Meta-Programming Design-Flow Patterns for Automating Reusable Optimisations

Jessica Vandebon
Imperial College London
United Kingdom
jessica.vandebon17@imperial.ac.uk

Jose G. F. Coutinho
Imperial College London
United Kingdom
gabriel.figueiredo@imperial.ac.uk

Wayne Luk
Imperial College London
United Kingdom
w.luk@imperial.ac.uk

ABSTRACT

Continuing advances in heterogeneous and parallel computing enable massive performance gains in domains such as AI and HPC. Such gains often involve using hardware accelerators, such as FPGAs and GPUs, to speed up specific workloads. However, to make effective use of emerging heterogeneous architectures, optimisation is typically done manually by highly-skilled developers with in-depth understanding of the target hardware. The process is tedious, error-prone, and must be repeated for each new application. This paper introduces *Design-Flow Patterns*, which capture modular, recurring application-agnostic elements involved in mapping and optimising application descriptions onto efficient CPU and GPU targets. Our approach is the first to codify and programmatically coordinate these elements into fully automated, customisable, and reusable end-to-end design-flows. We implement key design-flow patterns using the meta-programming tool Artisan, and evaluate automated design-flows applied to three sequential C++ applications. Compared to single-threaded implementations, our approach generates multi-threaded OpenMP CPU designs achieving up to 18 times speedup on a CPU platform with 32-threads, as well as HIP GPU designs achieving up to 1184 times speedup on an NVIDIA GeForce RTX 2080 Ti GPU.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Multicore architectures**; • **Software and its engineering** → **Compilers**.

KEYWORDS

Heterogeneous Computing, Parallel Computing, Meta-Programming, GPU, Multi-Core, Patterns

ACM Reference Format:
Jessica Vandebon, Jose G. F. Coutinho, and Wayne Luk. 2022. Meta-Programming Design-Flow Patterns for Automating Reusable Optimisations. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2022), June 9–10, 2022, Tsukuba, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3535044.3535050

HEART2022, June 9–10, 2022, Tsukuba, Japan

ACM ISBN 978-1-4503-9660-8/22/06…$15.00
https://doi.org/10.1145/3535044.3535050

1 INTRODUCTION

The mainstream compute landscape is rapidly evolving, becoming increasingly parallel and heterogeneous to harness the massive performance potential of specialised hardware accelerators such as GPUs and FPGAs. With this shift, the gap between traditional software application descriptions and optimised designs targeting heterogeneous platforms is becoming more pronounced. While device-specific optimising compilers continue to improve their ability to achieve high performance from high-level source-code, there is still significant restructuring and transformation required to craft sequential software application descriptions into designs amenable for compilation onto specialised hardware.

In this context, developers must identify computationally intensive 'hotspots' for acceleration; select a suitable partition and mapping across different processing elements; identify and expose concurrent code sections to enable parallelisation by compilers; apply well-known datatype, memory, and throughput optimisations; and perform systematic design space exploration (DSE) to determine optimal runtime configuration parameters. These tasks are all part of source-level *design-flows*, performed manually by human developers, to map unoptimised high-level descriptions into optimised designs for specific hardware targets.
Automating such design-flows poses the following challenges:

C1. **Abstraction**: design-flow components should be suitably abstracted to hide implementation details so they can be employed by non-expert developers.
C2. **Efficiency**: automatically optimised code should be as efficient as manually optimised code, which currently requires expertise and effort from developers with in-depth hardware knowledge and experience.
C3. **Customisability**: automated design-flows should be flexible and extensible to support new search algorithms, optimisation techniques, and emerging technologies in the massive and evolving design space.
C4. **Reusability**: design-flows should be built using reusable, existing components to decrease time and development effort.
C5. **Application-Agnostic**: automated design-flows should operate on arbitrary applications, or within a specific application domain.

This paper introduces *Design-Flow Patterns*, which provide a novel approach that addresses all challenges C1-C5 described above (see Fig. 1). Design-flow patterns allow common and recurring elements of a design-flow to be codified as meta-programs, including tasks currently performed manually by expert developers. Codified patterns can then be programmatically coordinated into

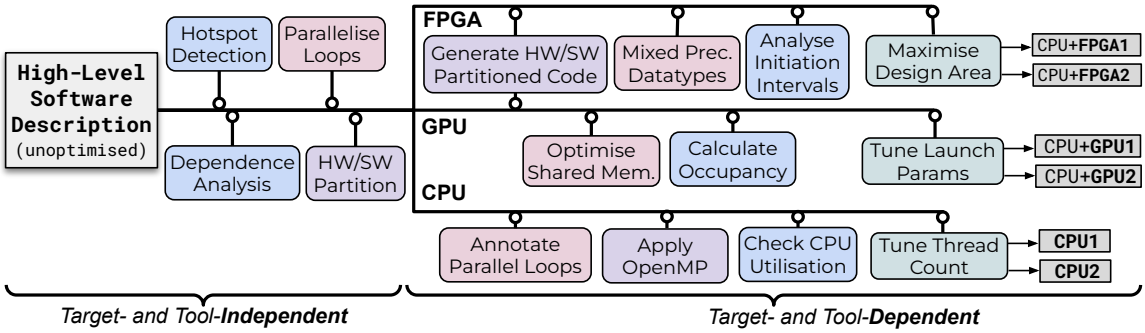

Figure 1: Design-Flow Patterns: common, recurring, application-agnostic elements for building, customising, and automating end-to-end design-flows that optimise high-level descriptions for heterogeneous CPU/GPU/FPGA platforms.

automatic design-flows. Our contributions are: (i) an initial catalogue of design-flow patterns for optimising high-level application descriptions onto multi-threaded CPU and GPU targets (Section 2); (ii) implementations of patterns and automated design-flows using the meta-programming tool Artisan (Section 3); (iii) an evaluation of implemented design-flows applied to three C++ applications generating optimised OpenMP CPU designs and HIP GPU designs achieving up to a 18 times speedup on a CPU platform with 32-threads and up to 1184 times speedup on an NVIDIA GeForce RTX 2080 Ti GPU compared to a single-threaded implementation (Section 4). Section 5 discusses related work, and Section 6 concludes and reports ongoing and future work.

2 DESIGN-FLOW PATTERNS

2.1 Overview

We define a *design-flow* as the explicit orchestration of manual and/or automated tasks that map and optimise a high-level software description onto a specific hardware platform. Fig. 2(a) illustrates the current manual design-flow methodology. Starting with a high-level implementation, developers perform code analysis to understand application requirements and bottlenecks, identifying code regions worth accelerating. They manually partition and map computations onto available processing elements, generating framework-specific device management and data transfer code. They apply known optimisation techniques, as documented in tool vendors' best practices guides [1][9][18], to tune performance on each target device. Experienced developers make decisions driven by insights into the application's characteristics, as well as knowledge of the target platform, typically involving a combination of informed trial and error and systematic DSE. This manual effort must be repeated and modified for each new application.

To automate this process, we propose *Design-Flow Patterns* as a means to capture, catalogue, and codify common and recurring design-flow tasks with the purpose of building customised, reusable, automated design-flows (see Fig. 2(b)). Similar to *design patterns*, popularised in the context of object-oriented software design [5] and extended to cover parallel as well as GPU and FPGA design [4][14][15][17], *Design-Flow Patterns* abstract recurring solutions to provide developers with a reusable base of experience and a common vocabulary. In contrast to design-patterns, which capture principles for developing application descriptions, *design-flow patterns* capture principles for developing automated mapping and optimisation tasks, addressing the challenges outlined in the introduction as explained below.

Fig. 2(b) illustrates two specific envisioned roles: (1) the application developer whose expertise and primary concern is algorithmic behaviour; and (2) the design-flow developer who builds design-flows that automate mapping and optimisation. This is possible through our meta-programming methodology [21], which allows programs to be seen as data, and therefore programmatically analysed and manipulated. This means that all currently manual design-flow tasks, from analysis to code transforms and DSE, can be potentially codified and coordinated into an end-to-end design-flow that operates on a high-level unoptimised software description to derive an optimised design with little intervention from an application developer.

High-level design-flow pattern descriptions abstract implementation details, allowing design-flow tasks to be accessible for experimentation by developers without in-depth hardware knowledge **(C1)**. With our approach, meta-programs operate at the same level as human developers (source-to-source), hence end-to-end flows have the potential to achieve performance close to hand-tuned designs or be further manually fine-tuned **(C2)**. Pattern implementations can be parameterised, replaced, and extended as plug-and-play building blocks **(C3)**. Moreover, modular end-to-end design-flows facilitate pattern component reuse (for instance, common analysis techniques) in order to reduce development time and effort (**C4**). Finally, our approach decouples optimisation concerns from high-level descriptions, supporting design-flows that are application agnostic (**C5**). This way, optimisation effort codified once can be reapplied across multiple applications.

2.2 Template

To facilitate modular implementations as well as reasoning about coordination of patterns into end-to-end design-flows, a uniform template should be used to catalogue them. We propose using a subset of the form contributed by Gamma et al. [5] for OO design patterns, including the following fields:

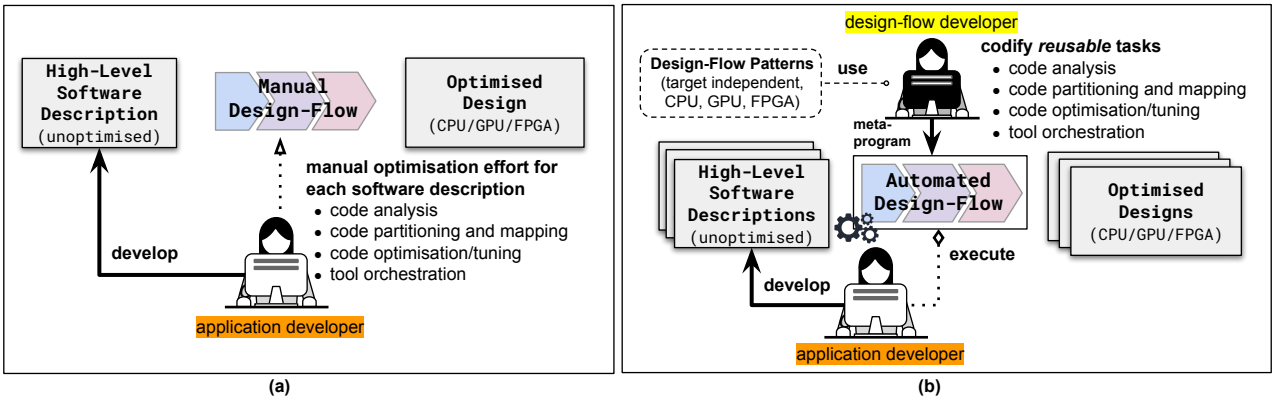

Figure 2: (a) Current design-flow methodology, (b) Proposed methodology based on design-flow patterns and meta-programming.

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 for the pattern to be applied?*
Related Patterns (optional): *Are there related patterns? (e.g. components, often used together)*

While a uniform template is important for cataloguing and reasoning about design-flow patterns, pattern definitions are still relatively informal and text-based. It is critical to clearly capture the intent and applicability of each pattern such that design-flow developers can unambiguously codify expected behaviour, and reason about how to combine patterns into design-flows, while still leaving room for flexibility in implementation. The following subsection poses an initial catalogue of design-flow patterns defined using this template, focusing on optimising for CPU and GPU targets.

2.3 Catalogue

In our catalogue, we classify design-flow patterns into four types:

i. **Analysis patterns** cover common static or dynamic application analyses. By operating at source-level, design-flow patterns have a broader scope for analysis than optimising compilers, including runtime analysis.
ii. **Code-Generation patterns** inject or generate new source-code to format the application as necessary for execution on a new target or to facilitate further analysis and transformation. For example, generating framework-specific GPU management code.
iii. **Transform patterns** perform source-to-source transformations. For example, replacing mathematical expressions with specialised built-in operations provided by a target programming model.
iv. **Optimisation patterns** employ analysis and transform patterns to optimise some aspect of the application, typically involving DSE driven by a performance objective. For example, tuning the number and configuration of threads in order to minimise execution time. The integration of different patterns (e.g. *analysis* to support decision making, and *transforms* to manipulate code) allow us to build design-flows that can operate with little or no intervention from application developers (see Fig. 5).

Patterns can be target- and tool- independent or dependent. For instance, time all program loops and generate a report (target-independent analysis), or insert a framework-specific pragma to indicate to a known vendor's compiler that it can safely make an assumption (target-dependent transform). Furthermore, tool-independent patterns may have tool-dependent implementations - most GPU vendors support a concept of shared memory, but how this should be indicated in code varies. Table 1 provides example patterns of each type defined using the proposed template in our approach. This catalogue of patterns is a starting point to demonstrate the scope and value of recurring, application-agnostic design-flow components - it is not an exhaustive list of all possible patterns.

The following section describes how we codify these patterns using meta-programs to develop coordinated end-to-end programmatic design-flows targeting CPU and GPU platforms.

3 CODIFYING DESIGN-FLOW PATTERNS

3.1 The Artisan Meta-Programming Framework

We implement the design-flow patterns in our catalogue using the meta-programming tool Artisan [21]. Artisan provides a unified environment for source-code analysis, instrumentation, and execution, with support for an array of key features that facilitate modular pattern implementation as well as coordination into end-to-end design-flows. Artisan currently employs the *libclang* framework [11] to parse C++ descriptions, so meta-programs operate at a true source-to-source level with no progressive lowering. Source-code is represented as an abstract-syntax-tree (AST) closely reflecting the code as written by a developer without losing information. Artisan meta-programs are written in Python 3, and are therefore accessible to design-flow developers with varying expertise (platform, tool, domain). With a familiar programming environment in which source-code, tools, and platforms are exposed as first-class Python objects Artisan meta-programs can seamlessly

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)

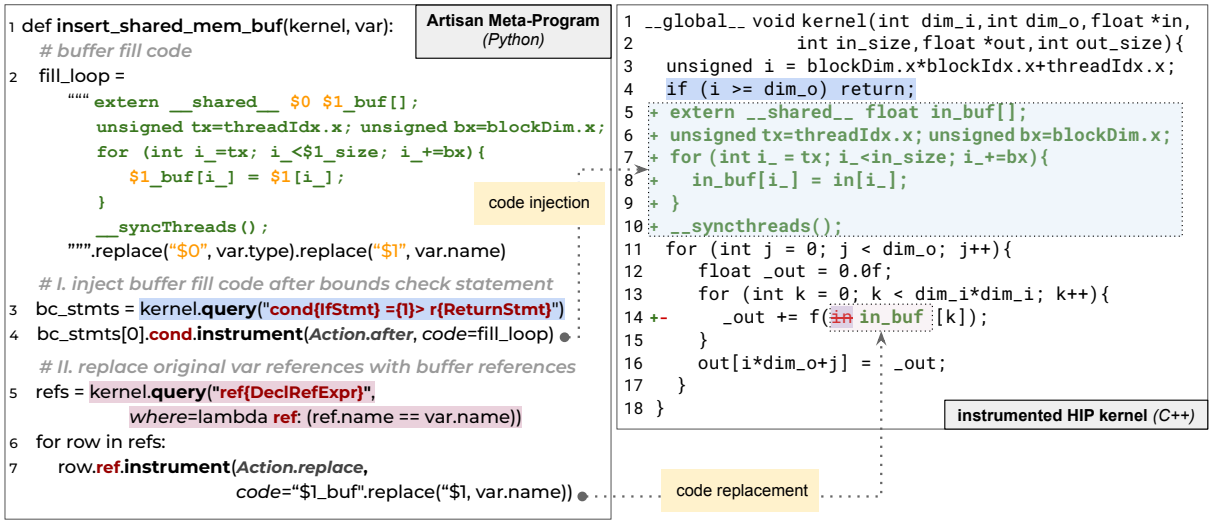

Figure 3: Artisan meta-program code (left) and example instrumented HIP kernel (right) for design-flow pattern *Shared Memory Buffer* (T2, Table 1). This example demonstrates Artisan's static code query and instrumentation mechanisms.

integrate with other Python libraries. Note that while Artisan has proved to be a convenient platform for implementing design-flow patterns, other frameworks could be used.

3.2 Meta-Program Examples

In this section, we detail two pattern implementations from our catalogue, demonstrating four key Artisan mechanisms: (1) source-code queries, (2) code instrumentation, (3) application execution, and (4) runtime reporting. These mechanisms are used to implement all patterns referenced in Table 1. Fig. 3 presents a simplified version of the *shared memory buffer* pattern (T2, Table 1), and Fig. 4 illustrates the key elements of the *hotspot detection* pattern (A1, Table 1), demonstrating their operation on C++ source-code. These patterns are application-agnostic and do not rely on any specific knowledge about the original application code. Note that both meta-programs have been adapted to adjust for space and increase legibility for readers not familiar with the Python language.

3.2.1 ***Shared Memory Buffer (Fig. 3).*** The aim of this meta-program is to instrument a GPU kernel to use faster shared memory instead of global memory. Our implementation demonstrates two key Artisan mechanisms, namely *querying* (left:3,5) and *instrumentation* (left:4,7). The meta-program operates as follows:

- The C++ kernel (right-hand side) was derived automatically by meta-programs G1 and G3 (see Table 1). First, G1 extracts a loop into an isolated function. In turn, G3 translates this function into a GPU kernel by **(a)** associating the outermost loop index with the global work-item ID (right:3), **(b)** adding a bounds check based on the loop exit condition (right:4), and **(c)** inserting arguments to the kernel function corresponding to all pointer sizes (in this example, `in_size` and `out_size`). G3 also generates framework specific host code for managing the GPU kernel launch (not included in the excerpt shown).
- The Python meta-program (left-hand side) accepts two arguments: (1) `kernel`, the AST node representing the GPU kernel function; and (2) `var`, the AST node representing the definition for the variable pointing to global memory that we wish to copy into shared memory (in our example, the `float *in` kernel parameter).
- The C++ code that declares, initialises, and fills a shared memory array is instantiated as a string (left:2), where `$0` and `$1` are respectively substituted with the variable type and name stored in the AST representation.
- The meta-program injects (left:4) the above code string into the kernel (right:5–10) *after* the `bounds_check` statement. We can identify this statement (right:4) since it is automatically generated by G3 for any application (see the above explanation). Because all kernels will exhibit this exact code pattern, we query the kernel for the first `if`-statement that immediately encloses a `return` statement (left:3).
- Finally, references to the original global pointer are replaced with references to the shared memory pointer. The kernel is queried for variable reference expression nodes with the original variable name (left:5), and matched references are instrumented, replacing them with the new buffer variable name (left:6–7). In the example kernel, there is only one reference to `in`, and it is replaced with a reference to `in_buf` (right:14).

3.2.2 ***Hotspot Loop Detection (Fig. 4).*** The aim of the *hotspot* loop detection is to identify computationally intensive regions of code that when optimised can substantially speedup the overall execution time of an application. In this section, we demonstrate how Artisan can dynamically extract runtime behaviour through code instrumentation to generate applications that self-report information. Our strategy for detecting hotspots is to create an augmented application that auto-profiles the execution time of (1) the overall

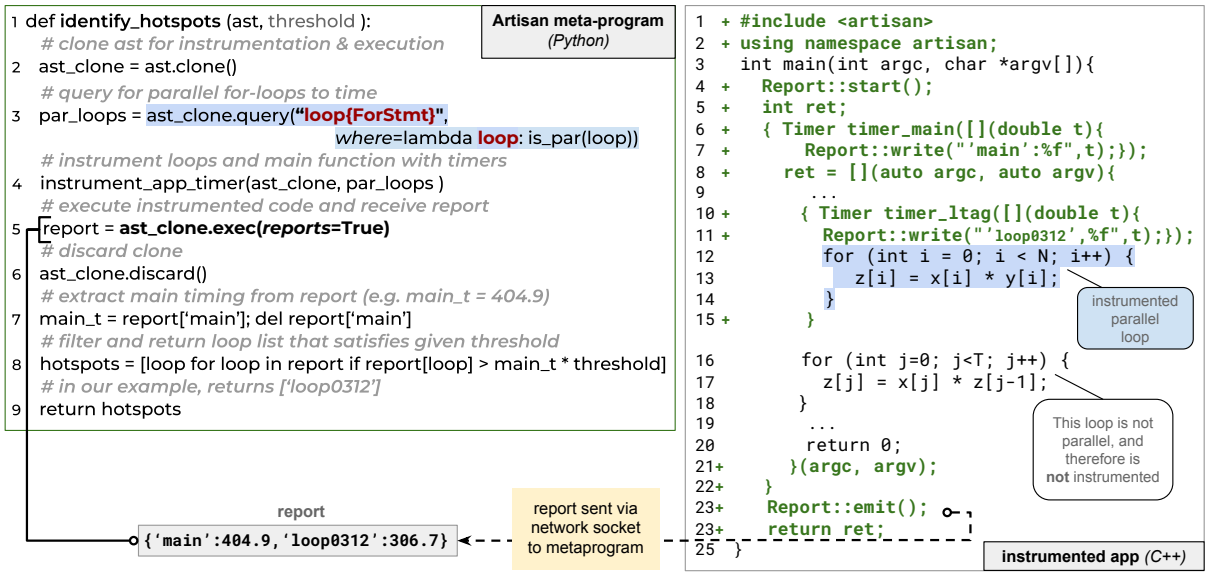

Figure 4: Artisan meta-program code (left) and instrumented HIP Kernel (right) based on design-flow pattern *Hotspot Loop Detection* (A1, Table 1). This example demonstrates Artisan's runtime execution and reporting mechanisms. Note that in this example, the function `instrument_app_timer` (left:4) injects all the lines into the HIP kernel (presented in +green) using the query and instrumentation mechanisms described in the previous example (Section 3.2.1).

application and (2) every parallel program loop. The meta-program operates as follows:

- The meta-program accepts two parameters: an `ast` representing the full application source-code, and a `threshold` which represents the fraction of overall execution time to be considered a hotspot (left:1). For instance, a 0.5 threshold indicates that a hotspot must execute for at least half of the overall execution time.
- We clone the `ast` (left:2) to avoid modifying the original code when generating the augmented version;
- We query all `for`-statements in the application, and filter the results with the `is_par` meta-program using polyhedral analysis (A3) to determine statically if a loop is parallel [10] (left:3).
- The `instrument_app_timer` function instruments the C++ code (left:4) and times the identified parallel loops (right:10–15), as well as the `main` function to capture the overall execution time (right:6–22). In addition, runtime primitives are injected to record the elapsed time for each code region (right:7, 11).
- Once the augmented application is built, it is executed (left:5). The meta-program waits for a report, sent by the application (right:23) at the end of its execution. The data received is translated into a map (dict) data-structure containing code region keys and corresponding timing values. The "main" key corresponds to the overall execution time, while the remaining correspond to uniquely identified loop regions (e.g. `loop0312`).
- At this stage, we have the timing report, and the instrumented clone can be discarded (left:6).
- The hotspots are now filtered by checking if the loop times pass the given threshold (left:7–8).

4 EVALUATION

4.1 Experimental Setup

To validate our approach, we implement two automated design-flows using Artisan which generate multi-threaded OpenMP CPU and HIP [1] GPU designs, respectively. Both design-flows are illustrated in Fig. 5, comprising of modular Artisan meta-programs which implement the patterns in our catalogue (Table 1).

Both design-flows operate on a complete unoptimised C++ application description, and begin with a common path. This starts with hotspot loop detection (A1), in which parallel application loops are identified (A3), timed (A2), and filtered based on a parameterised threshold of overall execution time (see Fig. 4). The loop with the longest execution time that meets the threshold is selected and extracted into an isolated function (G1), then target-independent pointer analysis and transforms common to both OpenMP and HIP are applied as applicable (A4, T1, T7). Next, the GPU and CPU design-flows diverge with target-dependent code generation (G3, G2), before a series of target-dependent transforms on the GPU design (T2, T3, T4, T5, T6). Finally, platform-specific DSE is performed in both design-flows. For OpenMP, the number of threads is tuned to minimise execution time on a known target (O1, T9, A2). For HIP, kernel launch parameters are tuned to maximise occupancy and minimise execution time on a specific GPU (O2, T8, A5, A6).

Our OpenMP designs are executed on a platform with two Intel Xeon Silver 4110 CPUs @ 2.10GHz, totalling 16 cores and 32 threads with simultaneous multi-threading (SMT), and are built with the g++

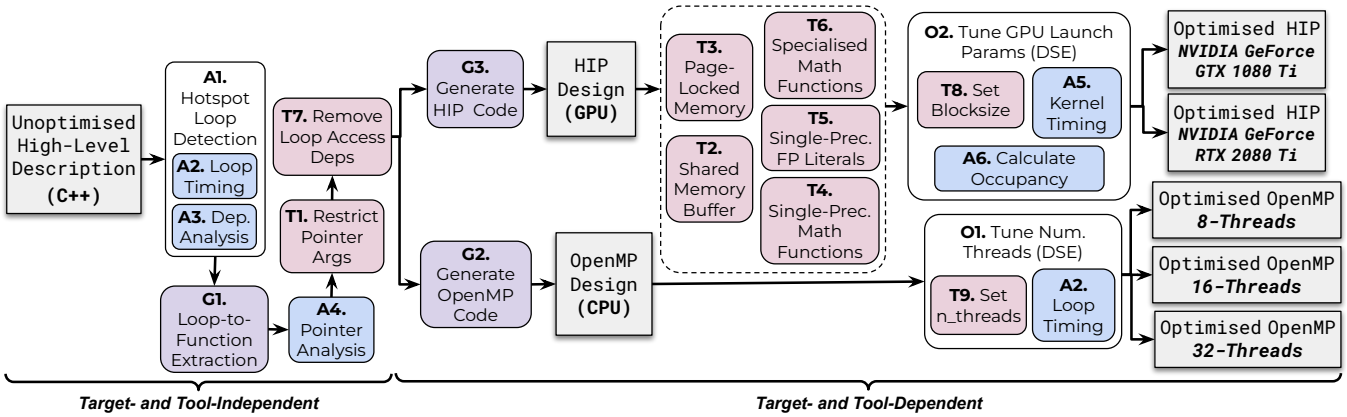

Figure 5: Overview of the implemented end-to-end HIP GPU and OpenMP CPU design-flows. Note that this design-flow is application-agnostic, and is designed to work on any C++ code without user intervention.

compiler using `-O2`. Our HIP designs target two NVIDIA GeForce GPUs: GTX 1080 Ti and RTX 2080 Ti, using the `hipcc` compiler with `-O2`. We evaluate our design-flows on three case-study applications in different domains: N-body particle simulation (physics) [16], Bezier surface generation (graphics) [6], and a Rush Larsen ODE solver (mathematics) [7]. These are representative examples of HPC applications with computationally intensive loops amenable for parallel acceleration. We choose workloads that require several minutes of execution time on a single-threaded CPU.

4.2 Reusability and Application-Agnosticity

To validate design-flow pattern reuse and application-agnosticity, Table 2 outlines the patterns used in each design-flow, and each pattern's applicability to our case-studies. Seven patterns are shared by both design-flows, this is more than half of the multi-threaded CPU design-flow, and almost half of the HIP GPU design-flow. By codifying design-flow patterns as modular meta-programs, they are implemented once then reused for different hardware targets. Furthermore, the target-dependent patterns in Fig. 5 are common to devices within a target family (e.g. NVIDIA GTX 1080 Ti and RTX 2080 Ti).

In addition, both design-flows are application-agnostic, operating on any high-level C++ description. Unlike manual optimisation efforts which need to be repeated and modified for every application, design-flows are codified once and applicable to multiple applications. As shown in the bottom row of Table 2, all ten patterns composing the CPU design-flow were applied to all three case-studies. Of the seventeen patterns composing the GPU design-flow, fourteen are applicable to all three case-studies. While specific transforms may not be universally suitable, the design-flows need not be modified for each application, but rather check individual pattern applicability and profitability programmatically. For instance, pattern **T6** (Table 1), which employs specialised math operations, was only applied to Rush Larsen, replacing expensive divide by square-root expressions (`1/sqrt(x)`) with optimised reciprocal square-root functions (`rsqrt(x)`). The other two case studies did not contain any expressions that could be replaced with specialised versions, so the design-flow skips the transform automatically.

4.3 Multi-threaded CPU Performance

For our multi-threaded CPU experiments, we consider three scenarios where we set a limit of 8, 16, and 32 available threads (see Fig. 6). In each scenario, the automated DSE (**O1**, Table 1) uses hill-climbing to identify the minimum number of CPU threads that maximises performance.

In general, execution time decreases with increasing threads. Execution with 8 threads achieves an 8 times speedup for N-Body Simulation compared to the single-threaded reference, and nearly doubles to 15 times with 16 threads available. For both Bezier Surface and Rush Larsen, a 7 times speedup is achieved on 8 threads, increasing to 12 times on 16 threads. The speedup is improved but not quite doubled, due to thread management and scheduling overhead.

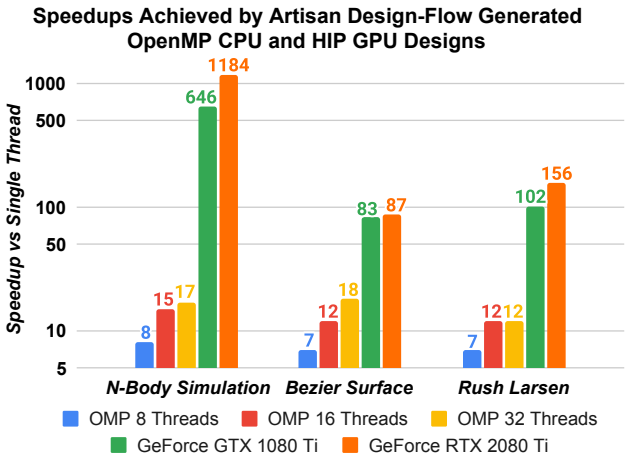

Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).

Table 2: Design-flow pattern reuse for different design-flows (Multi-Threaded CPU and HIP GPU) and applicability to evaluated case-studies. Pattern IDs referenced in Table 1. $N/3$ case-studies indicates pattern applicability to N of our three case-studies.

Pattern ID	A1	A2	A3	A4	A5	A6	G1	G2	G3	T1	T2	T3	T4	T5	T6	T7	T8	T9	O1	O2
MT-CPU	✓	✓	✓	✓			✓	✓		✓						✓		✓	✓	
HIP GPU	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
Case-Studies	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	1/3	3/3	3/3	3/3	1/3	1/3	3/3	3/3	3/3	3/3

Further increasing the number of threads from 16 to 32 provides a more modest speedup, since our CPU platform only contains 16 cores. However with the support of SMT, which allows two threads to maximise the utilisation of a single core, some performance gain can be expected. This is the case for N-Body Simulation and for Bezier Surface, which reach 17 times and 18 times speedup respectively compared to the reference implementation. However, for Rush Larsen, there is no observed execution time improvement on 32 threads compared to 16 threads, due to the complex mathematical logic which limits SMT execution.

4.4 HIP GPU Design-Flow Performance

Due to the massive parallelism available on the GPU targets, the generated GPU designs execute significantly faster than the reference sequential and multi-threaded CPU implementations. Our GPU design-flow automatically selects a block-size that maximises occupancy and minimises execution time for each case-study and specific GPU (**O2**, Table 1). In all cases, the observed performance, as explained below, is better on the RTX 2080 than the GTX 1080 (see Fig. 6). This is expected, as the RTX 2080 has roughly 20% more CUDA cores and also has wider cores with advanced features.

For N-Body Simulation, RTX 2080 execution is almost twice as fast as the GTX 1080, achieving, respectively, 1184 times and 646 times speedup compared to a single-threaded implementation. The HIP kernel requires 33 registers per thread, limiting the maximum occupancy on the GTX 1080 to 75%, while the RTX 2080 achieves 100% occupancy. Our N-Body Simulation design fully saturates both GPUs, requiring more work items to complete than concurrently available. With 1.6 times more active work items on the GTX 2080 Ti, we observe a 1.8 times performance improvement.

The generated Bezier Surface design is only slightly faster on the RTX 2080 than the GTX 1080 - 87 times vs. 83 times speedup compared to the reference sequential CPU design. Similar to N-Body Simulation, the generated Bezier surface HIP kernel requires 33 registers per thread, limiting the GTX 1080 maximum occupancy. The workload, however, does not saturate either GPU as the number of work items required is less than that concurrently available on either device, so the performance of the two devices is comparable.

The generated HIP kernel for Rush Larsen requires 255 registers per thread due to the complexity of the solver logic. This limits the occupancy achievable on the GTX 1080 to 12.5% and on the RTX 2080 to 25%. Application execution saturates the GTX 1080 design, but not the RTX 2080 design, achieving 102 times and 156 times speedup, respectively.

In all cases, our automatically generated designs achieve equivalent performance to our manually crafted versions. However, it is expected that a more experienced GPU developer could further improve performance with extra optimisation techniques, which could in turn be codified to extend the automatic design-flow. The presented performance gains for both CPU and GPU are achieved automatically and derived from unoptimised high-level descriptions. The performance comes free with little or no intervention from the application developer. Furthermore, the code generated is human-readable at the same level of abstraction as the original code, so developers can further fine-tune if necessary.

5 RELATED WORK

It is established that there are well-known, recurring methods for optimising heterogeneous applications. Several domain specific languages (DSLs) and libraries embed optimising transformations to abstract low-level GPU details and separate optimisation from behavioural application concerns [3][8][19][20]. These solutions enable efficient and customisable optimisation, but require effort and expertise. For instance, developers need to learn new programming models, DSL syntax, and/or familiarise themselves with available library functions, employing insight into target platforms to make design decisions. This effort needs to be repeated for each new application. With the proposed design-flow patterns, well-known optimisation methods can be captured, codified and coordinated into automated design-flows that can be reused across multiple applications by developers with or without in-depth hardware knowledge.

Various projects tackle programmatic mapping and/or optimisation. The approaches in both [12] and [24] generate OpenCL GPU code from data-parallel software inputs, profiling to map computations onto CPU or GPU targets. In [22], Haskell meta-programs tune GPU kernel launch parameters for designs expressed in an embedded DSL. In [23] automatic source-to-source transformations optimise CUDA stencil computations. Work under the ParaPhrase project uses pattern-based development and code refactoring techniques to transform sequential applications into parallel equivalents [2]. Togpu [13] similarly parallelises sequential code by performing C++ to CUDA transformations to generate GPU designs, with intermediate lowering based on Clang and LLVM. These automatic approaches effectively parallelise and/or optimise input computations and can be reused across applications. However, their design-flows are tightly coupled to an implementation based on a particular tool and/or framework, limiting customisability. In most cases, applications must be described in a way that explicitly exposes parallelism, requiring expertise from application developers and an effort that must be repeated for each application. Where sequential input applications are supported, lowering to an IR limits the scope for analysis and auto-generated source-code may lose the original structure, becoming difficult to maintain.

The Artisan meta-programming framework has previously been used to programmatically optimise FPGA designs starting from sequential C++ source-code [21], with efficient, customisable, and reusable optimisation strategies automated using meta-programs in order to decouple behavioural application descriptions from optimisation descriptions. In this paper, we employ this framework to support design-flow patterns, while providing a higher-level of abstraction above meta-program implementations, broadening the scope for reasoning about modular components and their coordination into design-flows for different targets, not tied to a particular implementation.

6 CONCLUSION

This paper introduces *design-flow patterns* as a means to capture common and recurring elements of design-flows for optimising C++ high-level application descriptions onto CPU and GPU targets. We report an initial catalogue of patterns that have known to be effective in accelerating CPU and GPU designs, and codify modular patterns using the meta-programming tool Artisan. Target-independent patterns are combined with target-dependent patterns to automate end-to-end programmatic design-flows that map unmodified sequential C++ application descriptions into optimised CPU and GPU designs. We apply our programmatic design-flows to three case-study HPC applications in different domains (physics, graphics, mathematics), and evaluate the performance of automatically generated OpenMP and HIP designs on three multi-core CPU and two GPU target platforms. We reuse the same design-flows for all three case-studies, and achieve up to 18 times speedup on a CPU platform with 32-threads and up to 1184 times speedup on an NVIDIA GeForce RTX 2080 Ti GPU compared to a sequential single-threaded reference implementation.

Ongoing and future work includes extending our pattern catalogue and programmatic design-flow strategies to support more advanced GPU optimisations leveraging the HIP programming model, incorporating FPGAs using OneAPI [9], as well as covering patterns which are specific to particular application domains.

ACKNOWLEDGMENTS

The support of SRC Artificial Intelligence Hardware Task 3020.001, AMD, Intel, and the U.K. EPSRC (EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1, EP/V028251/1) is gratefully acknowledged.

REFERENCES

[1] AMD. 2022. HIP Programming Guide v4.5. Webpage. Retrieved January, 2022 from https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html
[2] Christopher Brown, Marco Danelutto, Peter Kilpatrick, Kevin Hammond, and Sam Elliott. 2014. Cost-Directed Refactoring for Parallel Erlang Programs. *Int'l Journal of Parallel Programming* 42.
[3] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proc. of the 6th Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*.
[4] Andre Dehon, Joshua Adams, Michael Delorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. 2004. Design patterns for reconfigurable computing. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*.
[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP*.
[6] HeCBench. 2022. Bezier Surface. Webpage. Retrieved March, 2022 from https://github.com/zjin-lcf/HeCBench
[7] HeCBench. 2022. Rush Larsen. Webpage. Retrieved March, 2022 from https://github.com/zjin-lcf/HeCBench
[8] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. 2012. Declarative Parallel Programming for GPUs. In *Advances in Parallel Computing*, Vol. 22.
[9] Intel. 2022. Intel oneAPI DPC++ FPGA Optimization Guide. Webpage. Retrieved March, 2022 from https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html
[10] ISLPY. 2021. islpy 2020.2.2 Documentation. Webpage. Retrieved February, 2021 from https://documen.tician.de/islpy/index.html
[11] LLVM Developer Group. 2022. Clang: a C language family frontend for LLVM. Webpage. Retrieved March, 2022 from https://clang.llvm.org/
[12] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2016. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proc. of the 25th Int'l Conf. on Compiler Construction (CC 2016)*.
[13] Matthew Marangoni and Thomas Wischgoll. 2016. Paper: Togpu: Automatic Source Transformation from C++ to CUDA using Clang/LLVM. In *Visualization and Data Analysis*.
[14] Berna Massingill, Tim Mattson, and Beverly Sanders. 2000. A Pattern Language for Parallel Application Programs. In *Euro-Par 2000*.
[15] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (1st ed.). Addison-Wesley Professional.
[16] Maxeler App Gallery. 2015. N-Body Particle Simulation. Webpage. Retrieved January, 2020 from https://github.com/maxeler/NBody
[17] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc.
[18] NVIDIA Developer Zone. 2022. CUDA C++ Best Practices Guide. Webpage. Retrieved January, 2022 from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
[19] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *2011 IEEE Int'l Symp. on Parallel and Distributed Processing Workshops and Phd Forum*.
[20] Joel Svensson, Mary Sheeran, and Koen Claessen. 2008. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *Proc. of the 20th Int'l Conf. on Implementation and Application of Functional Languages (IFL'08)*.
[21] Jessica Vandebon, Jose G. F. Coutinho, Wayne Luk, and Eriko Nurvitadhi. 2021. Enhancing High-Level Synthesis Using a Meta-Programming Approach. *IEEE Trans. Comput.* 70, 12 (2021).
[22] Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan Newton. 2015. Meta-Programming and Auto-Tuning in the Search for High Performance GPU Code. In *Proc. of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*.
[23] Mohamed Wahib and Naoya Maruyama. 2015. Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery.
[24] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. 2015. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. In *ACM TACO '15*, Vol. 11.