Trireme: Exploration of Hierarchical Multi-Level Parallelism for Hardware Acceleration

GEORGIOS ZACHAROPOULOS, Harvard University, USA
ADEL EJJEH, University of Illinois at Urbana-Champaign, USA
YING JING, University of Illinois at Urbana-Champaign, USA
EN-YU YANG, Harvard University, USA
TIANYU JIA, Harvard University, USA
IULIAN BRUMAR, Harvard University, USA
JEREMY INTAN, University of Illinois at Urbana-Champaign, USA
MUHAMMAD HUZAIFA, University of Illinois at Urbana-Champaign, USA
SARITA ADVE, University of Illinois at Urbana-Champaign, USA
VIKRAM ADVE, University of Illinois at Urbana-Champaign, USA
GU-YEON WEI, Harvard University, USA
DAVID BROOKS, Harvard University, USA

The design of heterogeneous systems that include domain specific accelerators is a challenging and time-consuming process. While taking into account area constraints, designers must decide which parts of an application to accelerate in hardware and which to leave in software. Moreover, applications in domains such as Extended Reality (XR) offer opportunities for various forms of parallel execution, including loop level, task level and pipeline parallelism. To assist the design process and expose every possible level of parallelism, we present `Trireme`, a fully automated tool-chain that explores multiple levels of parallelism and produces domain specific accelerator designs and configurations that maximize performance, given an area budget. FPGA SoCs were used as target platforms and Catapult HLS [7] was used to synthesize RTL using a commercial 12nm FinFET technology. Experiments on demanding benchmarks from the XR domain revealed a speedup of up to 20×, as well as a speedup of up to 37× for smaller applications, compared to software-only implementations.

CCS Concepts: • **Computer Aided Design Tools for Embedded Systems**; • **Compilers, Code Synthesis, Parallelization Techniques for Embedded Applications**;

Authors' addresses: Georgios Zacharopoulos, georgios@seas.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221; Adel Ejjeh, aejjeh@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; Ying Jing, yingj4@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; En-Yu Yang, enyu_yang@g.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221; Tianyu Jia, tjia@g.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221; Iulian Brumar, ibrumar@g.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221; Jeremy Intan, jintan2@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; Muhammad Huzaifa, huzaifa2@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; Sarita Adve, sadve@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; Vikram Adve, vadve@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Champaign, IL, USA; Gu-Yeon Wei, guyeon@seas.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221; David Brooks, dbrooks@eecs.harvard.edu, Harvard University, P.O. Box 1212, Cambridge, MA, USA, 43017-6221.

1539-9087/2023/1-ART $15.00
https://doi.org/10.1145/3580394

Additional Key Words and Phrases: accelerators, ASICs, compiler techniques and optimizations, design tools, heterogeneous systems parallelism

1 INTRODUCTION

The breakdown of Dennard scaling [9], and the seemingly inescapable end of Moore's law [30], present new challenges for computer architects striving to achieve increased performance in modern computing systems. Heterogeneous Computing has emerged to address these issues, but the complexity of heterogeneous systems, consisting of software (SW) processors and hardware (HW) accelerators, has also increased dramatically. Hardware designers assigned with accelerating a certain application domain are required to have a deep knowledge and understanding of both the software applications and the underlying platform characteristics. Additionally, a great deal of manual effort is required to identify and extract the information that is necessary to explore various possible optimizations for every design.

These optimizations include exploiting application level parallelism, in the form of Instruction Level (ILP), Loop Level (LLP), Task Level (TLP) and Pipeline Parallelism (PP). The use of such parallelism has been limited in tools for designing hardware accelerators, in two ways. First, in the few tools [19, 27] that accommodate application level parallelism, it is limited to TLP and LLP. Second, these approaches do not usually perform Design Space Exploration (DSE) in early design stages. We call *early DSE* the software analysis and estimation of performance that take place before implementing a particular hardware design, for instance using a High Level Synthesis (HLS) tool, in order to explore a broad range of possible designs and combinations of optimisations, including different types of parallelism.

A hardware DSE flow for a System on Chip (SoC) with hardware accelerators, that automatically extracts and uses parallelism information, requires three main components: a) A program representation that captures and exposes various levels of parallelism in an application, and also potential data movement requiring communication or memory system demands. b) An analysis tool that explores various HW/SW partitioning options, while taking into account not only the execution time and area, but also SoC interconnect bandwidth and communication latency. c) An integration of (a) and (b), such that (a) can provide the information that (b) requires, and (b) can use this information to build efficient performance and cost models to apply in the DSE process.

Spatial [13] is a tool that performs early DSE focusing on parallelism, but it has a number of limitations. First, Spatial aims to support hardware designers by providing a hardware-centric design language, and does not support applications written in high level languages (e.g., C,C++). Second, it is restricted to modeling performance on FPGAs and CGRAs, and cannot be used to effectively perform DSE for SoCs. In particular, communication latency and memory bandwidth are not taken into account during DSE. Finally, the parts of the computation to be accelerated need to be specified by the user and no automatic exploration of acceleration candidates takes place, which is the primary goal of our work.

To address these issues we present `Trireme`,[1] an automated tool-chain that integrates the AccelSeeker [36] and Heterogeneous Parallel Virtual Machine (HPVM) [14] tools. AccelSeeker offers automatic identification and selection of HW accelerators based on models of performance, and HPVM is a parallel program representation for heterogeneous systems that exposes all the major forms of parallelism (loop level, task level and pipeline parallelism) relevant to accelerator design. We extend `Trireme` with novel models of parallel performance evaluation (described below) to enable early DSE that accounts for various forms of parallelism. Moreover, `Trireme` is able to account for SoC interconnect bandwidth and latency, which enables strong synergy with the explicit dataflow information captured in the HPVM parallel representation (a hierarchical dataflow graph). The

[1]Trireme was an ancient Greek/Roman boat having three main rows of oars (similar to the three types of parallelism that we explore) and requiring parallel work to flow. A lightweight, quick boat, taking advantage of parallelism is ideal for explorations, hopefully also for early Design Space Exploration.

integration of the two thus offers the basis for an extensive exploration of multiple levels of parallelism, provides an early estimation of performance, and outputs HW/SW designs that maximize speedup within specific area budgets.

For each type of potential parallelism that `Trireme` extracts (LLP, TLP, PP, and combinations of them), we introduce novel models of performance (in terms of latency) and area demands (in terms of hardware resources). With the aid of these models, we carry out comprehensive early DSE that selects combinations of parallel accelerator designs with increasing area budgets. Additionally, we study a variety of architectural configurations of target SoCs to distinguish the impact of every type of parallelism in accordance with the characteristics and complexity of novel benchmarks from the Extended Reality (XR) domain.[2] `Trireme` achieves speedups of up to 20× for complex XR application components (e.g., audio decoder) and up to 37× for single-kernel applications (e.g., gemm-blocked). FPGA SoCs serve as target platforms for our experiments and High Level Synthesis (Catapult HLS [7]) is used during the validation stage of our experimental section. For the latter, RTL was synthesized, placed and routed by ASIC EDA tools utilizing a commercial 12nm FinFET technology.

Our contributions are as follows:

- We present `Trireme`, a fully-automated tool integrating HPVM [14] and AccelSeeker [36], that offers identification, estimation of performance and selection of hardware accelerators that exploit task level, loop level, and pipeline parallelism (Section 3).
- We introduce novel models for estimating performance and resource demands (area) for task level, loop level, and pipeline parallelism (Section 4).
- We demonstrate `Trireme`'s HW/SW partitioning choices while sweeping area budgets and varying the configuration of memory latency and accelerator invocation overhead, thereby covering a wide range of possible designer scenarios (Sections 5 and 6).
- We evaluate our tool using a broad spectrum of applications, spanning from smaller, single-kernel applications, to complex and demanding state-of-the-art application components from the XR domain (derived from a recently released XR testbed [12]) (Section 6).

The rest of the document is organized as follows. Section 2 offers a description of AccelSeeker and HPVM basic characteristics. Section 3 presents `Trireme` and its main features. In Section 4 the performance and area models are formally defined. Section 5 provides the experimental setup and Section 6 showcases experimental results using Extended Reality applications and targeting FPGA SoCs, as well as generating RTL using Catapult HLS. In Section 7 recent research literature is reviewed and compared to our methodology. Finally in Section 8 we offer our closing thoughts and in Section 9 limitations of our tool-chain and future endeavors are discussed.

2 BACKGROUND

`Trireme` performs extensive early DSE of potential parallelism possibilities for HW acceleration, in comparison to tools such as TAPAS [19] and Peruse [15] that offer limited or late DSE. Furthermore, our tool explores a number of different platform configurations, with respect to memory latency and overhead due to the invocation of the accelerators, that can drastically affect the performance of a HW/SW design.

Achieving such a thorough and early DSE, while investigating the different parallelism opportunities, is a *challenging endeavor* because it requires both: a) automatic extraction of any parallelism-related information from the applications to be accelerated and b) automatic identification and early evaluation of potential accelerators. HPVM and AccelSeeker, both developed within the LLVM [17] infrastructure, support the former and the latter requirements respectively, and hence, serve as the basis of the `Trireme` tool-chain.

AccelSeeker is a tool that performs automatic identification and selection of hardware accelerators, and HPVM is a parallel program representation for heterogeneous systems. `Trireme` uses components of AccelSeeker to

[2]Extended Reality combines Augmented, Virtual and Mixed Reality.

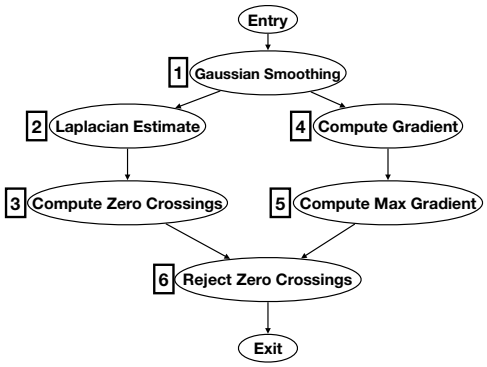

Fig. 1. Task Graph for edge `detection`.

perform an initial estimation of performance and an estimation of area requirements. We use HPVM to analyze the applications and collect required information regarding the three types of parallelism (TLP, LLP, PP) that we can exploit. In the following sections, we provide detailed background of both tools.

2.1 AccelSeeker

AccelSeeker is an LLVM-based tool, comprised of analysis passes, that analyzes applications represented by the LLVM Intermediate Representation (IR). It can be used in the early stages of the HW/SW partitioning process and can reveal the most promising parts of an application for HW acceleration. The tool has three main phases: a) Candidate Identification for HW acceleration, b) Performance and Area Estimation and c) Selection of Candidates for acceleration that maximize speedup under a user-defined area constraint.

Candidate Identification. The granularity of the candidates for acceleration is defined as that of a subgraph of the call-graph of an application that satisfies two properties: It has a root and there are no outgoing edges. Effectively this translates to a candidate that is a function/task, whose calls to other functions included in it (if any) are part of its computation as a potential HW accelerator. As an example, in Figure 1, every one of edge detection Task Graph nodes, which corresponds to a function in the call graph, can be a candidate for acceleration. Candidates that contain system calls, as well as any non-synthesizable constructs, such as dynamic memory allocation, many levels of indirection (pointer chasing), are excluded.

Performance and Area Estimation. AccelSeeker uses models that estimate speedup (merit) and area usage (cost). Through LLVM static analysis and dynamic profiling [38], these models assess software and hardware latency, area, and I/O data transfer requirements for every identified candidate. A default Zynq Programmable System-on-Chip target platform is assumed for the architectural characterization, though it can be configured to adapt to different platforms. The HW accelerators are designed as loosely coupled — their implementations exploiting ILP within the boundaries of a Basic Block (BB). This type of accelerator, exploiting parallelism within the BB granularity, will be referred to as Basic Block Level Parallelism (BBLP) accelerators in Section 6.

Selection of Candidates. Having assigned a specific speedup estimation (merit) and HW resource requirement (cost) to every identified candidate, the selection phase takes place. For a given area budget (which can be varied from small to large) a subset of the initial candidate list is selected that maximizes speedup. The tool's output is the design of a heterogeneous system that distinguishes the part of the computation that stays in software from the part that is accelerated by hardware.

2.2 HPVM

Heterogeneous Parallel Virtual Machine (HPVM) [14] is a parallel program representation for heterogeneous systems, designed to be a virtual ISA, compiler Intermediate Representation (IR) and run-time representation.

Designed as an extension of LLVM IR[17], HPVM exploits all the optimization and code generation potential of LLVM, both for scalar and vector code, while adding support for parallel computation and heterogeneous systems. This is achieved by representing programs using a hierarchical Data Flow Graph (DFG). An HPVM program consists of host code together with one or more DFGs. The DFGs can also be seen as Directed Acyclic Graphs (DAGs). All code suitable for acceleration is contained in the DFG nodes. A DFG node can either contain a part of the computation (called a leaf node) or an entire *nested* data flow graph. This hierarchical representation enables multiple levels of nested parallelism. Every DFG node has a *node function* associated with it, and node functions for leaf nodes contain ordinary scalar and vector LLVM IR. Every DFG edge represents an explicit, logical data transfer between two nodes. Each static node in the graph can have multiple, independent dynamic instances specified as a replication factor (similar to the grid of threads for a CUDA or OpenCL kernel). Put together, this structure allows HPVM to capture loop level data parallelism (via the dynamic instances of a node), fine-grain data parallelism (via LLVM vector instructions within a leaf node), task parallelism between concurrent nodes (via pairs of subgraphs that are not connected by any path), and pipelined streaming parallelism (via streaming dataflow edges), all in a single parallel program representation.

The HPVM representation promotes optimizations such as node fusion, data mapping to local accelerator memory (e.g., GPU scratchpads), and memory tiling. So a number of transformations can be performed on the HPVM IR to optimize execution on specific target devices. The HPVM code generator traverses the DFG, translating each DFG node into code for one or more processing elements in the target system. The HPVM design is able to leverage LLVM's well-tuned back-ends, such as NVIDIA PTX, Intel AVX and X86-64. The HPVM run-time, invoked by the host code, interfaces with the corresponding device run-time to launch a kernel and copy needed data to and from the device.

3 TRIREME

An overview of the entire methodology of the `Trireme` tool-chain is depicted in Figure 2. Boxes C, D and E in the figure represent new components developed for this work, while the other boxes represent existing AccelSeeker and HPVM components. The source code (C,C++) of every application, annotated using the HPVM front-end language (HeteroC++) is used as input. The HeteroC++ annotations are hardware-agnostic annotations, similar in nature to OpenMP, which are used to mark parallel tasks and loops in the program. No manual optimizations or other modifications of the initial source code are performed. As such, any manual, hand-tuned implementations, would only improve the performance that is presented in the experimental section (Section 6). Therefore, in terms of performance, Trireme results can be viewed as the lowest bound of expected outcome. With the aid of AccelSeeker, we analyze its IR to identify candidates for acceleration (Box A). Next we estimate the SW and HW latency, area and the amount of data required for every identified candidate. Their potential performance gain (speedup) is estimated and attached to them as *Merit*, as well as the *Cost* required in terms of HW resources (Box B).

The list of candidates and the DFG of the application, automatically generated by HPVM from the source code, are then passed as input to a tool that extracts all necessary information regarding potential parallel execution, as detailed in Subsection 3.1 (Box C). With the aid of novel models for loop level (LLP), task level (TLP) and pipeline parallelism (PP), described in the following sections, we estimate potential speedup (Merit) and area (Cost), including through combination of parallel approaches wherever applicable, i.e., task level+loop level parallelism (TLP-LLP) and pipeline parallelism+task level parallelism (PP-TLP) (Box D). Figure 3 shows the DFG of the edge detection benchmark and its respective parallelism opportunities.

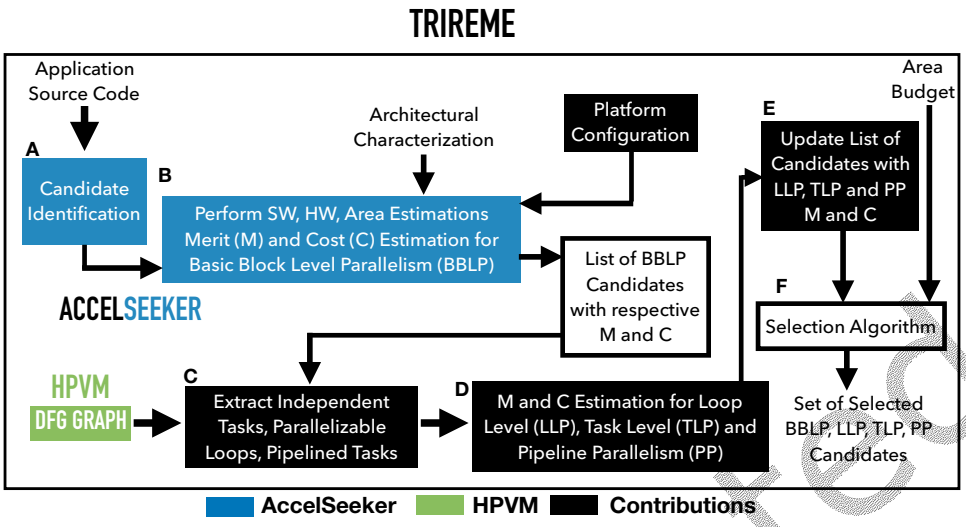

Fig. 2. Overview of the `Trireme` methodology.

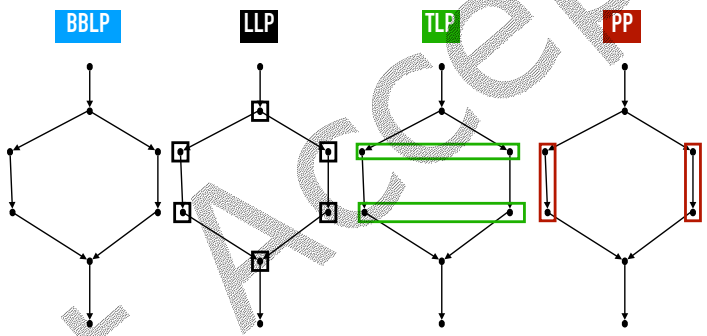

Fig. 3. DFG of edge `detection` depicting Basic Block level parallelism (BBLP) acceleration candidates, loop level (LLP), task level (TLP) and pipeline parallelism (PP) opportunities.

We update the list of accelerators with the newly formed candidates for acceleration that can exploit any (or all) of the three extracted types of parallelism (LLP, TLP, PP), and combinations of them (Box E). Finally, a selection algorithm provides the HW/SW design that maximizes the potential speedup within a given area budget (Box F).

3.1 AccelSeeker-HPVM Integration

We have integrated AccelSeeker and HPVM to exploit any parallelism information that can be provided by the latter and guide the selection process. In particular, we developed a C++ tool within the HPVM infrastructure that receives a list of the most promising candidates (functions) for acceleration, as evaluated by AccelSeeker, along with their corresponding estimated software (T_s), and hardware (T_h) execution times. In addition, the HPVM bitcode file of the application being analyzed is provided as input. The tool builds the DFG of the provided application and creates a mapping between the DFG leaf nodes and the respective input functions from AccelSeeker, such that each input function corresponds to a leaf node. For the scope of this work, we only consider candidate functions that correspond to leaf nodes in the HPVM DFG. Any functions called within a leaf

node are accounted for as part of the leaf node's analysis, and not analyzed separately. The tool then performs a set of HPVM DFG analyses that extract the different types of parallelism, as described below.

First, a *node-reachability* analysis is performed (Algorithm 2) that queries the HPVM DFG to determine whether each of the candidate DFG nodes has a path connecting it to any of the other candidates. We consider nodes that belong to separate DFGs to be sequential. For every node i, we build a list of nodes that are parallel to it, such that any node j that is found to be unreachable to/from i is added to that list. The output of this analysis is the set of nodes that can run in parallel with each candidate.

Second, a critical-path analysis is performed to calculate the Earliest Start Time (EST) and Earliest Finish Time (EFT) of each candidate node. Two full traversals through the DFG are performed: a) calculating the times while the entire run-time is in SW and b) calculating the times while the computation is implemented in HW. In each traversal, the EST, EFT, and Duration (D) of a leaf node (N) are calculated as follows:
$D(N) = T_s$ or T_h depending on the current traversal.
$EST(N) = MAX(EFT(Pred(N)))$ where $Pred(N)$ is the list of N's predecessors in the graph.
$EFT(N) = EST(N) + D(N)$.

Algorithm 1 Algorithm of **`edge detection`**

```
// Gaussian Smoothing
for row ← 1 to m do
    for col ← 0 to n do
        parallel computation
    end for
end for
// Laplacian Estimate
for row ← 1 to m do
    for col ← 0 to n do
        parallel computation
    end for
end for
// Compute Zero Crossings
for row ← 1 to m do
    for col ← 0 to n do
        parallel computation
    end for
end for
// Compute Gradient
for row ← 1 to m do
    for col ← 0 to n do
        parallel computation
    end for
end for
// Compute Max Gradient
for i ← 1 to m * n do
    parallel computation
end for
// Reject Zero Crossings
for row ← 1 to m do
    for col ← 0 to n do
        parallel computation
    end for
end for
```

Algorithm 2 Node reachability analysis for task parallelism

```
for i ∈ Nodes do
    for j ∈ Nodes do
        if !existsForwardPathP(i, j) and !existsForwardPath(j, i) then
            mark j parallel to i
        end if
    end for
end for
```

For cases with separate DFGs, we set *EST* of the first node in a DFG i to be the *EFT* of the last node in the previous DFG $i-1$. The output of this analysis is the software and hardware *EST*s for each candidate function. This information is used in conjunction with the reachability analysis results at a later stage to determine task level parallelism (Section 4.2).

Finally, a third round of analysis detects for every candidate node whether or not it has dynamic replication. Its output is a table containing the nodes that have dynamic replication, along with the number of dimensions they are replicated on. Additionally, if the replication factors of a node are constants, those factors are included as well. This information is used at a later stage to determine loop level parallelism (Section 4.1).

3.2 Tool-chain Features

Accelerator Granularity. We consider the granularity of the candidates to be within the boundaries of a function, as identified by an LLVM-based analysis. Furthermore, under the scope of our work, and in order to integrate AccelSeeker analysis with HPVM, HW accelerators correspond to leaf nodes in the HPVM DFG, as seen in the example of Figure 1. In this instance, every (indexed) node of the DFG of `edge detection` serves as a potential candidate for acceleration.

Software, Hardware Latency and Area Estimation. We perform estimation of software and hardware latency for every identified candidate both by static analysis at the IR level and dynamic, by extracting run-time profiling information for cases where the application is input dependant (e.g. a loop trip count that is not statically resolved and may depend on an input parameter). Furthermore, an estimation of LUTs and mm^2 is carried out in order to account for the hardware resource requirements of every accelerator. The former is estimated with AccelSeeker and its characterization of area in LUTs, by synthesizing a number of micro-benchmarks on a Zynq Programmable System-on-Chip (PSoC). The latter is retrieved by employing the Aladdin [28] area characterization in mm^2. Our method, however, is not constrained to a specific platform and it can easily be adapted for different computing systems (e.g., FPGA boards, ASIC implementations, etc.). In accordance to the limitations of HLS tools, dynamic memory allocations are not supported.

I/O Communication Estimation. The amount of data required by each candidate is also extracted by static analysis and by parsing its dynamic trace, when the latter is available. This data requirement is subsequently used to estimate latency due to communication between an accelerator and memory (e.g., DRAM, last level cache, etc.).

Merit and Cost Estimation. Given the characteristics of the platform for which we are going to implement HW accelerators, we estimate potential speedup (Merit) for every acceleration candidate and the hardware resources required (Cost) to achieve that speedup. To obtain an accurate estimate, we use the AccelSeeker model for *Merit*, which translates to cycles saved, and its model for *Cost*, which accounts for the area budget in terms of LUTs (Section 2.1).

Automatic Extraction of Parallelism. Using the tool developed for AccelSeeker-HPVM integration (3.1) we automatically extract information about the potential for loop level, task level and pipeline parallelism. This serves as input, along with AccelSeeker's list of candidates for (BBLP) acceleration, for the novel performance models of multiple levels of parallelism explored by `Trireme`. These models are presented in detail in Section 4.

Selection Algorithm/HW-SW Partitioning. The updated list of candidates for acceleration is generated, including both the Basic Block Level Parallelism (BBLP) accelerators from AccelSeeker and the candidates that exploit all types of parallelism explored by our tool-chain. The selection algorithm recursively explores the subsets of the updated list of candidates, in a similar manner to the Bron-Kerbosch algorithm [2]. The output returned is the set with the highest speedup (cumulative Merit) that stays within the user defined area budget (Cost).

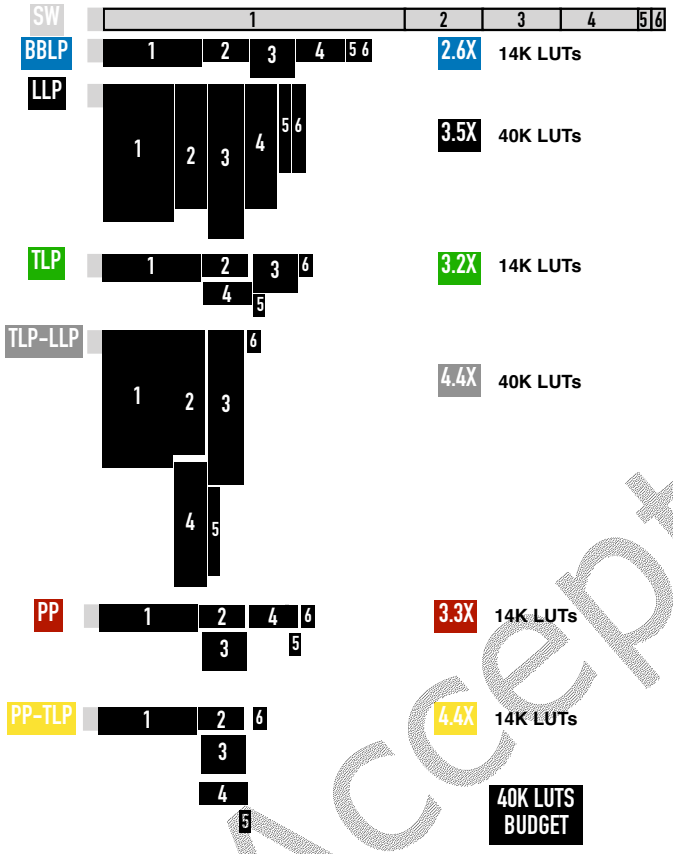

Fig. 4. Designs exploiting Basic Block level (BBLP - AccelSeeker [36]), loop level (LLP), task level (TLP), and pipeline parallelism (PP) in **edge detection** provided a 40×10^3 LUTs area budget. The size of the black rectangles represents area usage.

4 MERIT AND COST MODELS

As mentioned in the previous section, we introduce novel models for estimation of speedup, which we denote as *Merit*, and an estimation of the area required for every HW accelerator implementation, denoted as *Cost*. These models, inspired by the respective ones from RegionSeeker [37] and AccelSeeker [36], are introduced to accommodate the estimation of loop level, task level and pipeline parallelism extracted by HPVM. Having an early estimation of speedup and area budget needs, for every possible design exploiting any, or a combination, of these three types of parallelism can lead to better design choices and significantly less engineering effort.

4.1 Loop Level Parallelism (LLP)

With the aid of the tool described in 3.1, information regarding the DFG nodes loop-level parallelism is retrieved. As shown in the example of Figure 3, the marked DFG nodes of edge detection are identified as nodes that contain a fully-parallelizable loop and, thus, are analyzed further so that multiple versions of the same functions are generated with an increasing LLP factor. For each factor, the loop is parallelized by replicating its body, and

the corresponding speedup and cost estimates are computed. To simplify the estimation we assume an equal workload for every iteration of the loop.

LLP Merit and Cost Estimation. Let $S = \{S_1, S_2, \ldots, S_N\}$ be a set of parallelizable loop candidates, with associated SW latency (SW_i), HW computation latency ($HWcomp_i$), HW communication latency ($HWcom_i$), invocation overhead ($OVHD_i$) and area cost (A_i). Also let LLP factor $j = 1 \ldots, K \mid K = max(Loop\ Trip\ Count)$ be the factor by which we parallelize each loop. To simplify the analysis, we assume that the loop is perfectly load-balanced, and communication latency is constant, independent of j.

Under these simplifying assumptions, for every loop candidate $\{S_{ij} \mid i = 1, \ldots, N,\ \ j = 1, \ldots, K\}$, we compute the merit $M(S_{ij}) = SW_i - \frac{HWcomp_i}{j} - HWcom_i - OVHD_i$, and the loop candidate area cost $C(S_{ij}) = A_i \times j$, respectively.

As anticipated, by increasing the replication factor, better performance is achieved with the higher cost of area required. LLP, where applicable, can yield tremendous speedup benefits but at a high area budget cost, as seen in Figure 4 (LLP vs software-only implementations) and discussed in greater extent in Section 6.

4.2 Task Level Parallelism (TLP)

To compute the potential speedup of a number of tasks that can be run in parallel we need first to extract all possible sets of independent candidates, i.e., all candidates that have no data flow dependencies. As depicted in the example in Figures 1 and 3, edge detection candidates indexed {2,4} and {3,5} are independent sets and can therefore be invoked in parallel. The same applies for candidates {2,5} and {3,4}. For this analysis, we use the SW and HW estimated times, as well as the Earliest Start Time (EST) provided from the tool described in 3.1 as described below.

Merit and Cost Definition/Estimation of TLP. Let $S = \{S_1, S_2, \ldots, S_N\}$ be a set of independent candidates (tasks), with associated SW latency (SW_i), HW computation latency ($HWcomp_i$), HW communication latency ($HWcom_i$), invocation overhead ($OVHD_i$) and area cost (A_i). In the best case, all candidates in the set will be able to start execution at the same time, and the total HW latency of this set of candidates S would be $MAX(S_{H_W}) = max(HWcomp_i + HWcom_i + OVHD_i) \mid i = 1, \ldots, N$.

In practice, some candidates may have varying starting times (e.g., {2,5}) because of dependencies on previous tasks that may not exist in the candidate set (e.g., node 5 must wait for node 4 to complete). To account for these delays, we add an extra overhead based on the difference of ESTs of the nodes in the candidate set: $EST\_OVHD = max(EST_i) - min(EST_j) | i, j = 1, \ldots, N$, where N is equal to the number of tasks. The i index can vary independently hence all possible pairs estimates are taken into account. Intuitively, the overhead allows us to mark the candidate set {2,4} as a better candidate for acceleration compared to {2,5}.

We denote the merit of set S, by $M(S) = \sum_{i \in [1,N]} SW_i - MAX(S_{H_W}) - EST\_OVHD$ and we denote the cumulative cost of set S in area by $C(S) = \sum_{i \in [1,N]} A_i$.

Task level parallelism, in applications that have independent tasks, can offer significant speedup compared to, for instance, sequential accelerators exploiting only Basic Block level parallelism (BBLP) that require the same HW resources. Figure 4 provides a comparison between TLP and BBLP when accelerating the edge detection application. The example of edge detection (Figure 3) shows instances of two tasks running in parallel at any given time, however the estimations above can be applied to any N number of tasks that may be executed in parallel. The Merit and Cost estimation formulas remain the same and no further modification is required.

4.3 Pipeline Parallelism (PP)

An illustration of the pipeline parallelism is shown in Figure 5. We assume that the pipeline has K stages, S_1, S_2, ... S_K, and the time needed on stage i is T_i. We also assume that the stage that requires the longest time is S_j (i.e.,

$\forall i \in \{1, 2, ..., K\}, T_j \geq T_i$). Now we will prove that the total execution time for pipeline parallelism, where N is the number of iterations, is $T_{total} = \sum_{i=1}^{K} T_i + T_j \times (N-1)$.

The first term $\sum_{i=1}^{K} T_i$ is the time spent on the first iteration. The second term $\max_i T_i \times (N-1)$ is the timing overhead caused by the following $(N-1)$ iterations. We prove the second term in two steps: (1) For iteration n, it can finish at T_j later than the finishing time of iteration $(n-1)$; (2) For iteration n, it cannot finish at time t later than the finishing time of iteration $(n-1)$ if $t < T_j$.

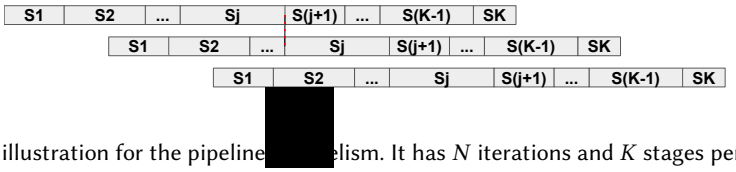

Fig. 5. An illustration for the pipeline [redacted]elism. It has N iterations and K stages per iteration.

Step 1. Provided that the inter-stage dependencies are not considered (e.g., S_2 cannot start before S_1 finishes, etc.), the earliest starting time for each stage is the ending time of the same stage in the previous iteration. This is shown in Figure 6.

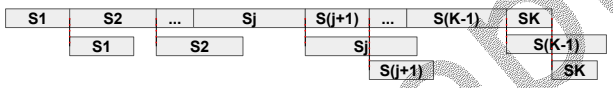

Fig. 6. The earliest starting point of each stage in the second iteration.

If we start the second iteration after T_j, since $T_j \geq T_i, \forall i \in \{1, 2, ..., K\}$, the starting time of every stage in the second iteration will be no later than the ending time of the same stage in the previous iteration. In other words, there should not be any idle time. Thus, the ending time for the second iteration is T_j later than the first iteration.

For the third iteration, the ending time is T_j later than the second iteration. Thus, based on mathematical induction, we can prove that at iteration n, execution is completed T_j later than the previous iteration.

Step 2. We prove that iteration n cannot finish at time t later than the previous iteration, if $t < T_j$. Provided that the ending time of the second iteration is t later than the first iteration, the ending time of stage S_K in the second iteration is t later than the one in the first iteration. Therefore, the starting time of stage S_K in the second iteration is t later than the one in the first iteration. Due to the inter-stage correlation, the ending time of stage S_{K-1} in the second iteration should be no more than time t later than the one in the first iteration.

Hence, if we trace back to S_j, we can say that the ending time of S_j in the second iteration should be no more than time t later than the one in the first iteration. However, since $t < T_j$, the starting time of S_j in the second iteration will be $T_j - t$ earlier than the ending time of S_j in the first iteration. In other words, there will be an overlap between two consecutive iterations on stage S_j (Figure 7).

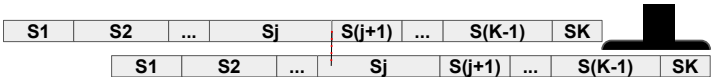

Fig. 7. Overlap on stage S_j.

Merit and Cost Definition/Estimation of PP. Based on the previous illustration, let $S = \{S_1, S_2, \ldots, S_K\}$ be a set of pipelined candidates (tasks) and N be the number of iterations, with associated SW latency (SW_i), HW computation latency ($HWcomp_i$), HW communication latency ($HWcom_i$), invocation overhead ($OVHD_i$), HW

latency $HW_i = HWcomp_i + HWcom_i + OVHD_i$ and area cost (A_i). We compute the HW latency, using the previous proof, as $HW_{TOTAL} = \sum_{i=1}^{K} HW_i + \max_i HW_i \times (N-1)$. This formula can be applied to both a balanced pipeline and an unbalanced pipeline.

We denote the merit of set S, by $M(S) = \sum_{i\in[1,K]} SW_i - HW_{TOTAL}$ and we denote the cumulative cost of set S in area by $C(S) = \sum_{i\in[1,K]} A_i$.

5 EXPERIMENTAL SETUP

For our experiments, we assume a heterogeneous system constituted by a single SW processor and multiple loosely coupled HW accelerators. The processor invokes the accelerators via a memory-mapped interface. DMA is used to transfer data from main memory to accelerator scratchpads and vice versa in order to store the accelerators output to main memory and be available to the SW processor. As AccelSeeker, used as a baseline, targets by default an FPGA SoC (Zynq UltraScale SoC), we also use FPGA SoCs in our experiments.

Benchmarks. We evaluated the `Trireme` tool-chain in a variety of applications, spanning from smaller, single-kernel ones, to larger and more demanding ones. The type of potential parallelism extracted from every benchmark, as expected, also varies. The kernels from Parboil [31] and MachSuite [25] offer opportunities for loop level parallelism only. Medium and large size applications from the XR domain, such as 3D spatial `audio encoder` from a recently released XR testbed [12] and Camera Vision Pipeline `cava`[34], where both loop level parallelism and pipelining would be feasible, and visual inertial odometry (VIO), often referred to as `SLAM`, where 70% of its run-time is evaluated and loop level and task level parallelism opportunities are present. Larger and more complex applications, where all types of parallelism can be retrieved (as well as combinations of them), are also rigorously evaluated. These include 3D spatial `audio decoder` (XR domain) from the XR testbed [12] and `edge detection`, a six stage image processing pipeline used in [14].

Parallelism Strategies. We evaluate and compare the following parallelism strategies for HW acceleration:
a) Basic Block Level Parallelism (BBLP). Function (Task) accelerators that exploit Instruction Level Parallelism within a Basic Block. It corresponds to the accelerators selected by AccelSeeker [36].
b) Loop Level Parallelism (LLP). Replication and parallel execution of fully parallelizable loops, represented in HPVM as leaf nodes with multiple dynamic instances.
c) Task Level Parallelism (TLP). Sets of two or more tasks (HPVM leaf nodes) that have no data flow dependencies between them (i.e., no path in the HPVM dataflow graph connecting any pair of nodes in the set) and can therefore all run in parallel with each other.
d) Pipeline Parallelism (PP). Sequences of HPVM nodes (tasks) connected by streaming dataflow edges, and therefore can be pipelined.
e) Task and Loop Level Parallelism (TLP-LLP). Sets of tasks that can be either executed as parallelizable loops or run as parallel tasks or both. The final design may have any of these forms of parallelism applied.
f) Pipeline and Task Level Parallelism (PP-TLP). Sets of pipelined tasks that can also be run in parallel. This setting supports the execution of pipelines (two or more) in parallel. (e.g. In the case that there are two independent pipelines, then they can be computed in parallel.

Validation. For the validation of our models we evaluated HW acceleration with Aladdin [28] HW accelerator simulator. The run-time of the non-accelerated part was measured using gem5 [1]. The processor modelled is an ARMv8-A processor of issue width of 1, having an atomic model, in-order execution and clocked at 100 MHz. It is interfaced with a separate data cache (64 KB) and instruction cache (16 KB), where the access latency is one clock cycle. This setup is realistic for resource-constrained embedded systems, however it is conservative for high-performance systems, as an L1 cache that never incurs in any misses places SW execution at a slight advantage. Additionally, we used Catapult HLS[7] to synthesize the HW accelerators for further validation. The

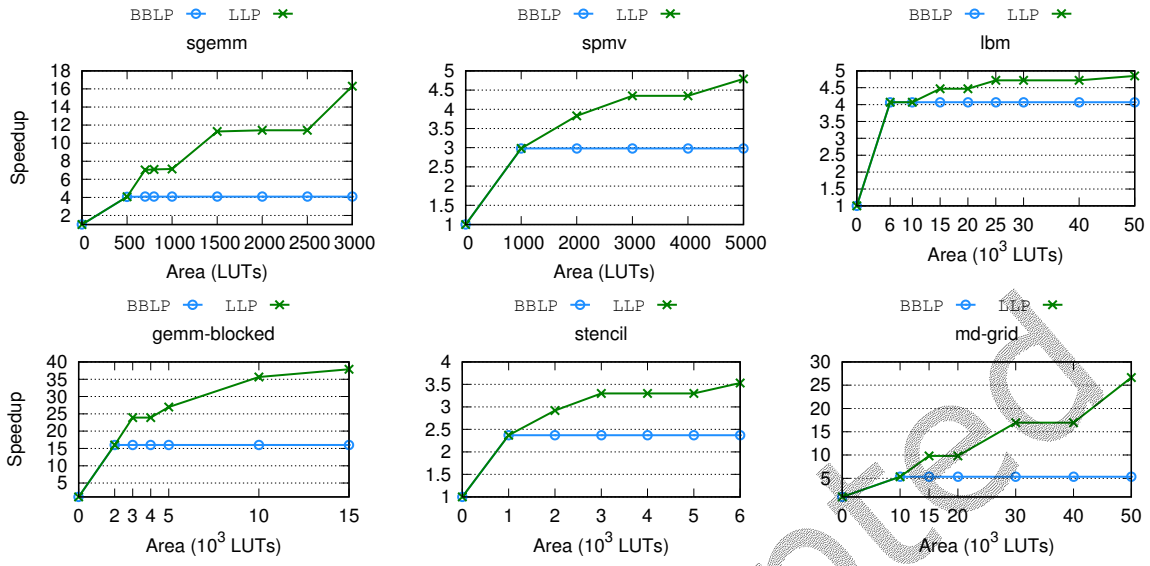

Fig. 8. Speedup obtained for applications from Parboil [31] and MachSuite [25] benchmark suites, varying the area budget constraint. We evaluate AccelSeeker [36] (BBLP) and LLP, while the baseline is a SW-only implementation.

latter, i.e., synthesis performed by Catapult HLS in order to perform validation was carried out manually, as `Trireme` does not support automatic HLS.

6 EXPERIMENTAL RESULTS

In the following subsections, we showcase the speedup achieved from the hierarchical multi-level parallelism strategies explored by our tool-chain. We group the results by different types of parallelism exploited by `Trireme`. First, the performance benefits in single-kernel applications that solely exploit LLP are presented. Then, we investigate XR applications with pipelines (`audio encoder, cava`) and independent tasks (`SLAM`), where both LLP/PP or LLP/TLP can be applied. Finally, we study larger ones (`audio decoder, edge detection`), where LLP/TLP/PP and combinations of them can be used, such as TLP-LLP and PP-TLP, as described in the previous section. We evaluate the above against SW-only implementations, and against state-of-the-art AccelSeeker. As such, we target FPGA SoCs in all our experiments.

We validate the designs selected by our tool, given increasing area constraints, first using Aladdin [28] (for the latency of HW accelerators) and gem5 [1] (for the software latency), and second using Catapult HLS for real hardware measurements. Finally, we study the effects of varying the bandwidth of data transfers between host and accelerator, and the overhead of accelerator invocation, on the `audio decoder` and `edge detection` benchmarks.

6.1 Loop Level Parallelism

`Trireme`, extracting information exposed by HPVM, identifies the application kernels that contain a fully parallelizable loop or loop nest. Subsequently, the Merit/Cost estimation models for loop level parallelism, as described in Section 4, are used to estimate the speedup and hardware resource utilization for varying LLP factor. Figure 8 shows the speedup obtained on six benchmarks from Parboil (`sgemm, lbm, spmv`) and MachSuite (`gemm-blocked, md-grid, stencil`), compared to a SW-only baseline.

All applications benefit significantly from replicating their loop-bodies and running them in parallel, and the parallelism enables the designs to take advantage of larger area resources to achieve greater speedups than is possible without loop level parallelism. For an area budget of 3×10^3 LUTs, `sgemm` and `gemm-blocked` reach a 16× and 25× speedup respectively, compared to the baseline, and a 3× and ~2× speedup compared to BBLP, which corresponds to state-of-the-art AccelSeeker selections.

Kernels such as `spmv` and `stencil` realize a 4.7× and 3.4× speedup compared to a SW-only implementation respectively, for a budget of 5×10^3 LUTs, whereas `lbm` having a smaller loop body, i.e., fewer instructions and less computation time within the loop body compared to the previous ones, has little benefit from extra area resources and LLP. Finally, `md-grid` requires more area compared to the previous kernels and, having a large potential for loop level parallelism, reaches a 27× speedup compared to the SW baseline and 5.4× compared to state-of-the-art BBLP accelerators. Overall, `Trireme` is able in many cases to achieve substantial performance improvements for given hardware resources by exploiting loop level parallelism alone.

6.2 Loop vs. Pipeline and Loop vs. Task Parallelism

Richer applications, such as components from the XR testbed [12], contain a variety of opportunities to exploit parallelism. For `audio encoder` and `cava`, in addition to parallelizable loops, the DFG nodes can also be pipelined. For `SLAM`, apart from LLP, independent tasks are present as well. `Trireme` automatically generates designs exploiting this information.

Figure 9 shows the speedup obtained from applying LLP and PP on `audio encoder` and `cava`, for a number of increasing area budgets. For a budget of 5×10^3 LUTs `audio encoder` achieves an 8× (for LLP) and 9× (for TLP) speedup compared to SW-only baseline, as the entire pipeline fits the budget. Additionally, a slight improvement over BBLP (AccelSeeker selection) is achieved. Nonetheless, more area is required to parallelize the loops within the selected accelerators, which is evident by the increasing trend line for LLP.

For the same area budget in `cava`, the pipeline does not fit. Thus, the speedup gain for PP is the same as for BBLP (10× over the baseline). LLP on the other hand benefits from loop parallelization and achieves a 20× speedup.

For larger budgets, we can observe significant benefits in speedup for LLP, both in `audio encoder` and `cava`. With 15×10^3 LUTs `audio encoder` achieves a ~17× speedup compared to baseline, and with 10×10^3 LUTs `cava` attains a 33× speedup. These are respectively about 2× and 3× the speedup achieved with BBLP alone.

Figure 9 shows that `SLAM` benefits from LLP, reaching up to 7× speedup, as the area budget allows for more loop level parallelism. On the other hand, since only two tasks — with small latency relative to the total run-time — can be parallelized, TLP offers no performance gain.

For `audio encoder` and `cava`, PP produces little improvement in performance. This is due to the unbalanced pipelines in these workloads. One of the functions (DFG nodes) in each application dominates the computation time, therefore applying the PP strategy yields little benefit. However, as demonstrated in the following round of experiments, this is not the case for the next two applications evaluated: `audio decoder` and `edge detection`.

6.3 Loop/Task/Pipeline Parallelism

In the previous subsection we encountered applications that could only exploit LLP and PP, whereas `audio decoder`, a state-of-the-art XR application component, and `edge detection`, a six-stage image processing pipeline, can offer LLP, TLP, PP, as well as combinations of them. Such applications are ideal candidates to employ `Trireme` and unlock their full parallelism potential. Figure 10 presents the speedup achieved by multiple levels of parallelism explored by our tool-chain, for increasing area budgets.

On `audio decoder`, Figure 10 (left) and Table 1, for an area budget of 12×10^3 LUTs, LLP and PP reach a 13.2× and 13.7× speedup respectively, compared to a SW-only baseline. This budget is enough to fit one of the

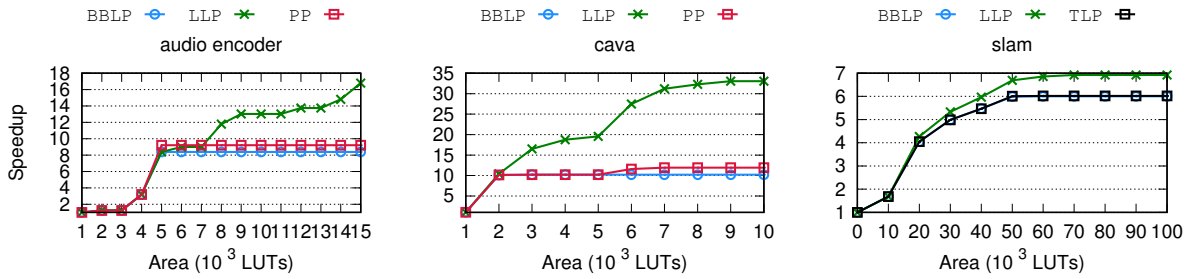

Fig. 9. Speedup obtained over the entire run-time of audio encoder [12], cava [34] and OpenVINS algorithm for SLAM [12], varying the area constraint. We evaluate AccelSeeker [36] (BBLP), LLP, PP and TLP, while the baseline is a SW-only implementation.

two `audio decoder` pipelines, and since the workloads are fairly balanced, we see the benefit obtained from this strategy. TLP and TLP-LLP achieve the same 15.1× speedup, as not enough area is available to benefit from parallelizing the loops, while the selected independent tasks are accelerated in parallel.

Increasing the budget to 14×10^3 LUTs, almost equivalent to Xilinx Artix Z-7007S PSoC [33], we can see that LLP and TLP-LLP are making use of the larger area and increase their respective speedups to 14.21× and 15.74×. Conversely, BBLP, TLP and PP extract no benefit, using only 85% of the available resources, as their potential candidate choices require more area to be selected (Table 1 - row 2). A budget of 15×10^3 LUTs, however, accommodates all available tasks to be parallelized (TLP-16.7×), as well as the pipelines (PP-16.5×), including the possibility to parallelize the independent pipelines (PP-TLP-18.31×), yielding the maximum possible speedup for these strategies.

The latter point can also be seen in the last row of Table 1. A larger area budget, almost equivalent to Xilinx Artix Z-7012S PSoC [33], allows LLP and TLP-LLP to benefit from increased parallelization of the loop bodies of their accelerators. TLP, PP and PP-TLP show no benefit from the doubling of the hardware resources as they have already reached their better-performing designs. An interesting aspect is that PP-TLP, the strategy that achieves the best speedup, along with TLP and PP require fewer hardware resources to reach their maximum speedup compared to LLP and TLP-LLP, the latter achieving an almost equivalent speedup to PP-TLP but for much larger area. Also BBLP is consistently outperformed by all parallelism strategies explored.

Similar trends can be seen in `edge detection` while investigating its potential for parallelism (Figure 10 – right). For a 14×10^3 LUTs area budget TLP (3.2×), PP (3.4×) and PP-TLP (4.4×) can accommodate all their respective HW/SW designs and reach their top speedups compared to the SW-only baseline. For the same budget, LLP and TLP-LLP can achieve 2.5× and 3.2× respectively, requiring more area to reach better performance. An area budget of 40×10^3 LUTs, equivalent to Artix Z-7014S PSoC, would allow for more parallelization of the loop bodies for LLP an TLP-LLP, the latter reaching an equivalent of the PP-TLP maximum speedup (4.4×).

For even larger area budgets, such as 100×10^3 LUTs, we notice that LLP reaches a 4× speedup and TLP-LLP surpasses the highest-performing PP-TLP design by achieving 4.7× speedup compared to the baseline. This is because, unlike `audio decoder`, all of the accelerated functions in `edge detection` have parallelizable loops, which allows for increasing speedup as the area increases.

6.4 Aladdin/gem5 and Catapult HLS

To validate the selection of the HW/SW designs for every parallelism strategy explored and evaluated by our tool-chain, we use Aladdin [28], a HW accelerator simulator, and the gem5 [1] simulator. Aladdin was chosen as a faster, yet accurate, alternative to commercial HLS tools that offer latency and area results. For `audio decoder`,

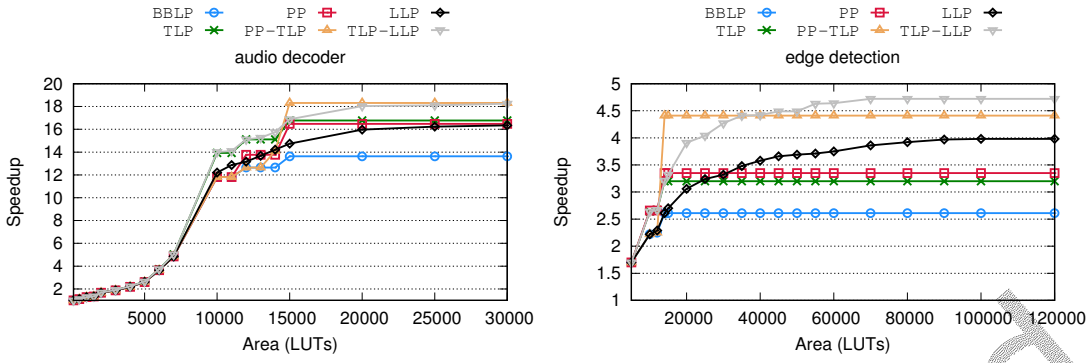

Fig. 10. Speedup over the entire runtime for different versions of audio decoder (left) and edge detection (right), varying the area constraint. We evaluate AccelSeeker [36] (BBLP), TLP [13, 21], PP, a combination between PP and TLP (PP-TLP), LLP [8, 15] and a combination between TLP and LLP (TLP-LLP), while the baseline is a SW-only implementation.

Benchmark	**Parallelism Version**	**Area Budget** (LUTs)	**Area Used** (LUTs)	**Speedup vs. SW**
audio decoder	BBLP	12 000	11916 (99%)	12.65
	LLP		11655 (97%)	13.2
	TLP		11916 (99%)	15.1
	TLP-LLP		11916 (99%)	15.1
	PP		11916 (99%)	13.7
	PP-TLP		11916 (99%)	12.65
	BBLP	14 000	11916 (85%)	12.65
	LLP	**Artix Z-7007S**	13889 (99%)	14.21
	TLP	[33]	11916 (85%)	15.1
	TLP-LLP		13889 (99%)	15.74
	PP		11916 (85%)	13.7
	PP-TLP		13861 (99%)	14.09
	BBLP	15 000	14166 (94%)	13.62
	LLP		14722 (98%)	14.7
	TLP		14166 (94%)	16.7
	TLP-LLP		14471 (96%)	16.9
	PP		14166 (94%)	16.5
	PP-TLP		14166 (94%)	**18.31**
	BBLP	30 000	14166 (47%)	13.62
	LLP	**Artix Z-7012S**	29773 (99%)	16.3
	TLP	[33]	14166 (47%)	16.7
	TLP-LLP		29773 (99%)	18.24
	PP		14166 (47%)	16.5
	PP-TLP		14166 (47%)	**18.31**

Table 1. Area Budget and Area Used for audio decoder.

we gather the HW latency and area of the available candidates for acceleration with Aladdin, and their respective SW latency with gem5, as well as the run-time of the application as detailed in Section 5.

Figure 11 shows the speedup over increasing area budgets. For every area budget, the outputs of applying the parallelism strategies explored in this work match the ones generated by the Aladdin/gem5 simulations. This

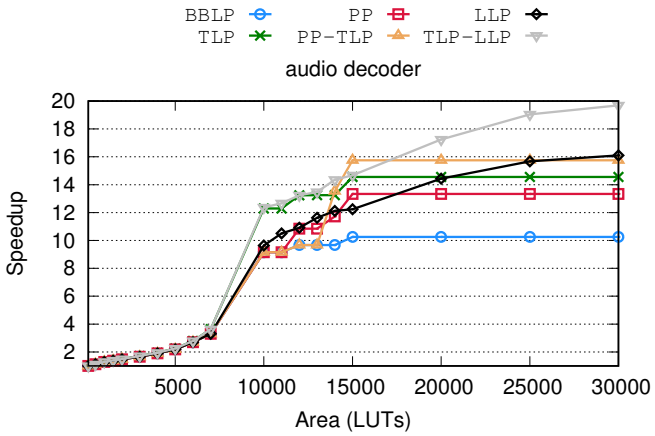

Fig. 11. Speedup obtained for audio decoder varying the area constraint using Aladdin [28] for the HW acceleration parts and gem5 [1] for the SW-only implementation.

reinforces our expectation that our tool-chain selects the most promising designs with respect to performance and area usage.

As expected, speedup absolute values for `audio decoder` (Figures 10 and 11) differ. This is due to two factors: A) Our performance and area models are not based on cycle-accurate estimations, but aim to enable the selection of high-performance HW/SW choices automatically, and faster than performing demanding simulations or RTL synthesis. B) The characterization of latency for Aladdin is performed targeting OpenPDK 45nm technology, which is different to the characterization of our tool targeting a Zynq Programmable SoC.

Benchmark	**Parallelism Version**	**Area Used** (uM^2)	**Speedup vs. state-of-the-art AccelSeeker (BBLP)**
audio encoder	BBLP	3854	1
	LLP	5415	2
	LLP	8578	4
	LLP	15072	8
	LLP	27491	16
audio decoder	BBLP	92 738	1
	LLP	85 602	1.5
	TLP-LLP	85 602	2
	BBLP	125 865	1
	LLP	171 385	2
	TLP	**125 865**	**3**
	TLP-LLP	**125 865**	**3**
	TLP-LLP	251 641	6

Table 2. `Trireme` vs. AccelSeeker [36] by Catapult HLS [7].

To further evaluate our tool flow, we designed accelerator prototypes using SystemC, guided by `Trireme`. To gather HW latency and area requirements, the accelerators were synthesized using Catapult HLS [7]. The RTL was then synthesized, placed and routed by ASIC EDA tools using a commercial 12nm FinFET technology. The accelerators were clocked at 500MHz frequency and cycle-accurate Catapult simulations were used to measure the HW latency.

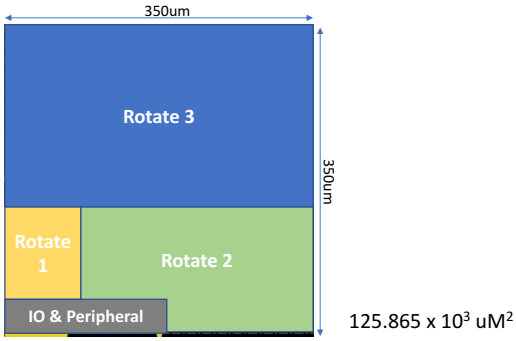

Fig. 12. HLS design of audio decoder guided by `Trireme`.

Table 2 shows the HW latency comparison of Trireme (LLP, TLP, TLP-LLP) to AccelSeeker (BBLP). For `audio encoder`, LLP designs guided by `Trireme` achieve impressive performance gains at the expense of more HW resources. In `audio decoder`, LLP designs achieve smaller speedup and require the same or more resources compared to TLP-LLP. The latter can be up to six times faster compared to the respective AccelSeeker design for a large area budget (~$252 \times 10^3 uM^2$). A medium area budget of ~$126 \times 10^3 uM^2$ can yield significant speedup for TLP and TLP-LLP where accelerators Rotate 1-3 are operating in parallel. Figure 12 shows the physical layout of this design for `audio decoder`.

6.5 Configurations of the Target Platform

To gain better intuition on how different platform configurations affect potential speedup in HW accelerated systems, we apply a round of experiments varying the bandwidth of the data transfers to and from the HW accelerators (affecting memory latency), and the overhead of invoking them. Note that for Subsections 6.1, 6.2 and 6.3 we have been assuming a configuration of 1 GBps bandwidth and 1μs overhead per accelerator invocation.

Figure 13 (left) shows the `audio decoder` speedup due to varying the bandwidth over 100 MBps, 1 GBps and 10 GBps, and the area budgets over 12, 15 and $30{\times}10^3$ LUTs. We observe that low bandwidth (100 MBps), even when the area budget is increased, offers little speedup from BBLP, LLP, TLP, TLP-LLP and PP. This reveals the limitation of platforms where communication to memory can severely affect the speedup of a HW/SW design.

Overall, as expected, all parallelism strategies reach greater speedup when both bandwidth and area are increased. Nonetheless, LLP and TLP-LLP are favored, compared to the rest of the strategies, when bandwidth is increased for a given area budget. This result is even more evident for `edge detection` compared to `audio decoder`, as seen in Figure 13 (right), as it has more parallelizable loops than the latter. For the largest area budget of 100×10^3 LUTs we notice that the second and fourth bars increase vastly reaching 4.2× and 4.9× speedup respectively, as bandwidth increases, surpassing the previous better performing strategy (PP-TLP) for a smaller budget of 15×10^3. We can also notice this for `audio decoder` for the largest area budget of 30×10^3 LUTs where TLP-LLP reaches the maximum speedup (20×), compared to the rest of the parallelism approaches.

Finally, we evaluated the effect of both latency due to communication between the accelerators and memory, as well as the invocation overhead of the accelerators, on edge detection. In Figure 14, we observe that even for a low bandwidth, such as 100 MBps, a high speedup can be obtained if the invocation overhead remains low. For a low overhead of 300 ns per invocation, the speedup reached by parallelizing the pipelines (PP-TLP) almost doubles (5×) compared to the same bandwidth and a higher invocation overhead of 2×10^3 ns per invocation.

A similar trend is observed for all evaluated bandwidths, where a low invocation overhead with a 1 GBps bandwidth can yield better speedups compared to a higher invocation overhead with a 10 GBps bandwidth. For instance, all the parallelization strategies with a 1 GBps bandwidth and a 500 ns invocation overhead achieve

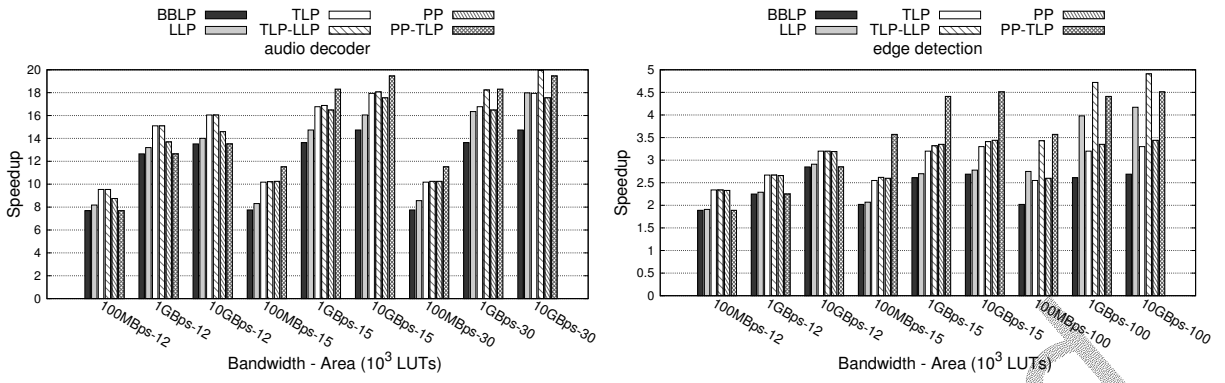

Fig. 13. Speedup of audio decoder (left) and edge detection (right), for increasing bandwidth and area. Baseline is SW-only. We evaluate all parallelism strategies explored by Trireme, while the baseline is a SW-only implementation.

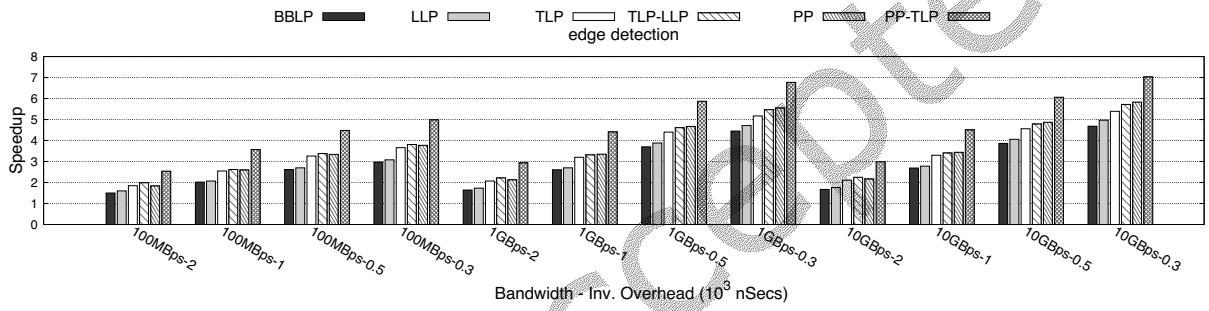

Fig. 14. Speedup obtained of edge detection for different bandwidth and invocation overhead values. All parallelism strategies are evaluated for an area budget of 15×10^3 LUTs.

better performance compared to all the respective configurations that have a 10 GBps bandwidth with a higher invocation overhead.

7 RELATED WORK

We classify related research literature across five dimensions, as shown in Table 3. The types of parallelism supported by each piece of research vary from ILP within Basic Block boundaries [36, 37], to loop level [8, 13, 15, 21], task level parallelism [19, 22] and Tensor level [18]. Early DSE, one of the most important aspects of Trireme, is in many instances not supported by tools developed to expose and exploit parallelism in HW acceleration [15, 18, 19, 23, 27].

FCUDA [23] is a source-to-source tool that translates CUDA code to FPGA accelerators, however offers no DSE or estimation of HW acceleration performance. On the other hand, Spatial [13] is an early DSE infrastructure that uses Hypermapper 2.0 [21] in order to apply early DSE, however the parts to be accelerated need to be user-defined and high level languages are not supported as input. Aetherling [8] applies early DSE as well and can be configured onto FPGAs, but it is restricted to loop level parallelism only and it does not support high level languages (C/C++). Spatial [13] also employs early DSE using Hypermapper 2.0 [21], however the parts to be accelerated need to be user-defined and high level languages are not supported as input. Early DSE that serves the purpose of merging accelerators has been explored in [3].

Methodologies that combine static analysis and machine learning have been used in Peruse [15], in [10, 11] and in [35] to predict the potential speedup of loop accelerators. TAPAS [19] is a tool-chain focusing on loop and task level parallelism by leveraging the TAPIR [27] Parallel IR representation of the code. Although TAPIR is able to generate parallelism at arbitrary granularities, HPVM is able to expose nested parallelism which is leveraged by `Trireme`.

HeteroCL [16], developed within a Python-based domain specific language, performs early DSE and offers estimations on performance and area targeting FPGAs. It uses parallel processing pipelines and shifts towards tensor-related computations, used in Linear Algebra, Computer Vision and Machine Learning. Since HeteroCL is domain specific, it uses the domain expertise to trade accuracy for performance aggressively by reducing the bitwidth for key functional units.

High Level Synthesis (HLS) tools have improved substantially in recent years [20]. Commercial tools like Xilinx Vivado HLS [32] and Cadence Stratus HLS [4], and academic tools like Bambu [24] and Legup [6], carry out the design of computation-heavy accelerators from application source code. They achieve performance on a par with that of hand-crafted implementations written in low level hardware description languages like VHDL and Verilog. But these HLS tools provide no DSE or early estimation of accelerator performance; hence, they are complementary to `Trireme` in an application-driven hardware-design workflow.

Feature	**FCUDA** [23]	**Spatial** [13, 21]	**Peruse** [15]	**TAPAS** [19]	**CIRCT** [18]	**Aethering** [8]	**AccelSeeker** [36]	**`Trireme`**
Levels of Parallelism	Loop Task	Loop Task	Loop	Loop Task	Tensor	Loop	Intra-BB ILP	Intra-BB Loop Task Pipeline
Early DSE	✗	✓	✗	✗	✗	✓	✓	✓
Performance Estimation	✗	✓	✓	✗	N/A	✓	✓	✓
Automated	✓	✓	✓	✓	✓	✓	✓	✓
Configurations of Target SoCs	✗	✗	✗	✗	N/A	✗	✗	✓

Table 3. Taxonomy Table. DSE Methodologies and tools comparison.

Tools that perform HW acceleration simulation and can be used for DSE such as Aladdin [28], gem5-aladdin [29] and gem5-SALAM [26] can achieve high cycle and power accuracy, comparable to that of commercial HLS tools. Furthermore optimizations, such as loop unrolling and loop pipelining, can be applied. However, a considerable amount of manual work is required and the simulation process is fairly time-consuming, even though significantly less than the time required by commercial HLS tools. Finally, frameworks used for automatic binary parallelization [39] and for automatic parallelization of non-numerical applications [5] by decoupling communication from computation, in order to avoid the overhead due to synchronization, have also been proposed.

8 CONCLUSIONS

Early DSE in modern applications, along with the extraction of critical information about parallelism, can be crucial to the outcome of a final HW/SW design and its respective performance on SoCs. `Trireme` leverages

information automatically retrieved by HPVM and applies it to accelerators automatically identified and evaluated by AccelSeeker. Using novel performance models, `Trireme` is able to thoroughly explore a variety of parallelism strategies and select the highest performing HW/SW design as output for area budgets of increasing size. We have explored multiple SoC configurations, varying the data transfer bandwidth between memory and accelerators, as well as accelerator invocation overhead. Application of `Trireme` to the XR domain yields substantial speedup gain with fixed resources when compared with state-of-the-art tools (e.g., AccelSeeker [36]) that do not consider loop level, task level and pipeline parallelism.

9 FUTURE WORK

Our tool-chain, in its current state, does not offer automatic code generation for HW accelerators. Therefore, we plan to extend `Trireme`'s capabilities by either adding a High Level Synthesis step for automatic code generation or linking it to tools that already offer RTL synthesis, such as Catapult HLS [7]. Also, apart from SW CPUs coupled with HW accelerators, we wish to target more complex heterogeneous systems that may have GPUs, TPUs, DPUs etc. and formulate their respective evaluation/cost models. Finally, more optimizations (e.g. loop transformations, custom memory buffers etc.) can be considered, so that they can be implemented automatically, apart from the various forms of parallelism that are explored and studied in this work.

10 ACKNOWLEDGEMENTS

This work was supported in part by the 'Software Analysis for Heterogeneous Computing Architectures' (grant no. 191497) project funded by the Swiss National Science Foundation (SNSF), by the National Science Foundation (US) under grant CCF 16-19245 and NSF grant CNS-1718160, by DARPA through the Domain-Specific System on Chip (DSSoC) program, by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA and by two gifts from Intel Corp.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Feb. 2011), 1–7.

[2] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. In *Communications ACM*, Vol. 9. 575–577.

[3] Iulian Brumar, Georgios Zacharopoulos, Yuan Yao, Saketh Rama, Gu-Yeon Wei, and David Brooks. 2022. Early DSE and Automatic Generation of Coarse Grained Merged Accelerators. *ACM Trans. Embed. Comput. Syst.* (jun 2022). https://doi.org/10.1145/3546070

[4] Cadence. 2016. Stratus High-Level Synthesis. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[5] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 217–228.

[6] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. 2013. From software to accelerators with LegUp high-level synthesis. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. IEEE, 18.

[7] Catapult. 2017. Catapult High-Level Synthesis. https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/.

[8] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. 408–422. https://doi.org/10.1145/3385412.3385983

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, Vol. 39. 365–376.

[10] Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. 2021. A Graph Deep Learning Framework for High-Level Synthesis Design Space Exploration. *arXiv preprint arXiv:2111.14767* (2021).

[11] Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. 2022. Graph Neural Networks for High-Level Synthesis Design Space Exploration. *ACM Transactions on Design Automation of Electronic Systems* (2022).

[12] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. 2021. ILLIXR: Enabling End-to-End Extended Reality Research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 24–38. https://doi.org/10.1109/IISWC53511.2021.00014

[13] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[14] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80.

[15] Snehasish Kumar, Vijayalakshmi Srinivasan, Amirali Sharifian, Nick Sumner, and Arrvindh Shriraman. 2016. Peruse and profit: Estimating the accelerability of loops. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–13.

[16] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. 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*. 242–251.

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*. 75–88.

[18] LLVM/CIRCT. [n.d.]. llvm/circt. https://github.com/llvm/circt

[19] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–257.

[20] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (Sept. 2012), 31–51.

[21] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 425–426.

[22] Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. FCUDA-SoC: Platform integration for field-programmable SoC with the CUDA-to-FPGA compiler. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 5–14.

[23] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *2009 IEEE 7th Symposium on Application Specific Processors*. IEEE, 35–42.

[24] Christian Pilato and Fabrizio Ferrandi. 2012. Bambu: A Free Framework for the High Level Synthesis of Complex Applications. In *2013 23rd International Conference on Field programmable Logic and Applications*.

[25] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 110–119.

[26] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. 2020. gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 471–482.

[27] Tao B Schardl, William S Moses, and Charles E Leiserson. 2017. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 249–265.

[28] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE, 97–108.

[29] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[30] Tom Simonite. 2016. Moore's Law is Dead. Now What? *MIT Technology Review, May* 13 (2016), 40–41.

[31] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[32] Xilinx. 2017. Vivado High-Level Synthesis. www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[33] Xilinx. 2017. Xilinx All Programmable SoC portfolio. www.xilinx.com/products/silicon-devices/soc.html.

[34] Yuan Yao and Saketh Rama. [n.d.]. yaoyuannnn/cava. https://github.com/yaoyuannnn/cava

[35] Georgios Zacharopoulos, Andrea Barbon, Giovanni Ansaloni, and Laura Pozzi. 2018. Machine Learning Approach for Loop Unrolling Factor Prediction in High Level Synthesis. *2018 IEEE International Conference on High Performance Computing & Simulation (HPCS)* (2018), 91–97.

[36] Georgios Zacharopoulos, Lorenzo Ferretti, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. 2019. Compiler-Assisted Selection of Hardware Acceleration Candidates from Application Source Code. *Proceedings of the International Conference on*

Computer Design (2019), 1–9.

[37] Georgios Zacharopoulos, Lorenzo Ferretti, Emanuele Giaquinta, Giovanni Ansaloni, and Laura Pozzi. 2019. RegionSeeker: Automatically Identifying and Selecting Accelerators from Application Source Code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (April 2019), 741–754.

[38] Georgios Zacharopoulos and Laura Pozzi. 2017. *ClrFreqCFGPrinter: A Tool for Frequency Annotated Control Flow Graph Generation.* Technical Report. European LLVM Developers Meeting.

[39] Ruoyu Zhou and Timothy M Jones. 2019. Janus: statically-driven and profile-guided automatic dynamic binary parallelisation. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 15–25.