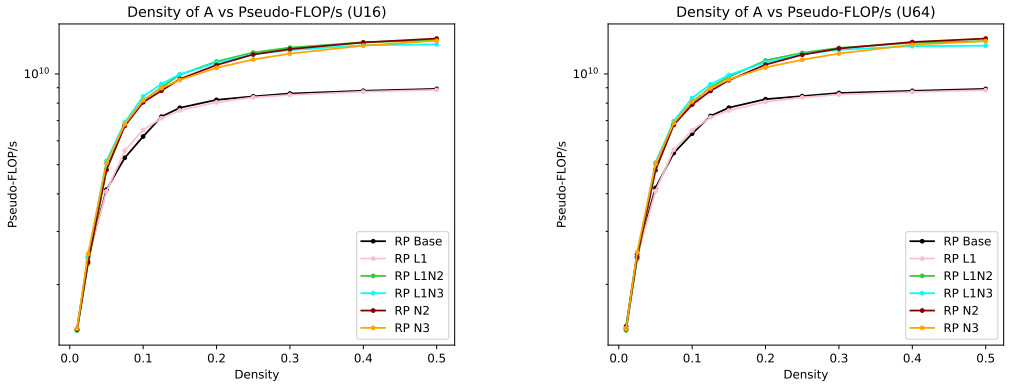


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 9.6: Register Packing L1 N Blocking: Performance vs Number of Columns

Vary Number of Density

As the density increases in Figure 9.7, the performance when using L1 selector operands remains the same as the kernels not using them for $N = 2$ and $N = 3$. For small matrices, there is sufficient L1 data cache space and bandwidth to load L1 selector operands as well as the multiple strides of **B**, up to the density of 0.5. This experiment strongly supports hypothesis **7**, showing that it holds for matrices that fit within the target dimensions of the sparse-dense MM routine.



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 9.7: Register Packing L1 N Blocking: Performance vs Density

Vary Number of Unique Non-Zeros

Figure 9.8 shows how using L1 selector operands allows the routine to continue to provide over 2x the performance of the dense routine when the number of unique non-zeros increases. The kernels using L1 selector operands consistently performed around 1.01x faster than the kernels that didn't use the L1 selector operands for $N = 3$. For $N = 2$ they provided the same performance. The results support hypothesis **7**.

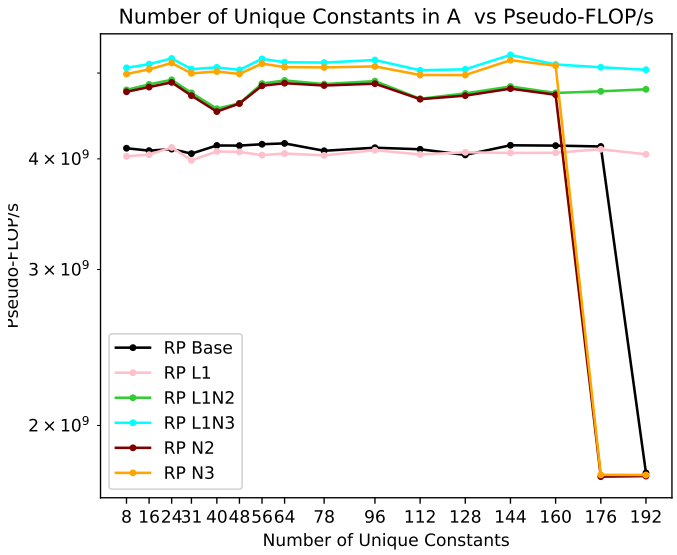


Figure 9.8: Register Packing L1 N Blocking: Performance vs Number of Unique Non-Zeros

9.2.3 Summary

We evaluated $N = 2$ and $N = 3$ blocking being used in conjunction with register packing that also uses L1 selector operands. For matrices where the number of columns was large (> 128), using L1 selector operands in conjunction with additional N blocking led to slower performance than just using additional N blocking. Also, we found that for dense PyFR operator matrices, the combination did not always maintain the performance levels of the kernels that do not use L1 selector operands. This meant that hypothesis **7** was false. The extra cache space taken by the operands is harmful for larger matrices and for denser matrices, the increase in L1 cache bandwidth that is required can over-saturate it and cause slowdowns. However, we found that for the smaller and sparser matrices, using L1 selector operands with additional N blocking was an adequate method to support more unique non-zeros without causing slowdowns in performance.

9.3 L1 M Blocking Evaluation

In this section we will evaluate $M = 2$ and $M = 3$ blocking being used in conjunction with register packing that also uses L1 selector operands. We will compare performance against the base register packing (RP Base) from Chapter 6 (black lines) and the register packing with L1 selector operands (RP L1) from Chapter 7 (pink lines). Roofline plots are not provided for this evaluation as previous results have shown that the kernels would be appear at around the same areas on a roofline plot. Throughout the evaluation, we will discuss the following hypothesis.

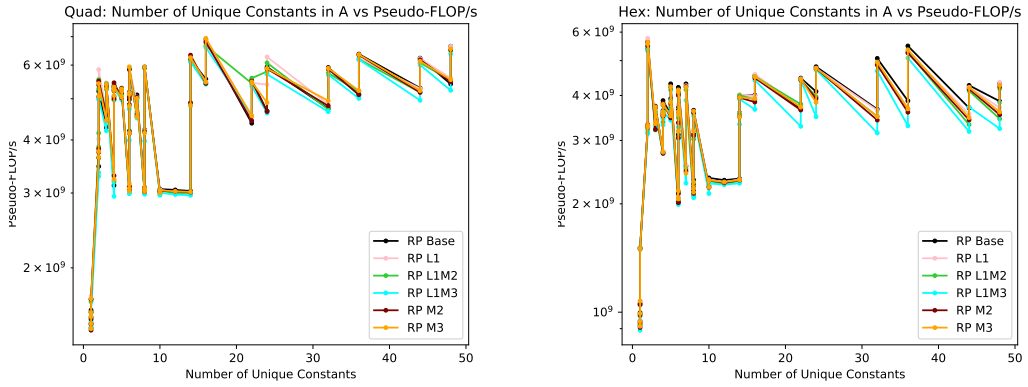
Hypothesis

8: Using L1 selector operands does not decrease performance when also using additional M blocking with register packing. The reason for this is that there should be enough bandwidth to L1 data cache and opportunities to load selector operands without penalty if there are stalls when loading strides of **B**.

9.3.1 PyFR Suite

Sparse Operator Matrices

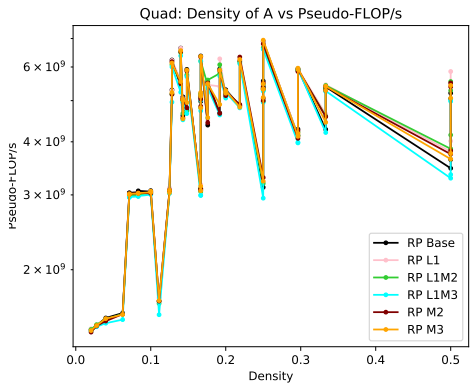
$M = 2$ blocking appears to not be impacted by using L1 selector operands, unlike $M = 3$ blocking, which shows performance decreases, as seen in Figure 9.9. Unlike additional N blocking, additional M blocking increases the number of 'parallel' broadcasts (linearly with M), and so the number of selector operands loaded in parallel increases. That is why $M = 3$ blocking is affected more than $M = 2$. Figure 9.10a shows that for quadrilaterals, the largest drop in performance with $M = 3$ is around 0.85x and occurs at a density of 0.5. Denser matrices lead to an increase in L1 data cache read pressure, which can end up becoming a bottleneck. For hexahedra, Figure 9.11b shows that when the number of columns is around 400 and greater, additional M blocking with L1 selector operands has around 0.8x the performance of kernels without L1 selector operands. $M = 2$ does not show a similar drop in performance, which suggests $M = 3$ with L1 selector operand kernels over-saturate bandwidth to L1 data cache. For $M = 2$, hypothesis **8** is true, but not for $M = 3$.



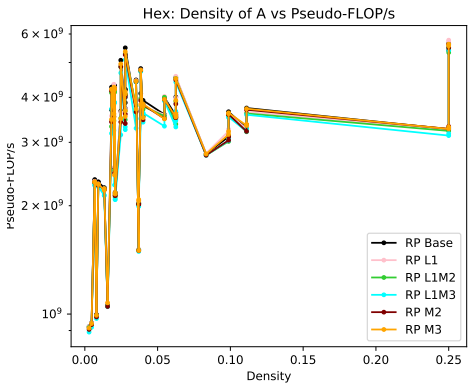
(a) Quadrilaterals

(b) Hexahedra

Figure 9.9: Register Packing L1 M Blocking - PyFR Sparse Examples (Number of Unique Non-Zeros)

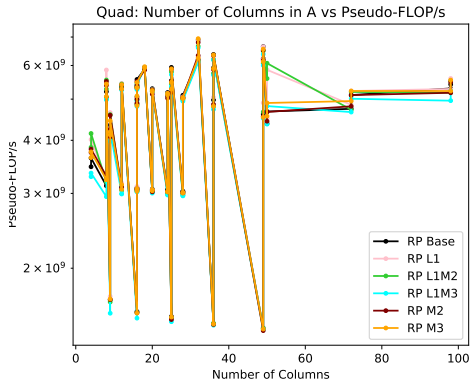


(a) Quadrilaterals

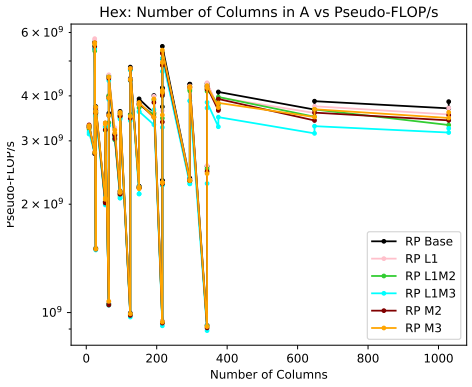


(b) Hexahedra

Figure 9.10: Register Packing L1 M Blocking - PyFR Sparse Examples (Density)



(a) Quadrilaterals

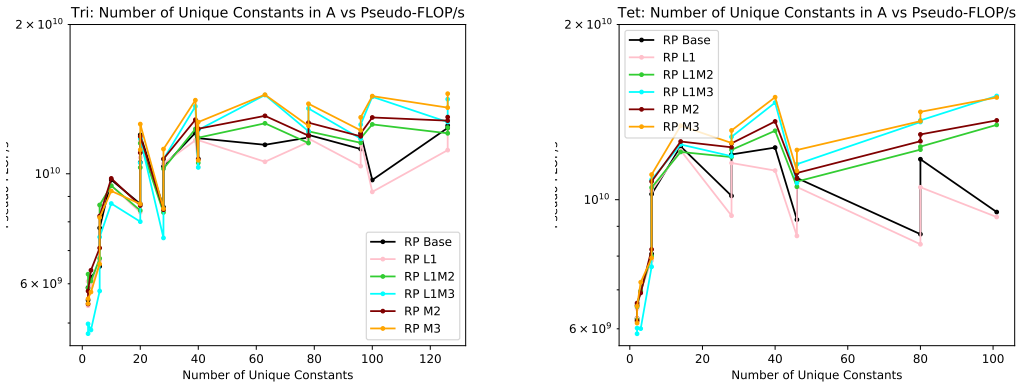


(b) Hexahedra

Figure 9.11: Register Packing L1 M Blocking - PyFR Sparse Examples (Number of Columns)

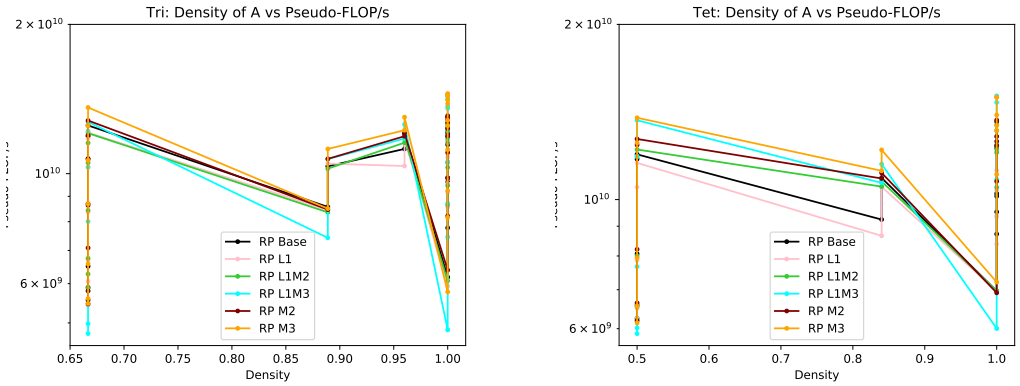
Dense Operator Matrices

Figure 9.12 shows that both $M = 2$ and $M = 3$ blocking perform slower when using L1 selector operands than when not using them. $M = 3$ with L1 selector operands exhibit performance around 0.8x with them compared to without, which can be better seen in Figure 9.13. In comparison, $M = 2$ blocking is shown to drop to around 0.95x the performance. This shows that at higher densities, $M = 2$ starts to also saturate L1 data cache bandwidth. For $M = 3$, the effects are even more severe for dense matrices than sparse matrices. Hypothesis **8** should be rejected based on these results.



(a) Triangles (b) Tetrahedra

Figure 9.12: Register Packing L1 M Blocking - PyFR Dense Examples (Number of Unique Non-Zeros)



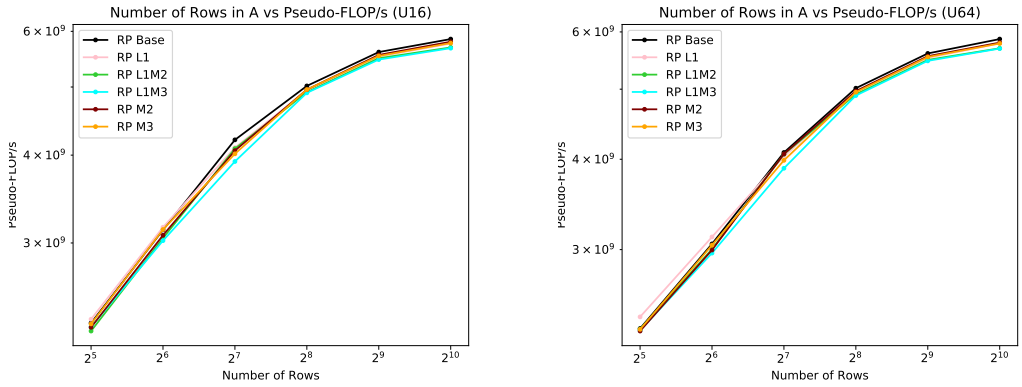
(a) Triangles (b) Tetrahedra

Figure 9.13: Register Packing L1 M Blocking - PyFR Dense Examples (Density)

9.3.2 Synthetic Suite

Vary Number of Rows

Performance is about the same for $M = 2$ blocking with and without L1 selector operands as the number of rows is varied in Figure 9.14. For $M = 3$, beyond 64 rows, using L1 selector operands leads to around 0.9x the performance compared to when not using them. Increasing the number of rows increases the number of times the strides of **B** are reused. This then leads to L1 data cache bandwidth becoming the bottleneck once filled with the mini-chunk of **B**, as opposed to the main memory bandwidth when first loading the strides. As $M = 3$ loads more selector operand in parallel than $M = 2$, it runs into this bottleneck whereas $M = 2$ does not. The results suggest that hypothesis **8** is false.
As seen in the experiment from varying the density, further below, it is likely that the increase in reuse of strides of **B** is not only the reason $M = 3$ suffered at higher number of rows. It is possible that a mixture of that and the Skylake-SP micro-architecture resulted in slower performance. If tested with a density of 0.25 instead of 0.05, we might have seen similar or even faster performance for $M = 3$ when using L1 selector operands versus when not using them.



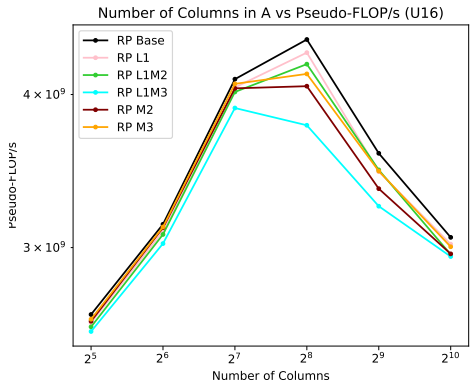
(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

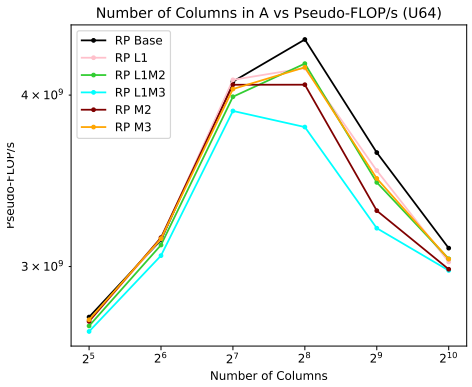
Figure 9.14: Register Packing L1 M Blocking: Performance vs Number of Rows

Vary Number of Columns

For $M = 2$ blocking, using L1 selector operands led to similar performance than without. When the number of columns was 512 it performed about 1.05x faster, as shown in Figure 9.15. For $M = 3$, performance was always slower when using L1 selector operands. When spilling to L2 cache occurred from 512 columns and beyond, the use of L1 selector operands led to around 0.9x the performance compared to just $M = 3$ blocking. Once spilling occurs, the space taken up by the selector operands causes even more evictions from L1 data cache. This is a bigger issue for $M = 3$ as it loads more strides of **B** in parallel, and so the L2 cache latency is considerably impacting the kernel performance.
An unexpected result is that we see the effects of spilling to L2 cache for $M = 3$ blocking with L1 selector operands at just 256 columns, where the mini-chunk of **B** should easily fit within the L1 data cache. This suggests that the L1 selector operands cause conflict cache misses for the strides of **B**.
For $M = 3$ we reject hypothesis **8**, but it holds for $M = 2$.



(a) 16 Unique Non-Zeros

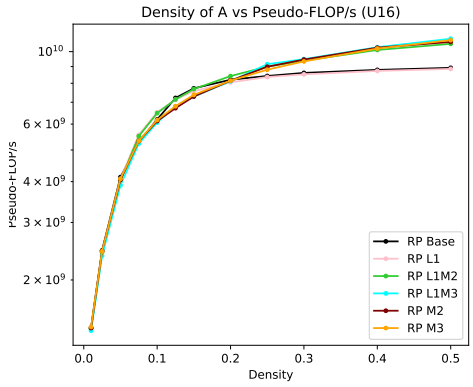


(b) 64 Unique Non-Zeros

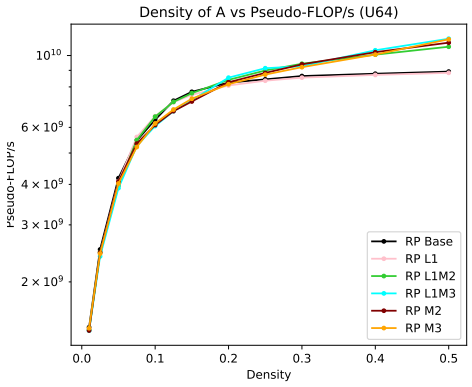
Figure 9.15: Register Packing L1 M Blocking: Performance vs Number of Columns

Vary Number of Density

Interestingly, as the density increased, the performance for both $M = 2$ and $M = 3$ remained around the same when using L1 selector operands compared to when not using them, as seen in Figure 9.16. There are a couple of densities for $M = 3$, where using L1 selector operands was slower (0.97x at density of 0.05) or faster (1.03x at density of 0.25), but this can be down to micro-architecture quirks, and so further testing would need to be carried out to draw solid conclusions. This suggests the results from the experiment where we varied the number of rows could have had a different outcome if we tested with a density of 0.25 instead of 0.05. Hypothesis **8** is supported by the results from this experiment.



(a) 16 Unique Non-Zeros



(b) 64 Unique Non-Zeros

Figure 9.16: Register Packing L1 M Blocking: Performance vs Density

Vary Number of Unique Non-Zeros

As we can see from Figure 9.17, $M = 3$ with L1 selector operands performed around 0.97x as fast compared to without, when both versions support the number of unique non-zeros. This would most likely be *slightly* different if tested at a different density. For $N = 2$, performance is mostly the same when both versions support the number of unique non-zeros. However, we see that using L1 selector operands allow the routines to provide a roughly similar of performance for small matrices that have a greater number of unique non-zeros. We can roughly say the results provide support for hypothesis **8**, once we take into account the exact density (0.05) of the matrices tested.

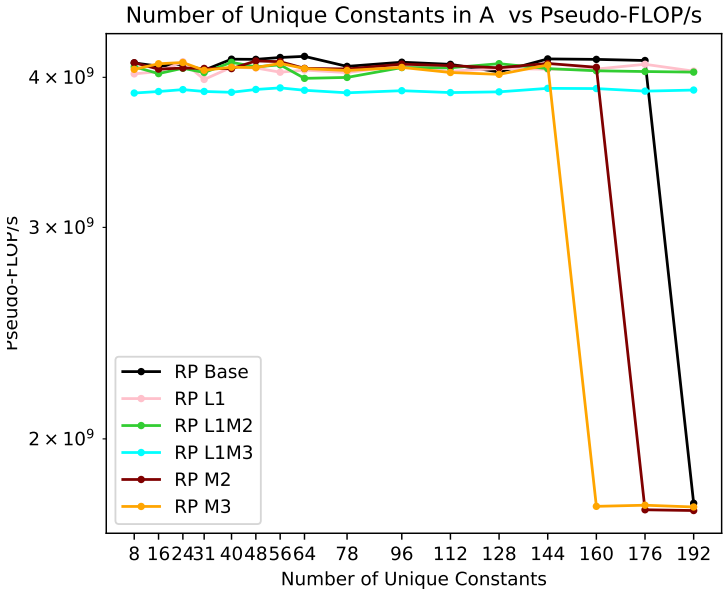


Figure 9.17: Register Packing L1 M Blocking: Performance vs Number of Unique Non-Zeros

9.3.3 Summary

We evaluated $M = 2$ and $M = 3$ blocking being used in conjunction with register packing that also uses L1 selector operands. For matrices where the number of columns was large (> 256), using L1 selector operands in conjunction with $M = 3$ blocking led to slower performance than just using $M = 3$ blocking. This was due to the space taken up by selector operands in L1 data cache causing more spilling to L2 cache for the strides of **B**. Also, we found that for high density PyFR matrices, using L1 selector operands led to slower performance when used with either $M = 2$ or $M = 3$ blocking.
Thus, we conclude that hypothesis **8** is false for the general case of additional M blocking. However, for $M = 2$ blocking, the hypothesis is true when the matrices are small and sparse. So, using L1 selector operands with $M = 2$ blocking for small and sparse matrices is a method to support a greater number of unique non-zeros without sacrificing performance.

9.4 Summary

We outlined solutions that mixed together additional N/M blocking with L1 selector operands, detailing the increase in number of unique non-zeros supported. However, this method increases the number of loads from and the space taken in the L1 data cache, so we evaluated the new solutions against kernels just using additional N/M blocking.
For operator matrices with a large number of columns (128/256) and for dense PyFR operator matrices, additional N/M blocking when used with L1 selector operands performed slower than kernels just using additional N/M blocking.
For smaller (number of columns ≤ 128) and sparser matrices (density ≤ 0.5), using L1 selector operands did not impact performance when also using additional N blocking or $M = 2$ blocking. This was not the case for $M = 3$ blocking, where performance decreased on those matrices when also using L1 selector operands.

Chapter 10

Hybrid LIBXSMM vs GiMMiK

In this chapter we will outline a strategy that chooses one of the previously evaluated solutions depending on a couple of heuristics. We refer to this as the *hybrid routine*. We will evaluate this hybrid routine against GiMMiK 2.1 on the benchmark. We aim to provide a suggestion on which library to use to accelerate the PyFR matrix multiplication, as well as using the synthetic suite to investigate the similarities and differences between the solutions.
Appendix F contains additional Figures that compare the performance between the hybrid routine and GiMMiK 2.1 on the PyFR example operator matrices.

10.1 Solution

10.1.1 GiMMiK 2.1

The benchmark was adapted to use GiMMiK 2.1 [32] to generate 'C' kernels. Instead of passing in 'chunks' of **B** within a loop to the kernel, like in LIBXSMM, the kernel is only called once with the entire matrix. Performance was measured in the same environment as used for LIBXSMM, outlined in Chapter 4. The 'C' kernel code was compiled using Intel's C/C++ compiler, *ICC*, with the following compiler flags: "-std=c11 -O3 -pthread -D_GNU_SOURCE -DNDEBUG -mavx512f -mavx512cd -mavx512vl -mavx512dq -mavx512bw -mfma -march=skylake-avx512 -qopt-zmm-usage=high -qopenmp".
The 'C' kernel generated by GiMMiK contains one for loop, which iterates over the columns in **B**. Within the loop, dot-products are calculated between the rows of **A** and the current column of **B**, then stored to **C**. Sparsity is eliminated and this loop is vectorised by the compiler.
We inspected the assembly code generated and found that the code was being vectorised for AVX-512 and the loop was being fully unrolled for most of the matrices tested. Kernels compiled for the following matrices had the warning of "*was not vectorized with "simd"*":

- PyFR - Hexahedra, Sixth Order, Gauss-Legendre-Lobatto - m132
- PyFR - Hexahedra, Sixth Order, Gauss-Legendre-Lobatto - m460
- PyFR - Hexahedra, Sixth Order, Gauss-Legendre - m132
- PyFR - Hexahedra, Sixth Order, Gauss-Legendre - m460
- Synthetic - Vary Density ($U = 16$) - 0.5
- Synthetic - Vary Density ($U = 64$) - 0.5
- Synthetic - Vary Number of Rows ($U = 16$) - 1024

- Synthetic - Vary Number of Rows ($U = 16$) - 1024

This was due to the main body of the for loop in the 'C' kernel being too large for the compiler, ICC, to handle.

10.1.2 Hybrid Routine - Strategy

The strategy for the hybrid routine is listed in Algorithm 1. The conditions were chosen based on results from previous evaluations in the thesis.
Starting with the first condition, if the density falls within the dense PyFR operator matrices range, the reference dense routine is used. Next, for slightly less dense matrices that are not considered small, the dense routine is also used. From then on, the sparse routine using additional N blocking is preferred. The exact method used is determined by the number of unique non-zeros in the matrix, with the more performant subroutine is chosen if possible. Finally, if the number of unique non-zeros cannot be supported, the fallback of the reference dense routine is used.
Ideally, there would be a fallback routine that can handle any amount of unique non-zeros which also performs sparsity elimination. This would essentially be a routine that implements the strategy GiMMiK uses, but the code generation would use the LIBXSMM API and be fully in our control, instead of relying on a third-party compiler. However, in the suite of matrices tested, there is no small and sparse matrix that is not covered by the available sparse subroutines within the hybrid routine.

Algorithm 1: Hybrid Routine - Strategy to choose subroutine (DP values)

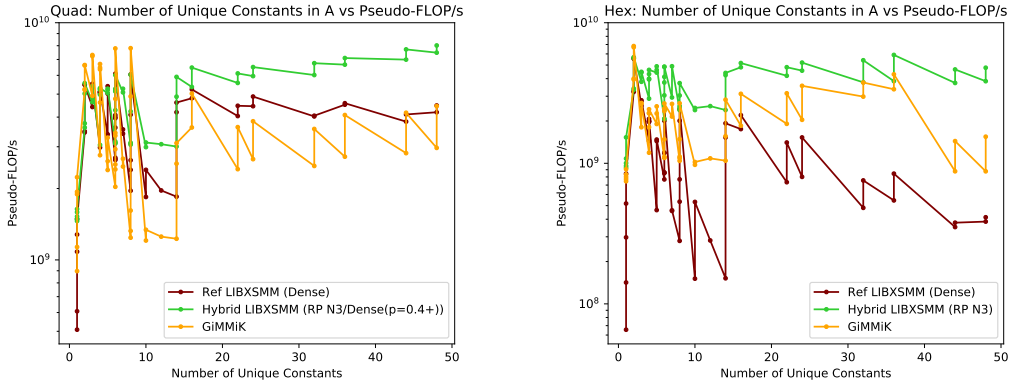
```
ρ = density;
U = number of unique non-zeros;
R = number of rows;
C = number of columns;
if ρ ≥ 0.5 then
    use dense routine;
else if ρ ≥ 0.4 and R ≥ 128 and C ≥ 128 then
    use dense routine;
else
    if U ≤ 160 then
        use sparse - register packing with N = 3 blocking;
    else if U ≤ 224 then
        use sparse - register packing with N = 3 blocking and L1 selector operands;
    else if U ≤ 232 then
        use sparse - register packing with N = 2 blocking and L1 selector operands;
    else if U ≤ 240 then
        use sparse - register packing with L1 selector operands;
    else
        use dense routine;
    end
end
```

10.2 Hybrid vs GiMMiK Evaluation

10.2.1 PyFR Suite

Sparse Operator Matrices

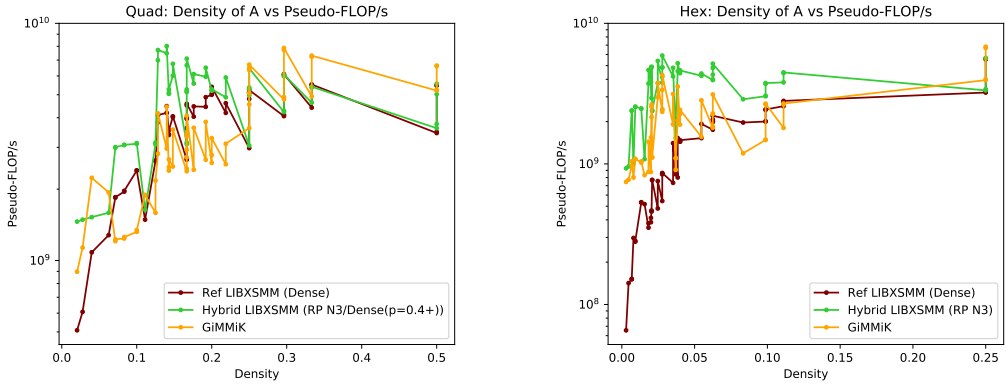
In general, the hybrid routine performed faster or the same as GiMMiK for sparse PyFR operator matrices. In a couple of cases, GiMMiK performed around 1.3x faster for quadrilaterals, shown in Figure 10.1a. However, there are more cases where the hybrid routine performed 2x faster. In those cases, GiMMiK performed slower than the dense routine. For hexahedra, GiMMiK was faster than the dense routine, but the hybrid routine was usually faster than GiMMiK, by over 2x in a few cases, including the four cases when GiMMiK does not produce SIMD code, shown in Figure 10.1b.
Figure 10.2a shows that as the density varies there is no clear trend as to which routine was faster for quadrilaterals. However, when looking at Figure 10.3, it is clear that as the number of rows increases, the hybrid routine outperformed GiMMiK for sparse PyFR matrices. As the number of rows increases, so does the length of the body of the for loop inside of a GiMMiK 'C' kernel. It is possible that ICC struggles to generate as optimal code for larger loop bodies as it does for smaller loop bodies. We should note that *all* of the GiMMiK kernels for *quadrilaterals* were using AVX-512 code.
Figure 10.4 shows that whilst the hybrid routine kernels appear to be mainly memory bound, the slower GiMMiK kernels do not saturate the bandwidth to main memory. We confirmed that the kernels were fully vectorised and also unrolled. So the slowdown could be due to ICC generating code with worse low level cache (L1) usage compared to the hybrid routine. Overall, the hybrid routine beat GiMMiK for sparse PyFR operator matrices.



(a) Quadrilaterals

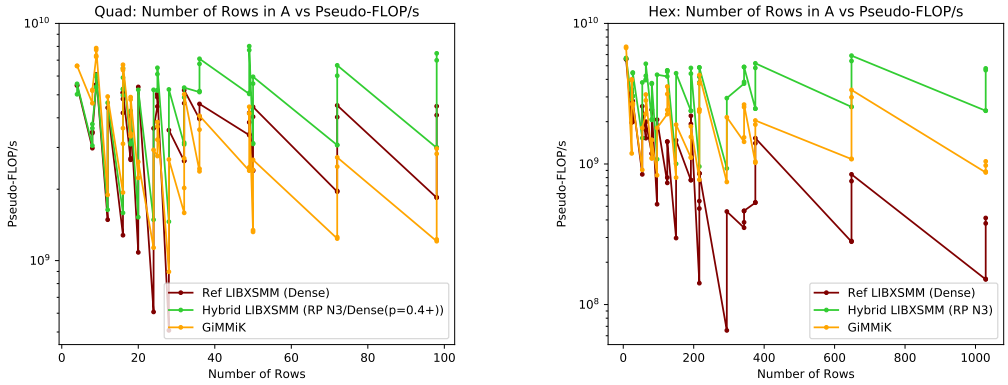
(b) Hexahedra

Figure 10.1: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Number of Unique Non-Zeros)



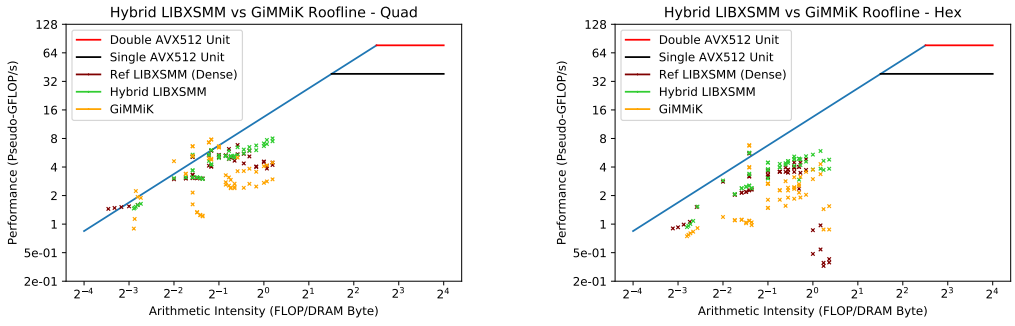
(a) Quadrilaterals (b) Hexahedra

Figure 10.2: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Density)



(a) Quadrilaterals (b) Hexahedra

Figure 10.3: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Number of Rows)



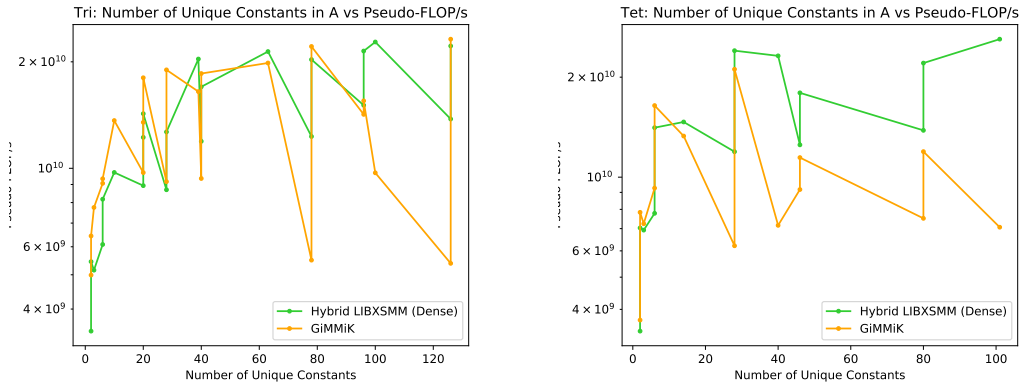
(a) Quadrilaterals (b) Hexahedra

Figure 10.4: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Roofline Plots)

Dense Operator Matrices

The hybrid routine uses the reference LIBXSMM dense routine for dense (density ≥ 0.5) PyFR operator matrices. Figure 10.5b shows that in general, for tetrahedra the hybrid routine performed faster than GiMMiK, and by up to 3x in one case. For triangles, the story is more complex. Figure 10.6a suggests that for triangles, GiMMiK was faster at the higher densities. If we instead look at Figure 10.7a, it cleat that beyond 20 rows in the operator matrix, GiMMiK performed slower, and that below 20 rows, it performed around the same or faster. For triangle meshes using third order accurate and lower solutions, the number of rows is ≤ 20. So for even higher-order solutions, the hybrid (dense) routine would offer faster performance. In the case of tetrahedra, only the first order accurate solution has all of its operator matrices contain fewer than 20 rows. Figure 10.7b shows that as the number of rows increases, GiMMiK performed slower than the hybrid routine for tetrahedra, and so shouldn't be used for tetrahedra meshes using a second order accurate or greater solution.

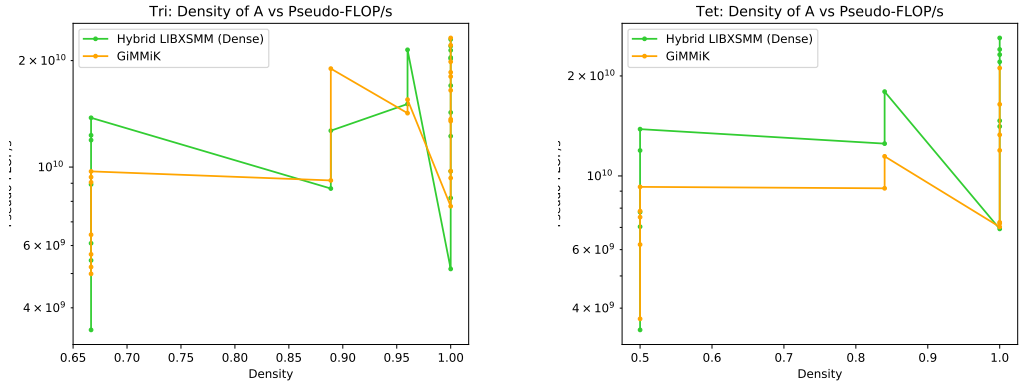
Interestingly, in the cases where the hybrid routines are close to being memory bound, GiMMiK performed faster, especially for triangles, shown in 10.8a. This suggests ICC is better able to schedule the loads from main memory than the pre-fetching that LIBXSMM applies, and so achieved a higher peak bandwidth. When not memory bound, as in a few cases in Figure 10.8b for tetrahedra, GiMMiK performed slower. It is possible that the GiMMiK kernels are not making good use of the fast L1 data cache, and so struggle to achieve a high pseudo-FLOP/s. This could be confirmed via profiling and using performance counters.



(a) Triangles

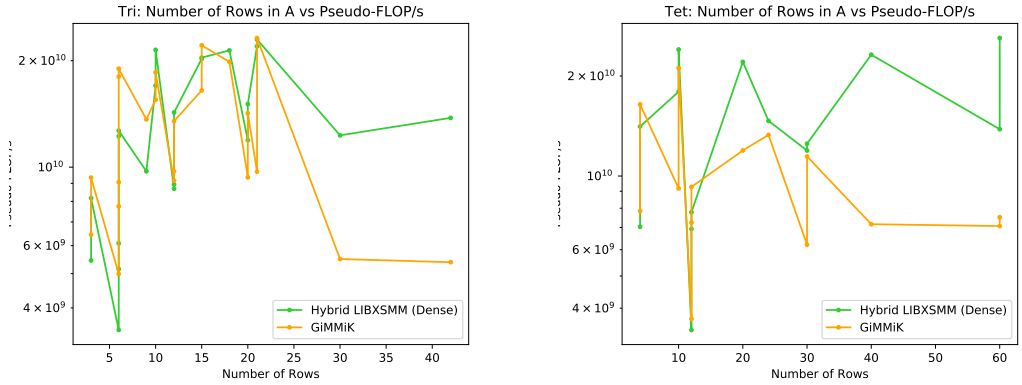
(b) Tetrahedra

Figure 10.5: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Number of Unique Non-Zeros)



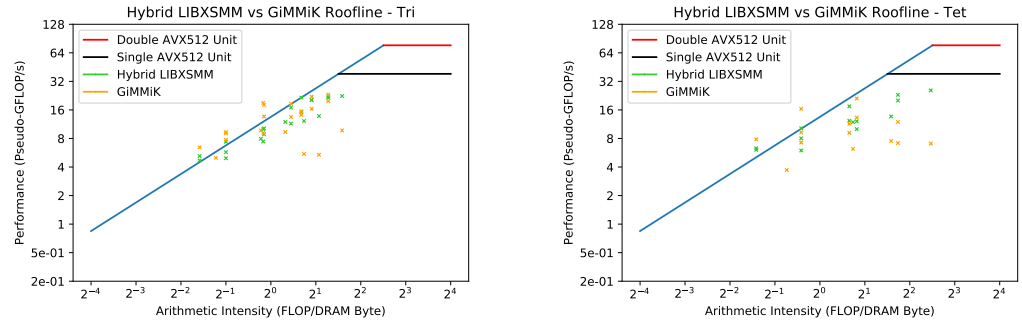
(a) Triangles (b) Tetrahedra

Figure 10.6: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Density)



(a) Triangles (b) Tetrahedra

Figure 10.7: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Number of Rows)



(a) Triangles (b) Tetrahedra

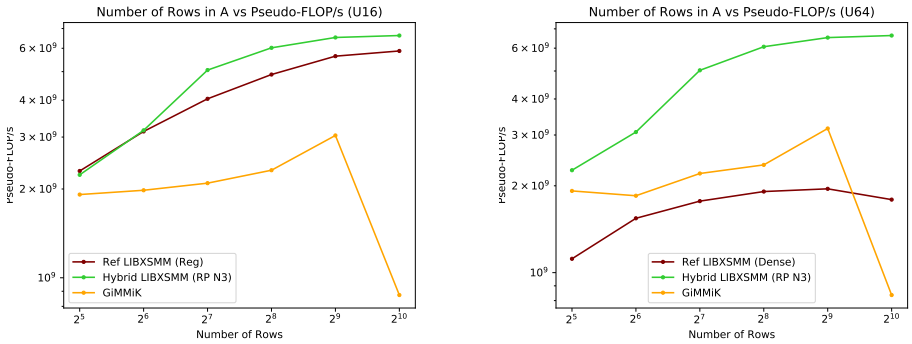
Figure 10.8: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Roofline Plots)

10.2.2 Synthetic Suite

For the synthetic suite, the hybrid routine mainly uses the Register Packing with $N = 3$ blocking (RP N3) routine, apart from a few cases. In the density experiment, at densities of 0.4+, the dense routine is used. In the number of unique non-zeros experiment, for over 160 unique non-zeros, the Register Packing with $N = 3$ blocking and L1 selector operands (RP L1 N3) routine is used.

Vary Number of Rows

At the lowest number of rows tested, 32, the performance gap between GiMMiK and the hybrid routine was at its smallest, shown in Figure 10.9. As the number of rows increases, so did the performance difference; where the hybrid routine performed 1.1x faster at 32 rows, to around 2.6x faster at 256 rows. At 512 rows, the GiMMiK kernels performance increased faster than the hybrid routine, when compared to 256 rows. This suggests the strategy ICC uses to compile the code works better for higher kernels with a higher AI. However, at 1024 rows the performance of the GiMMiK kernel deteriorated, when ICC no longer generates SIMD code, and the hybrid routine performed 10x faster. Table 10.1 shows the binary size of the compiled GiMMiK kernels for 256+ rows. The size is smaller for 1024 rows, at 177KB, than for 512 rows, where it was 263KB. We would expect the code size to increase if ICC was applying the same technique to generate code as it does for the matrices with fewer than 1024 rows. Instead, this binary size confirms the kernel was not being fully unrolled, which when combined with the non-SIMD code, led to the kernel performing poorly.
It is possible that for very small matrices with at least fewer than 32 rows, GiMMiK might offer competitive performance.



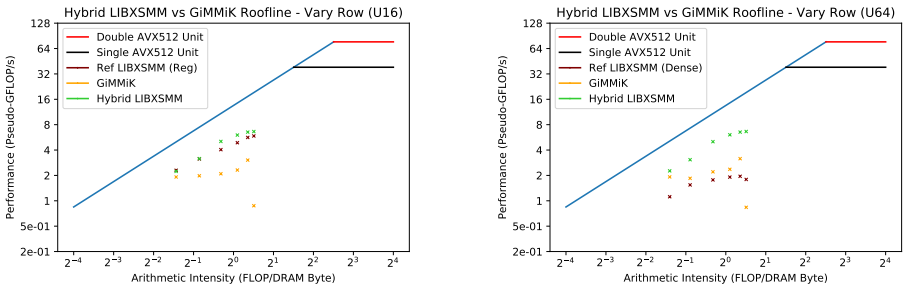
(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure 10.9: Hybrid LIBXSMM vs GiMMiK: Performance vs Number of Rows

Number of Rows	Binary Size (KB)	Compiler Warning
256	153	None
512	263	None
1024	177	was not vectorized with "simd"

Table 10.1: GiMMiK Kernel Binary Size - Vary Number of Rows

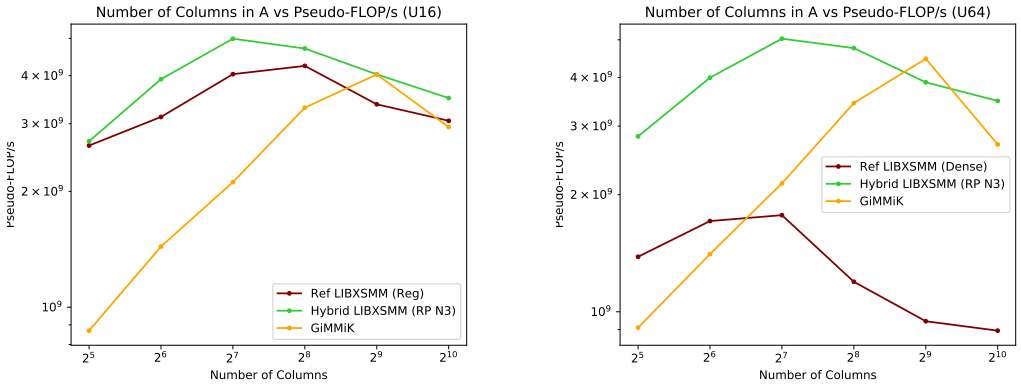


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 10.10: Hybrid LIBXSMM vs GiMMiK: Roofline Plot - Vary Number of Rows

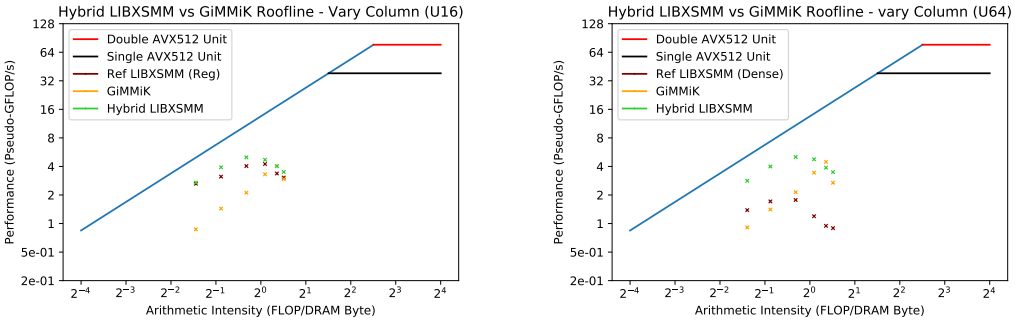
Vary Number of Columns

Interestingly, as the number of columns increases from 128 to 512, the performance gap between GiMMiK and the hybrid routine decreased, where GiMMiK then performed around 1.1x faster for $U = 64$, shown in Figure 10.11b. When $U = 16$, they performed the same as seen in Figure 10.11a. The hybrid routine performed the same between $U = 16$ and $U = 64$. We cannot determine a solid reason as to why GiMMiK did not.
At 1024 columns, GiMMiK's performance dropped, but the kernel was still being fully unrolled by ICC and it still used AVX-512 code. It is possible that the strategy used by ICC to compile the kernel has worse cache hit-rates at 1024 columns than it has for 512 columns and below.
From 32 to 128 columns, the performance difference was maintained. From 256 columns, RP N3 starts to spill to L2 cache. It appears that ICC uses a strategy that isn't reliant on keeping 'mini-chunks' of **B** in L1 data cache, and so gains the benefit of a higher AI without spilling outside of whatever cache (not L1, likely L2) strategy it is using. This is highlighted in Figure 10.12 where the increase in columns leads to an increase in AI, and so does the higher peak pseudo-FLOP/s. This experiment reveals that the LIBXSMM strategy of keeping to L1 data cache can be significantly advantageous for small matrices.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 10.11: Hybrid LIBXSMM vs GiMMiK: Performance vs Number of Columns

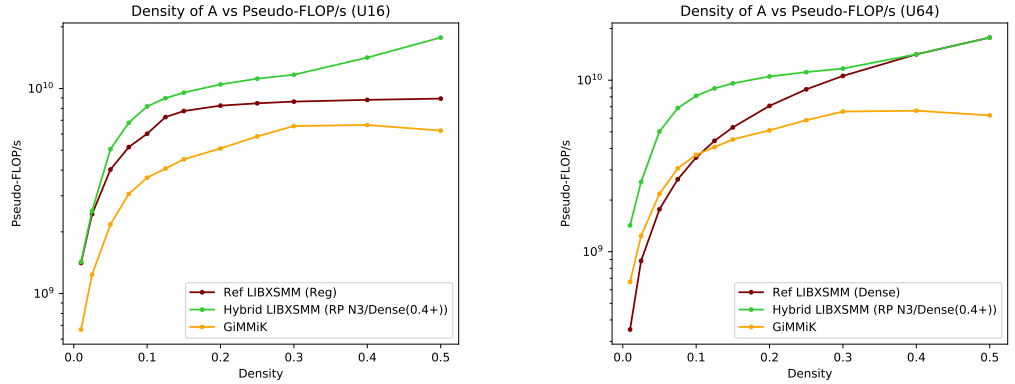


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 10.12: Hybrid LIBXSMM vs GiMMiK: Roofline Plot - Vary Number of Columns

Vary Number of Density

As the density varied, the hybrid routine maintained around a 1.8x speedup over GiMMiK until the density reached 0.3, shown in Figure 10.14, for both 16 and 64 unique non-zeros. Once the hybrid routine switched to the dense routine at a density of 0.4 and greater, it started to increase its performance lead over GiMMiK, which instead plateaued. The drop in performance for GiMMiK at density of 0.5 is explained by ICC failing to generate SIMD code for it.
If operator matrices in this test were smaller than $128 \cdot 128$, we would then expect, based on the previous two experiments, that the performance difference would at least be closer. This experiment shows that the performance gap between GiMMiK and hybrid routine kernels is more sensitive to the dimensions of the operator matrix, than the density.

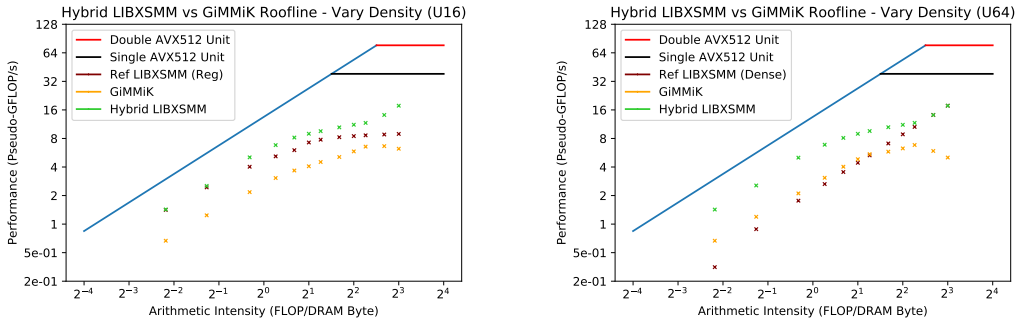


(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 10.13: Hybrid LIBXSMM vs GiMMiK: Performance vs Density

Vary Number of Unique Non-Zeros

Increasing the number of unique non-zeros did not *significantly* impact the performance of the GiMMiK kernels, as seen in Figure 10.15. This was expected as GiMMiK encodes the values into the kernel code. However, we see that as the hybrid routine is able to switch to the RP L1 N3 routine when $U > 160$, it maintained just over a 2x speedup over GiMMiK.



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure 10.14: Hybrid LIBXSMM vs GiMMiK: Roofline Plot - Vary Density

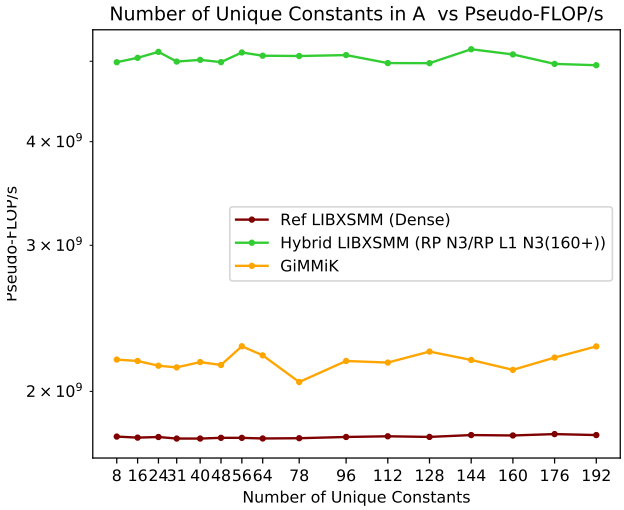


Figure 10.15: Hybrid LIBXSMM vs GiMMiK: Performance vs Number of Unique Non-Zeros

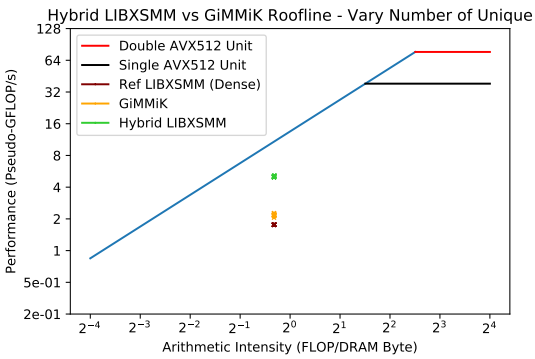


Figure 10.16: Hybrid LIBXSMM vs GiMMiK: Roofline Plot - Vary Number of Unique Non-Zeros

10.3 Summary

In this chapter we outlined a strategy that chooses one of the previously evaluated solutions depending on a couple of heuristics, and named this the *hybrid routine*. We evaluated this hybrid routine against GiMMiK 2.1 on the benchmark.
For sparse PyFR operator matrices, (quadrilateral and hexahedra meshes), we found that the hybrid routine, using the sparse-dense, register packing with $N = 3$ blocking routine, performed faster than GiMMiK for the majority of the matrices, and so should be used for those matrices. The hybrid routine selects the dense routine from LIBXSMM for the dense PyFR operator matrices. For triangular meshes, where a third-order accurate or lower solution is being computed, GiMMiK was shown to offer faster performance. For tetrahedra meshes, the hybrid LIBXSMM routine offered faster performance for second-order accurate and greater solutions.
Evaluation on the synthetic suite showed that the performance difference between the hybrid LIBXSMM routine and GiMMiK is strongly linked to the dimensionality of the operator matrix. GiMMiK was shown to perform closer to the hybrid routine when the matrix had a small number of rows. For a larger number of columns, the hybrid routine 'spills' to L2 cache when GiMMiK does not spill, as it is likely already operating at that level. That allowed GiMMiK to close the performance gap, and in some cases, perform faster for those larger matrices.

Chapter 11

Conclusions & Further Work

In this chapter we summarise the findings of this thesis, have a look at the initial objectives set out and cover the contributions made. We then discuss a range of further work which could potentially deliver further performance improvements not only for AVX-512, but also for other SIMD architectures.

11.1 Summary

The main objective of this thesis was to find and evaluate a method to support a greater number of unique non-zeros for the sparse-dense matrix multiplication routine within LIBXSMM, to support more operator matrices found in PyFR applications. The next aim was to explore methods to utilise any resulting free registers to gain additional performance speedups. The primary hardware target was AVX-512 on the Skylake-SP microarchitecture. In Chapter 2 we cover in detail the characteristics of the operator matrices found in PyFR, SIMD architectures, how matrix multiplication can be performed using them and we describe how the LIBXSMM JIT process is used within PyFR. Chapter 3 covers pieces of work where values where intricately packed within registers and where cross-lane communication was used to move data around. In Chapter 4 we outline the benchmarking process and how a synthetic suite of operator matrices was created to be used with a suite of example operator matrices encountered in PyFR applications, for the evaluation of solutions. Chapter 5 introduces two schemes to pack more unique values into vector registers that also allow for efficient runtime broadcasting, and in Chapter 6 we evaluate the performance against the reference LIBXSMM for both PyFR and synthetic operator matrices. Chapters 7, 8 and 9 detail further iterations on the base register packing scheme, where some make use of free registers, and we evaluate their performance using the benchmark. In Chapter 10 we compare a new hybrid strategy for LIBXSMM against GiMMiK by evaluating on both the PyFR and synthetic suite of operator matrices.

The following contributions were made in this thesis:

- We explored how to broadcast any selected vector lane, which leads to more unique non-zeros being able to be packed into the vector register file for the sparse-dense matrix multiplication routine within LIBXSMM. We evaluated the performance of kernels generated by this updated routine and report speedups between 0.976 and 10.915 times for some sparse PyFR operator matrices on the Intel Xeon 8175M.
- We compared with a suite of 170 matrices that arise in common PyFR applications, the performance of the new kernels against the characteristics of those operator matrices. We highlighted when the new kernels perform faster and when they perform slower than kernels generated by the reference LIBXSMM version.

- We presented a synthetic suite of operator matrices that are used to further evaluate the new code generator. By controlling the other characteristics of an operator matrix, we were able to confidently draw conclusions about the effect on performance when one characteristic varies at a time.

- We experimentally explored the impact on performance when free registers are used to increase the parallelism of the kernels, by working on accumulating multiple strides of C concurrently. We report speedups between 0.997 and 11.500 times using these optimisations over the reference LIBXSMM for some sparse PyFR operator matrices.

- We presented a hybrid strategy that uses heuristics to select between existing LIBXSMM routines and routines we developed, to achieve good performance based on the characteristics of the operator matrix. The heuristics were formed using the results from the evaluation on existing and new routines.

- We compared the hybrid strategy against GiMMiK and report speedups between 0.484 and 5.457 times on a suite of example operator matrices encountered in PyFR applications. For the sparse examples (quadrilateral and hexahedra based meshes) we report speedups of up to 5.457 times. When a solution accuracy of third-order and greater is used for dense examples (triangle and tetrahedra based meshes), we report speedups of up to 2.325 times.

The following contributions were made to Intel's open-source LIBXSMM as a result of this thesis:

- We submitted code [20] that was merged to the LIBXSMM repository on GitHub, that increases the number of unique non-zeros supported by the sparse-dense routine from 31 to 176 (224) for DP (SP) data types.

- We submitted code [21] that was merged to the LIBXSMM repository on GitHub, that further increases the number of unique non-zeros supported by the sparse-dense routine to 240 (480) for DP (SP) data types. The method was built on top of the first submission, but the former is preferred for fewer than 176 (224) DP (SP) unique non-zeros.

We believe that there are other subject areas, in and out of CFD, that could benefit from using the sparse-dense matrix multiplication routine within LIBXSMM. As we have contributed code to the open-source Intel library, adopters of the JIT compiler can benefit from the work in this thesis. Furthermore, the findings in this thesis can be used to support the application of register packing and runtime broadcasting for matrix multiplication implemented on other SIMD architectures, and potentially for other memory bound algorithms as well.

11.2 Further Work

Register Packing with AVX2

We have implemented various versions of register packing for the small sparse operator matrix when performing matrix multiplication with the AVX-512 instruction set. Currently, most x86 CPUs do not have support for AVX-512, only the higher end Intel workstation and server class CPUs do. This limits the range of target hardware that the new version of the sparse-dense matrix multiplication routine within LIBXSMM can be used on. Most modern x86 CPUs from both Intel and AMD do however support AVX2, the 256-bit SIMD architecture.

AVX2 also has good support for permutes and shuffles, allowing for cross-lane communication. However, it has *half* as many vector registers (16) and *half* the vector width compared to AVX-512.
The AVX2 version of *VPERMQ/PD* encodes the *selector* operand within the instruction, using 8-bits, similar to the AVX-512 *VSHUFF64X2* instruction. Whenever a permute or shuffle can only chose from 4 sources, it is possible to encode selections within 8-bits. In AVX-512, the *VPERMQ/PD* has 8 sources, hence a vector register is used to store the selector operand, not an encoded 8-bit integer. Interestingly, even though the vector width doubled from AVX2 to AVX-512, the width of the space for integer operands encoded into the vector instructions did not increase. This led to AVX-512 losing the ability to use permutes to broadcast DP values without having to store selector operands in vector registers.
By encoding the selection within the instruction, the basic register packing on AVX2 leaves 14 free registers, and so can pack 56 unique DP values. For SP in AVX2, *VPERMD/PS* requires the use of vector registers to store the selector operands, as there are 8 sources for each destination lane. This only leaves 6 vector registers for packing unique values, hence a total of 48 unique SP values. Table 11.1 summarises the number of unique values basic register packing can support via permute instructions, in AV2 and AVX-512. Table 11.2 shows the number of unique values that can be packed if the selector operands for the permute operands were to be read from L1 cache.

Data Type	AVX2	AVX-512
SP	48	224
DP	56	176

Table 11.1: Number of Unique Values that can be packed using permutes in AVX2 vs AVX-512

Data Type	AVX2	AVX-512
SP	112	480
DP	56*	240

Table 11.2: Number of Unique Values that can be packed using permutes with L1 selector operands in AVX2 vs AVX-512 (* AVX2 DP does not require selector operands to be stored as vectors)

However, in AVX2, each stride of B (4 sequential elements on the same row) does not take up an entire cache-line. So it would be interesting to see if the performance gains for the operator matrices that both would support, are significantly different to the performance gains for AVX-512.
This also shows that register packing could be applied to other SIMD architectures, possibly without the use of LIBXSMM for non x86 hardware, such as the ARM Scalable Vector Extension (SVE) or one of the RISC-V SIMD extensions. In the case of SVE where each implementation can have a different width, the use of a JIT is beneficial as the SIMD width only needs to be known at runtime. The main requirement to apply register packing on any SIMD architecture is the availability of *hardware supported cross-lane communication operations*. It would also be interesting to see if the register packing scheme could be adapted and applied for GPUs.

Hybrid Register Packing

When using multiple accumulators with register packing and L1 selector operands, there is a potential for the runtime broadcasting to hinder the performance. Steps can be taken to reduce this:

- Pre-Broadcast most commonly used values: After accounting for any selector operands and registers required for accumulators and runtime broadcasting, there could be free registers due to the operator matrices having fewer unique non-zero values than the maximum supported. The most commonly used non-zero values can be stored in a pre-broadcast format in a free register, and not stored with the other packed values. This should reduce the amount of runtime broadcasting in proportion to the relative frequency of this non-zero value appearing in the operator matrix.

- Store some Permute Selector Operands in registers and some in memory: To reduce read pressure on the L1 data cache, some of the selector operands for the permute operands can be stored in the registers, while other less frequently used ones can be stored in L1. The values with a higher frequency of appearing in the operator matrix could be packed in a way so that certain permute operands have a much higher frequency of use over others. An additional benefit of this is that more registers are available to use either to pack more unique non-zeros, or to use for other optimisations.

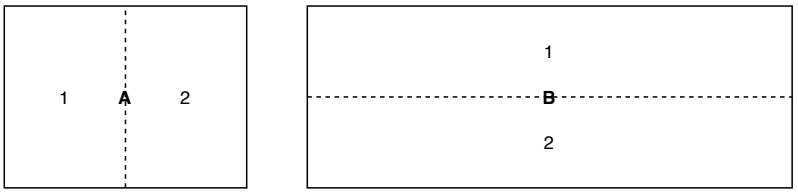
Use Free Registers to Store B

Alternatively, another use of free registers is to keep strides of B in a temporary register if they are heavily used. For example, an operator matrix where one column of A of is dense, but others are sparse, the corresponding stride of B for the dense column could be kept within a free temporary register. This could greatly reduce the number of reads from the L1 data cache. It is possible this could now lead to the runtime broadcasting adding to the critical path, where in basic register packing it wasn't.
As long as there are free registers due to using some form of register packing, there potentially exists multiple optimisations that can make use of them. A model could be made that considers the benefits to either reducing the critical path and/or reducing read pressure to memory. The model could also consider further increasing the number of accumulators and the effects that would have on the previous two factors, as well as the increase in parallelism. As all of these techniques make use of the free registers, a cost/weight could be associated with adding the optimisation. A minimum spanning tree could be calculated as part of the JIT process that selects the most beneficial uses of the free registers, in order to generate more optimal code for a given operator matrix.

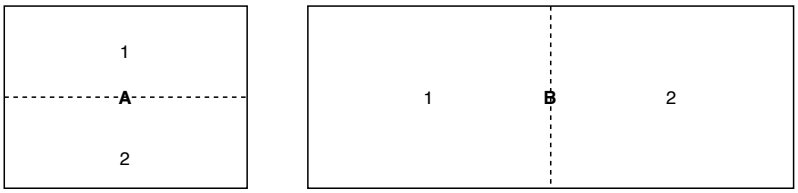
Tiling to Reduce Number of Unique Non-Zeros

When an operator matrix has more unique non-zeros than the amount supported by a given register packing scheme, LIBXSMM defaults to using a well tuned dense routine. However, if the matrix is small and sparse, this dense routine will generate kernels with slower performance than a register packing based routine (where the number of unique non-zeros was supported).
If the operator matrix is split into n tiles, then n smaller kernels could be generated for those respective tiles. If each tile has a number of unique non-zeros less than or equal to the maximum number supported by the register packing scheme, then the register packing routine could be used over the dense routine. There are few methods to logically split the operator matrix, each one requires a corresponding tiling scheme to be applied to the B

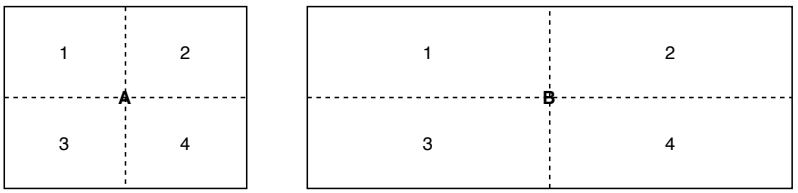
matrix, shown in Figure 11.1.
The first basic tiling scheme splits A vertically, shown in Figure 11.1a. This has the advantage that the tiling applied to B does not split a cache line, other than possibly at the final element, assuming row-major storage. However, the JIT process has to more carefully traverse the CSR layout of A. Figure 11.1b shows a simple horizontal tiling. While traversing the CSR layout becomes easier, now care has to be taken so that the tiling on B does not split cache lines. For AVX-512, each stride of B is a cache line, so this doesn't apply, but for other SIMD architectures it might. These can be combined together to split A even further, as shown in Figure 11.1c.
The drawback of using a basic approach, based on equal splits, is that no consideration is given to the locations and possible grouping of unique non-zeros in A. Park [13] presented a 'tailored tiling scheme' in 2016 that would create tiles by counting up to an amount of non-zero elements in the operator matrix, and then splitting when a threshold was reached. However, the uniqueness of the non-zero values was not considered. A more complex JIT process could analyse those factors and decide on a tiling that possibly uses *irregular* shapes to split the matrices. This is a more complex process but could yield good results.



(a) Vertically Split A



(b) Horizontally Split A



(c) Split A in two directions

Figure 11.1: Example tiling schemes for operator matrix

CSE and Graphs

The use of the free registers due to packing in this work has focused on how to execute a set number of FLOPS as fast as possible. Alternatively, the free registers could be used to reduce the required FLOPS by finding and eliminating redundant calculations by saving results in temporary (free) registers. This is known as common sub-expression elimination (CSE) and examples of related work can be found in [25] and [10]. We summarise a few possible CSE tactics:

- Column Wise CSE when there are non-zero elements in a column of A that are the same: The broadcasted value can be multiplied with a stride of B and stored in a temporary register. This result is then added to the corresponding strides of C. If

the value is repeated in n times in a column, then there would be a reduction of $n-1$ FLOPS. However, if using FMA, then the same number of FMA instructions would still be used, as the partial result is added to each stride of C. The benefit would come from having to perform $n-1$ fewer broadcasts and $n-1$ fewer reads of B by keeping the partial result within a register. For hardware without FMA support or where the FMA is slower than an addition, the resulting performance improvement could be even greater.

- Row Reordering without changing memory layout of A: Whilst possible to keep the temporary results from column wise CSE in a register for the whole kernel duration, it would be better to free up the register once no longer required, i.e. no more rows of A have that specific value in that specific column. The column wise CSE above can work without the values being sequentially repeated in a column. However, if they were, then once the kernel has passed the last row with that common value, the register used to store the partial result can be freed and used for other CSE opportunities.
  The matrix A could be re-ordered row-wise to find an optimal order where the common values in (multiple) columns are grouped together. However, changing the actual storage of A in memory can be avoided, by just traversing the rows of A in the new optimal order.

- Row Wise CSE to reduce the number of broadcasts: If sequential non-zero values of A are the same for a given row of A, then the number of required FLOPS can be reduced. If the value is repeated n times, then n fewer FLOPS can be used. By adding together the n corresponding strides of B ($n-1$ FLOPS) and then multiplying by the broadcasted value of A (1 FLOP), a total of n FLOPS are required with one additional free temporary register, compared to the $2n$ FLOPS the basic register packing routine would need. Again, if FMA hardware is available and just as fast, no performance would be gained from the reduction in FLOPS. However, this tactic of row wise CSE would reduce the number of runtime broadcasts by an amount of $n-1$. So for hardware with fast FMAs, this tactic can solely be used to issue fewer runtime broadcasts.

- Computing a minimum spanning tree to decide on which CSE options to take: All of the above tactics require the use of a free register, which is a limited resource. A model could be proposed that considers the changes in the number of broadcasts, the number of reads from memory and the FLOPS for the kernel. Then, a graph could be constructed where a new node is formed by applying a single CSE tactic to a given row or column. By using directed edges to connect nodes, the graph construction could avoid creating edges where CSE tactics conflict with each other. Finally, a minimum/maximum (depending on the edge weight calculation) spanning tree could be obtained, to inform the code generation which specific CSE tactics to apply.
  The use of free registers for 'Hybrid Register Packing' and 'Use Free Registers to Store B' could also be considered as an alternative use of the registers in this model, with the cost/benefit being calculated for those uses as well. To further increase the model complexity, the use of multiple accumulators could be considered for the free registers, if not already utilised.

Bibliography

[1] R. Kastner A. Hosangadi, F. Fallah. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:2012 – 2022, 2006.

[2] E. Biham. A fast new des implementation in software. In E. Biham, editor, *Fast Software Encryption*, pages 260–272, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[3] Intel Corporation. LIBXSMM, 2020 (Accessed 05/01/20). `https://libxsmm.readthedocs.io`.

[4] Intel Corporation. Math Kernel Library, 2020 (Accessed 05/01/20. `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`.

[5] NVIDIA Corporation. cuSPARSE, 2020 (Accessed 05/01/20). `https://developer.nvidia.com/cusparse`.

[6] Daytime. SIMD instruction list, December 2019 (Accessed 30/03/20). `https://www.officedaytime.com/simd512e/simdimg/si.php?f=vshufi64x2`.

[7] P.E. Vincent F.D. Witherden, A.M. Farrington. PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185:3028–3040, 2014.

[8] P.E. Vincent F.D. Witherden, B.C. Vermeire. Heterogeneous computing on mixed unstructured grids with PyFR. *Computers and Fluids*, 120:173–186, 2015.

[9] A. Fog. Instruction tables, 2019 (Accessed 30/04/20). `https://www.agner.org/optimize/instruction_tables.pdf`.

[10] P.H.J. Kelly F.P. Russel. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software*, 39, 2013.

[11] H. Zang D.H. Park H. Zhang, X. Cheng. Compiler-level matrix multiplication optimization for deep learning. *ArXiv*, abs/1909.10616, 2019.

[12] H.T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous galerkin methods. *AIAA paper*, 4079, 2007.

[13] W. Wen P.T.P. Tang H. Li Y. Chen P. Dubey J. Park, S. Li. Faster CNNs with direct sparse convolutions and guided pruning. *arXiv*, 1608.01409, 2016.

[14] I. Cutress J.D. Gelas. Sizing up servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - the server CPU battle of the decade?, Jul 2017 (Accessed 30/04/20). `https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade/12`.

[15] R.A. Rutenbar Rob J.Y.F. Tong, D. Nagle. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8:273 – 286, 2000.

[16] W.R. Mark M. Pharr. ispc: A SPMD compiler for high-performance CPU programming. *Innovative Parallel Computing*, pages 1–13, 05 2012.

[17] J. N. P. Martel, M. Chau, M. Cook, and P. Dudek. Pixel interlacing to trade off the resolution of a cellular processor array against more registers. In *2015 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, 2015.

[18] R. Neely. Fast chapter 16 surferu, 1999 (Accessed 07/05/20). `https://www.nas.nasa.gov/Software/FAST/RND-93-010.walatka-clucas/htmldocs/chp_16.surferu.html`.

[19] P.E. Vincent M.R.L. Morales P. Castonguay, D.M. Williams. On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids. *20th AIAA Computational Fluid Dynamics Conference*, 2011.

[20] M.V. Paribartan. Issue #378: Add run-time broadcasting to csr_asparse_reg using permutes, 2020 (Accessed 13/05/20). `https://github.com/hfp/libxsmm/pull/381`.

[21] M.V. Paribartan. Issue #385: Add support to load permute selector operands from memory in csr_asparse_reg, 2020 (Accessed 15/06/20). `https://github.com/hfp/libxsmm/pull/386`.

[22] J.M. Park. Full unrolling combined with tiling for small sparse matrix multiplies in fluid dynamics. *MEng Thesis, Department of Computing, Imperial College London*, 2016.

[23] T. Price. Optimising small sparse special matrix multiplies through cunning vectorisation. *MEng Thesis, Department of Computing, Imperial College London*, 2019.

[24] H. Dietz R. Fisher. Compiling for SIMD within a register. *Languages and Compilers for Parallel Computing*, pages 290–304, 1998.

[25] L.R. Scott A.R. Terrel R.C. Kirby, A. Logg. Topological optimization of the evaluation of finite element matrices. *Society for Industrial and Applied Mathematics*, 28:224–240, 2006.

[26] B.C. Kuszmaul C.E. Leiserson K.H. Randall Y. Zhou R.D. Blumofe, C.F. Joerg. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[27] M. Roth. Compiler optimisations in high-performance matrix multiplication kernels. *MEng Thesis, Department of Computing, Imperial College London*, 2015.

[28] P. Vandermersch J. Cohen J. Tran B. Catanzaro E. Shelhamer S. Chetlur, C. Woolley. cuDNN: Efficient primitives for deep learning. *arXiv*, 1410.0759, 2014.

[29] D. Gregg S. Xu. Customizable precision of floating-point arithmetic with bitslice vector types. *arXiv*, 1602.04716, 2016.

[30] A. Turner. UK National HPC Benchmarks, 2016 (Accessed 15/06/20). `https://www.archer.ac.uk/documentation/white-papers/benchmarks/UK_National_HPC_Benchmarks.pdf`.

[31] Z.J. Wang, Y. Liu, C. Lacor, and J.L.F. Azevedo. Chapter 9 - spectral volume and spectral difference methods. In C. Shu R. Abgrall, editor, *Handbook of Numerical Methods for Hyperbolic Problems*, volume 17 of *Handbook of Numerical Analysis*, pages 199 – 226. Elsevier, 2016.

[32] F.D. Witherden. GiMMiK 2.1 Release, 2016 (Accessed 04/06/20). `https://github.com/PyFR/GiMMiK/releases/tag/v2.1`.

[33] F.D. Witherden, P.E. Vincent, and A. Jameson. Chapter 10 - high-order flux reconstruction schemes. In C. Shu R. Abgrall, editor, *Handbook of Numerical Methods for Hyperbolic Problems*, volume 17 of *Handbook of Numerical Analysis*, pages 227 – 263. Elsevier, 2016.

[34] B.D. Wozniak. Gimmik-generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics. *MEng Thesis, Department of Computing, Imperial College London*, 2014.

Appendix A

PyFR Example Operator Matrices Characteristics

The follwing tables describe the example operator matrices from PyFR in terms of:

- Number of Rows, R
- Number of Columns, C
- Density, ρ
- Number of unique non-zeros, U

Matrix	R	C	ρ	U
m0	8	4	0.5000	2
m3	4	8	0.5000	2
m6	8	8	0.2500	4
m132	4	8	0.5000	2
m460	8	4	0.5000	2

(a) First Order

Matrix	R	C	ρ	U
m0	12	9	0.3333	3
m3	9	12	0.3333	3
m6	18	12	0.1667	6
m132	9	18	0.2963	8
m460	18	9	0.2963	8

(b) Second Order

Matrix	R	C	ρ	U
m0	16	16	0.2500	4
m3	16	16	0.2500	4
m6	32	16	0.1250	8
m132	16	32	0.2500	16
m460	32	16	0.2500	16

(c) Third Order

Matrix	R	C	ρ	U
m0	20	25	0.2000	5
m3	25	20	0.2000	5
m6	50	20	0.1000	10
m132	25	50	0.1920	24
m460	50	25	0.1920	24

(d) Fourth Order

Matrix	R	C	ρ	U
m0	24	36	0.1667	6
m3	36	24	0.1667	6
m6	72	24	0.0833	12
m132	36	72	0.1667	36
m460	72	36	0.1667	36

(e) Fifth Order

Matrix	R	C	ρ	U
m0	28	49	0.1429	7
m3	49	28	0.1429	7
m6	98	28	0.0714	14
m132	49	98	0.1399	48
m460	98	49	0.1399	48

(f) Sixth Order

Table A.1: Quadrilateral Gauss-Legendre Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	12	9	0.1111	1
m3	9	12	0.3333	3
m6	18	12	0.1667	6
m132	9	18	0.2963	6
m460	18	9	0.2963	6

(a) Second Order

Matrix	R	C	ρ	U
m0	16	16	0.0625	1
m3	16	16	0.2500	4
m6	32	16	0.1250	6
m132	16	32	0.2188	14
m460	32	16	0.2188	14

(b) Third Order

Matrix	R	C	ρ	U
m0	20	25	0.0400	1
m3	25	20	0.2000	4
m6	50	20	0.1000	8
m132	25	50	0.1760	22
m460	50	25	0.1760	22

(c) Fourth Order

Matrix	R	C	ρ	U
m0	24	36	0.0278	1
m3	36	24	0.1667	6
m6	72	24	0.0833	8
m132	36	72	0.1481	32
m460	72	36	0.1481	32

(d) Fifth Order

Matrix	R	C	ρ	U
m0	28	49	0.0204	1
m3	49	28	0.1429	5
m6	98	28	0.0714	10
m132	49	98	0.1283	44
m460	98	49	0.1283	44

(e) Sixth Order

Table A.2: Quadrilateral Gauss-Legendre-Lobatto Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	24	8	0.2500	2
m3	8	24	0.2500	2
m6	24	24	0.0833	4
m132	8	24	0.2500	2
m460	24	8	0.2500	2

(a) First Order

Matrix	R	C	ρ	U
m0	54	27	0.1111	3
m3	27	54	0.1111	3
m6	81	54	0.0370	6
m132	27	81	0.0988	8
m460	81	27	0.0988	8

(b) Second Order

Matrix	R	C	ρ	U
m0	96	64	0.0625	4
m3	64	96	0.0625	4
m6	192	96	0.0208	8
m132	64	192	0.0625	16
m460	192	64	0.0625	16

(c) Third Order

Matrix	R	C	ρ	U
m0	150	125	0.0400	5
m3	125	150	0.0400	5
m6	375	150	0.0133	10
m132	125	375	0.0384	24
m460	375	125	0.0384	24

(d) Fourth Order

Matrix	R	C	ρ	U
m0	216	216	0.0278	6
m3	216	216	0.0278	6
m6	648	216	0.0093	12
m132	216	648	0.0278	36
m460	648	216	0.0278	36

(e) Fifth Order

Matrix	R	C	ρ	U
m0	294	343	0.0204	7
m3	343	294	0.0204	7
m6	1029	294	0.0068	14
m132	343	1029	0.0200	48
m460	1029	343	0.0200	48

(f) Sixth Order

Table A.3: Hexahedra Gauss-Legendre Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	54	27	0.0370	1
m3	27	54	0.1111	3
m6	81	54	0.0370	6
m132	27	81	0.0988	6
m460	81	27	0.0988	6

(a) Second Order

Matrix	R	C	ρ	U
m0	96	64	0.0156	1
m3	64	96	0.0625	4
m6	192	96	0.0208	6
m132	64	192	0.0547	14
m460	192	64	0.0547	14

(b) Third Order

Matrix	R	C	ρ	U
m0	150	125	0.0080	1
m3	125	150	0.0400	4
m6	375	150	0.0133	8
m132	125	375	0.0352	22
m460	375	125	0.0352	22

(c) Fourth Order

Matrix	R	C	ρ	U
m0	216	216	0.0046	1
m3	216	216	0.0278	6
m6	648	216	0.0093	8
m132	216	648	0.0247	32
m460	648	216	0.0247	32

(d) Fifth Order

Matrix	R	C	ρ	U
m0	294	343	0.0029	1
m3	343	294	0.0204	5
m6	1029	294	0.0068	10
m132	343	1029	0.0183	44
m460	1029	343	0.0183	44

(e) Sixth Order

Table A.4: Hexahedra Gauss-Legendre-Lobatto Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	6	3	1.0000	3
m3	3	6	1.0000	6
m6	6	6	0.6667	6
m132	3	6	0.6667	2
m460	6	3	0.6667	2

(a) First Order

Matrix	R	C	ρ	U
m0	9	6	1.0000	10
m3	6	9	1.0000	20
m6	12	9	0.6667	20
m132	6	12	0.8889	28
m460	12	6	0.8889	28

(b) Second Order

Matrix	R	C	ρ	U
m0	12	10	1.0000	20
m3	10	12	1.0000	40
m6	20	12	0.6667	40
m132	10	20	0.9600	96
m460	20	10	0.9600	96

(c) Third Order

Matrix	R	C	ρ	U
m0	15	15	1.0000	39
m3	15	15	1.0000	78
m6	30	15	0.6667	78
m132	15	30	0.9600	216
m460	30	15	0.9600	216

(d) Fourth Order

Matrix	R	C	ρ	U
m0	18	21	1.0000	63
m3	21	18	1.0000	126
m6	42	18	0.6667	126
m132	21	42	0.9796	432
m460	42	21	0.9796	432

(e) Fifth Order

Matrix	R	C	ρ	U
m0	21	28	1.0000	100
m3	28	21	1.0000	200
m6	56	21	0.6667	200
m132	28	56	0.9796	768
m460	56	28	0.9796	768

(f) Sixth Order

Table A.5: Triangles Williams-Shunn Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	12	4	1.0000	3
m3	4	12	1.0000	6
m6	12	12	0.5000	6
m132	4	12	0.5000	2
m460	12	4	0.5000	2

(a) First Order

Matrix	R	C	ρ	U
m0	24	10	1.0000	14
m3	10	24	1.0000	28
m6	30	24	0.5000	28
m132	10	30	0.8400	46
m460	30	10	0.8400	46

(b) Second Order

Matrix	R	C	ρ	U
m0	40	20	1.0000	40
m3	20	40	1.0000	80
m6	60	40	0.5000	80
m132	20	60	0.9100	200
m460	60	20	0.9100	200

(c) Third Order

Matrix	R	C	ρ	U
m0	60	35	1.0000	101
m3	35	60	1.0000	202
m6	105	60	0.5000	202
m132	35	105	0.9339	608
m460	105	35	0.9339	608

(d) Fourth Order

Matrix	R	C	ρ	U
m0	84	56	1.0000	214
m3	56	84	1.0000	428
m6	168	84	0.5000	428
m132	56	168	0.9541	1568
m460	168	56	0.9541	1568

(e) Fifth Order

Matrix	R	C	ρ	U
m0	112	84	1.0000	425
m3	84	112	1.0000	850
m6	252	112	0.5000	850
m132	84	252	0.9637	3520
m460	252	84	0.9637	3520

(f) Sixth Order

Table A.6: Tetrahedra Shunn-Ham Operator Matrices Characteristics

Appendix B

LIBXSMM Code Buffer Size

LIBXSMM has a maximum code buffer size of 128 KB (v1.15 [3]). This limit is set to prevent the JIT process from writing too large a function for the user process. If the limit was reached, a fallback strategy is used. If the limit is reached in the case of A of being sparse, then LIBXSMM v1.15 falls back to a dense matrix multiply routine.
However, in our evaluation, the aim was to compare the algorithms used to generate the code, and not necessarily what would be best for the general user in a production release of LIBXSMM. For that reason, the code buffer size was doubled to 256 KB, to be able to handle larger loop unrolling in Chapter 6. This removed buffer size limits for 5 operator matrices in the benchmark.
The following list details which matrices ran into the previous 128 KB limit, when using the Register Packing update from Chapter 5:

- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre $m132$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre $m460$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre-Lobatto $m132$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre-Lobatto $m460$

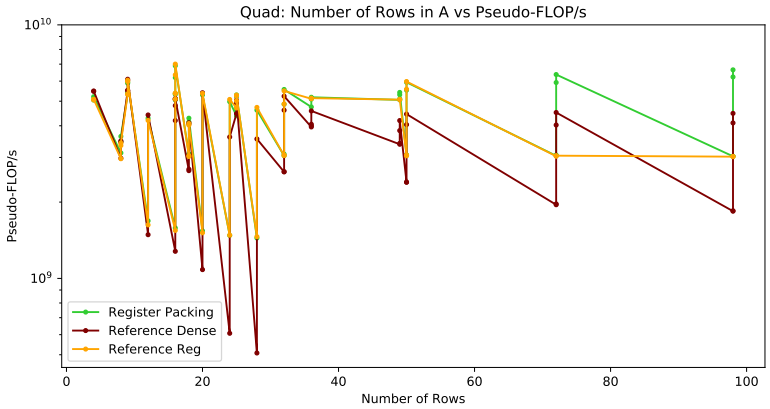
These were the largest size operator matrices in the benchmark, all having a size of $1029 * 343$.
Only one matrix from the synthetic suite ran into the buffer size limit, when using the reference LIBXSMM version:

- Synthetic: Density of 0.5, with 64 unique non-zeros. Size of $128 * 128$

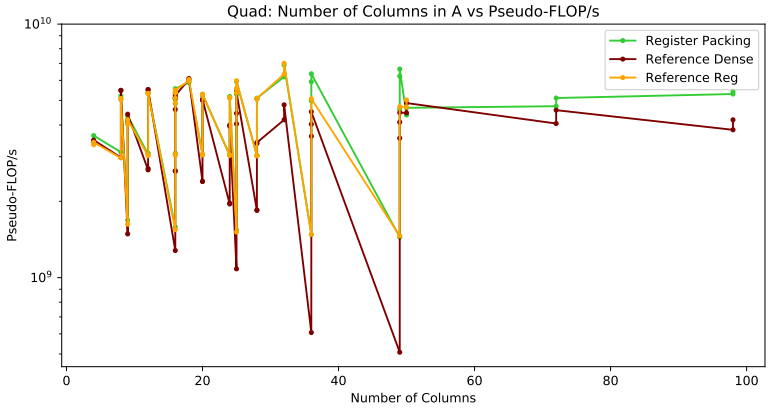
For Chapters 7, 8 and 9, the code buffer size was increased by a factor of 8, to 1 MB. This provided sufficient space for the additional instructions being issued by the updated routines that were evaluated in those chapters.

Appendix C

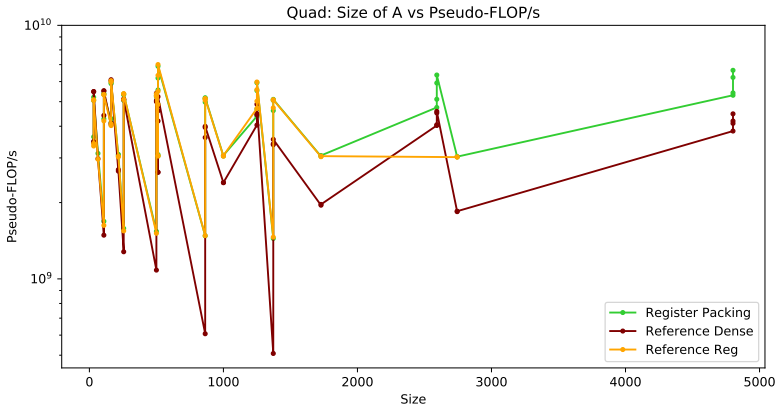
Register Packing Evaluation on PyFR Suite Additional Plots



(a) Plotting against Number of Rows

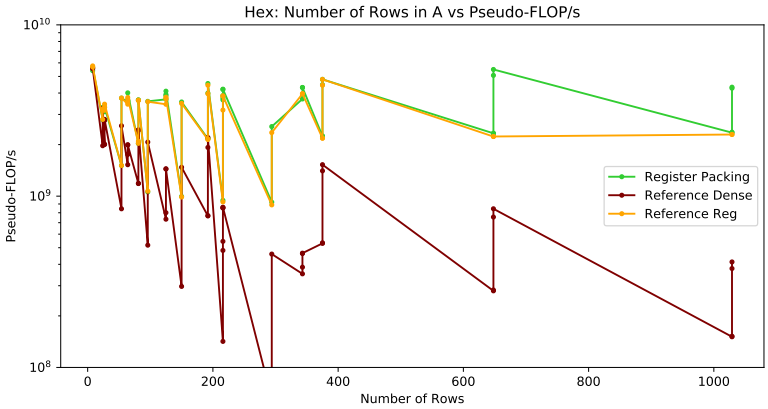


(b) Plotting against Number of Columns

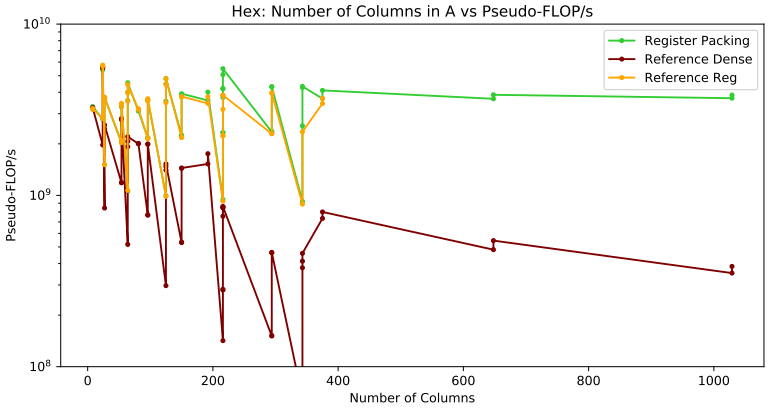


(c) Plotting against Size

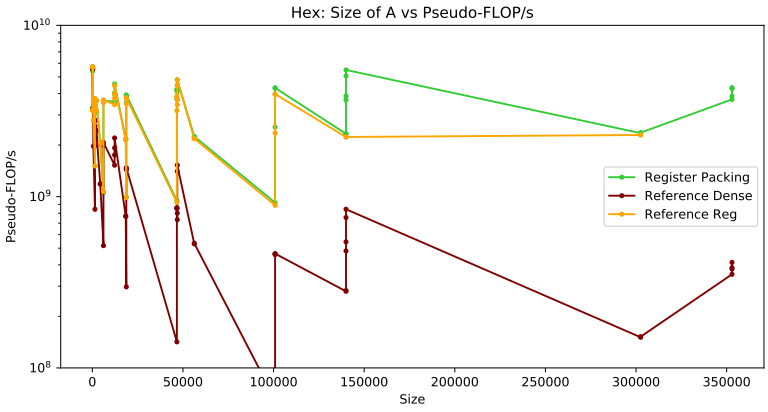
Figure C.1: Register Packing vs Reference Performance - PyFR Quadrilateral Examples



(a) Plotting against Number of Rows

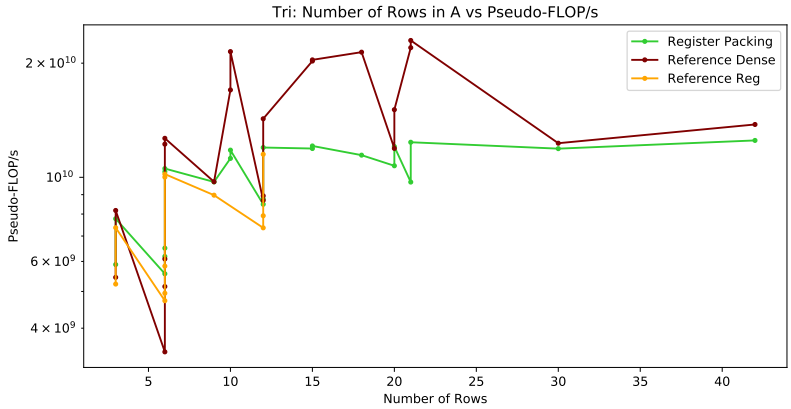


(b) Plotting against Number of Columns

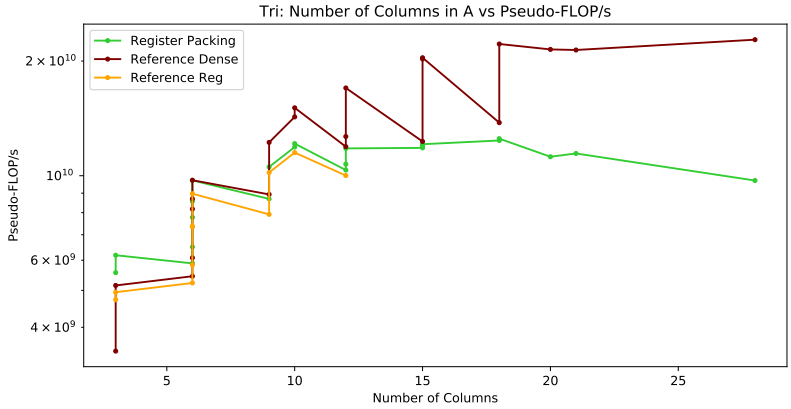


(c) Plotting against Size

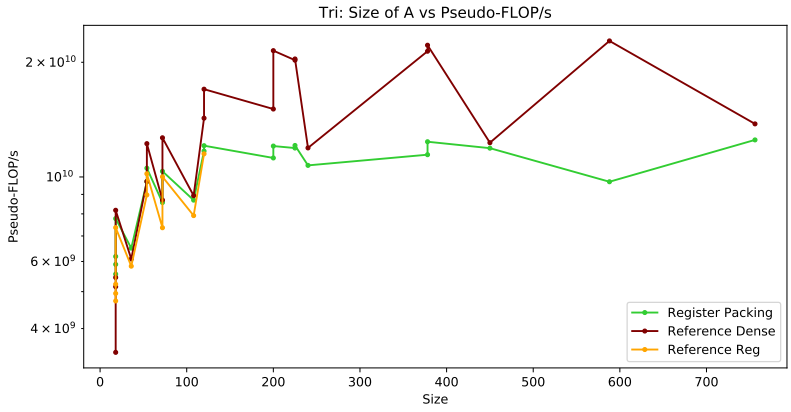
Figure C.2: Register Packing vs Reference Performance - PyFR Hexahedra Examples



(a) Plotting against Number of Rows

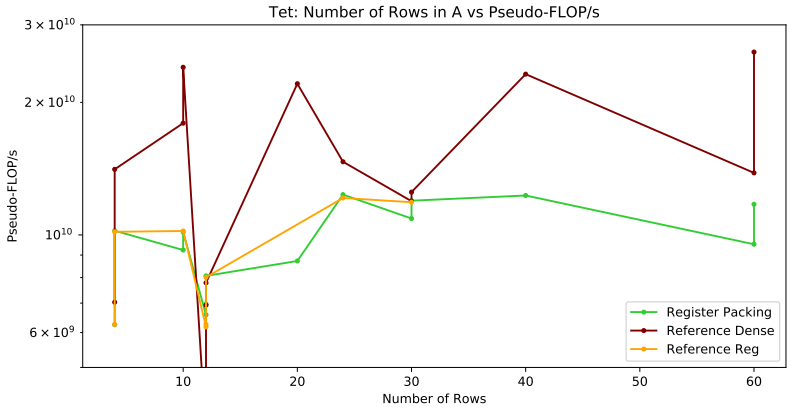


(b) Plotting against Number of Columns

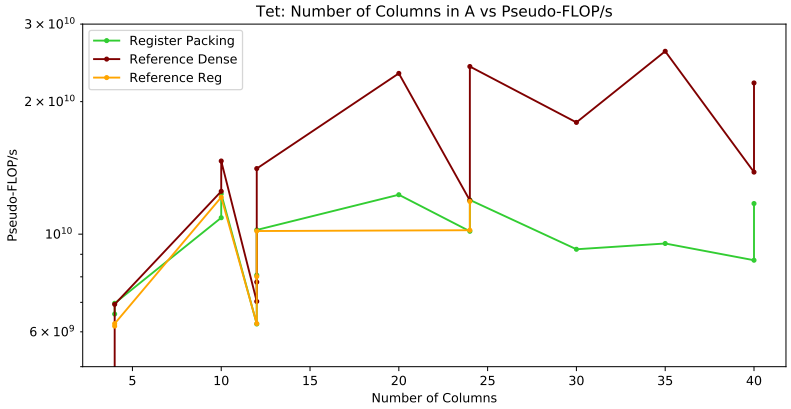


(c) Plotting against Size

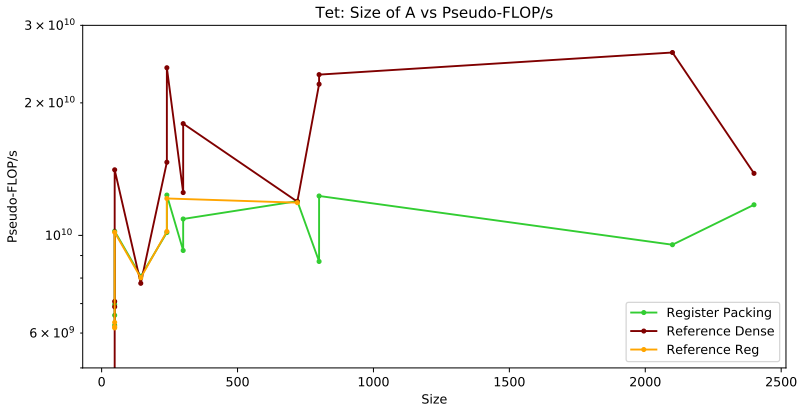
Figure C.3: Register Packing vs Reference Performance - PyFR Triangles Examples



(a) Plotting against Number of Rows



(b) Plotting against Number of Columns



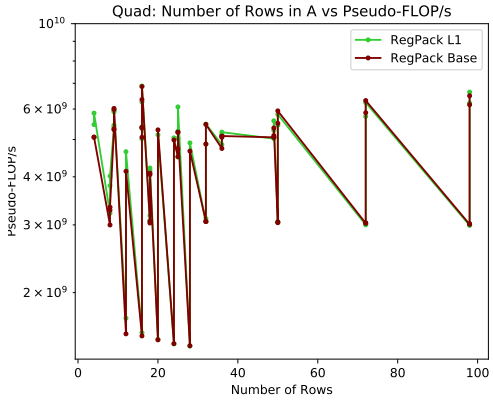
(c) Plotting against Size

Figure C.4: Register Packing vs Reference Performance - PyFR Tetrahedra Examples

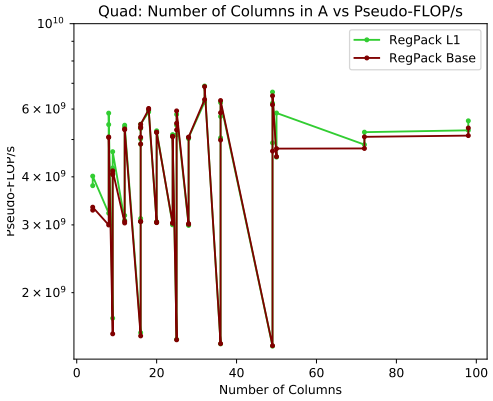
Appendix D

Register Packing L1 Evaluation on PyFR Suite Additional Plots

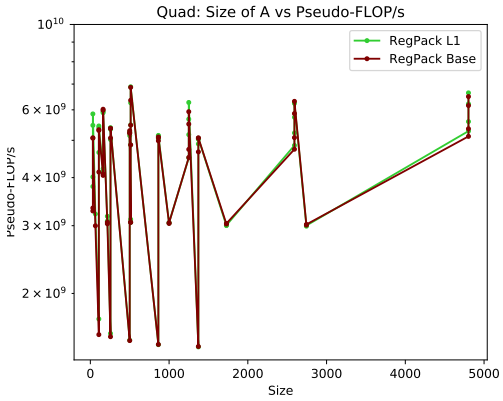
D.1 Against RP Base



(a) Plotting against Number of Rows

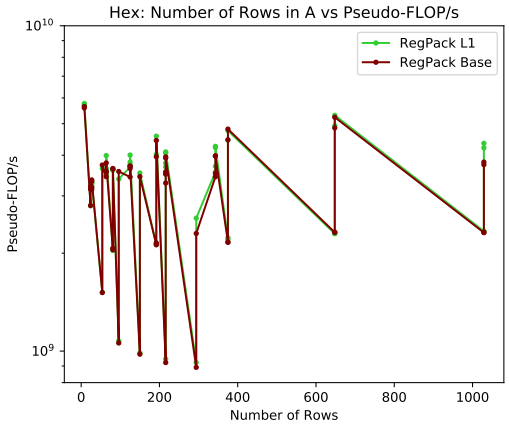


(b) Plotting against Number of Columns

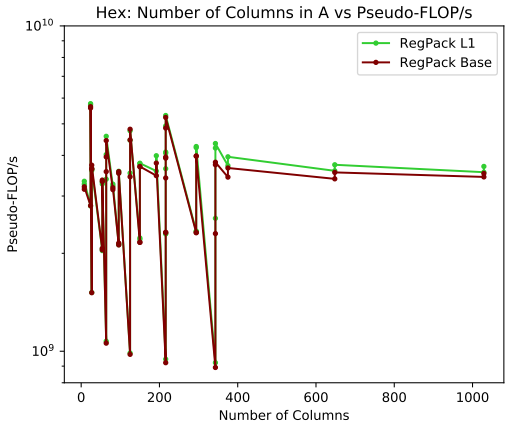


(c) Plotting against Size

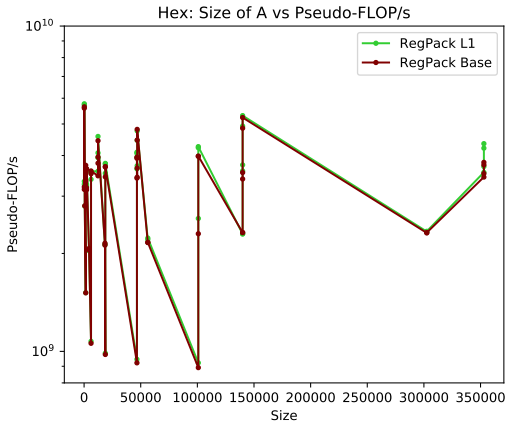
Figure D.1: Register Packing L1 vs Base Performance - PyFR Quadrilateral Examples



(a) Plotting against Number of Rows

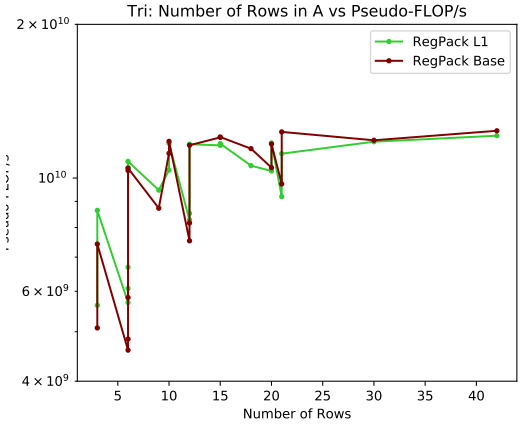


(b) Plotting against Number of Columns

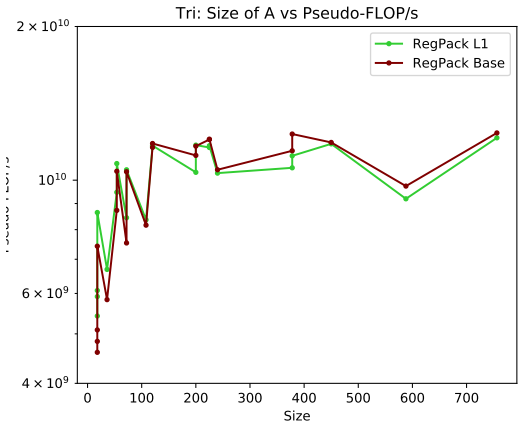


(c) Plotting against Size

Figure D.2: Register Packing L1 vs Base Performance - PyFR Hexahedra Examples

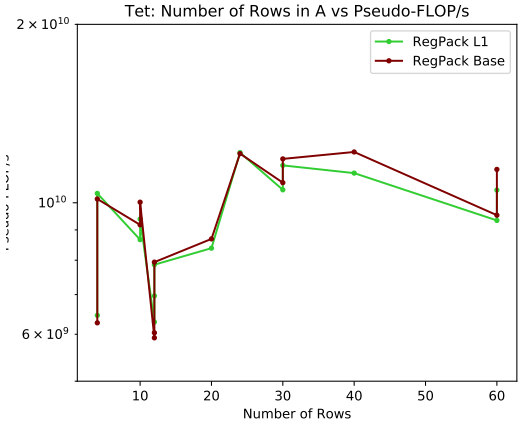


(a) Plotting against Number of Rows

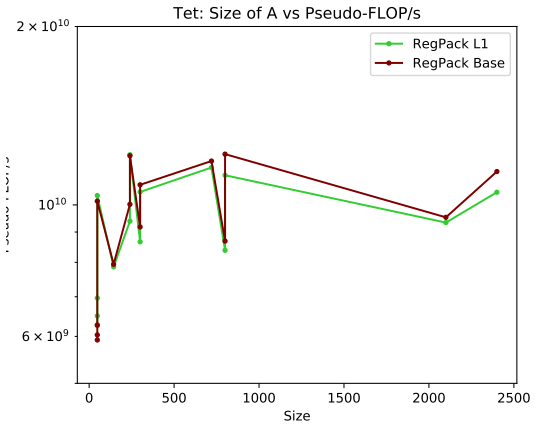


(b) Plotting against Size

Figure D.3: Register Packing L1 vs Base Performance - PyFR Triangles Examples



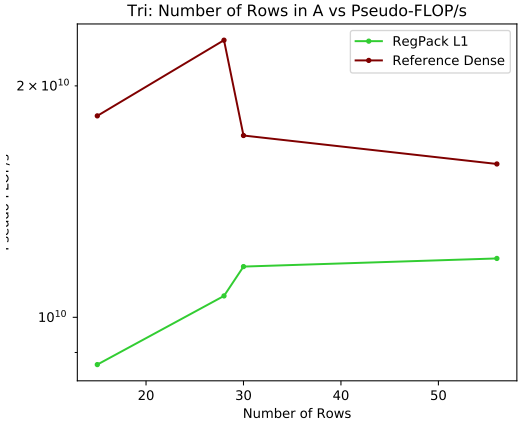
(a) Plotting against Number of Rows



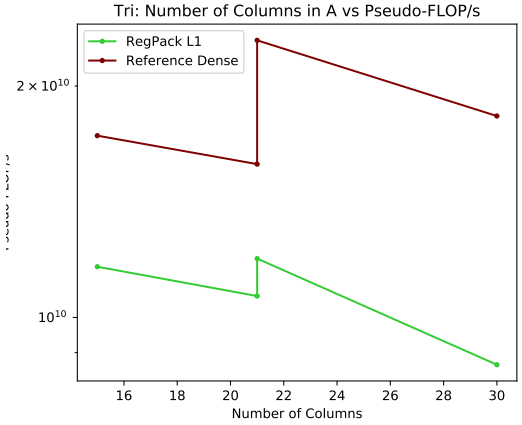
(b) Plotting against Size

Figure D.4: Register Packing L1 vs Base Performance - PyFR Tetrahedra Examples

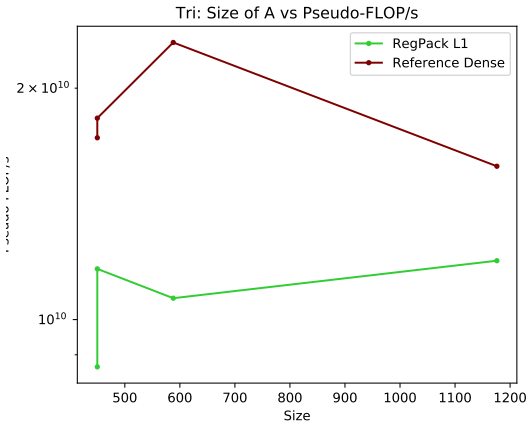
D.2 Against Reference Dense Routine



(a) Plotting against Number of Rows

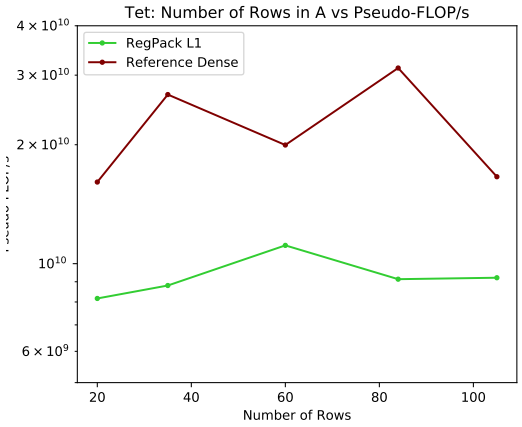


(b) Plotting against Number of Columns

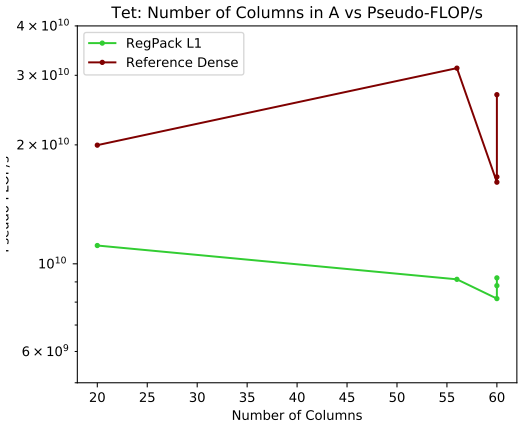


(c) Plotting against Size

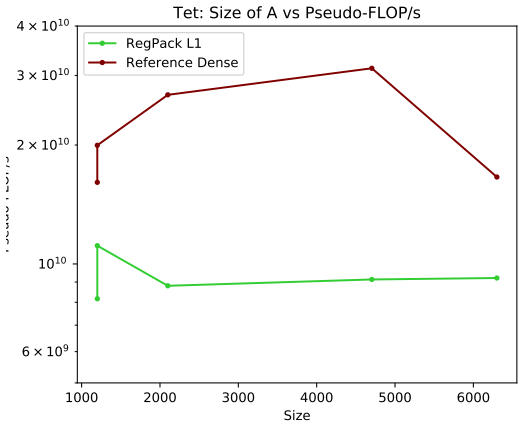
Figure D.5: Register Packing L1 vs Dense Performance - PyFR Triangles Examples



(a) Plotting against Number of Rows



(b) Plotting against Number of Columns



(c) Plotting against Size

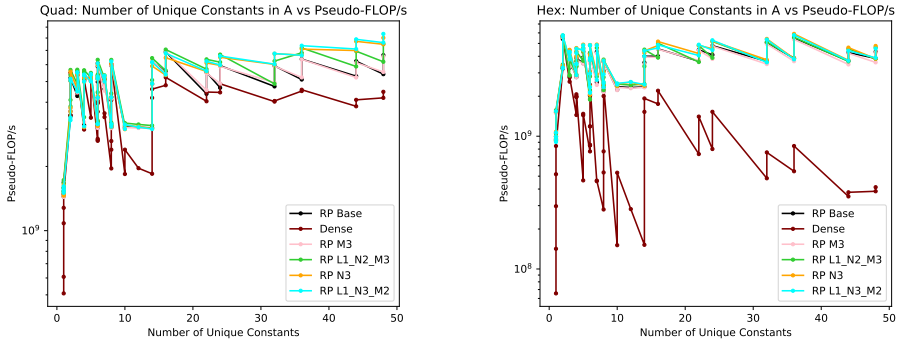
Figure D.6: Register Packing L1 vs Dense Performance - PyFR Tetrahedra Examples

Appendix E

Register Packing Combination Extra Evaluation Plots

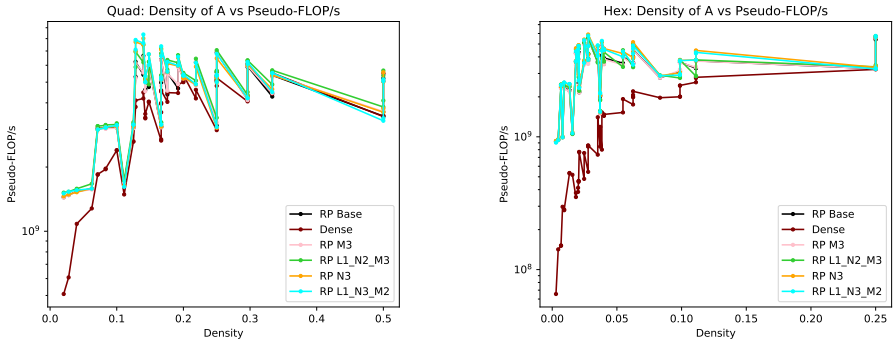
E.1 PyFR Suite

Sparse Operator Matrices



(a) Quadrilaterals (b) Hexahedra

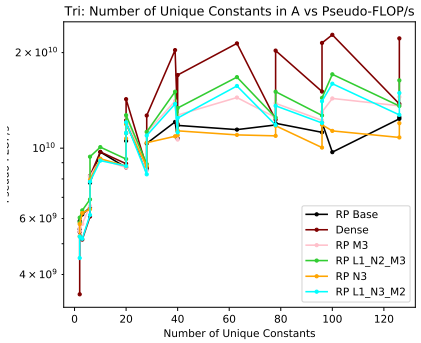
Figure E.1: Register Packing Combination - PyFR Sparse Examples (Number of Unique Non-Zeros)



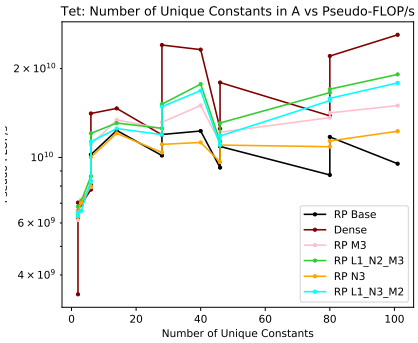
(a) Quadrilaterals (b) Hexahedra

Figure E.2: Register Packing Combination - PyFR Sparse Examples (Density)

Dense Operator Matrices

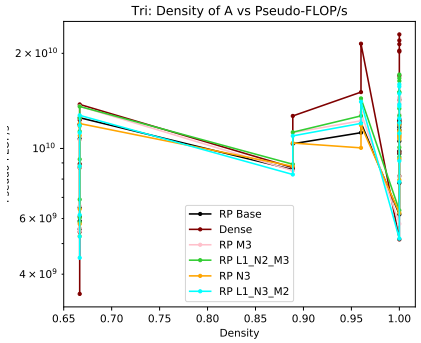


(a) Triangles

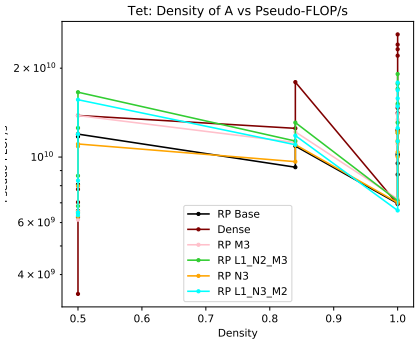


(b) Tetrahedra

Figure E.3: Register Packing Combination - PyFR Dense Examples (Number of Unique Non-Zeros)



(a) Triangles

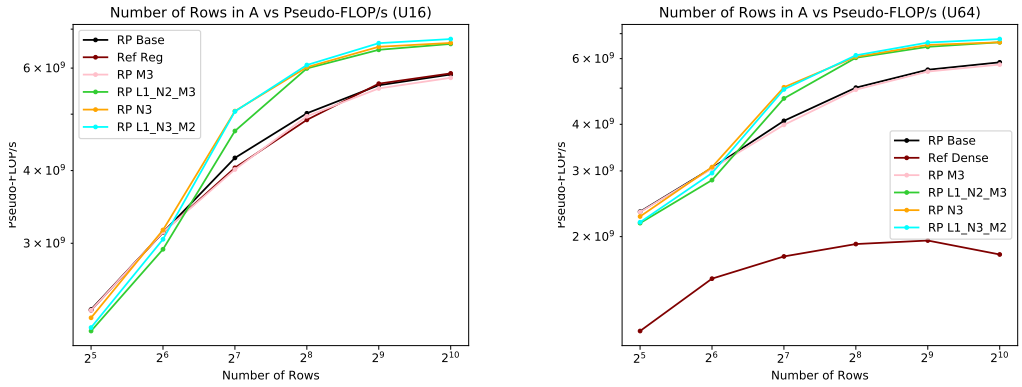


(b) Tetrahedra

Figure E.4: Register Packing Combination - PyFR Dense Examples (Density)

E.2 Synthetic Suite

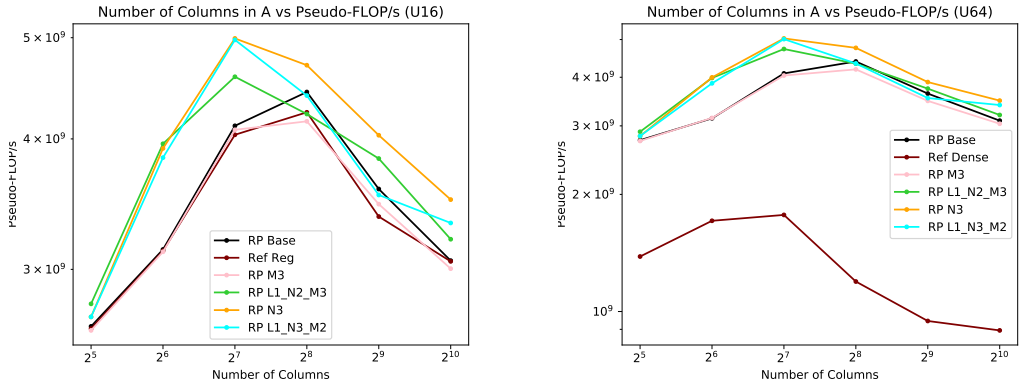
Vary Number of Rows



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure E.5: Register Packing Combination: Performance vs Number of Rows

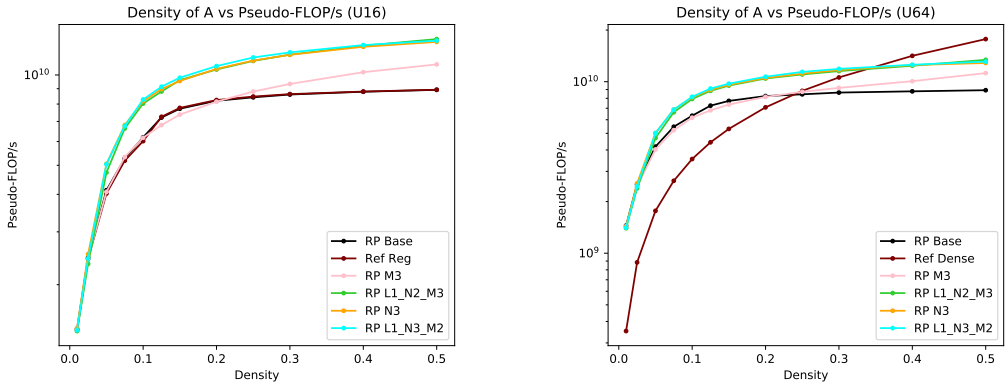
Vary Number of Columns



(a) 16 Unique Non-Zeros (b) 64 Unique Non-Zeros

Figure E.6: Register Packing Combination: Performance vs Number of Columns

Vary Number of Density



(a) 16 Unique Non-Zeros

(b) 64 Unique Non-Zeros

Figure E.7: Register Packing Combination: Performance vs Density

Vary Number of Unique Non-Zeros

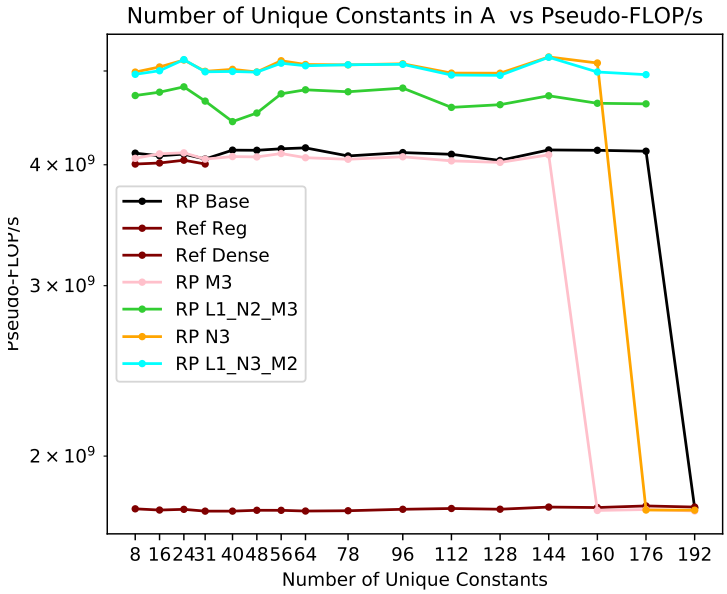
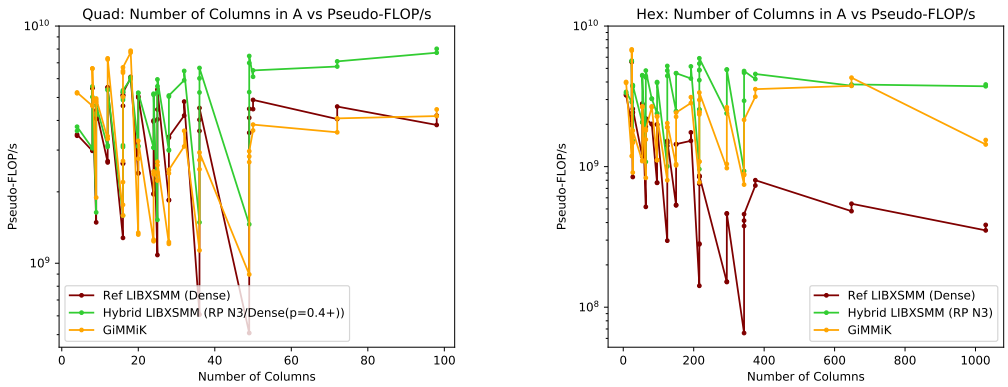


Figure E.8: Register Packing Combination: Performance vs Number of Unique Non-Zeros

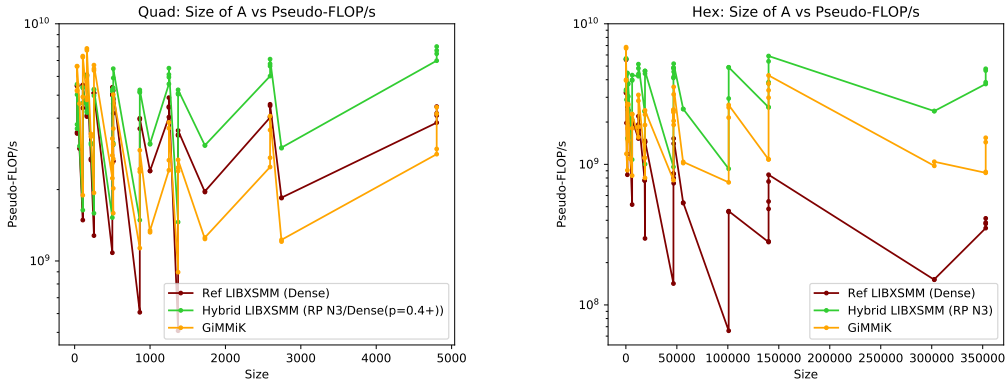
Appendix F

Hybrid LIBXSMM vs GiMMiK Evaluation on PyFR Suite Additional Plots



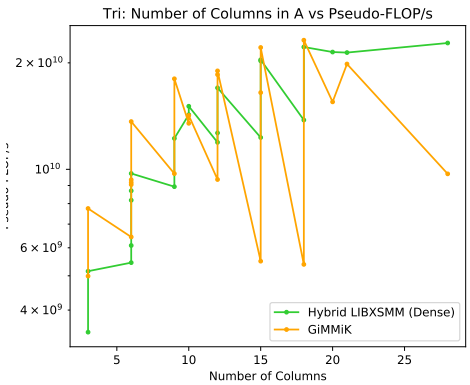
(a) Quadrilaterals (b) Hexahedra

Figure F.1: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Number of Columns)

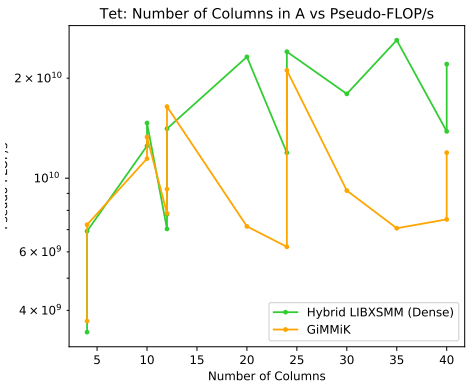


(a) Quadrilaterals (b) Hexahedra

Figure F.2: Hybrid LIBXSMM vs GiMMiK - PyFR Sparse Examples (Size)

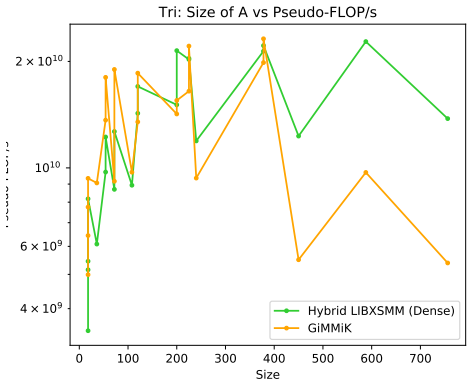


(a) Triangles

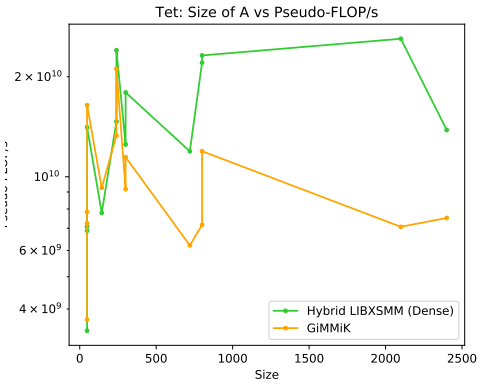


(b) Tetrahedra

Figure F.3: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Number of Columns)



(a) Triangles



(b) Tetrahedra

Figure F.4: Hybrid LIBXSMM vs GiMMiK - PyFR Dense Examples (Size)