

NANDA: the Next Frontier

Wayne Luk

Imperial College London, United Kingdom

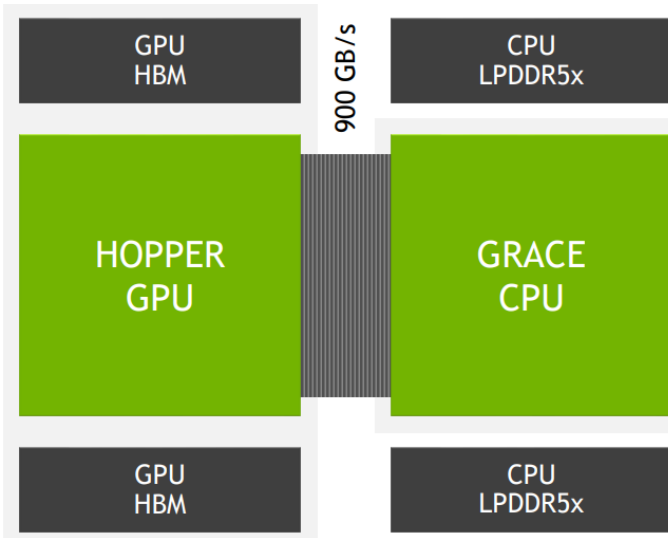
NANDA 2022

Acknowledgement: Jessica Vandebon, José G. F. Coutinho, Eriko Nurvitadhi, Stewart Denholm
EPSRC, SRC. AMD, Intel

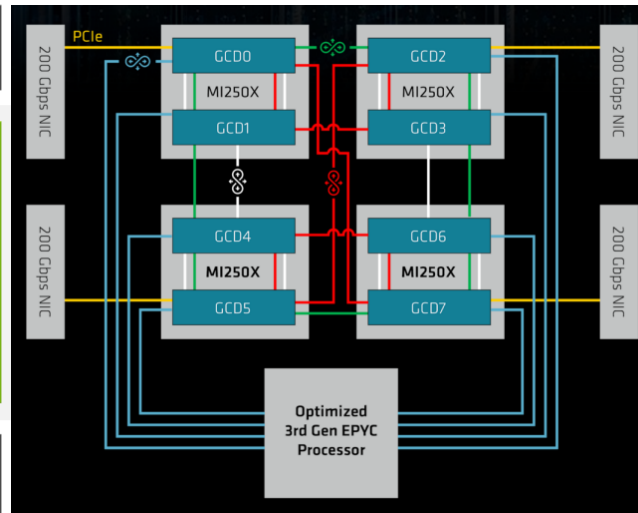
Agenda

- A. Context, Problem, Challenges
- B. Approach: Meta-Programming Design-Flow Patterns
- C. Design-Flow Pattern Catalogue For CPU and GPU Optimisation
- D. Implementing Design-Flow Patterns as Meta-Programs
- E. Evaluation: Automated Design-Flow Performance + Reusability
- F. Ongoing Work and Big Picture

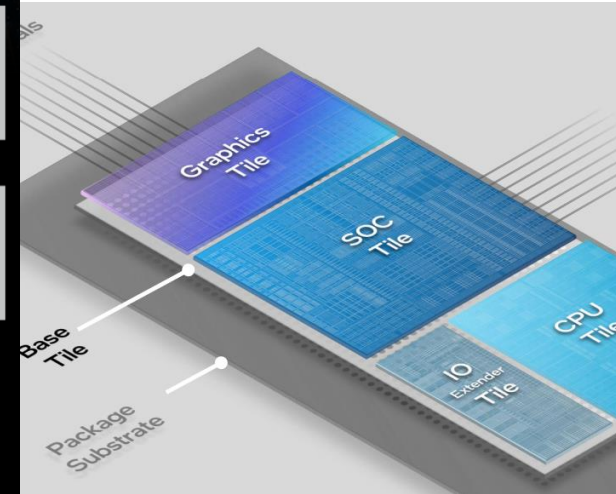
Processor Architecture: Heterogeneous Trend



NVIDIA GRACE HOPPER

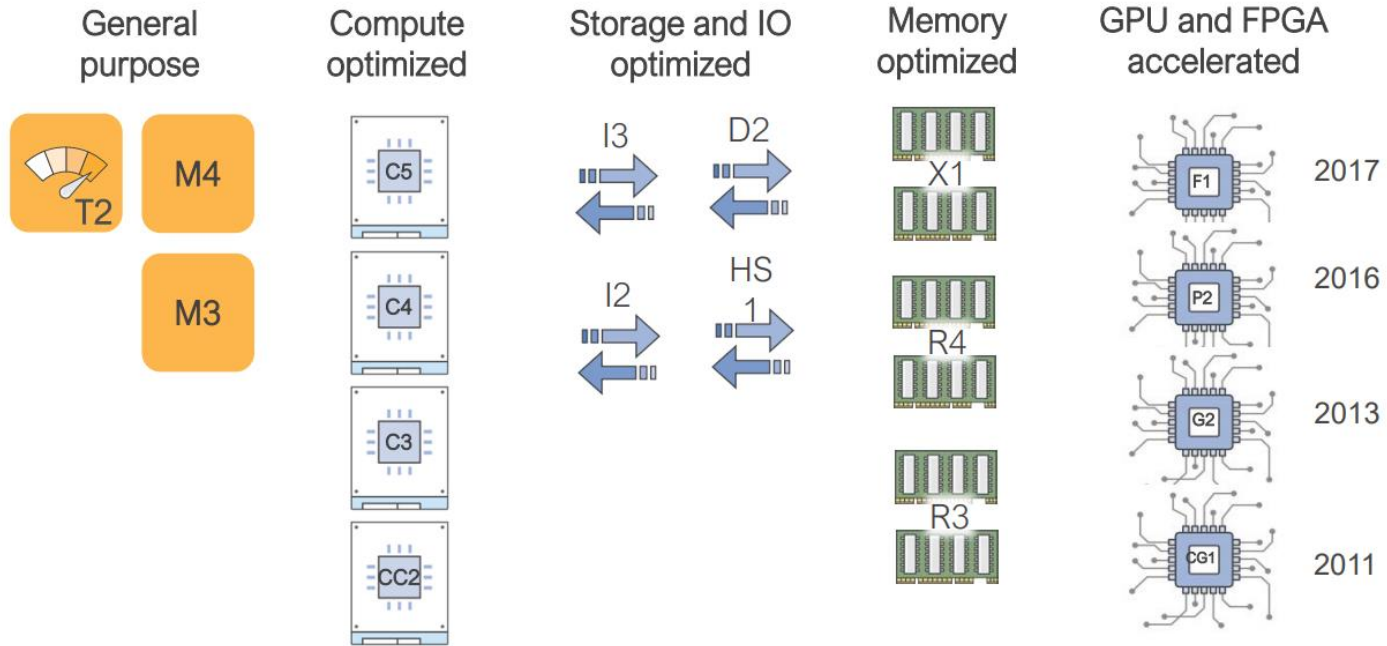


AMD Instinct M1200
(source: Hotchips 2022)



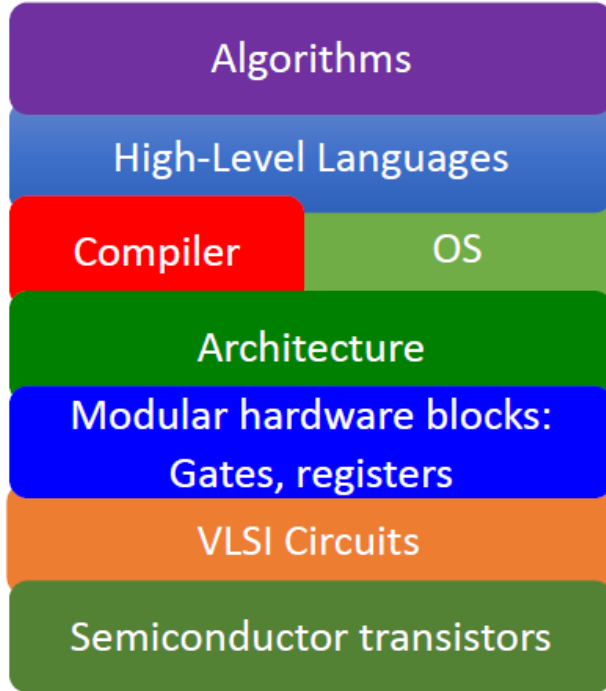
Intel New Flexible Tile

Cloud Architecture: Heterogeneous Trend

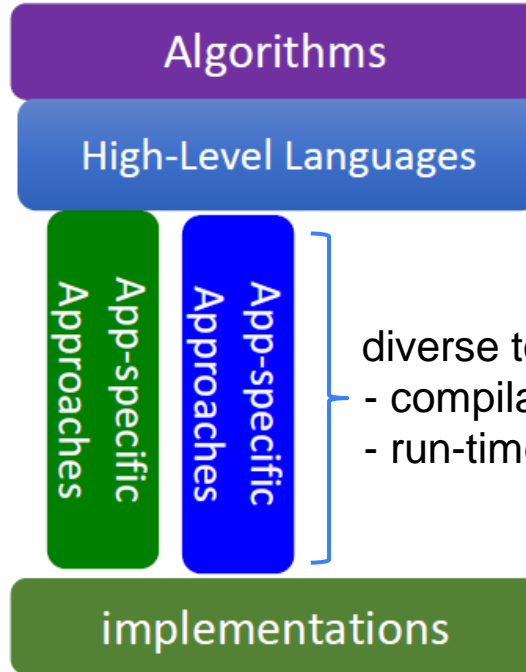


AWS Compute Instance Types (source: AWS)

Design Automation: Support for Heterogeneity



Classical: Monolithic



Current + Future: Diverse

diverse tool-flow for high-level programs

- compilation
- run-time management

(adapted from: Martonosi)

Design Automation: Support for Heterogeneity

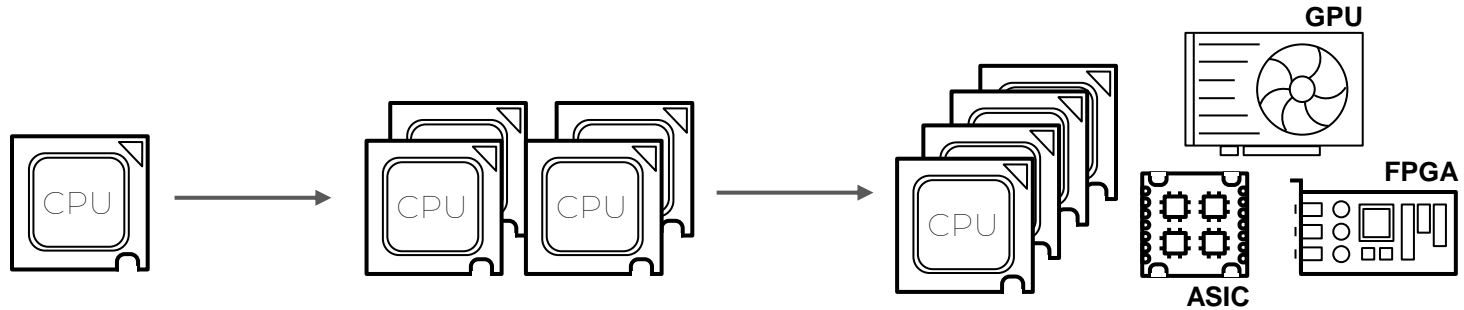
- Compiling for ***heterogeneous systems and processors***
 - Meta-programming Design Flow Patterns
- Managing ***heterogeneous clouds***
 - Function-as-a-Service
- Managing ***heterogeneous FPGA resources***
 - Pool of functions

Design Automation: Support for Heterogeneity

- Compiling for ***heterogeneous systems and processors***
 - Meta-programming Design Flow Patterns - IEEE Trans. Computers, HEART 2022
- Managing ***heterogeneous clouds***
 - Function-as-a-Service - Journal of Signal Processing Systems
- Managing ***heterogeneous FPGA resources***
 - Pool of functions - FPL 2022

Heterogeneous Systems

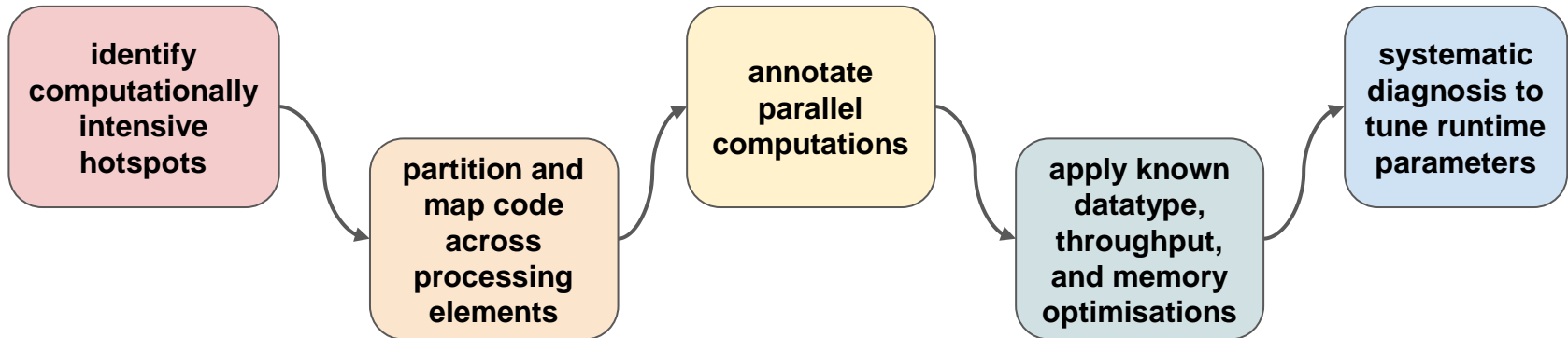
- compute landscape is rapidly evolving → increasingly **parallel** and **heterogeneous**
 - potential of specialised accelerators (GPUs, FPGAs) for demanding applications, e.g. AI, HPC



- gap between software descriptions and optimised heterogeneous designs: getting larger
 - device-specific compilers: achieve high performance from high-level source-code
 - but significant code restructuring is required

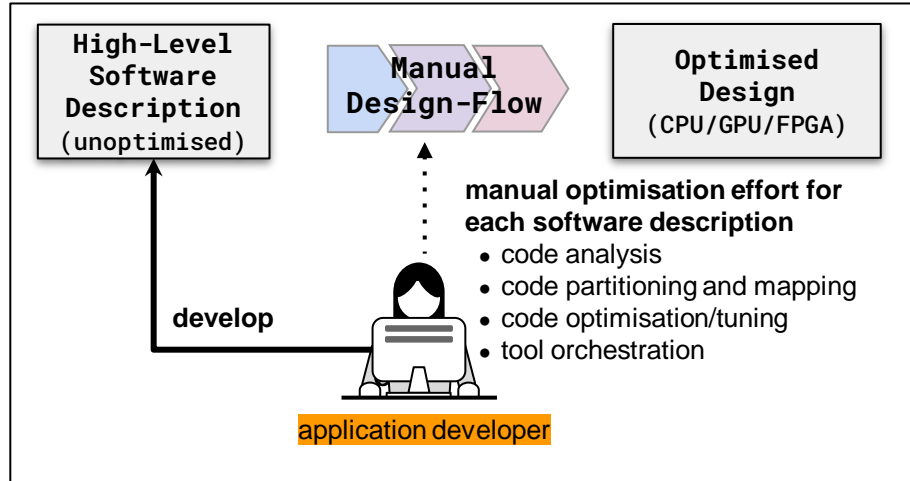
Problem

- heterogeneous application optimisation: typically done manually
 - requires highly-skilled developers with in-depth target hardware understanding
- manual optimisation tasks:



⇒ this process is **tedious**, **error-prone**, and **must be repeated** for each new application

Current Design-Flow: State Of The Practice (SOP)




- current SOP: human developers **manually** perform source-level design-flows
- **design-flow:** - explicit orchestration of manual and/or automated tasks
 - map and optimise a high-level software description onto hardware

Design-Flow Automation Challenges

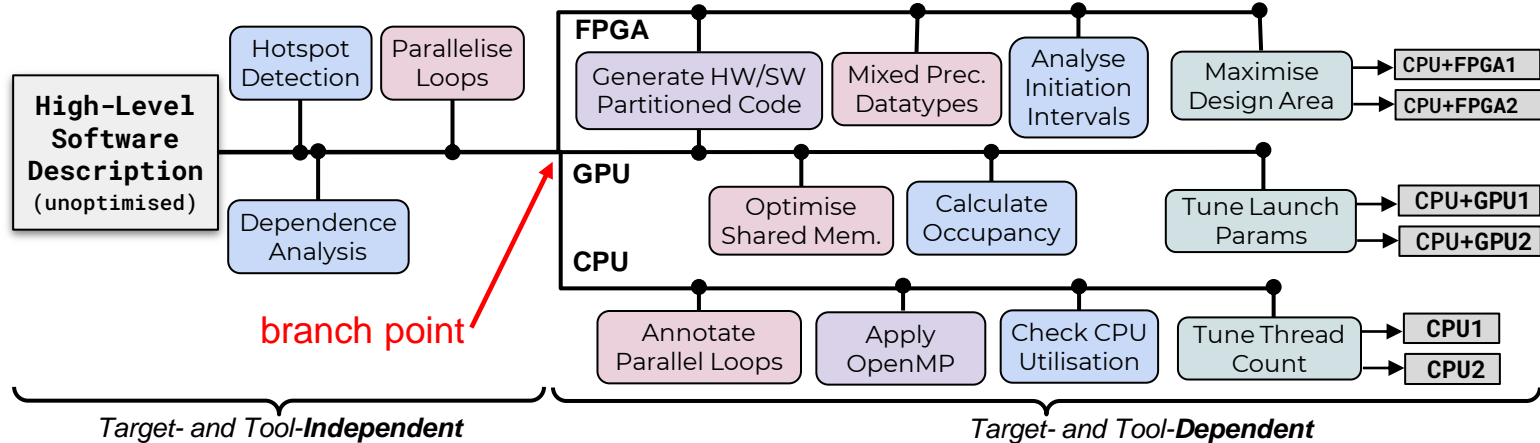
- C1. *Abstraction*:** diverse components should be abstracted to hide implementation details
 - so they can be employed by non-experts
- C2. *Efficiency*:** automatically optimised code should be as efficient as manual optimisation
 - currently requires expertise, experience, and effort
- C3. *Customisability*:** automated design-flows should be flexible and extensible
 - support new techniques and technologies in the massive, evolving design space
- C4. *Reusability*:** design-flows should employ existing, reusable components
 - reduce time and development effort
- C5. *Application-Agnosticity*:** automated design-flows should operate on multiple applications
 - within a specific application domain

Contributions

- ***design-flow patterns*** to capture common and recurring elements of design-flows
 - for optimising high-level descriptions onto diverse hardware targets
 - an initial catalogue of patterns
 - for accelerating CPU and GPU designs
 - codify modular patterns as ***Artisan meta-programs***
 - combine target-independent and -dependent patterns into automated design-flows
 - map unmodified sequential C++ descriptions into optimised CPU and GPU designs
 - apply our design-flows to:
 - 3 case-study HPC applications in different domains (physics, graphics, mathematics)
 - evaluate performance of automatically generated OpenMP and HIP designs
 - results:
 - up to 18 times speedup on a CPU platform with 32-threads
 - up to 1184 times speedup on an NVIDIA GeForce RTX 2080 Ti GPU

compared to a sequential single-threaded reference implementation
- 
- CUDA to C++
Open Multi-Processing

Key Observation

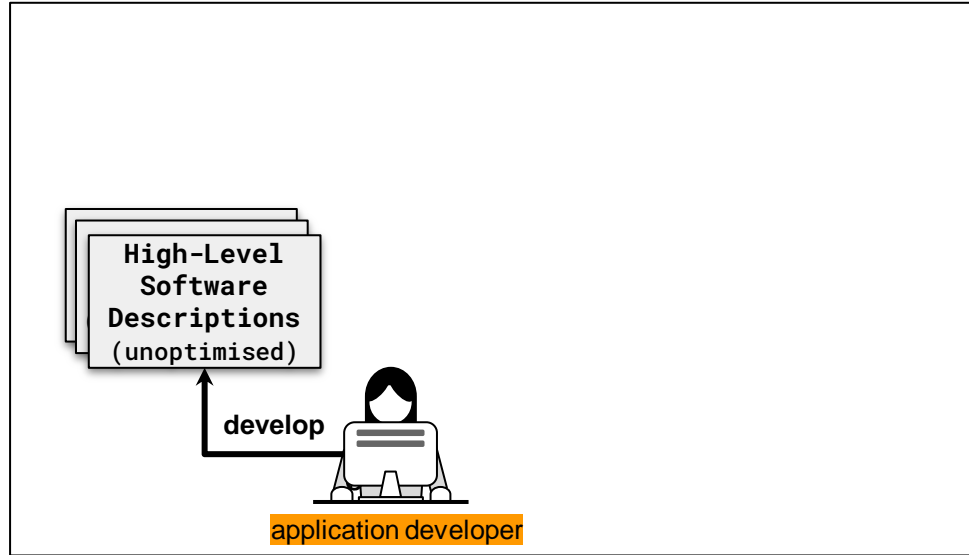


- design-flows for diverse hardware targets often involve:
 - common, recurring, application-agnostic elements
 - elements that can be target and tool-independent or tool-dependent
- ⇒ can we capture and codify these recurring building blocks, highlighting the **branch points** for introducing diverse designs?

Proposed Solution: Meta-Programming Design-Flow Patterns

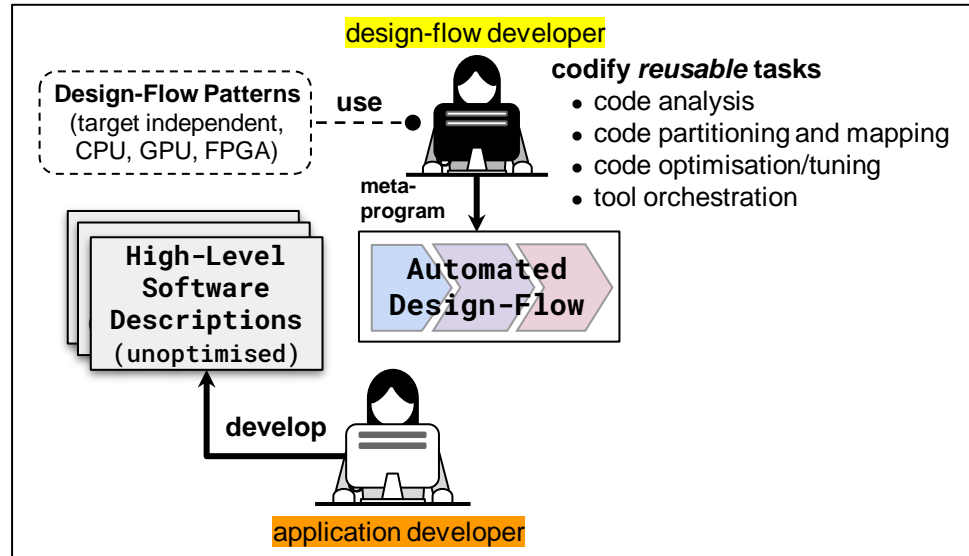
- ***Design-Flow Patterns:***
 - capture, catalogue, and codify common and recurring design-flow tasks
 - for building customised, reusable, automated design-flows
- similar to design patterns:
 - abstract recurring solutions
 - provide reusable base of experience and a common vocabulary
- modular patterns implemented as meta-programs
 - can be coordinated into automated end-to-end design flows

Two Design-Flow Roles



(1) *application developer*: writes functionally correct high-level application description

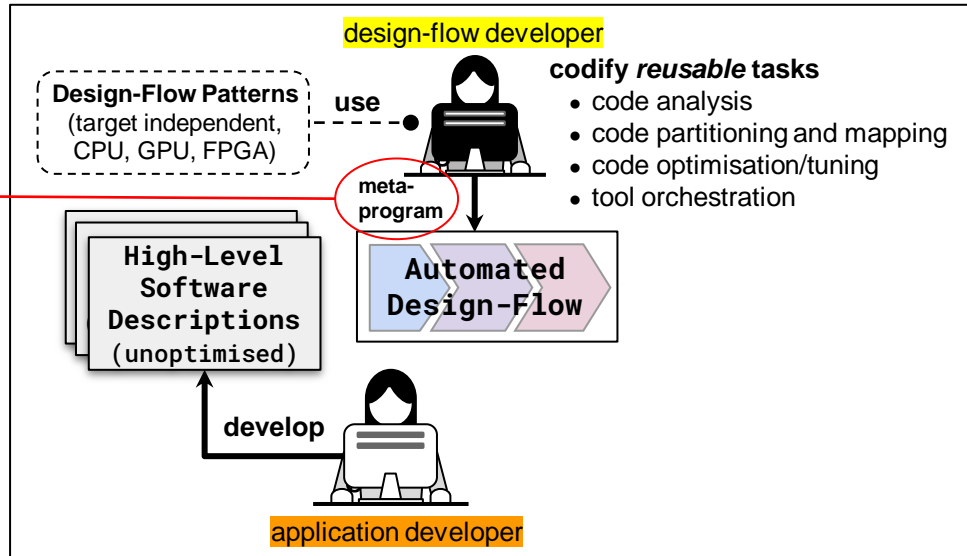
Two Design-Flow Roles



(1) **application developer**: - writes functionally correct high-level application description

(2) **design-flow developer**: - uses design-flow patterns and meta-programs
- to automate design-flows for mapping and optimisation

Two Design-Flow Roles

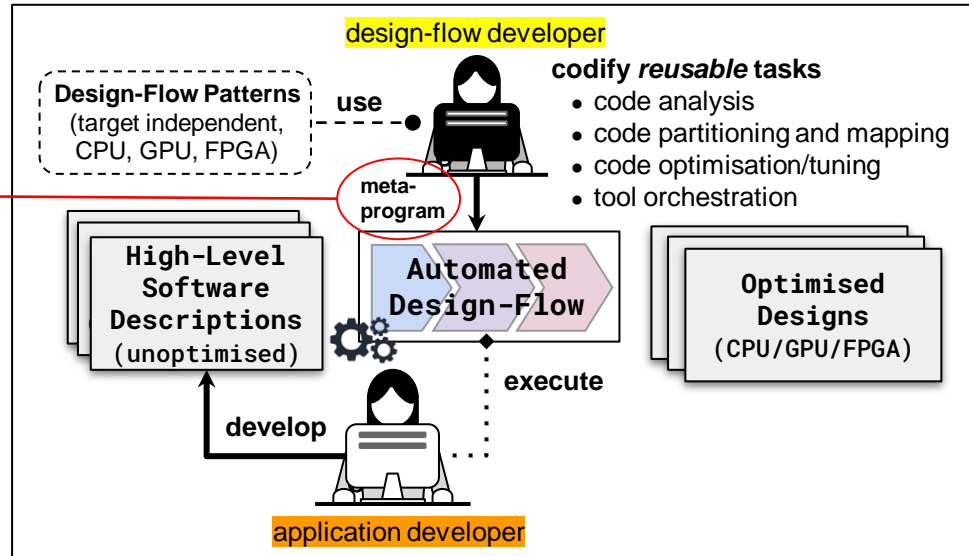


Artisan meta-programs treat programs as data, enabling programmatic analysis and source-code manipulation

(1) **application developer:** - writes functionally correct high-level application description

(2) **design-flow developer:** - uses design-flow patterns and meta-programs
- to automate design-flows for mapping and optimisation

Two Design-Flow Roles



Artisan meta-programs treat programs as data, enabling programmatic analysis and source-code manipulation

- ⇒ manual design-flow tasks are codified and coordinated to produce:
- end-to-end design-flows operating on high-level software descriptions
 - optimised designs with little intervention from application developers

Addressing Design-Flow Automation Challenges

- C1. *Abstraction*:** high-level pattern descriptions
 - abstract implementation details for diverse targets
- C2. *Efficiency*:** source-to-source meta-programs
 - automate manual optimisation with **static** and **dynamic** analysis
- C3. *Customisability*:** pattern implementations as plug-and-play building blocks
 - can be parameterised, replaced, and extended
- C4. *Reusability*:** modular design-flows facilitate patterns
 - implemented once and reused in multiple design-flows (e.g. common analysis)
- C5. *Application-Agnosticity*:** optimisation is decoupled from application descriptions
 - so design-flows are application-agnostic

Design-Flow Pattern Catalogue: Overview

- current catalogue contains patterns for CPU and GPU parallel targets
- requirement: facilitate modular implementations and reasoning about coordination
- a uniform template* is used to describe design-flow patterns:
 - NAME: a succinct, descriptive name for the pattern
 - INTENT: what does the pattern do?
 - MOTIVATION: why is the pattern used?
 - APPLICABILITY: what conditions must be met to apply the pattern?
 - RELATED PATTERNS (OPTIONAL): are there related patterns?
(e.g. components, often used together)
- text-based description should clearly capture intent and applicability
 - developers can unambiguously codify expected behaviour
 - ongoing work: formalise design-flow pattern specification

*a subset of design patterns for OOP from Gamma et al.

Design-Flow Pattern Classification

- 4 types of design-flow patterns:
 - I. ***Analysis patterns***: perform static or dynamic app analysis
 - II. ***Code-generation patterns***: inject or generate new source-code
 - III. ***Transform patterns***: perform source-to-source transformation
 - IV. ***Optimisation patterns***: employ analysis and transform patterns
 - optimise a target metric (typically involving Design Space Exploration)

Analysis Design-Flow Pattern Example

- NAME: ***HOTSPOT LOOP DETECTION***
- INTENT: identify computationally intensive parallel loops to accelerate
- MOTIVATION: - loops are often where most time is spent during execution
- suitable for acceleration (Amdahl's law)
- APPLICABILITY: applicable to any application source code
- RELATED PATTERNS: Loop Timing, Dependence Analysis

Code-Generation Design-Flow Pattern Example

- NAME: ***HIP GPU MANAGEMENT CODE GENERATION***
- INTENT: insert code required to execute an identified kernel function on GPU
- MOTIVATION: device management code is required
 - to inform the runtime system what to run on the GPU vs CPU
 - to ensure data are where they need to be for application execution
- APPLICABILITY: applicable to application code with a specified kernel function

Transform Design-Flow Pattern Example

- NAME: ***SHARED MEMORY BUFFER***
- INTENT:
 - copy the contents of a pointer argument
 - into shared memory in a GPU kernel
- MOTIVATION: on-chip shared memory
 - has limited size
 - has higher bandwidth and lower latency than global memory
- APPLICABILITY: applicable to any pointer argument for a GPU kernel
 - if pointer contents fit in shared memory

Optimisation Design-Flow Pattern Example

- NAME: ***TUNE KERNEL LAUNCH PARAMETERS***
- INTENT: determine the kernel launch parameters
 - to minimise execution time, and/or
 - to maximise occupancy (e.g. block size)
- MOTIVATION: launching kernels with different thread configurations
 - can affect execution time and GPU occupancy
- APPLICABILITY: applicable to an application source with a GPU kernel
- RELATED PATTERNS: Set Blocksize, Kernel Timing, Calculate GPU Occupancy

C. Design-Flow Pattern Catalogue For CPU and GPU Optimisation

Table 1: Analysis (A1-A6), Code-Generation (G1-G3), Transform (T1-T9) and Optimisation (O1-O2) Design-Flow Patterns

ID	NAME (RELATED)	INTENT	MOTIVATION	APPLICABILITY
A1	Hotspot Loop Detection (A2,A3)	Identify computationally intensive loops to accelerate.	Loops are often regions where most time is spent during the program's execution.	Application code
A2	Loop Timing	Measure execution time for all loops in the application.	To identify application bottlenecks and regions worth optimising.	Application code
A3	Dep. Analysis	Identify dependencies in a program loop.	To parallelise and/or transform loops.	Loop
A4	Pointer Analysis (T1)	Determine if pointer arguments could alias within a function scope.	Certain compiler optimisations can only be applied if it is indicated that pointers do not alias.	Function definition
A5	Kernel Timing	Time all GPU kernels in an executed application.	To understand the impact of code changes, identify bottlenecks, and compare performance.	Application code + GPU kernel
A6	Calculate GPU Occupancy	Determine the occupancy for a kernel on a target GPU.	Calculating occupancy helps to understand performance and to tune GPU launch parameters.	Application code + GPU kernel
G1	Loop-to-Function Extraction	Extract a program loop into an isolated function.	To enable isolated analysis and annotation to indicate it should be offloaded to an accelerator.	Loop
G2	Multi-Threaded Code Generation	Insert the framework-specific code required to multi-thread a loop.	Loop annotation, header file inclusion, and runtime parameter specification is needed for runtime system to use multiple parallel threads.	Application code + loop
G3	GPU Mgmt Code Generation	Insert the framework-specific code required to execute a kernel on a GPU.	Device management code is required to inform the runtime system what to run on the GPU vs CPU, and to ensure data is where it needs to be.	Application code + function
T1	Restrict Pointer Arguments (A4)	Indicate to the compiler that pointer arguments do not alias.	Device compilers that cannot determine if pointers could alias conservatively assume that they might, limiting the scope for optimisation.	Non-aliasing function args + target with restrict keyword
T2	Shared Memory Buffer	Copy the contents of a pointer argument into shared memory in a GPU kernel.	Limited on-chip shared memory has higher bandwidth and lower latency than global memory.	Pointer + GPU kernel, if pointer contents fit in shared mem
T3	Page-Locked Memory	Allocate memory as page-locked.	Limited page-locked memory has the highest bandwidth between host and device, but has heavier weight allocations than regular memory.	App code + GPU kernel + target with page-locked memory
T4	Single-Precision Math Functions	Use single-precision versions of math functions. (e.g. sqrtf).	Avoid implicit intermediate rounding to double-precision operations.	GPU kernel + library math function call
T5	Single-Precision FP Literals	Employ single-precision floating point literals.	Explicitly use single precision literals (e.g. 0.0f) so compiler does not assume double precision.	Expressions with single-precision types.
T6	Specialised Math Operations	Use available specialised math operations.	Framework-provided specialised math functions are more optimised than general equivalents.	Consult tool documentation (e.g. pow(x, 2) to exp2(x))
T7	Remove Loop Dep (A3)	Remove dependent array accesses in loops by introducing intermediate variables.	To ease loop dependency bottlenecks.	Loops with dependent array accesses
T8	Set Blocksize	Specify the thread block size for GPU kernel execution.	Runtime GPU thread configurations must be set when launching a kernel.	GPU kernel
T9	Set Num Threads	Set the number of parallel threads for loop execution.	To control the number of threads used for multi-threaded execution.	Loop + multi-threaded target
O1	Tune Number of Threads (T9,A2)	Determine the number of threads that minimises loop execution time.	The number of threads can affect performance depending on available cores and workload size.	Loop(s) + multi-threaded target
O2	Tune Kernel Launch (T8,A5,A6)	Determine the kernel launch parameters that minimises execution time and/or maximises occupancy.	Launching kernels with different thread configurations can affect execution time and GPU occupancy.	Application code + GPU kernel(s)

Design-Flow Pattern Catalogue

- refer to our HERRT'22 paper for the full catalogue
- a starting point to demonstrate scope and value: recurring, application-agnostic design-flow
- not an exhaustive list of GPU/CPU patterns

Design-Flow Patterns as Meta-Programs

- codify patterns using the Artisan meta-programming framework
 - based on libclang, supports C++ parsing and manipulation
 - unified Python environment for code analysis, instrumentation, and execution
 - true source-to-source: no progressive lowering
- key Artisan features
 - ***query*** and ***instrument***
 - ⇒ enables static source-code analysis and manipulation
 - application ***execution*** and runtime ***reporting***
 - ⇒ enables application self-reporting for dynamic analyses

Design-Flow Patterns as Meta-Programs

- example meta-programs:
 - (1) GPU shared memory buffer (transform)
 - (2) parallel hotspot loop detection (dynamic analysis)
- for more details on Artisan, refer to our paper in *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2043-2055, 1 Dec. 2021

Enhancing High-Level Synthesis using a Meta-Programming Approach

Jessica Vandebon*, Jose G. F. Coutinho*, Wayne Luk*, Eriko Nurvitadhi†

*Imperial College London, United Kingdom

Email: {jessica.vandebon17, gabriel.figueiredo, w.luk}@imperial.ac.uk

†Intel Corporation, San Jose, USA

Email: eriko.nurvitadhi@intel.com

Abstract—In today's increasingly heterogeneous compute landscape, there is high demand for design tools that offer seemingly contradictory features: portable programming abstractions that hide underlying architectural detail, and the capability to optimise and exploit architectural features. Our meta-programming approach, Artisan, decouples application functionality from optimisation concerns to address the complexity of mapping high-level application descriptions onto heterogeneous platforms from which they are abstracted. With

Hotspot Detection

D. Implementing Design-Flow Patterns as Meta-Programs

```
1 def identify_hotspots (ast, threshold):  
    # clone ast for instrumentation & execution  
2 ast_clone = ast.clone()  
    # query for parallel for-loops to time  
3 par_loops = ast_clone.query("loop{ForStmt}",  
                             where=lambda loop: is_par(loop))  
    # instrument loops and main function with timers  
4 instrument_app_timer(ast_clone, par_loops)  
    # execute instrumented code and receive report  
5 report = ast_clone.exec(reports=True)  
    # discard clone  
6 ast_clone.discard()  
    # extract main timing from report (e.g. main_t = 404.9)  
7 main_t = report['main']; del report['main']  
    # filter and return loop list that satisfies given threshold  
8 hotspots = [loop for loop in report if report[loop] > main_t * threshold]  
    # in our example, returns ['loop0312']  
9 return hotspots
```

Artisan meta-program
(Python)

```
1 + #include <artisan>  
2 + using namespace artisan;  
3 int main(int argc, char *argv[]){  
4 + Report::start();  
5 + int ret;  
6 + { Timer timer_main([](double t){  
7 +     Report::write("'main':%f",t);});}  
8 + ret = [](auto argc, auto argv){  
9     ...  
10 + { Timer timer_ltag([](double t){  
11 +     Report::write("'loop0312',%f",t);});}  
12     for (int i = 0; i < N; i++) {  
13         z[i] = x[i] * y[i];  
14     }  
15 + }  
16     for (int j=0; j<T; j++) {  
17         z[j] = x[j] * z[j-1];  
18     }  
19     ...  
20     return 0;  
21+ } (argc, argv);  
22+ }  
23+ Report::emit();  
23+ return ret;  
25 }
```

instrumented
parallel
loop

This loop is not
parallel, and
therefore is
not instrumented

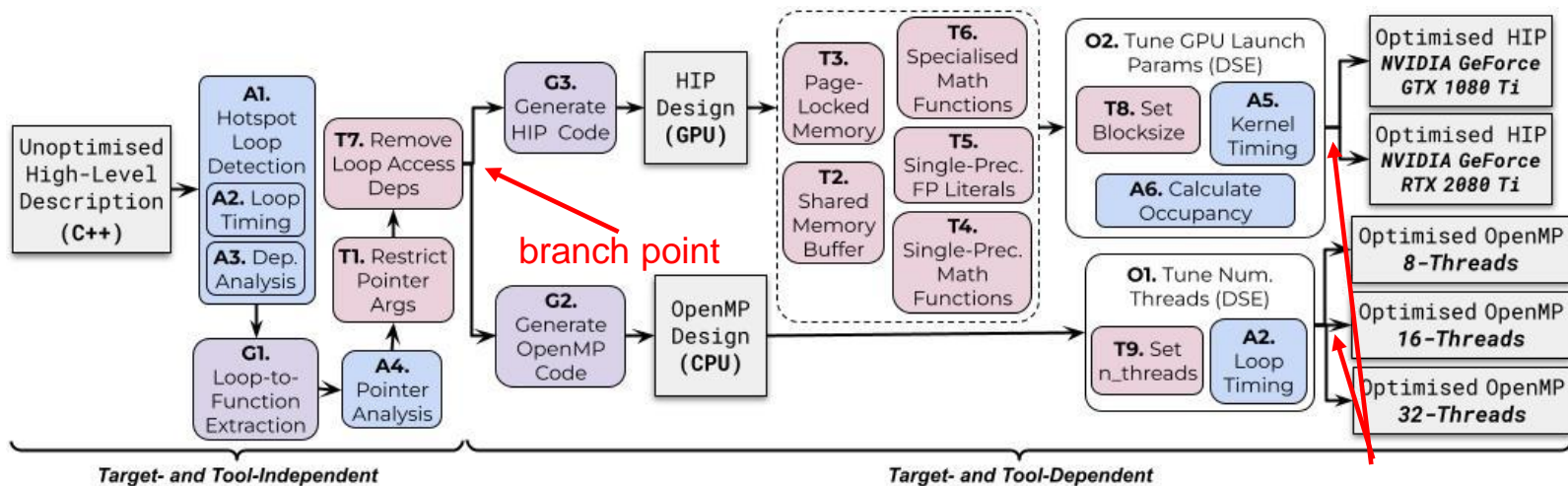
report

{'main':404.9, 'loop0312':306.7}

report sent via
network socket
to metaprogram

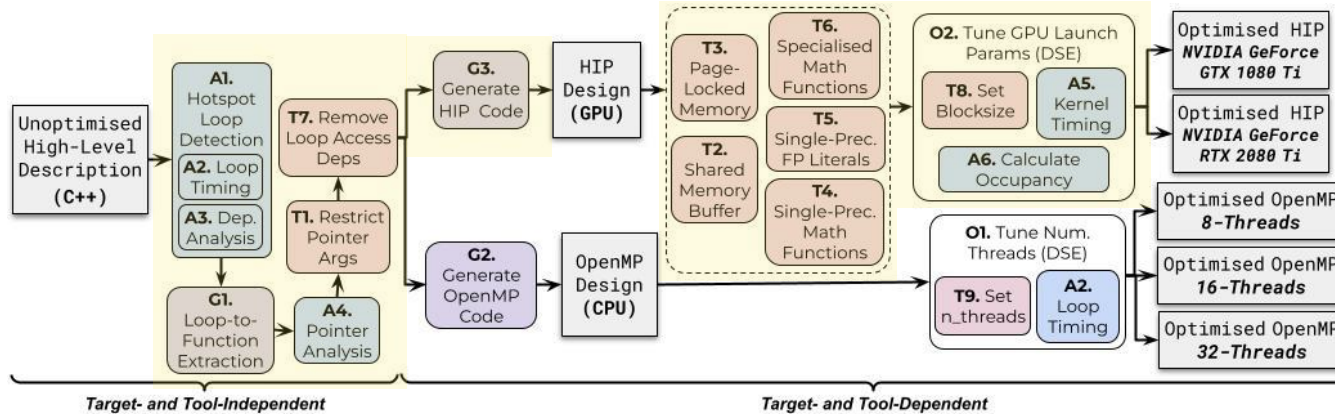
instrumented app (C++)

Automated End-To-End Design Flows



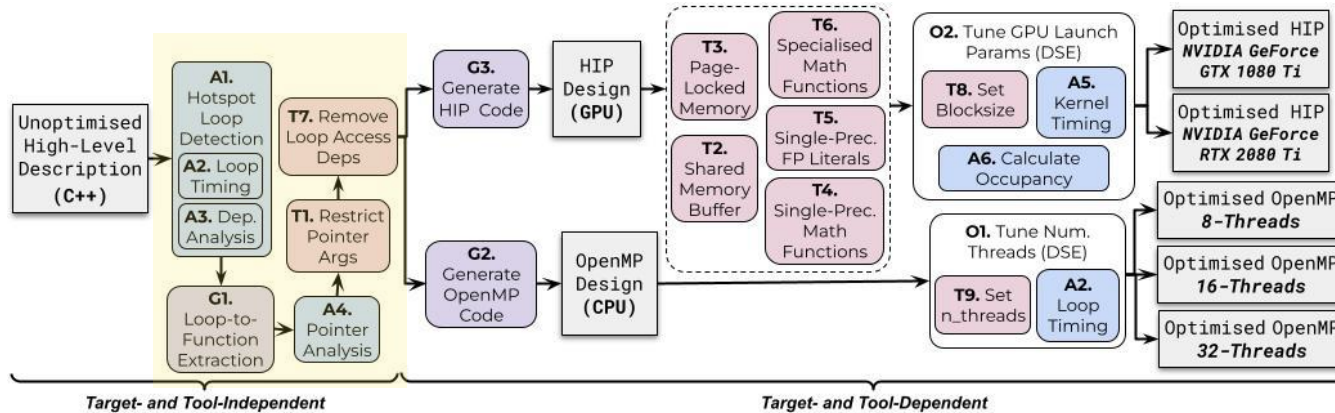
- implemented end-to-end **HIP GPU** and **OpenMP CPU** design-flows
 - comprised modular meta-programs codifying patterns from our catalogue
- applied to three HPC case-study applications:
 - N-Body Simulation (physics),
 - Bezier Surface Generation (graphics),
 - Rush Larsen ODE Solver (maths)

Evaluating Design-Flow Pattern Reuse



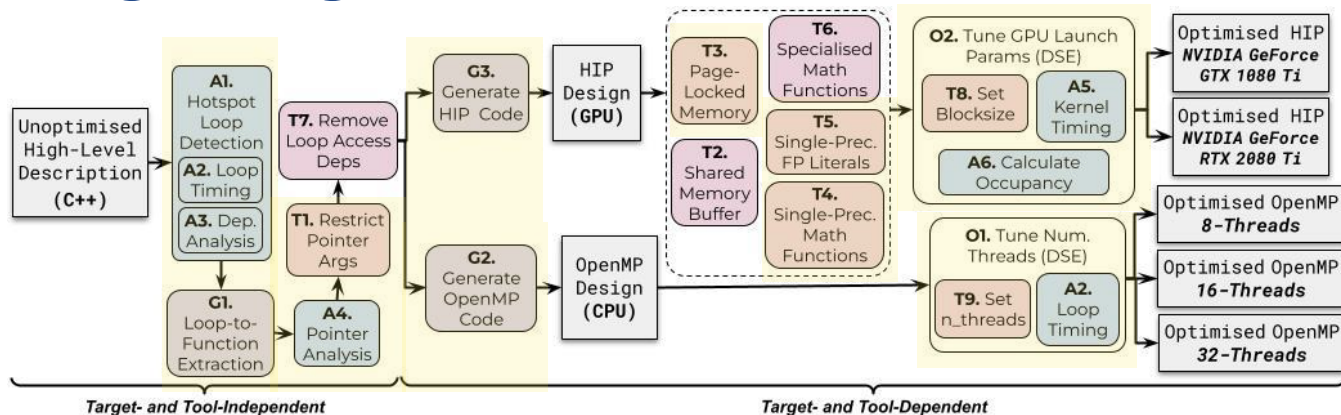
- 20 patterns implemented
 - 10 employed by OpenMP design-flow
 - 17 employed by HIP GPU design-flow

Evaluating Design-Flow Pattern Reuse



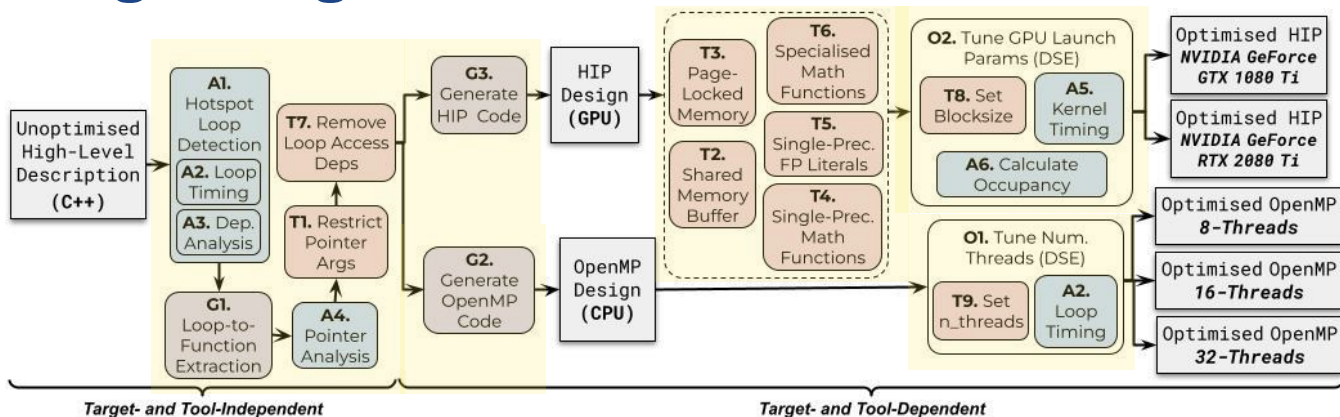
- 20 patterns implemented
 - 10 employed by OpenMP design-flow
 - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows

Evaluating Design-Flow Pattern Reuse



- 20 patterns implemented
 - 10 employed by OpenMP design-flow
 - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows
- 17/20 patterns applicable to all three case-study applications

Evaluating Design-Flow Pattern Reuse



- 20 patterns implemented
 - 10 employed by OpenMP design-flow
 - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows
- 17/20 patterns applicable to all three case-study applications
- 20/20 patterns are application agnostic

Evaluating Design-Flow Performance

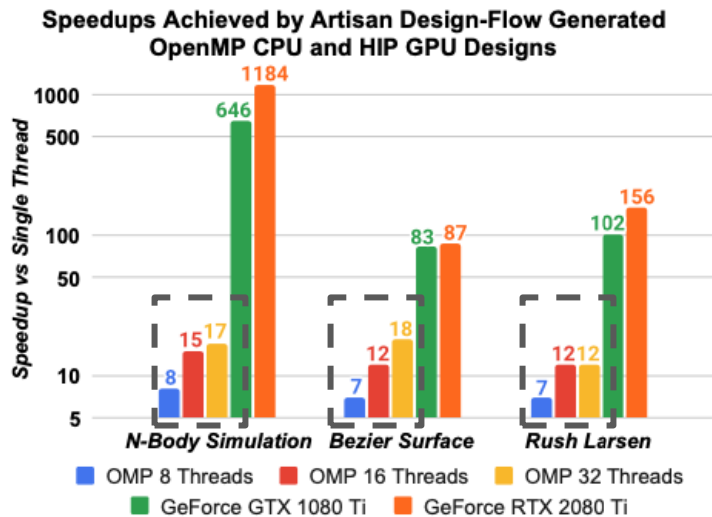


Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).

OpenMP CPU Experiments:

- experimental set-up:
 - 2 Intel Xeon Silver 4110 CPUs, 16 cores with SMT
 - g++ -O2
 - consider 8, 16, 32 available threads
- performance results:
 - generally: increasing threads decreases execution time (nonlinear due to scheduling/mgmt overhead)
 - above 16 threads: speedup limited by SMT support
 - **12X-18X** maximum speedup across case-studies

Evaluating Design-Flow Performance

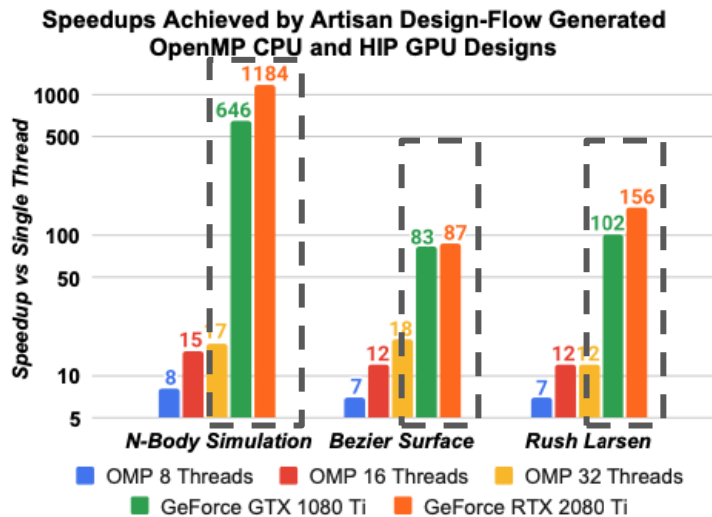
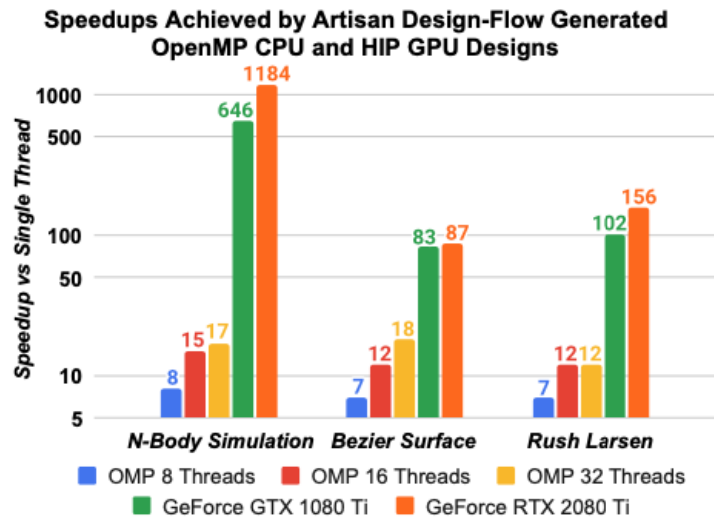


Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).

HIP GPU Experiments:

- experimental set-up:
 - 2 NVIDIA GeForce GPUS:
 - NVIDIA GeForce GTX 1080 Ti
 - NVIDIA GeForce RTX 2080 Ti
 - hipcc -O2
- performance results:
 - generally: RTX 2080 faster than GTX 1080 (wider cores with advanced features)
 - **87X-1184X** maximum speedup across case-studies

Evaluating Design-Flow Performance



- performance comes free
 - little or no intervention from application developer
- generated code is human-readable
 - same level of abstraction as original code
 - can be further hand-tuned

Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).

Ongoing and Future Work

- formalising the specification and description of design-flow patterns
 - ◆ using functional programming

- extending our design-flow pattern catalogue:
 - ◆ FPGA OneAPI mapping and optimisation patterns
 - ◆ patterns to support more advanced GPU optimisations
 - ◆ application-domain specific patterns

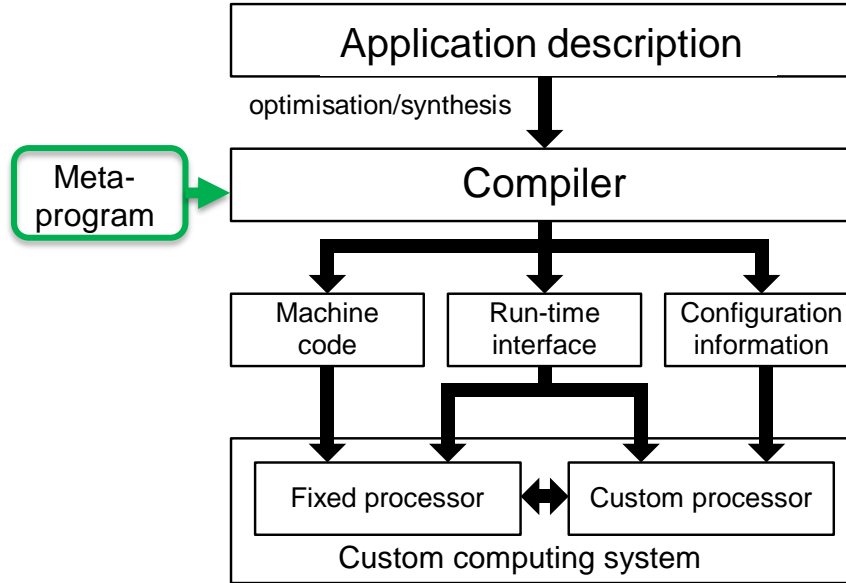
Big Picture: Automating Design

design space exploration,
goals and constraints

partition, compile

system-specific
programming interface

system-specific adaptation:
clouds to edge devices



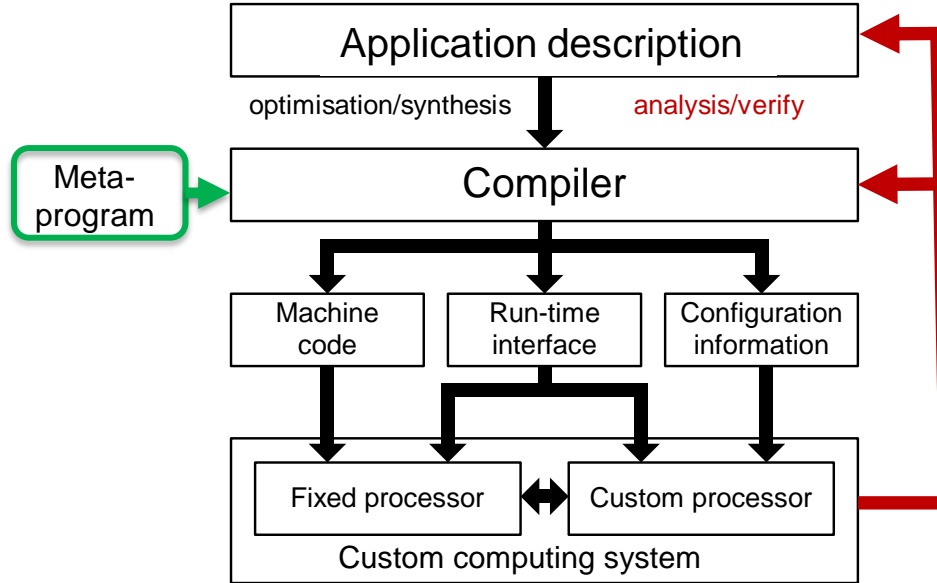
Big Picture: Automating Design + Debug + Verify

design space exploration,
goals and constraints

partition, compile,
analysis, verify

system-specific
programming interface

system-specific adaptation:
clouds to edge devices



FCCM 2021: Flexible Instrumentation for Live On-Chip Debug of Machine Learning Training on FPGAs
 JSA 2021: In-Circuit Tuning of Deep Learning Designs