Auto-Generating Diverse Heterogeneous Designs

Jessica Vandebon, Jose G. F. Coutinho and Wayne Luk
Department of Computing, Imperial College London, UK
{jessica.vandebon17, gabriel.figueiredo, w.luk}@imperial.ac.uk

***Abstract*—This paper presents a novel architecture for end-to-end design automation, facilitating high-level design portability across diverse technologies. We introduce programmatic, customizable and reusable design-flows capable of generating multiple implementations (e.g., CPU, GPU, FPGA) from a single technology-agnostic high-level application source. Notably, our approach incorporates design-flow branch points and automated path selection strategies, mitigating the manual effort currently needed for efficient design production, particularly for heterogeneous platforms. To validate our approach, we implement optimizing design-flows tailored to different hardware platforms. Through experiments on five AI and HPC benchmarks, we demonstrate significant speed improvements compared to single-threaded CPU execution. Our approach generates multi-thread CPU, CPU+FPGA, and CPU+GPU designs from a single high-level source description, achieving speedups of up to 30 times for OpenMP multi-thread CPU, 32 times for oneAPI CPU+FPGA, and 779 times for HIP CPU+GPU designs. We also showcase cost-effective implementations targeting heterogeneous computing platforms. Additionally, these performance advancements are accompanied by gains in developer productivity, quantified based on added lines of code.**

I. Introduction

High-level programming languages, such as C/C++, have long offered a portable way to express functionality across diverse CPU architectures, demonstrating resilience in adapting to evolving platforms. This adaptability relies on the capability of CPU compilers to leverage increasingly advanced features, such as specialized instruction sets and memory hierarchies, without human intervention, while providing simple and flexible programming interfaces, such as the use of OpenMP [1], to further enhance performance.

However, such progress in CPU compilers has not extended to hardware accelerators such as FPGAs and GPUs when employed as co-processors. As HPC and cloud systems adopt heterogeneous architectures with various types of accelerators, there is a growing need for efficient implementations from high-level portable descriptions. Despite the availability of tools such as high-level synthesis (HLS) and cross-platform frameworks like oneAPI [2] and OpenCL [3], disparities persist between unoptimized high-level descriptions and actual implementations, often requiring manual code restructuring and optimizations to maximize performance on each device. The process of generating an optimized heterogeneous design involves manual steps like code partitioning, mapping, and optimization, requiring developer expertise. Transitioning to new platforms entails employing different techniques and optimization strategies.

This works seeks to automatically generate diverse and efficient heterogeneous designs from a single high-level application description, while aiming to enhance design quality and developer productivity by ensuring portability across CPU platforms with hardware accelerators (FPGAs and GPUs) as co-processors. Employing **meta-programming** [4] [5], strategies for static code analysis, runtime behavior characterization, and source-to-source optimizations are programmatically encoded independently from functional description to maintain code readability and improve maintainability. Our approach is further refined to support **PSA-flows**, which employs branching sequences with Path Selection Automation (PSA) to enable automatic specialization of different targets and optimization strategies (see Fig. 1). This capability facilitates programmatic decision-making, addressing the complexity of selecting the most suitable target platform for a given computation.

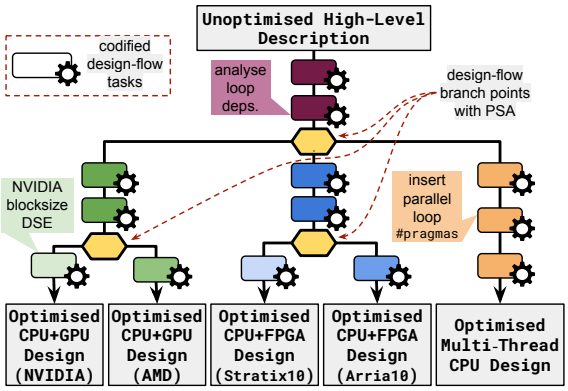

Fig. 1. Our automated approach for diverse *PSA-flows* stems from modular, codified *design-flow tasks* and *branch points* with path selection automation (PSA). Each *task* encapsulates a distinct code analysis, transformation, or optimization, and *branch points* allow specialization at various levels (e.g., for different targets, vendors, and/or optimization strategies).

This paper makes the following key contributions:

- **(1)** a novel approach automating custom PSA-flows for portable high-level descriptions across diverse hardware targets (Section II-B);
- **(2)** a complete implementation of the approach using Artisan meta-programs (Section III);
- **(3)** an evaluation of the approach (Section IV).

The paper is organized as follows: Section II provides an overview of our approach; Section II-A covers meta-programming design-flow tasks; Section II-B explains PSA-flow automation; Section III outlines implemented PSA-flows; Section IV presents the evaluation; Section V discusses related research; Section VI concludes and outlines future work.

II. Approach

This work addresses the significant technical challenge of automating the optimization of portable high-level code across diverse hardware targets, including HPC and cloud platforms. The tasks involve identifying and accelerating code hotspots, determining suitable mappings for specialized processors, and adapting source code with specialized languages and programming models, followed by device- and framework-specific optimizations. The complexity arises from factors such as memory footprint, data movement, and arithmetic intensity, contributing to a massive and non-linear design-space. The manual execution of these tasks proves challenging for non-experts, prompting the proposal of an automated approach. Our approach encodes and automates expertise within a design-flow, enabling specialization for different targets, vendors, and strategies using the PSA-flow architecture, thereby addressing challenges related to tedium, error-proneness, and the need for individualized efforts in each application and target. The encoding of design-flow tasks and the orchestration of PSA-flows are covered next.

A. Codifying Design-Flow Tasks

Meta-programs [4] leverage programmatic access to source code, tools, and platforms to automate design-flow tasks. Fig. 2 illustrates an example meta-program, described in pseudocode, which encodes an automated DSE task that unrolls FPGA kernel loops iteratively until resources are overmapped, thus maximising parallelism and resource utilization of the FPGA target. The operation of this meta-program on an example application is depicted on the right, which we explain next.

The meta-program takes as input a C++ source file (`src`) and the name of a function representing the FPGA kernel (`kernel_name`). In this example, partitioning and mapping have already taken place, so the kernel function is known. The meta-program outputs a modified source file (`mod_src`) with loops unrolled based on the results of the DSE, which operates as follows.

First, an internal abstract syntax tree (AST) representation of the source-code is created, illustrated by the purple tree structure on the right. The AST is *queried* for all outermost `for`-loops enclosed in the kernel function. In the example, there is one match, catching nodes representing a function (`knl()`) and enclosed top level `for`-loop, highlighted in red. The query does not match the second, nested `for`-loop within `knl()` since it is not outermost. Moreover, none of the loops in the non-kernel function (`main()`) are matched.

In a DSE loop, each matched loop is *instrumented* with a `#pragma unroll` directive. As such, the source-code is directly modified with a newly inserted pragma. An FPGA compiler tool is then run to gather estimated resource utilisation. This may, for example, run a partial compile with Intel's oneAPI tools to generate a high-level design report. The meta-program checks the generated report for estimated LUT usage, and doubles the unroll factor at every iteration, continuing the DSE until LUTs are overmapped (utilisation$> 90\%$). The

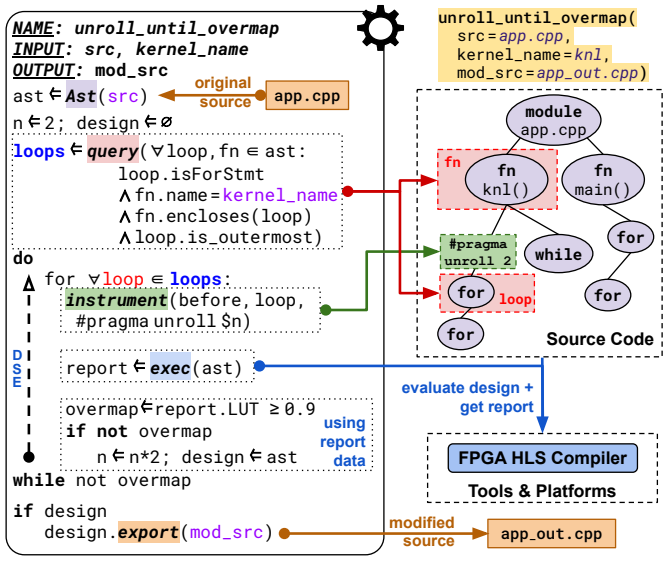

Fig. 2. Meta-programs [4] provide programmatic access to source-code, tools, and platforms, with built-in mechanisms for code querying and instrumentation, tool configuration, application execution, and platform monitoring. This enables self-contained analysis, transformation, and optimisation tasks to be codified and maintained separately from application descriptions. This example illustrates a meta-program written in pseudocode that unrolls program loops iteratively until the design overmaps a particular target FPGA.

final unrolled design is then exported to a new source file, `app_out.cpp`.

This example provides a glimpse of the extensive range of tasks that can be accomplished using meta-programs employing the query and instrument mechanisms mentioned earlier. Developers can effectively encode various types of static and dynamic analyses, as well as source-to-source transformations and programmatic DSE. In Fig. 4, we present a list of meta-programs utilised in this paper.

B. Orchestrating PSA-Flows

To construct a design-flow with a predetermined optimization strategy tailored to specific application domains or targets, a set of codified design-flow tasks must first be orchestrated. These tasks can be linearly composed into a sequence, but for supporting diverse targets and strategies within a single design-flow, branching is essential to accommodate varying task sequences at each level of specialization. Branch points in a PSA-flow, depicted by yellow blocks in Fig. 1 and Fig. 4, introduce divergence, enabling the consideration of different target types, devices within hardware/vendor families, or strategies based on distinct goals. These branches lead to increasingly specialized designs, requiring decisions based on specific requirements or objectives, a process facilitated by programmatic, customizable PSA at branch points.

Fig. 3 illustrates a branch point with a PSA strategy that selects a suitable device mapping based on information accrued from target-independent analysis tasks that are currently implemented in our repository. These tasks are captured as meta-programs, and executed with the reference unoptimised

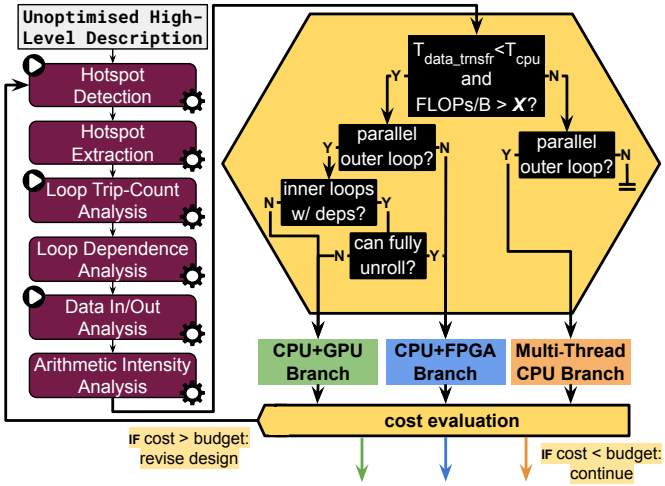

Fig. 3. An example PSA strategy for a branch point that specialises for different target types (GPU+CPU, FPGA+CPU, or Multi-Thread CPU) targeting, for instance, heterogeneous clouds. The strategy uses information obtained by a series of target-independent analyses as well as analytical cost evaluation. ◘ indicates a dynamic analysis that requires program execution.

high-level application description as input. Initially, hotspot loop detection and extraction are run on the input source-code. Hotspot detection instruments the application with loop timers and executes the instrumented code to dynamically identify time-consuming loops as candidates for acceleration. Once a hotspot is identified, it is extracted into an isolated function for further analysis and eventual offloading, replacing the original loop with a function call. This covers the partitioning stage of the design-flow. Next, various analyses are run on the new hotspot function, including dynamic loop trip-count analysis, static loop dependence analysis, dynamic data movement analysis, and static arithmetic intensity analysis.

The first step in the example PSA strategy is to compare the estimated time for data transfer to and from an accelerator (determined using data movement analysis and known device transfer bandwidths) to the recorded hotspot execution time on a single CPU thread. If data transfer is expected to exceed CPU execution time ($T_{data\_trnsfr} > T_{CPU}$), or if the arithmetic intensity ($FLOPs/B$) indicates that the hotspot is memory bound ($< X$, where X is a tunable parameter), there is no benefit to offloading the hotspot to an accelerator (GPU or FPGA). In this case, the strategy checks if the outer hotspot loop is parallel, in which case it is mapped for multi-threaded CPU optimisation, or not, in which case the design-flow terminates without modifying the input high-level reference.

If it is worthwhile to offload the hotspot to a GPU or FPGA co-processor (that is, if $T_{data\_trnsfr} < T_{CPU}$ and $FLOPs/B > X$), the strategy checks if the outer hotspot loop is parallel. If it is parallel, a GPU will likely perform well due to efficient data parallel execution. However, if there are any inner loops with loop-carried dependencies and fixed bounds under a certain threshold ('fully unrollable' on an FPGA), the FPGA could achieve better performance through pipelined execution. For a parallel outer hotspot loop, the decision between GPU and FPGA is made accordingly. If the outer hotspot loop is not parallel, the hotspot is mapped for FPGA optimisation. Note that while this strategy has proven effective empirically for our benchmarks and experiments, it could be adjusted to support different domains or target types.

After a path has been selected based on application characteristics, the example PSA-flow uses information about the selected target's capabilities to evaluate the cost of the design. Cost, in this context, could be based on performance, power, and/or monetary requirements, with various models available to predict these metrics on different targets [6]. If the evaluated cost exceeds a user-specified budget, the PSA-flow feeds back and the design is revised with updated information. Otherwise, the PSA-flow continues with the selected branch. With access to a full application representation, data collected by analysis tasks, and knowledge of target hardware capabilities, there is considerable opportunity for sophisticated PSA strategies incorporating, for example, machine-learning (ML) techniques to make intelligent decisions, which we are considering for future work.

To implement PSA-flow strategies, developers must take into account a variety of trade-offs. First, there are implications to the placement of branch points. The closer they are to the final implementation, the greater the opportunity for reuse of design-flow tasks across diverse targets. However, there could be cases where specialising too late might miss opportunities for optimisation. Second, developers need to consider different mechanisms used to make decisions at branch points. For example, when selecting paths between two different FPGA types, a PSA strategy could use the application representation to run performance estimation, bit-accurate simulation, or full compilation and synthesis to select the most profitable accelerator based on a certain goal (e.g. performance, power, or area). These mechanisms can vary in terms of the time taken to produce results and the level of accuracy achieved, and thus developers need to make decisions suitable and practical for their use cases (e.g. quick experimentation or fine-grained cost optimisation).

III. Implementation

To implement PSA-flows, we use the Artisan meta-programming framework [4] to codify design-flow tasks. Artisan provides a uniform Python development environment for code analysis, manipulation, and execution. In addition, we provide a high-level Python interface for developers to seamlessly compose their design-flow tasks and branch points.

Fig. 4 includes a list of the tasks we currently have codified in our repository, and illustrates our implemented PSA-flow for generating multi-thread CPU, CPU+FPGA, and CPU+GPU designs from a single C++ application source. The PSA-flow begins execution with target-independent tasks. Similar to the example in Fig. 3, the code is partitioned by identifying the most time-consuming program loop and extracting it into a *kernel* function (i.e. hotspot detection/extraction). Then, a series of static and dynamic analyses are automatically performed on the kernel: dynamic pointer alias analysis to ensure

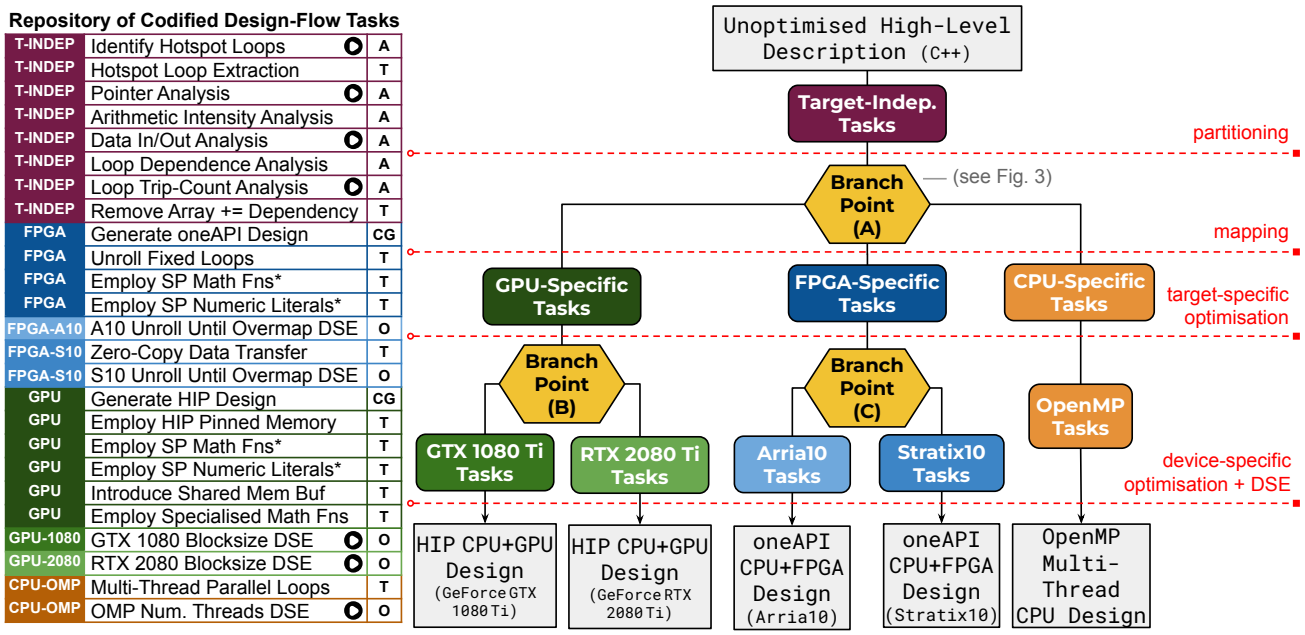

Fig. 4. Our implemented PSA-flow supporting HIP CPU+GPU, oneAPI CPU+FPGA, and OpenMP CPU platforms. The automated PSA-flow is composed of target-independent and -dependent design-flow task meta-programs from our repository included on the left, where A, T, CG, and O signify task classifications (Analysis, Transform, Code-Generation and Optimisation), and ● indicates a dynamic task that requires program execution.

that pointer arguments do not reference overlapping memory locations; static arithmetic intensity analysis to indicate if computations are compute- or memory-bound; dynamic data movement analysis to quantify data transfer requirements; static loop dependence analysis to identify loop-carried dependencies; and dynamic loop trip-count analysis to characterise the behaviour of program loops.

At branch point **A**, there are three paths corresponding to the supported targets: CPU+GPU, CPU+FPGA, and multi-thread CPU. A mapping decision is made following the PSA strategy from the yellow hexagon in Fig. 3, using the information accrued by the analysis tasks. Each path after the branch point comprises target-dependent tasks, beginning with generating the framework specific management code required for each programming model (HIP, oneAPI, or OpenMP), followed by target-dependent optimisations. For instance, unrolling fixed-size FPGA loops, or introducing suitable GPU shared memory buffers. Within the CPU+FPGA and CPU+GPU paths there are further branch points specialising for different FPGA and GPU devices (Arria10 or Stratix10 FPGAs, GeForce GTX 1080 or RTX 2080 Ti GPUs) often found in heterogeneous clouds.

Device-specific branching (**B**, **C**) enables fine-grained specialisation to support device-specific optimisations and DSE. For example, taking advantage of zero-copy host memory with oneAPI is supported on Intel Stratix10 FPGAs with support for unified shared memory (USM), but not on Arria10s. Moreover, the launch parameters that maximise occupancy and minimise latency for a GPU kernel (e.g. blocksize) are likely different for the same computation executed on different GPUs, as is the factor by which a given kernel loop can be unrolled on different FPGA devices. As such, device-specific tasks are performed in these final branches before outputting the final designs. The current implementation automatically selects *both* paths at **B** and **C**, generating two CPU+GPU designs or two CPU+FPGA designs respectively, with one for each device.

Note that since Artisan ASTs closely mirror the source-code as written without lowering, output implementations are human-readable and can be further hand-tuned if desired. Furthermore, the approach is not limited to the programming models or vendor device types in our implemented PSA-flow. To target new technology, target-specific design-flow tasks can be implemented and seamlessly plugged in.

IV. Evaluation

A. Experimental Setup

To validate our approach, we apply the implemented PSA-flow (Fig. 4) to five HPC and AI applications, namely: N-Body Simulation, K-Means Classification, AdPredictor, Rush Larsen ODE Solver, and Bezier Surface Generation. For CPU experiments, we compile with g++ and target an AMD EPYC 7543 32-Core Processor @2800MHz. For GPU experiments, we target two NVIDIA platforms with GeForce GTX 1080 Ti and RTX 2080 Ti GPUs, using the hipcc compiler. For FPGA experiments, we target two Intel FPGA platforms with PAC Arria10 and Stratix10 boards, using the *dpcpp* compiler.

The following subsections cover an evaluation of our PSA-flow automation approach in terms of performance and development productivity.

B. Performance of Auto-Generated Diverse Designs

To evaluate our automated PSA-flow (Fig. 4), we run it using two modes on each of the five benchmark applications:

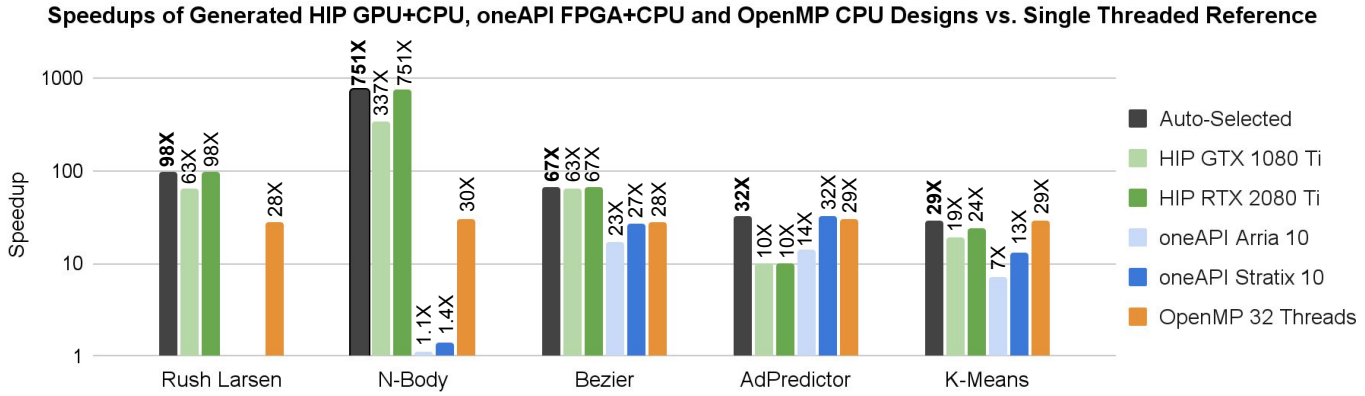

Fig. 5. Accelerated hotspot region speedups of the automatically generated designs from Fig. 4 compared to the input, unoptimised reference executed on a single CPU thread. 'Auto-Selected' bars represent the designs generated when using the PSA strategy from Fig. 3 in branch point A.

- **Informed**. We execute the PSA-flow as outlined in Section III, incorporating the PSA strategy from Fig. 3 at branch point **A**. This allows for the generation of either *one* OpenMP multi-threaded CPU design, *two* HIP CPU+GPU designs (1080 Ti and 2080 Ti), or *two* oneAPI CPU+FPGA designs (Arria10 and Stratix10).
- **Uninformed**. We modify branch point **A** to automatically select *all* paths, generating all design versions (one OpenMP multi-threaded CPU, two HIP CPU+GPU, *and* two oneAPI CPU+FPGA designs) for *all* applications.

Fig. 5 shows the speedups obtained for the hotspot regions of each generated design compared to the input reference software implementation executed on a single CPU thread. Note that for each benchmark, the PSA-flow applied is the same. The 'Auto-Selected' bar (leftmost) for each application represents the performance of the design generated by the *informed* PSA-flow. In the case where CPU+GPU or CPU+FPGA designs are generated, the Auto-Selected bar represents the fastest of the two generated designs (i.e. 1080 Ti or 2080 Ti for CPU+GPU, Arria10 or Stratix10 for CPU+FPGA). As shown, the *informed* PSA-flow selects the best target for all of the five benchmarks.

i. Multi-Thread CPU Performance (OpenMP). The OpenMP PSA-flow path selects the maximum number of threads available automatically for each of the five benchmarks with the *"OMP Num Threads DSE"* task, achieving speedups ranging from 28-30X. Since each application is embarrassingly parallel, we observe speedups close to the number of cores (32), as expected. For K-Means Classification, the OpenMP multi-threaded implementation achieves the best performance of the five generated designs. Since the identified hotspot is a memory-bound computation, the informed PSA strategy automatically selects the multi-thread CPU branch.

ii. CPU+GPU Performance (HIP). The HIP CPU+GPU designs similarly take advantage of the benchmarks' parallelism to achieve significant speedups. The CPU+GPU designs are best for the Rush Larsen solver, N-Body Simulation, and Bezier Surface Generation. All three of these applications are automatically determined to be compute bound. Rush Larsen comprises a single outer loop, N-Body Simulation comprises a double outer loop nest with bounds unknown at compile time, and Bezier Surface Generation contains a complex multi-nested inner loop structure. As such, the informed PSA strategy automatically selects the CPU+GPU branch for all three.

In the device-specific GPU+CPU paths, the DSE tasks for blocksize selection tailored to the GTX 1080 and RTX 2080 GPUs aim to minimize execution time and maximize occupancy for each benchmark. Generally, the RTX 2080 outperforms the GTX 1080, as expected due to its larger number of cores. For the Rush Larsen benchmark, the RTX 2080 achieves 1.6 times faster performance than the GTX 1080 (98X vs 63X). However, due to the complexity of the ODE solver logic, the GPU design requires 255 registers per thread, saturating the GTX 1080 but not the RTX 2080. Conversely, the N-Body Simulation workload fully saturates both GPUs, allowing the RTX 2080 to achieve more than 2 times faster performance than the GTX 1080 (751X vs 337X). For Bezier Surface Generation, where neither GPU is fully saturated, the difference in performance is less substantial (67X vs 63X), yet still significant compared to CPU performance.

iii. CPU+FPGA Performance (oneAPI). The oneAPI CPU+FPGA optimization strategy attempts to maximise pipeline parallelism on the target FPGAs, with device-specific *"unroll until overmap DSE"* tasks. In general for the CPU+FPGA designs, the Stratix10 performs better than the Arria10, as expected since it is a newer, larger FPGA with more advanced features, including support for zero-copy host memory.

This unrolled FPGA execution model is well suited to AdPredictor, for which the Stratix10 CPU+FPGA design achieves the best performance across all targets (32X speedup). The computations in AdPredictor are highly amenable to pipelined execution on an FPGA, with simple fixed-bound, fully-unrollable inner loops and an outer loop that can be unrolled to maximise resource utilisation on each FPGA target without affecting its initiation interval (II=1).

TABLE I
ADDED LINES OF CODE (LOC) FOR EACH GENERATED DESIGN COMPARED TO THE REFERENCE UNOPTIMISED HIGH-LEVEL SOURCE

Application	LOC Δ Per Design					Total LOC Δ (5 designs)
	OMP	HIP 1080	HIP 2080	oneAPI A10	oneAPI S10	
Rush Larsen	+.4%	+6%	+6%	n/a	n/a	n/a
N-Body	+2%	+37%	+37%	+52%	+69%	+197%
Bezier	+2%	+26%	+26%	+34%	+42%	+130%
AdPredictor	+2%	+31%	+31%	+42%	+63%	+169%
K-Means	+4%	+81%	+81%	+101%	+147%	+414%
Average	+2%	+36%	+36%	+57%	+81%	+212%

Although CPU+FPGA design descriptions for Rush Larsen are generated, their performance results are not included. This is because the resulting designs are sizeable and exceed the capacity of our current FPGA devices. As a result, additional strategies, like finer partitioning (e.g. loop splitting) and more effective resource area reduction, need to be incorporated into the PSA-flow. However, these adjustments may potentially impact performance negatively.

C. Development Productivity

In this section, we validate the efficacy of our approach towards enhancing development productivity. This validation is achieved through quantifying the increase in lines of code (LOC) for each automatically generated design in comparison to the input source reference. To facilitate a comprehensive assessment, we generate five heterogeneous designs for each benchmark under consideration. However, it is imperative to note that the generated CPU+FPGA designs for Rush Larsen are not synthesizable, as previously mentioned, and therefore, they are excluded from our LOC evaluation.

Table I showcases the outcomes of our analysis. The generation of five new implementations for a single application requires, on average, an additional 212% of the reference source-code LOC. Using automated PSA-flows removes the burden of manually writing this code from developers, thereby streamlining the design process and subsequently improving productivity. Note that LOC counts solely serve as a rudimentary indicator and do not encapsulate the expertise requisite for manually crafting code to map and optimize each application onto distinct hardware targets. Consequently, our LOC-based estimation presents a conservative perspective on the reduction in development effort.

Although the initial effort required to encode PSA-flows may seem substantial, its benefits become evident upon subsequent applications. Once codified, PSA-flows can be readily applied across various benchmarks, leading to significant time and resource savings in the long run. Furthermore, by encapsulating design-flow tasks into meta-programs, their adaptability and versatility are greatly enhanced. These meta-programs serve as building blocks that can be seamlessly integrated into customized PSA-flows, amplifying their utility and value proposition.

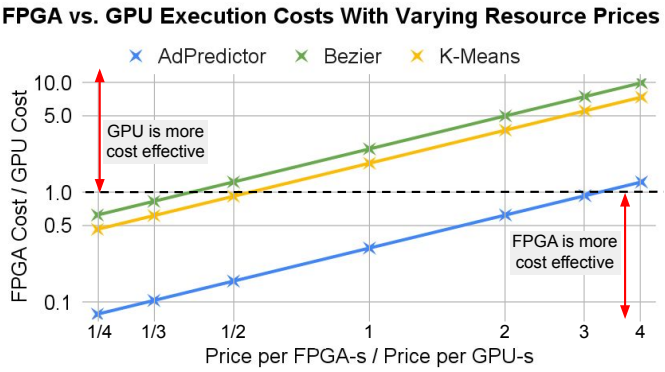

Fig. 6. Relative costs of FPGA vs. GPU execution for varying resource prices. FPGA-s and GPU-s correspond to 1s of execution on an FPGA or GPU.

D. Cost and Performance Trade-offs

With a set of generated diverse designs available for different targets (e.g. using the *uninformed* PSA-flow), there is scope for runtime experimentation beyond just identifying the best performing resource for a particular application and workload. For instance, performance models could be derived and used to inform runtime mapping decisions in heterogeneous cloud platforms [7] [8]. In a cloud environment particularly, factors such as cost also need to be considered. With models to predict performance on different resources and known cloud resource prices, computations can be mapped at runtime to minimise cost. Thus, the most performant design for a given application and workload might not be the most cost effective.

Cloud resources are typically priced based on the time for which they are provisioned (e.g. AWS EC2 [9] On-Demand instances are charged per hour). Costs for different resources depend on supply and demand considerations as well as platform traffic. Resource costs may be variable, with discounts at off-peak hours, for example. Fig. 6 shows the relative cost of FPGA and GPU execution for three applications based on the Stratix10 and 2080 Ti results from Fig. 5, illustrating different scenarios:

- if the FPGA price per unit time is $>$ 3.2 times the GPU price, it is more cost effective to execute on the CPU+GPU 2080 Ti platform, although AdPredictor executes fastest on the Stratix10 CPU+FPGA platform;
- if the GPU price is $>$ 2.5 times the FPGA price, it is more cost effective to execute Bezier on the Stratix10 CPU+FPGA platform, despite being slower than executing on the 2080 Ti CPU+GPU platform.

Similar analysis could be used to identify the most energy efficient implementation for a specific application. When all targets as well as factors such as cost, energy, and performance requirements are considered, the mapping problem becomes more nuanced. The ability to automatically generate multiple diverse implementations from a single technology-agnostic description facilitates streamlining experimentation in such scenarios, including deriving performance and cost models for different applications across multiple targets.

V. Related Work

There have been various efforts addressing issues similar to those covered by this paper. Table II compares approaches for partitioning, mapping, and optimising applications onto diverse hardware targets.

Cross Platform Frameworks (e.g. OpenMP [1], oneAPI [2], OpenCL [3]) support compilation of a single source description onto multiple targets (e.g. CPUs, GPUs, and/or FPGAs). However, they require manual partitioning, mapping, and optimisation. Furthermore, while they in principle support compilation of identical code for diverse targets, in practice substantial refactoring is necessary to achieve desired performance on each device.

Domain Specific Languages (DSLs) simplify the optimisation of specialised hardware designs by providing built-in constructs that are specifically tailored to a certain domain or platform. For instance, HeteroCL [10] is a Python-based DSL designed for FPGA configurations. It distinguishes between algorithm description and hardware customisation. Similarly, Halide [11] is a DSL embedded in C++ that is used for constructing CPU and GPU image processing pipelines. It separates the description of the algorithm from its scheduling. Both HeteroCL and Halide are designed to support a specific target (CPU, GPU, or FPGA) for any given design. However, they require that developers rework computation descriptions, manually partition and map their code onto suitable devices, and conduct optimisations solely at the kernel level.

Compiler frameworks, like Delite [12] and MLIR [13], facilitate development of customised DSLs with fine-grained optimisations, and flexibility to support multiple targets and full applications. While they enable effective built-in optimisations, they require developers to rework code and manually partition and map computations across targets. Compiler expertise is needed to develop new DSLs using them.

Automatic HLS DSE tools support programmatic black-box or analytical systematic DSE with the goal of generating optimised HLS FPGA designs considering various trade-offs (e.g. area, performance). These tools typically consider only directives-based optimisations ([14] [15] [16]), tightly coupled to a particular HLS compiler (e.g. Merlin [17], Vivado HLS [18]). ScaleHLS [19] is a full HLS compiler built on MLIR with an embedded DSE engine that considers multi-level optimisations (i.e. loops, graph, directives). These tools effectively optimise designs but are limited to FPGA targets and kernel scopes, and do not cover partitioning or mapping.

Automated partitioning and/or mapping tools, like StreamBlocks [20] and GenMat [21], aim to derive optimal code partitions and device mappings. StreamBlocks is an HLS compiler that automatically identifies code regions apt for hardware execution, using performance models to optimise partitioning and minimise execution time. However, it is limited to CPU+FPGA targets with implicit mapping. On the other hand, GenMat is an ML-driven tuner for heterogeneous platforms. It wraps applications in meta-programs that expose tunable parameters and uses automatic profiling and ML-based modelling to choose a target mapping. However, designs for different targets must already be available. Other runtime mapping approaches [22] [23] [24] select a device for computation execution at runtime, but do not aim to deduce mappings from high-level code before generating specialised designs.

TABLE II
Comparison of design approaches that partition (**P**), map (**M**), and/or optimise (**O**) applications onto specialised hardware.

Approach	P	M	O	Multiple Targets	Scope
Cross-Platform Frameworks [1]–[3]				✓	Full App.
HeteroCL [10]			✓		Kernel
Halide [11]			✓		Kernel
Delite [12]			✓	✓	Full App.
MLIR [13]			✓	✓	Full App.
HLS DSE [14]–[16], [19]			✓		Kernel
StreamBlocks [20]	✓				Full App.
GenMat [21]		✓	✓	✓	Kernel
Design-Flow Patterns [5]	✓		✓		Full App.
This Work	✓	✓	✓	✓	Full App.

Meta-Programming Design-Flow Patterns [5] catalogues and encodes modular, reusable design-flow tasks as 'patterns' for reuse in design-flows for different targets and optimisation strategies. Their approach covers automatic code partitioning and optimisation, but only supports *linear* design-flows for specific targets (e.g. CPU+GPU, or multi-thread CPU). As such, there is no support for automated mapping.

This work covers automated code partitioning, mapping, *and* optimisation, capturing the full scope of applications. It supports adaptability across *various* diverse targets within a single integrated PSA-flow, in contrast to manual cloud GPU and FPGA designs [25].

VI. Conclusion

This paper proposes a novel design-flow automation approach with potential for disrupting high-level design.

Our focus on unified PSA-flows with branch points enables a new way of understanding and documenting design development: algorithmic-level branch points separate optimisations applicable to multiple algorithms from those specific to a domain, while device-level branch points separate optimisations applicable to multiple devices types from those specific to a particular device. We demonstrate how unified PSA-flows can enhance the portability of high-level designs across specialised heterogeneous platforms, achieving speedups of up to 30 times for multi-thread OpenMP, 32 times for oneAPI CPU+FPGA, and a striking 779 times for HIP CPU+GPU designs, when compared to single-threaded references.

Current and uture work includes extending our repository of design-flow task meta-programs, developing sophisticated ML-based PSA strategies, and demonstrating the effectiveness of PSA-flows for various cloud and edge computing platforms.

Acknowledgement. The support of UK EPSRC (grant number EP/X036006/1, EP/V028251/1, EP/L016796/1 and EP/N031768/1), AMD, SRC and Intel is gratefully acknowledged.

REFERENCES

[1] "OpenMP," accessed Jan-2023. [Online]. Available: https://www.openmp.org/

[2] "oneAPI: A New Era of Heterogeneous Computing," accessed Jan-2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[3] "OpenCL Overview," accessed Jan-2023. [Online]. Available: https://www.khronos.org/opencl/

[4] J. Vandebon, J. G. F. Coutinho, W. Luk, and E. Nurvitadhi, "Enhancing High-Level Synthesis Using a Meta-Programming Approach," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2043–2055, 2021.

[5] J. Vandebon, J. Coutinho, and W. Luk, "Meta-Programming Design-Flow Patterns for Automating Reusable Optimisations," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART2022, 2022.

[6] K. O'Neal and P. Brisk, "Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 763–768.

[7] J. Vandebon, J. G. F. Coutinho, W. Luk, E. Nurvitadhi, and M. Naik, "Enhanced Heterogeneous Cloud: Transparent Acceleration and Elasticity," in *International Conference on Field-Programmable Technology (FPT)*, 2019, pp. 162–170.

[8] J. Vandebon, J. G. F. Coutinho, and W. Luk, "Scheduling Hardware-Accelerated Cloud Functions," *J. Signal Process. Syst.*, vol. 93, no. 12, p. 1419–1431, dec 2021. [Online]. Available: https://doi.org/10.1007/s11265-021-01695-7

[9] Amazon Web Services, "*Amazon EC2*," accessed Apr-2020. [Online]. Available: https://aws.amazon.com/ec2/

[10] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, 2019.

[11] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing," *Commun. ACM*, vol. 61, no. 1, p. 106–115, dec 2017. [Online]. Available: https://doi.org/10.1145/3150211

[12] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, 2014.

[13] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021.

[14] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators," 2020. [Online]. Available: https://arxiv.org/abs/2009.14381

[15] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-Traversing Design Space Exploration for High Level Synthesis," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 210–217.

[16] Q. Sun, T. Chen, S. Liu, J. Miao, J. Chen, H. Yu, and B. Yu, "Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 46–51.

[17] Xilinx, "*Introduction to the Merlin Compiler for FPGA Accelerator Designs*," accessed Jan-2023. [Online]. Available: https://github.com/Xilinx/merlin-compiler/discussions/2

[18] Xilinx Vivado, "Vivado Design Suite User Guide: High-Level Synthesis," 2014. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[19] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," 2021. [Online]. Available: https://arxiv.org/abs/2107.11673

[20] M. Emami, E. Bezati, J. W. Janneck, and J. R. Larus, "Auto-Partitioning Heterogeneous Task-Parallel Programs with StreamBlocks," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 398–411. [Online]. Available: https://doi.org/10.1145/3559009.3569659

[21] N. Zhang, A. Srivastava, R. Kannan, and V. K. Prasanna, "GenMAT: A General-Purpose Machine Learning-Driven Auto-Tuner for Heterogeneous Platforms," in *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*, 2021, pp. 1–9.

[22] A. Chikin, J. N. Amaral, K. Ali, and E. Tiotto, "Toward an Analytical Performance Model to Select between GPU and CPU Execution," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 353–362.

[23] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-Learning-Based Performance Heuristics for Runtime CPU/GPU Selection," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, ser. PPPJ '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 27–36. [Online]. Available: https://doi.org/10.1145/2807426.2807429

[24] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 149–160. [Online]. Available: https://doi.org/10.1145/2464996.2465007

[25] M. Shepovalov and V. Akella, "FPGA and GPU-based acceleration of ML workloads on Amazon cloud - A case study using gradient boosted decision tree library," *Integration, the VLSI Journal*, vol. 70, pp. 1–9, 2020.