1-2017

Exploiting Hardware Abstraction for Parallel Programming Framework: Platform and Multitasking

Hongyuan Ding
University of Arkansas, Fayetteville

Exploiting Hardware Abstraction for Parallel Programming Framework:
Platform and Multitasking

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Engineering
with a Concentration in Computer Engineering

by

Hongyuan Ding
Shandong University
Bachelor of Engineering in Electrical Engineering, 2012
University of Arkansas
Master of Science in Computer Engineering, 2015

May 2017
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council

Dr. Miaoqing Huang
Dissertation Director

Dr. David Andrews
Committee Member

Dr. Wing Ning Li
Committee Member

Dr. Zhong Chen
Committee Member

Abstract

With the help of the parallelism provided by the fine-grained architecture, hardware accelerators on Field Programmable Gate Arrays (FPGAs) can significantly improve the performance of many applications. However, designers are required to have excellent hardware programming skills and unique optimization techniques to explore the potential of FPGA resources fully. Intermediate frameworks above hardware circuits are proposed to improve either performance or productivity by leveraging parallel programming models beyond the multi-core era.

In this work, we propose the PolyPC (**Poly**morphic **P**arallel **C**omputing) framework, which targets enhancing productivity without losing performance. It helps designers develop parallelized applications and implement them on FPGAs. The PolyPC framework implements a custom hardware platform, on which programs written in an OpenCL-like programming model can launch. Additionally, the PolyPC framework extends vendor-provided tools to provide a complete development environment including intermediate software framework, and automatic system builders. Designers' programs can be either synthesized as hardware processing elements (PEs) or compiled to executable files running on software PEs. Benefiting from nontrivial features of re-loadable PEs, and independent group-level schedulers, the multitasking is enabled for both software and hardware PEs to improve the efficiency of utilizing hardware resources.

The PolyPC framework is evaluated regarding performance, area efficiency, and multitasking. The results show a maximum 66× speedup over a dual-core ARM processor and 1043× speedup over a high-performance MicroBlaze with 125× of area efficiency. It delivers a significant improvement in response time to high-priority tasks with the priority-aware scheduling. Overheads of multitasking are evaluated to analyze trade-offs. With the help of the design flow, the OpenCL application programs are converted into executables through the front-end source-to-source transformation and back-end synthesis/compilation to run on PEs, and the framework is generated from users' specifications.

Acknowledgements

I would like to thank my committee members, especially my advisor Dr. Miaoqing Huang for providing constructive advice and necessary facilities for my research.

I would like to thank my best friends in the US, China, and Europe. I'm thankful that we could be together to explore this wonderful world by supporting each other and sharing our happiness and sadness.

I would like to thank my beloved parents and relatives for their love, continued support, and understanding. I never expect a more important relationship other than that between family members.

I would like to believe that the algorithm is science; while the framework is art, which reflects designer's understanding of aesthetics.

Contents

List of Figures

List of Tables

Glossary and Definitions

CU Compute Unit. A compute device in the OpenCL may consist of CUs.

PE Processing Element. Multiple PEs in both OpenCL standard and PolyPC framework comprise a CU. Work-items execute on PEs.

hPE Each PE can be either a dedicated hardware accelerator or a general-purpose processor. The accelerator PE is called an hPE, and the processor PE is called an sPE.

sPE See **hPE**.

Group A CU is called a group in the PolyPC framework. Sometimes it is called as hardware groups.

PS Processing System. A complete Zynq device consists of the PS, and the programmable logic (PL). The PS part is a hard-copy dual-core ARM processor. The PL part is fabricated as traditional FPGAs.

PL Programmable Logic. See **PS**.

PR Partial Reconfiguration. PR techniques enable reprogramming part of FPGA fabrics without affecting others.

Module It usually refers to a hardware component that can be either from vendors or users. It can be embedded into FPGA EAD tools to comprise a system.

PolyTask Polymorphic Task. A PolyTask is associated with a kernel program and consists of multiple threads. It can only run on compute device.

PolyTT PolyTask Table. It records execution information to run a PolyTask and to switch between them.

Hardware Thread A thread associated with a work-item within the kernel program is executed

on a PE. As a PE can be either an sPE or an hPE, a thread is called as a software thread or a hardware thread when it is executed on an sPE or an hPE, respectively.

Software Thread See **Hardware Thread**.

VSM Virtual Socket Manager. The PR controller works with multiple VSMs), each of which is associated with a PR region.

ICAP Internal Configuration Access Port.

PCAP Processor Configuration Access Port.

API Application Programming Interface. The defined interface of a piece of software, often a library or operating system.

CPU Central Processing Unit. A programmable hardware component often referred to as processor or core.

DMA Direct Memory Access. Often referring to hardware devices that can perform memory-to-memory operations without processor assistance.

DSP Digital Signal Processor. A processor specialized for signal processing, often featuring vector and multiply-accumulate operations.

FIFO First-In, First-Out. A hardware component or data structure that exhibits First-In, First-Out behavior (e.g. a queue).

FPGA Field Programmable Gate Array. A hardware chip whose functionality can be changed post-fabrication.

GPGPU General-purpose Graphics Processing Unit. A hardware device specialized for data parallelism computation.

ACC Hardware Accelerator. Dedicated hardware logics that are designed to finish a particular task.

Chapter 1

Introduction

1.1 Multi-core Era and Beyond

Before the general-purpose computing community decided to shift to multi-core systems, general-purpose processors within the computer system kept pushing to their limits in terms of processing technology, and advanced microarchitecture. Benefiting from the periodic device feature size scaling predicted by Moore's law (Dennard scaling [24]), general-purpose processors were improving the performance accordingly for decades. Besides, with the help of multiple abstractions of software programming such as operating systems, compilers and programming models, general-purpose computer systems provide designers significant productivity, flexibility and additional performance improvement.

However, the transistor cannot shrink forever [63]. By 2004, the general-purpose computing community began to increase the number of cores to extend Moore's law instead of squeezing more performance out of a single-core processor [39]. This mainly resulted from the failure of Dennard scaling. The slowed processor clock frequency scaling that did not scale along with the feature size anymore, which thereby limits the increasing performance of a single-core processor. The overall performance is improved by parallelizing workloads on multi-core processors. Besides the processor clock frequency, the supply voltage fails to scale along with the feature size as well [37], and thus power density is increasing. As the number of cores increases, the incapability of heat dissipation system may prevent fully powering of all cores at the same time, requiring some cores to run at low speed not to exceed the processor thermal design power (TDP) [36, 90]. This phenomenon is called *dark silicon*: transistors that can be used on a chip are far less than the economic manufacturability sweet-spot [10, 38, 46].

Another constraint contributing to the shift to multi-core systems is the nearly-stalled

improvement of processors' microarchitecture. Advanced techniques applied in the general-purpose single-core processor, such as superscalar execution, deeper pipelines, and efficient cache design, have exploited its potential. Efforts to perfect the processors' microarchitecture design obtain fewer gains over generations.

The transition to multi-core era seemed to be inevitable at first but was forced indeed. This raises the following two questions:

1. ***How much more performance left in multi-core processors in the future?*** In the multi-core era key questions that need to answer are if the performance gain is worth scaling down or if we can scale down in the future after the failure of Moore's law. But one thing for sure is without breakthroughs of processing technologies multi-core systems (e.g., CPU and GPU) cannot benefit from performance gains that they used to have in the single-core era. Even considering parallel programming optimization along with processing scaling, the average annual performance gain is about 16% to 23% before 8 nm emerging in the short future [36, 37]. Besides power constraints, the community can find alternative solutions in realms such as multi-core microarchitecture, specific acceleration, and programming methods.

2. ***How much speedup we can get from parallelism on multi-core processors?*** For decades, the single-core processor and sequential programming models dominated the general-purpose computing and gave its way in the multi-core era. However multi-core systems bring new challenges for software programming, and it takes time for programmers and software techniques to catch up with new trends. Amdahl's law provides a method to estimate how much speedup obtained through parallelism. But what matters is how to parallelize applications. Programming models are evolving to meet new demands. In those symmetric multiprocessor (SMP) systems, standards interfaces, such as Posix threads (PThreads) or OpenMP, are used by software programmers [48, 82]. In distributed systems where there are fundamental differences from multi-core systems, programmers are able to

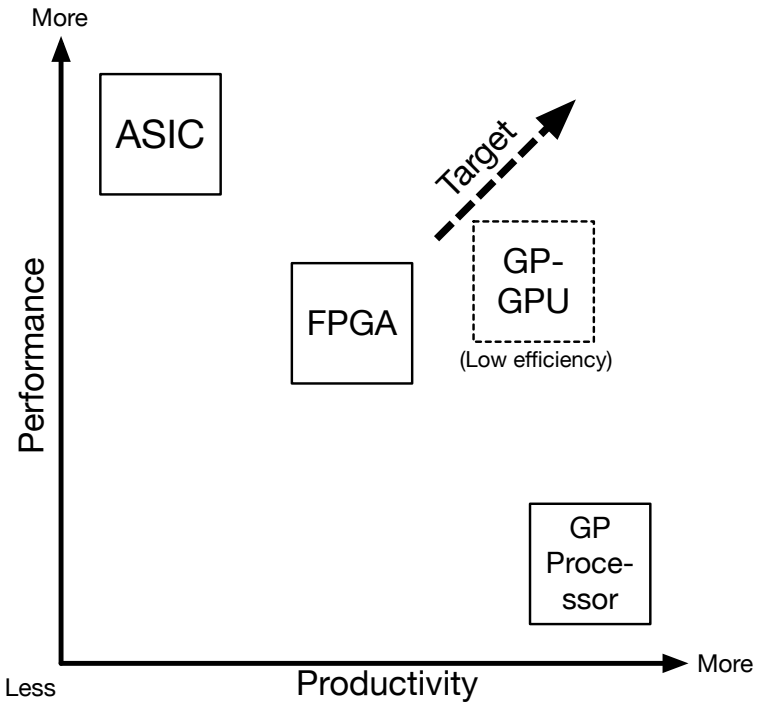

Figure 1.1: Heterogeneity Considerations

use to various standards such as message passing interface (MPI) and CORBA [101]. In recent years, domain-specific models such as MapReduce [23] and Spark [97] are used to handle big data analysis. However, speedup of small fractions of an application may be negligible due to overheads of these programming models. For heterogeneous multi-core systems, the standard SMP programming models may be unsuitable. Furthermore, some applications, such as non-computation intensive applications, run well on single-core processors; while computation intensive applications are able to have much speedup on SIMD systems such as emerging general-purpose graphic processing units (GPGPUs).

1.2 Heterogeneous Computing

In the multi-core era, one computing resource cannot meet all requirements in terms of performance, power efficiency, and productivity. The community is seeking solutions through heterogeneity. One of the most promising approaches is to leverage specification.

For a particular task, custom logic via application-specific integrated circuits (ASICs) can achieve up to three orders of magnitude differences in both power efficiency and performance over general-purpose processors [17, 29, 32, 44]. But the heavy use of ASICs on general applications remains many challenges. This is reflected in productivity and flexibility. Designing ASICs requires much more efforts than on general-purpose processors and implementing frequently-changed applications is infeasible. Programmable accelerators (i.e., FPGAs and GPGPUs) provide the trade-off between the performance of ASICs and the productivity of general-purpose processors (see Figure 1.1). GPGPUs leverage SIMD to accelerate data-intensive applications; while FPGAs accelerate applications through programmable logics. GPGPUs are still based on massive simple general-purpose processors. By using parallel programming models (i.e., CUDA and OpenCL), they can achieve noticeable productivity even compared to general-purpose processors. But recent research work and industrial trends reveal that FPGAs are better at web services requiring low-latency and high-frequency requests than GPGPUs in data centers [15, 66, 88].

1.3 FPGA as a Candidate

With the help of hardware accelerators, FPGAs are able to significantly improve the performance of many applications due to the parallelism provided by the fine-grained architecture. However, the dearth of flexibility and programmability affects their widespread adoption. In order to fully explore the computing potential of FPGA resources, designers are required to have excellent hardware programming skills and special optimization techniques. Usually, on a single computer system, software programs can be compiled to binaries running on processors and software programmers do not need to design application-specific hardware architectures. On the contrary, programming hardware circuits by using hardware description languages (HDLs) on FPGAs requires comprehensive hardware-oriented knowledge including pipelining, clock domains, data flows, and memory managements. Therefore, designing hardware accelerators usually results in

particular hardware architecture per application.

FPGA devices have grown to a point where advanced processing technology is adopted to achieve high capacity. State-of-the-art FPGAs can accommodate complicated heterogeneous systems. Thus, hardware designers are willing to utilize extra resources to trade for flexible architectures and productive design flows [3]. One of the most promising approaches is parallel computing platforms with multiple processing elements such as Graphic Processing Units (GPUs) and Multiprocessor System-on-Chip (MPSoC). Parallel programming languages such as OpenCL are used to program these platforms efficiently and productively. OpenCL is an open standard for parallel computing frameworks across GPUs, CPUs, and even FPGAs. OpenCL introduces several models including execution, platform, memory, and programming models. These models provide the unified hardware and software abstractions for programmers. Designers are released from the burden of designing hardware architectures, and productivities and efficiency are improved significantly.

One way to address the programmability of hardware logics within FPGAs is to use High-level Synthesis (HLS) techniques. In [85], applications' parallelism is expressed in a kernel program that is directly synthesized into FPGA logics. This technique is very efficient in terms of performance and power consumptions. Productivity is satisfying since designers do not have to dig into the hardware details. But when multiple kernels are launched, they have to be synthesized multiple times, which is significantly time-consuming. Besides, when one kernel is changed, the whole platform has to be synthesized again. Other approaches target using soft processors. They can be configured as many-core systems or overlays by using customized processors [65, 91] or multiprocessor systems with vendor-provided processors [27, 28, 31]. Recent works also focus on dedicated soft-GPU cores on FPAGs [3]. These approaches increase productivity by leveraging general-purpose processors on a pre-defined hardware architecture. Comparing to HLS techniques that every application has their logics, soft-processor-based solutions change different kernels by compiling, which is usually much faster than synthesizing the same functionality. However, the

trade-off is that they lose performance and area efficiency compared to dedicated logics.

However, heterogeneous systems on FPGAs still face the following challenges:

- Can platform specification details be removed from users' codes and be encapsulated within an automatic system builder?
- Can the complexity of designing parallel applications be reduced by applying standard parallel programming models on FPGAs?
- What are the benefits when heterogeneous computing resources are used on FPGAs?
- Can the system efficiency be increased by introducing the priority-aware multitasking?

1.4 Statements and Contributions

In this work, we want to answer the questions raised in the previous section by proposing the PolyPC (**Poly**morphic **P**arallel **C**omputing) framework. The PolyPC framework targets improving productivity without losing performance when designers want to implement their parallelized applications on FPGAs. The PolyPC framework implements a custom hardware platform, along with software intermediate framework and automatic system builders. On the one hand, designers can write their programs by using the easy-to-use OpenCL-like programming model to optimize their applications. Firstly, the automatic system builder takes the specifications to generate the hardware platform. Then the software programs are either synthesized into hardware accelerators or compiled to executable files running on general-purpose processors. On the other hand, different from previous work that the whole hardware system is generated per application, the PolyPC framework enables partially reloaded processing elements (PEs) for both sPEs and hPEs. Therefore, PEs can be generated off-line without changing the static system and loaded into the system during runtime environment. Benefiting from the flexibility of re-loadable PEs, the context switching feature is enabled for PEs to improve the efficiency to utilize hardware resources. The priority-aware scheduling policy is used to provide fast responses to tasks with high priorities.

1.5 Dissertation Organization

In the next chapter, we discuss the background in terms of parallel programming models and detailed illustration of the OpenCL standard working on the PolyPC framework. Chapter 3, Chapter 4 and Chapter 5 provide discussions of approaches for the PolyPC framework to address questions raised in this chapter.

The hardware architecture of the PolyPC framework, as well as the evaluation of pure performance part, is talked about in Chapter 3. Chapter 4 discusses the polymorphic multitasking feature. Besides, experiments and results related to this feature are addressed as well. Chapter 5 demonstrates how to improve the productivity to implement parallel applications on the PolyPC framework by using the design flow. Finally, Chapter 6 concludes this work and discusses the future work.

Chapter 2

Backgrounds

2.1 Parallel Programming Models

A parallel programming model is an abstraction of the computer architecture [25]. There are many programming models for parallel computing because how to put several processors or computing units together to form a complete system is different from one programming models to another. Besides, it is difficult to separate one particular programming model to its hardware implementation. In other words, the same programming model may have different software compilation flows and hardware architectures.

POSIX Threads In this model, several concurrent execution paths (threads) can be controlled independently. A thread is a lightweight process having its own program counter (PC) and execution stack [7]. The model is very flexible, but it usually associated with shared memory and operating systems.

The Pthreads, or Portable Operating System Interface (POSIX) Threads, is a set of C programming language types and procedure calls [11, 55, 60]. Pthreads is implemented as a header (pthread.h) and a library for creating and manipulating each thread. The Pthreads library provides functions for creating and destroying threads and for coordinating thread activities via constructs designed to ensure exclusive access to selected memory locations (locks and condition variables). This model is especially appropriate for the fork/join parallel programming pattern [75].

In general, Pthreads is not recommended as a parallel program development technology. While it has its place in specific situations, and in the hands of expert programmers, the

unstructured nature of Pthreads makes the development of correct and maintainable programs difficult. Also, recall that the number of threads is not related to the number of processors available. These characteristics make Pthreads programs not scalable to a large number of processors [93].

OpenMP OpenMP [16] is a shared memory application programming interface (API) whose aim is to ease shared memory parallel programming. The OpenMP multithreading interface [82] is specifically designed to support high-performance computing (HPC) programs. It is also portable across shared memory architectures. OpenMP differs from Pthreads in several significant ways. While Pthreads is purely implemented as a library, OpenMP is implemented as a combination of a set of compiler directives, pragmas, and a runtime providing both management of the thread pool and a set of library routines. These directives instruct the compiler to create threads, perform synchronization operations, and manage shared memory. Therefore, OpenMP does require specialized compiler support to understand and process these directives.

In OpenMP, the use of threads is highly structured because it was designed specifically for parallel applications. In particular, the switch between sequential and parallel sections of code follows the fork/join model [7]. This is a block-structured approach for introducing concurrency. A single thread of control splits into some number of independent threads (the fork). When all the threads have completed the execution of their specified tasks, they resume the sequential execution (the join). A fork/join block corresponds to a parallel region, which is defined using the *PARALLEL* and *END PARALLEL* directives.

The characteristics of OpenMP allow for a high abstraction level, making it well suited for developing HPC applications in shared memory systems. The pragma directives make it easy to obtain the concurrent code from serial codes. In addition, the existence of specific directives eases parallelizing loop-based code.

MPI Message Passing is a parallel programming model where communication between processes is done by interchanging messages. This is a natural model for a distributed memory system, where communication cannot be achieved by sharing variables. Over time, a standard has evolved and dominated for this model: the Message Passing Interface (MPI). MPI is a specification for message passing operations [42, 42, 84, 93]. MPI is a library, not a language. It specifies the names, calling sequences, and results of the subroutines or functions to be called from Fortran, C or C++ programs. Thus, the programs can be compiled with ordinary compilers but must be linked with the MPI library.

In this model, the processes executed in parallel have separate memory address spaces. Communication occurs when part of the address space of one process is copied into the address space of another process. This operation is cooperative and occurs only when the first process executes a send operation and the second process executes a receive operation. In MPI, the workload partitioning and task mapping have to be done by the programmer, similarly to Pthreads. Programmers have to manage what tasks are to be computed by each process.

In summary, MPI is well suited for applications where portability, both in space (across different systems existing now) and in time (across generations of computers), is important. MPI is also an excellent choice for task-parallel computations and for applications where the data structures are dynamic, such as unstructured mesh computations.

CUDA CUDA is a parallel programming model developed by NVIDIA [52]. The CUDA model is designed to develop applications scaling transparently with the increasing number of processor cores provided by the GPUs [49, 62]. CUDA provides a software environment that allows developers to use C as a high-level programming language. For CUDA, a parallel system consists of a host (i.e., CPU) and a computation resource or device (i.e., GPU). The computation of tasks is done in the GPU by a set of threads running in parallel. The GPU threads architecture consists of a two-level hierarchy, namely the block and the grid.

The block is a set of tightly coupled threads, each identified by a thread ID. On the other hand, the grid is a set of loosely coupled blocks with similar size and dimension. There is no synchronization between the blocks, and a single GPU handles an entire grid. The GPU is organized as a collection of multiprocessors, with each multiprocessor responsible for managing one or more blocks in a grid. A block is never divided across multiple multiprocessors. Threads within a block can cooperate by sharing data through some shared memory, and by synchronizing their execution to coordinate memory accesses.

Worker management in CUDA is done implicitly. That is, programmers do not manage thread creations and destructions. They just need to specify the dimension of the grid and block required to process an individual task. Workload partitioning and worker mapping in CUDA is done explicitly. Programmers have to define the workload to run in parallel by using the function *Global Function* and specifying the dimension and size of the grid and of each block.

A recent initiative by Intel is called Knights Ferry [95, 103], and is being developed under the Intel Many Integrated Core (MIC) architecture. Knight Ferry is implemented on a PCI card with 32 x86 cores. The MIC supports a more classical coherent shared memory parallel programming paradigm than CUDA. Moreover, it will be programmed using native C/C++ compilers from Intel.

OpenCL OpenCL [43] is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors. Primarily, OpenCL distinguishes between the devices (usually GPUs or CPUs) and the host (CPU). The idea behind OpenCL is to write kernels (functions that execute on OpenCL devices) and APIs for creating and managing these kernels. The kernels are compiled for the targeted device at runtime, during the application execution. This enables the host application to take advantage of all the computing devices in the system.

The OpenCL operational model can be described as four interrelated models: the platform,

execution, memory, and programming models. The platform model is viewed from a hierarchical and abstract perspective. In this model, a host coordinates execution, transferring data to and from an array of Compute Devices. Each Compute Device is composed of an array of Compute Units. Each Compute Unit is comprised of an array of Processing Elements.

Execution of an OpenCL program involves simultaneous execution of multiple instances of a kernel on one or more OpenCL devices. A kernel is the primary executable code unit. It is called work-item and has a unique ID. Work-items can be organized into work groups for synchronization and communication purposes. The different executions of a program are queued and controlled by the host application. This last sets up the context in which the kernels run, including memory allocation, data transfer among memory objects, and the creation of command queues used to control the sequence in which commands are executed. However, the programmer is responsible for synchronizing any necessary execution order.

OpenCL has been designed to be used not only in GPUs but also in other platforms like multi-core CPUs. Thus, it can support both data parallel [47], and task parallel [76] programming patterns [75], which are well suited for GPUs and CPUs architectures, respectively.

DirectCompute DirectCompute is Microsofts approach to GPU programming. DirectCompute is a part of the Microsoft DirectX APIs collection [50]. In fact, it is also known as DirectX11 Compute Shader. It was initially released with the DirectX 11 API but runs on both DirectX 10 and DirectX 11 graphics processing units. In particular, it was introduced thanks to the new Shader Model 5 [51] provided in DirectX 11, which allows computation independently of the graphic pipeline, therefore suitable for GPGPUs. The main drawback of DirectCompute is that it only works on Windows platforms.

Array Building Blocks Intels Array Building Blocks (ArBB) provides a generalized vector-parallel-programming solution for data-intensive mathematical computation [40, 56, 61]. Users express computations as operations on arrays and vectors. ArBB comprises a standard C++

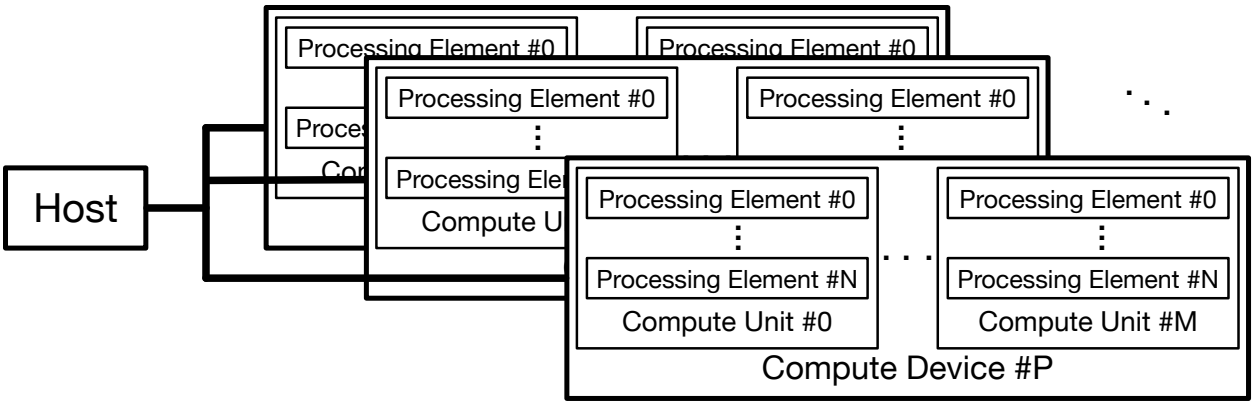

Figure 2.1: OpenCL Platform Model

library interface and a powerful runtime. A just-in-time (JIT) compiler supplied with the library translates the operations into the target-dependent code, where a target could be the host CPU or an attached GPU. As for runtime, ArBB uses Intels Threading Building Blocks [57], which contributes to abstract platform details and threading mechanisms for scalability and performance. Intels ArBB can run data-parallel vector computations on a possibly heterogeneous system. By design, Intel ArBB prevents parallel programming bugs such as data races and deadlocks.

2.2 The OpenCL Standard Overview

OpenCL is an open industrial standard providing the low-level hardware abstraction plus a framework to support programming on heterogeneous hardware platforms [43]. The OpenCL framework can be described by using several models:

- Platform model.
- Memory model.
- Execution model.

2.2.1 Platform Model

The platform model is an abstraction describing how OpenCL views the hardware. As shown in Figure 2.1, the OpenCL platform model consists of one host processor and one or more compute devices, each of which contains multiple compute units (CUs). A single compute unit is comprised of multiple processing elements (PEs), within which the computation occurs. The OpenCL framework can be deployed on heterogeneous compute devices, such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and even central processing units (CPUs). The specific hardware implementation of the platform model is different per vendor even for the same compute device [99]. Four major commercial solutions are listed as following:

- **NVIDIA GPUs** [21]. In a NVIDIA GPU, the compute unit is called as a streaming multiprocessor (SM). A streaming multiprocessor consists multiple stream processor (SP) cores. In modern GPU architecture, each SP core is highly multithreaded. SMs employ the NVIDIA-called SIMT (single-instruction, multiple-thread) architecture. Threads are divided into groups of 32 threads, which are called warps. Warps are executing on SMs, and further on SIMD units in lockstep. To improve throughputs, warps are scheduled and interleave their execution through the pipeline of SIMT units.

- **AMD GPUs** [5]. AMD GPU devices vary in architectures through generations. For devices in Evergreen and Northern Islands families, a compute unit is called a SIMD (single-instruction, multiple-data) engine that consists of multiple stream cores (SCs). SCs execute VLIW (very long instruction word) 4 (for Northern Islands) or 5 (for Evergreen) wide instructions from one wavefront (similar to warps in NVIDIA devices) in lockstep. For devices in Southern Islands family, which is also known as GCN (Graphic Core Next), each compute unit has one scalar unit and four vector unit. Each vector unit contains an array of 16 PEs and works as the SIMD architecture.

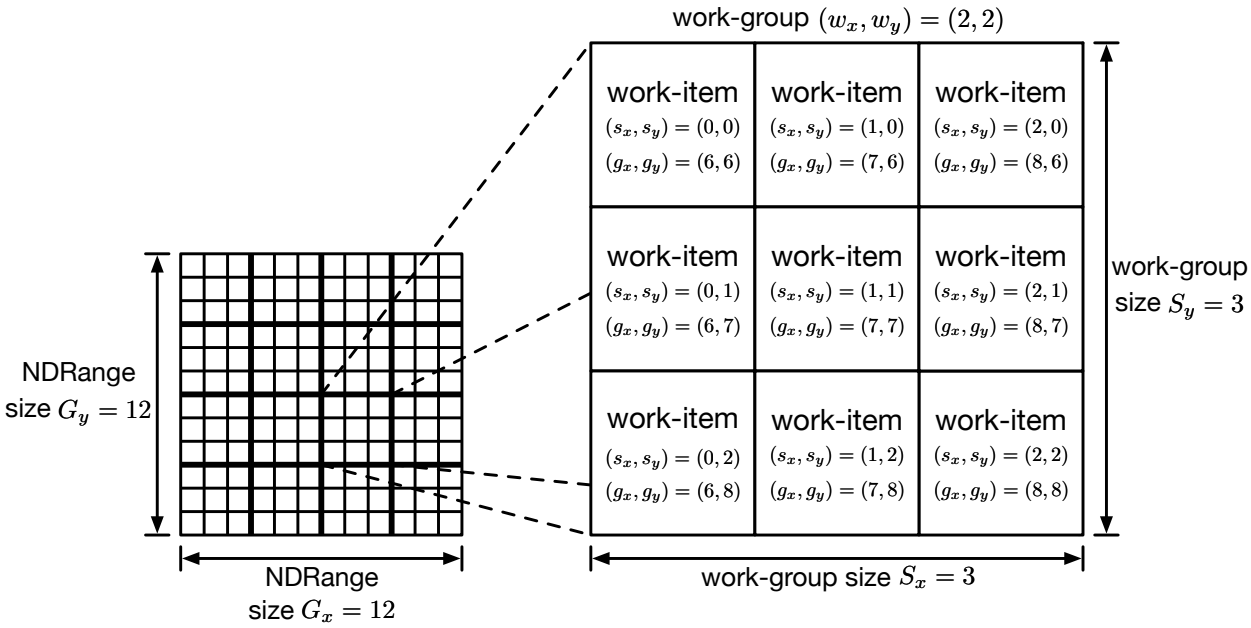

Figure 2.2: OpenCL Execution Model

- **Intel FPGAs** [94]. Different from how parallelism is revealed in GPUs, Intel FPGAs use the concept of pipeline parallelism. OpenCL programs are transformed into dedicated and pipelined hardware circuits on FPGAs by using their SDK tools. A PE is called a kernel pipeline, which is the dedicated and pipelined circuit module. Kernel pipelines duplicate themselves and group into multiple compute unit.

- **Xilinx FPGAs** [105]. SDAccel is the SDK tool that Xilinx launched for designers to implement the OpenCL framework on Xilinx FPGAs. OpenCL programs compatible with SDAccel are converted into the form that is acceptable to the Xilinx high-level synthesis (HLS) tool. One compute unit is generated from an OpenCL kernel program and can duplicate itself. PEs within the compute unit are revealed by unrolling the `for` loop in the kernel program. Same with Intel's solution, the dedicated hardware module can be pipelined.

2.2.2 Execution Model

An OpenCL application is combined with two programs: a set of kernel programs and a host program. A kernel program that executes on one or more OpenCL devices is defined as a unit of execution: kernel.

Kernel programs execute across a global domain of work-items, where an index space is defined. The work-items in a grid are broken up into work-groups. A work-item has a global ID according to the coordinate within the index space; while the local ID is defined based on the position within its work-group. The index space is called an NDRange in OpenCL. The NDRange is an N-dimensional index space, where N is up to three. The NDRange is defined by the size of the index space (global size) in each dimension, the size of a work-group (local size), and the offset F indicating the initial value of the indices. Each work-item's global ID is an N-dimensional tuple, as well as the local ID. Work-groups are also assigned IDs that indicating positions of the work-groups as blocks within the NDRange. Therefore, a work-item is calculated by using two ways: (1) a global index; (2) the size and index of work-group and local index.

As the example is shown in Figure 2.2, a 2-dimensional NDRange is demonstrated. The NDRange is defined by the NDRange size (G_x, G_y), the work-group size (S_x, S_y), and the global ID offset (F_x, F_y). Each work-item is identified by either its global ID (g_x, g_y) or the calculation of work-group ID (w_x, w_y), the work-group size (S_x, S_y), and the local ID (s_x, s_y) within its work-group such that

$$(g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y)$$

The number of work-groups are computed as [1]

$$(W_x, W_y) = (ceil(\frac{G_x}{S_x}), ceil(\frac{G_y}{S_y}))$$

[1] Beginning from OpenCL 2.0, work-group sizes could be non-uniform in multiple dimensions when the global size is not divisible by the work-group size in each dimension.

The real-world example of dot product is shown below. We want to calculate the dot product of two vectors $\mathbf{a} = [a_0, a_1, ..., a_{15}]$, and $\mathbf{b} = [b_0, b_1, ..., b_{15}]$, which is defined as $\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{15} a_i b_i$. The intermediate result $\mathbf{c} = [c_0, c_1, ..., c_{15}]$ is denoted as $c_i = a_i b_i$, where $0 \leq i \leq 15$. To calculate vector $\mathbf{c}$ by using OpenCL, the 1-dimensioned global index space is defined as 1×16, and the local index space (the size of workgroup) is as 1×4. Therefore, there are totally 4 workgroups, each of which contains 4 work-items, i.e., $c_0 ... c_3$, $c_4 ... c_7$, $c_8 ... c_{11}$, and $c_{12} ... c_{15}$.

A host program that executes on the host uses the OpenCL API to create and manage the context through command queues. A context contains several hardware and software resources including compute devices, kernel objects, program objects, and memory objects. Each single compute device within the context is associated with a command queue. The command queue is categorized into three types: (1) kernel-enqueue commands that enqueue a kernel program on a device; (2) memory commands that manage memory mappings and transfer data between memories; (3) synchronization commands that control the command execution order explicitly. Commands within its command queue execute in either in-order execution mode or out-of-order execution mode.

In OpenCL, there are different domains of synchronization, i.e., command synchronization and work-group synchronization. Command synchronization can constrain the execution order of commands within a command queue in an out-of-order execution mode. Work-items within a single work-group can be synchronized by using the work-group synchronization.

The work-group is scheduled to execute on a compute unit. Every work-item is physically executed on a processing element within its corresponding compute unit. Through this mapping of compute device$\leftrightarrow$kernel, compute unit$\leftrightarrow$work-group, processing element$\leftrightarrow$work-item, the data parallelism in an application is expressed.

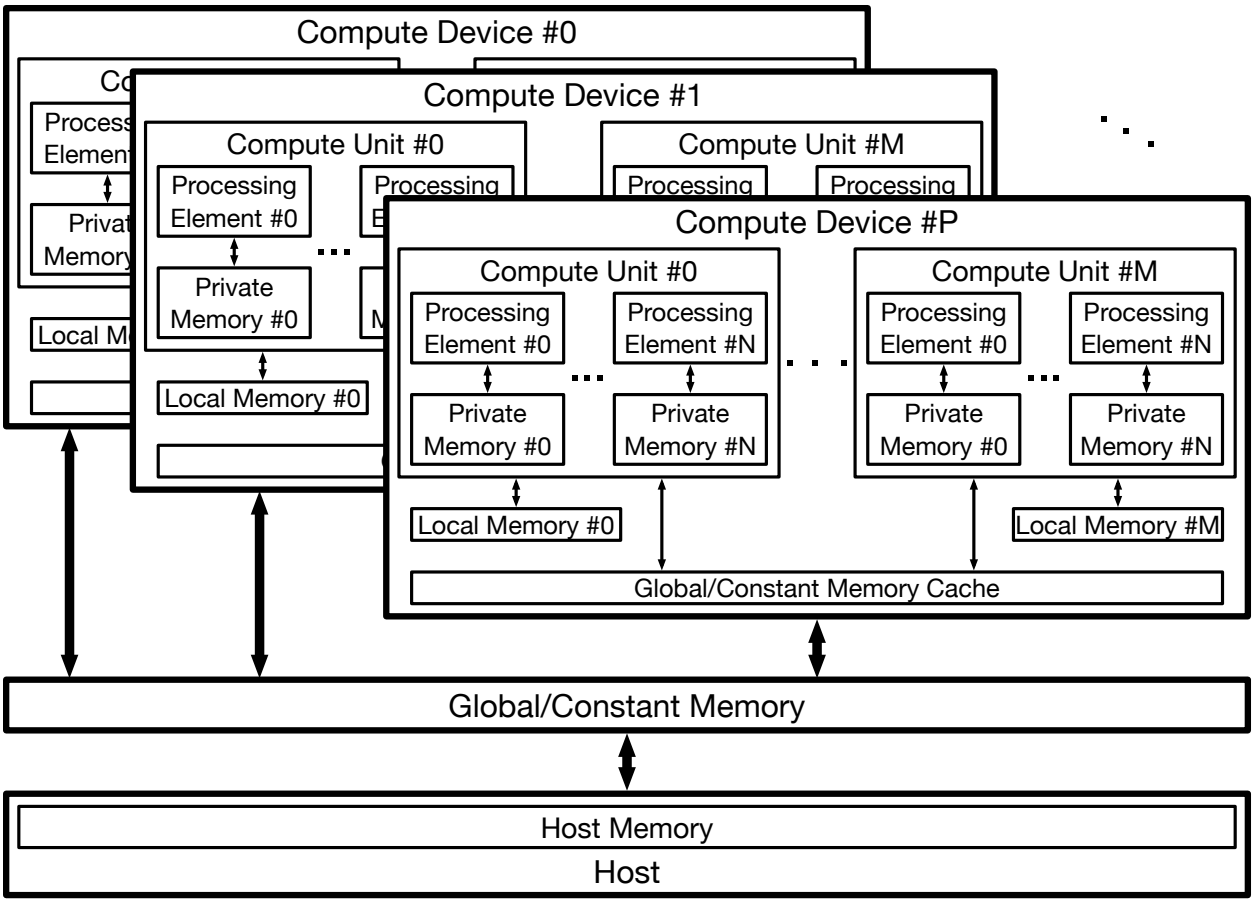

Figure 2.3: OpenCL Memory Model

2.2.3 Memory Model

Memory model showing in Figure 2.3 defines how memory resources are seen by multiple compute resources, and managed by the OpenCL applications. Memories in OpenCL are divided into two parts: host memory and device memory. Host memory is only accessible by the host, and device memory is only for the compute devices[2].

The device memory is further decomposed into a three-level memory hierarchy.

- **Global/Constant Memory.** The global memory is accessible by all work-items within all work-groups for reading and writing. The constant memory is also accessible by all

[2]OpenCL 2.0 brings the new feature of shared virtual memory (SVM), by using which the host and compute devices can share the same virtual memory space without moving data explicitly.

Table 2.1: Allocation and Accessibility to Memory Objects within Memory Hierarchy

		Global Memory	Constant Memory	Local Memory	Private Memory
Host	Allocation	Dynamic	Dynamic	Dynamic	-
	Accessibility	Read/Write	Read/Write	-	-
Kernel	Allocation	Static	Static	Static	Static
	Accessibility	Read/Write	Read	Read/Write	Read/Write

work-items but remains constant during the execution of kernels.

- **Local Memory.** The local memory is located in a work-group and is shared by all work-items within the work-group.
- **Private Memory.** The private memory is exclusively accessible by the work-item, and cannot be visible to other work-items.

The memory hierarchy guarantees the memory isolation among them referring to the logic concept. In host programs, the OpenCL API is used to create and manage memory objects to move data between host and device memories. A memory object is a handle pointing to a space within the memory. Table 2.1 summarizes the allocation and accessibility to memory objects for different memory hierarchies. Dynamic allocation means the memory space is managed at runtime while static allocation is at compile time.

Chapter 3

Hardware Architecture of the PolyPC Framework

3.1 Brief Introduction of Zynq Architecture

The PolyPC framework is built on top of the Xilinx Zynq-7000 series (i.e., Z7045 device on the ZC706 development kit in this work). A complete Zynq device consists of the processing system (PS) and the programmable logic (PL). The PS part is a hard-copy dual-core ARM processor. The PL part is fabricated as traditional Kintex-7 FPGAs.

One of the most important components within the PS is the dual-core ARM processor. Figure 3.1 demonstrates the overview architecture of the Zynq SoC. The PS and the PL parts communicate with each other through various interfaces including four general-purpose (GP) AXI interfaces, four high-performance (HP) AXI interfaces, and one AXI accelerator coherence port (ACP) interface.

- GP AXI interfaces provide two slave ports and two master ports. They are designed for general purposes (e.g., passing control signals from the PS to the PL) instead of achieving high performance. The GP AXI master bus is usually used to control the hardware modules in the PL part by the PS processor.
- The four HP AXI interfaces provide four PS master ports with high bandwidth data path to the PS DDR memory. Each interface includes two FIFO buffers for read/write traffic. The PL communicate with memory interconnect to the DDR interface through the four high-speed HP AXI ports. HP AXI interfaces are used by applications that need high throughputs between the PL and the PS.
- The ACP interface provides low-latency access to PL logics, with optional coherency with

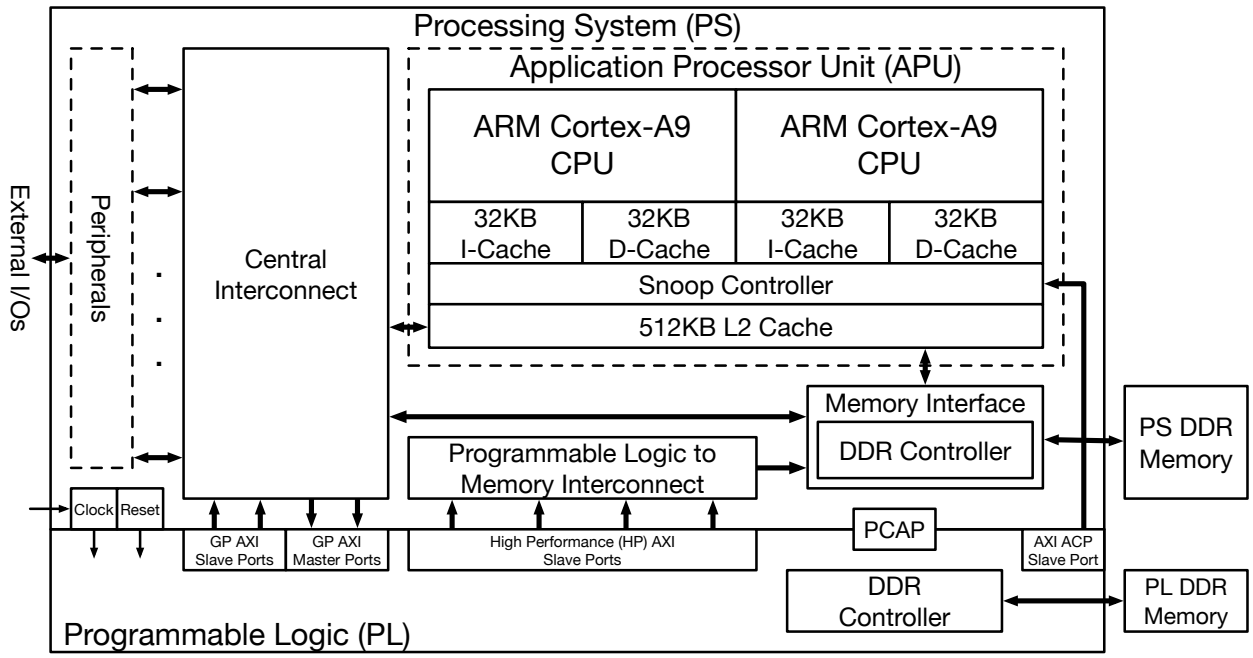

Figure 3.1: Zynq SoC Architecture

L1 and L2 cache. The master port is connected to the snoop controller that is responsible for data coherence and consistency. In other words, hardware modules that connects to the AXP slave port in the PL shares the same snoop controller with the PS processor.

The PS has a PS DDR memory that is for applications running on the ARM processor (i.e., Linux OS). The PL DDR memory is optional depending on the board design. If the PL DDR memory is connected to the PL, a DDR controller IP within the PL provides AXI interfaces that PL modules can access. The PL can be controlled via PS software through the processor configuration access port (PCAP) bridge with a bitstream file. The PS provides the clock and reset module that can drive PL logics.

3.2 PolyPC Architecture Overview

Different from existing hardware architecture, the PolyPC framework has its hardware architecture which adapts the hardware abstraction from the OpenCL platform model and extends to support unique features (i.e., polymorphic task and polymorphic multitasking). Figure 3.2

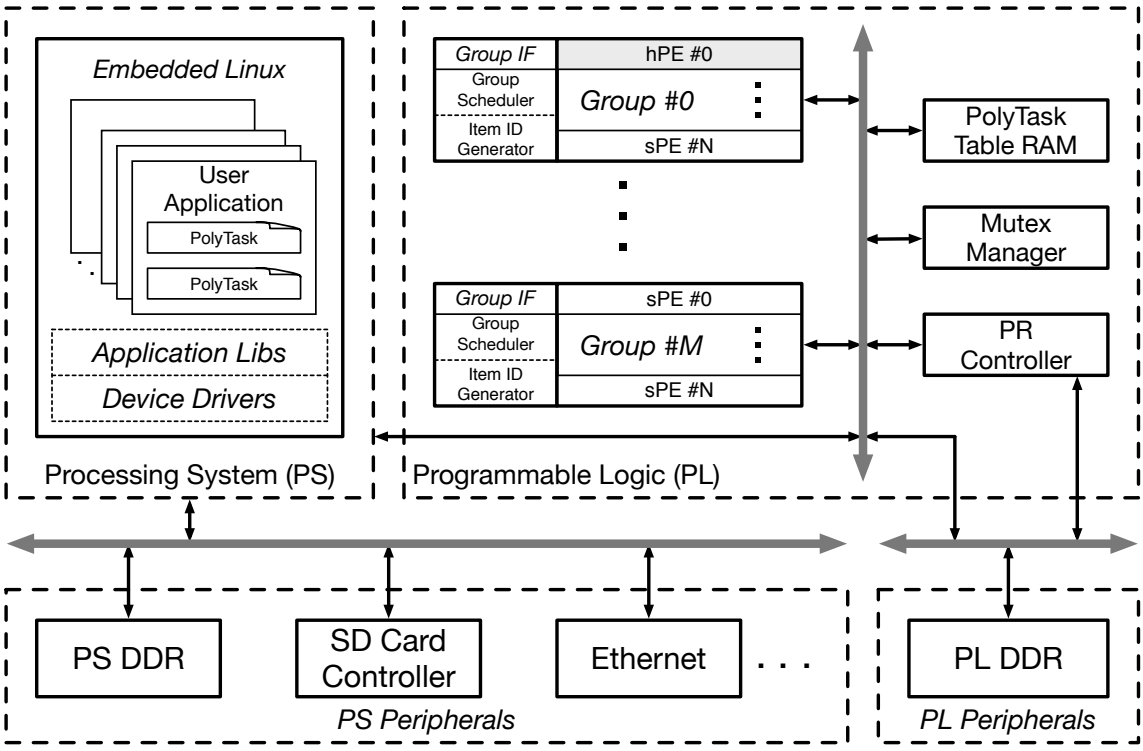

Figure 3.2: PolyPC Platform Architecture

shows the overview of the PolyPC framework architecture. In the PolyPC framework, the PS is equivalent to the host in the OpenCL platform model, and the architecture implemented on the PL is equal to the compute device. Currently, only one compute device is supported for one host in the PolyPC framework. Groups (sometimes, we call them hardware groups for clarification) are equivalent to and extends functionalities from Compute Units (CUs) within the OpenCL platform model. Each group may contain multiple processing elements (PEs) that are the same with PEs in OpenCL. The PolyPC framework supports a heterogeneous PE architecture. Each PE can be either a dedicated hardware accelerator or a general-purpose processor (i.e., MicroBlaze). The accelerator PE is called an hPE, and the processor PE is referred to as an sPE. PEs within one group can be all hPEs, sPEs, or the combination of them. hPEs can be either unchangeable or re-loadable by using the partial reconfigurable (PR) techniques. The sPEs are always re-loadable since they are general-purpose processors that can load any programs.

An embedded Linux is booted on the PS. Application libraries provide the OpenCL API

functions that are used in the host programs for the PolyPC framework. Device drivers are low-level interfaces between the Linux applications and devices within the PL part. User applications are similar to the host programs from the OpenCL execution model. In the PolyPC framework, a host program (user application) is a typical Linux user program written in the C language. In a host program, one or more polymorphic tasks (PolyTasks) are created, launched, and committed. A PolyTask is associated with a kernel program that is similar to the kernel program in the OpenCL execution model and runs on the compute device. It has polymorphic forms that can run on both sPEs and hPEs.

PS peripherals are exclusively accessible by the host processor. The Linux OS uses the PS DDR memory. Binary files and input data files are stored on the SD card for further loading into the DDR memory to process. The output results can also be moved back to the SD card after processing. The PL DDR memory is accessible by both PS and PL parts. During runtime, data elements are loaded from and stored into the PL DDR memory and processed on the PL. Other hardware modules within the PL part (i.e., PolyTask Table RAM, Mutex Manage, and PR Controller) support unique features of the PolyPC framework.

3.3 Memory Hierarchy

Similar to the OpenCL memory model, the PolyPC framework provides a three-level memory hierarchy within one compute device including private memory, local memory, and global/constant memory. The host memory is exclusively accessible by the host processor.

Figure 3.3 demonstrates the implementation of the three-level memory hierarchy in the PolyPC framework. The global/constant memory is visible to all PEs within their group on the compute device. Besides, global/constant memory is also accessible by the host processor and is used to temporarily store data and binary files that are utilized by the compute device. The local memory is located in a group and shared by PEs within the group. The private memory is exclusive owned per PE. Within each group, a group DMA can move datum between the local

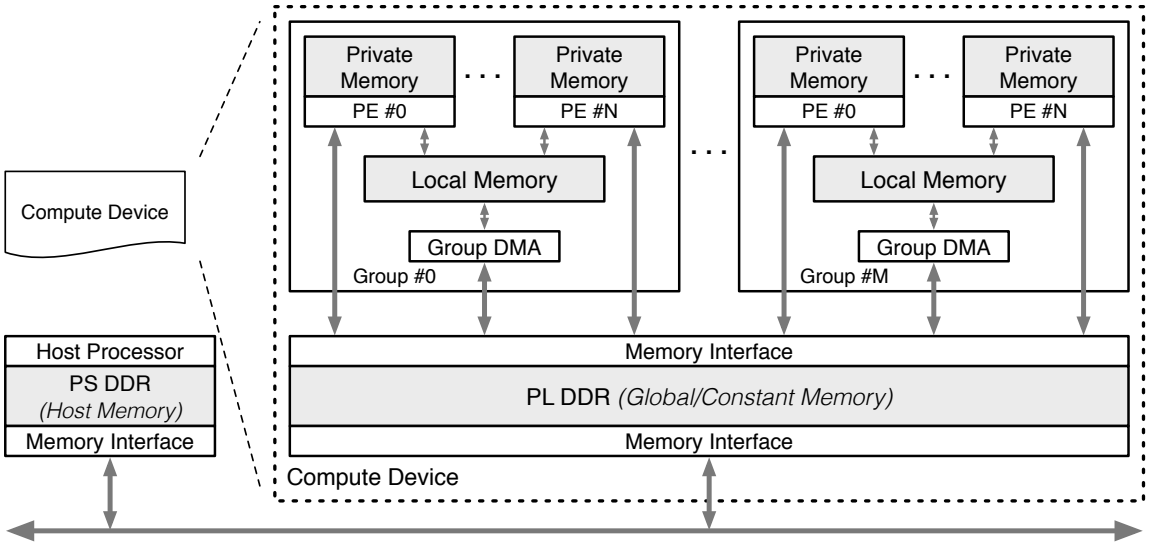

Figure 3.3: PolyPC Memory Hierarchy

memory and the global memory so that PEs are nearer to the datum that is ready to process than those in the global memory.

The global memory is connected to the host and the compute device through the AXI bus, as well as connections between memories and PEs. PS DDR and PL DDR memories are used to implement the host memory and the global/constant memory, respectively. Local memories use block RAMs (BRAMs), which are dedicated memory resources on FPGAs. Private memories are different per PE kind. For hPEs, the private memory is determined according to the kernel codes that define the size of the private memory. For sPE, the private memory uses the block RAM that can be only accessible by the general-purpose processor.

3.4 Bus Hierarchy

The PolyPC framework uses a hierarchical bus architecture based on the AXI memory-mapped (MM) bus. The AXI4 protocol provides the optional burst addressing and read and write operations works on multiple channels [106]. For example, reading data elements involves `Read Address` channel and `Read Data` channel. Under the non-burst mode, a read operation consists

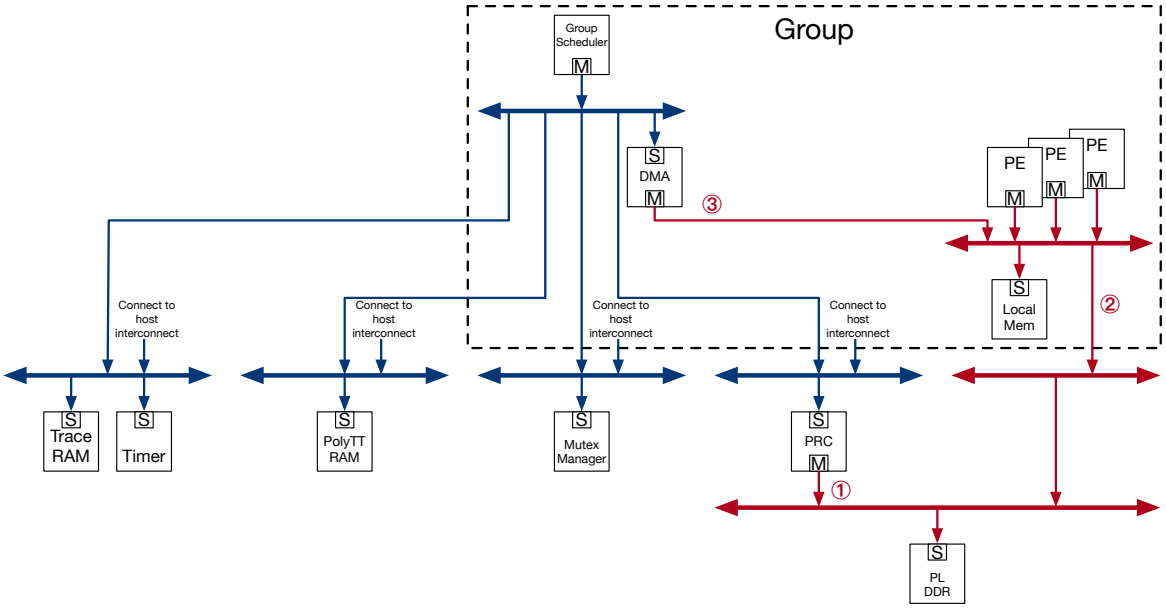

Figure 3.4: Bus Hierarchy of the Compute Device

of a read address request through the `Read Address` channel and a data read on the `Read Data` channel. If the size of data is larger than the bus length, the whole process will invoke multiple small read operations. The burst mode provides an efficient way for data transferring with consecutive addresses. Under the burst mode, a read operation consists of one read address request following by multiple data reads, which will significantly improve the bus bandwidth for a large amount of data.

The AXI MM bus is divided into two types: AXI-lite bus and AXI-full bus [9]. The former one does not have burst mode and usually is used for signal controlling. The latter one has burst mode and is designed for high throughput. In a practical design, the control flow bus architecture and data flow bus architecture are independent of each other to avoid interferences. As shown in Figure 3.4, the horizontal double-arrow lines represent AXI interconnects that have both slave and master ports. All memory-mapped IP modules have to communicate with others through AXI interconnects as either a slave device or a master device. The vertical single-arrow lines represent those connections from devices to interconnects.

The blue lines are the control path by using the AXI-lite protocol, and the red ones are data

flow by using the AXI-full protocol. Every peripheral such as PolyTT RAM and mutex manager has their own interconnect, each of which is connected to all groups and the host. Therefore, when different groups access different peripherals, the performance will not be affected since control signals go to different interconnect. The red lines are the data path by using the AXI-full protocol. Three data paths are demonstrated in Figure 3.4.

1. **PR Controller Data Path** The master port of the PR controller is used to fetch PR bitstream files from the PL DDR. When the PR controller is programming the PR region, the PR files are read to write to the ICAP module in burst mode.

2. **PE Data Path** Every PE has the data master port connect to the inner group data interconnect that is connected to the outer group data interconnect. Multiple groups can access the PL DDR through the same outer group data interconnect. The local memory is also attached to the inner group data interconnect. PEs operate on the local memory, which will not affect other groups to access data from the PL DDR. PEs can also use burst mode to fetch and update data from the PL DDR.

3. **Group DMA Data Path** The master port of the group DMA shares the same inner group data interconnect with PEs. Data elements between the PL DDR and the local memory are transferred in the burst mode.

Within each group, the group scheduler and PEs work on the group control interconnect and the inner group data interconnect, respectively. This design will leverage features of AXI-full and AXI-lite protocols to avoid the interference between control signals and the data transferring.

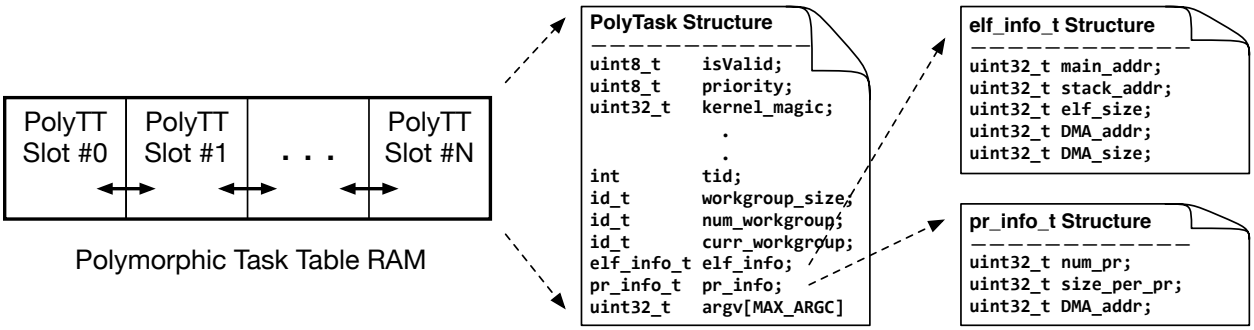

Figure 3.5: Structure of PolyTask Table RAM

3.5 Compute Device Architecture

3.5.1 PolyTask Table Queue

PolyTask table RAM is a doubly-linked list storing PolyTask tables in the block RAM. In host processor one user thread can create and launch one or more kernels that will run on PEs. A running kernel on compute device is called as a polymorphic task (PolyTask) in PolyPC context. Creating a PolyTask by a user thread is the equivalent process to create a PolyTask table in the PolyTask table RAM.

PolyTask table (PolyTT) records execution information to run a PolyTask and to switch between them. Figure 3.5 shows the detailed structure of the PolyTask table. The `priority` is set by users and is used for schedulers. The `tid` is copied from the thread ID from the user thread creating that particular PolyTask. The `kernel_magic` is defined manually. Together with `tid`, `kernel_magic` is a unique identifier to distinguish each PolyTask. `workgroup_size` (the size of each workgroup, i.e., the number of work-items within one workgroup) and `num_workgroup` (the number of workgroups totally) are kernel configurations that are the same with OpenCL. The `curr_workgroup` is the workgroup IDs indicating the latest workgroup that is running. The array `argv` stores arguments that pass to the kernel. `elf_info` and `pr_info` record the information of 'source codes' of a PolyTask in two forms: ELF executables and PR bitstream files. These two files are located in the FPGA DDR, which are initially loaded by user thread. Different from

Table 3.1: Accessibility of Registers on the Mutex Manager for a Mutex Lock

	ReqID	CurID	RelID
Execution Instances	W	R	WR
Mutex Manager	R	W	WR

`pr_info` that only stores the entry addresses and sizes of bitstream files for each hardware thread, `elf_info` stores the stack address additionally. To decrease the size of ELF files, sections that are allocated dynamically during run-time are eliminated with only labels remaining (e.g., head and stack sections). Therefore, the ELF file size and the DMA size are not the same and sections are loaded into FPGA DDR one by one according to its virtual memory address.

3.5.2 Mutex Manager

Mutex manager provides the underlying mutual exclusion mechanism across any execution instance across the whole system. It is a complement to the original OS mutex functions. In PolyPC, there are three kinds of execution instances: host programs on the host processor, scheduling programs on group schedulers, and PolyTasks on PEs. When acquiring the access of global resources (e.g., PolyTT Queue), threads within Linux can leverage OS-level mutex locks. Polymorphic tasks are not aware of these mutex locks at all. Instead of getting host processor involved to handle mutex requests from polymorphic tasks, a dedicated global mutex manager is set. Therefore, there are totally two-level mutex locks. For threads running on the host processor, firstly, they need to request the global mutex locks to exclude polymorphic tasks on PEs, and then request the OS-level mutex locks. Polymorphic tasks can request global mutex locks directly to exclude all the other threads.

Execution instances request and release mutex locks with the mutex manager through multiple 3-register bundles that are accessible to both execution instances and the mutex manager. One mutex lock occupies three registers to complete its functionality. Table 3.1 demonstrates the accessibility of three registers (i.e., `ReqID`, `CurID`, and `RelID`) from one mutex lock. To avoid

conflicts, one register may be read-only to execution instances and write-only to the mutex manager such as `ReqID`, and `CurID`. Since `RelID` cannot be read and written at the same by the mutex manager and execution instances, it is entirely accessible for both sides. To request and release a mutex lock, every execution instance should have one unique ID to let the mutex manager identify them. In the PolyPC framework, the host program can use the process ID (`PID`) from the Linux system while the group scheduler can use the group index that starts from 1. The algorithm of requesting a mutex lock from an execution instance is shown as:

1. Keep polling `CurID` to check if it is equal to its ID.

   If not, set `ReqID` with its ID and repeat this step (1); otherwise, jump to the next step (2).

2. Continue to run.

Possibly, there are multiple requests for one mutex lock by writing their IDs to `ReqID` during the same period. The mutex manager approves a random request by choosing its ID. Releasing the mutex lock is done by using only one register and is demonstrated as:

1. Keep `RelID` to check if it has been reset.

   If not, set `RelID` with its ID and repeat this step (1); otherwise, jump to the next step (2).

2. Continue to run.

Since one execution instance can only obtain one mutex lock at one time, the mutex manager monitors the `RelID` to check if the execution instance would release this mutex lock.

Figure 3.6 summaries how the mutex works for one mutex lock. The mutex manager keeps checking `CurID` to examine if this mutex lock has been obtained already. If `CurID` is equal to zero, which means this mutex lock is free to requests no matter there are requests or not, the mutex manager fills `CurID` with requesting ID from `ReqID` and resets `ReqID` in case this is the only request for this mutex lock. If `RelID` is not equal to zero, which means the execution instance wants to release this mutex lock, the mutex manager resets `RelID` to tell the execution

```
while true do
    if CurID == 0 then
        CurID = ReqID;
        ReqID = 0;
    end
    if RelID != 0 then
        RelID = 0;
        CurID = 0;
    end
end
```

Figure 3.6: Mutex Manager Algorithm for One Mutex Lock

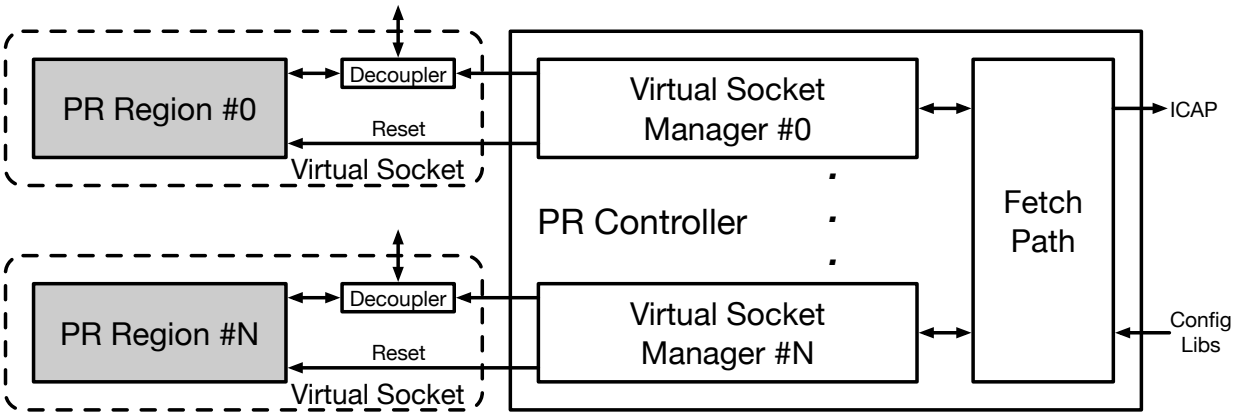

Figure 3.7: PolyTask Context Switching Demonstration

instance that the mutex lock has been released, and resets `CurID` to let the mutex manager know that current execution instance has released this mutex lock.

3.5.3 Partial Reconfiguration Controller

In the PolyPC framework, different hardware threads running on hPEs are switches by using the partial reconfiguration (PR) technique. Each loadable hardware thread exists as a hardware logic module and is placed within a PR region. To switch out a running hardware thread, the group scheduler requests the PR controller to load a PR bitstream file from the PL DDR into a particular PR region. For the group with multiple hPEs, the group scheduler issues multiple requests to the PR controller one at a time. The PR controller works with multiple virtual socket managers

(VSMs) simultaneously. Different group schedulers can issue requests at the same time. Since the PR controller uses the internal configuration access port (ICAP) IP module that can load only one bitstream file at a time, the PR controller queues requests from multiple VSMs and responses to them one by one.

As shown in Figure 3.7, the PR controller consists of VSMs that are connected to the fetch path. A VSM manages one particular virtual socket that refers to a PR region with static logics assisting the PR functionality. For example, the PolyPC framework uses decouplers to isolate PR regions with other static logics. In particular, during the period of loading one PR bitstream file into its corresponding PR region, the output wires of PR regions will be assigned with unexpected values that interfere with control logic on buses (e.g., `valid` signals on a master AXI Stream bus). This phenomenon may result in confusions of modules connecting to the PR region. For instance, if the PR region connects to a FIFO by using a master AXI Stream bus, the FIFO may be filled with garbage values during PR loading since the `valid` signal may be asserted for more than one clock cycle. Decouplers are comprised of multiple multiplexers of 2 inputs to isolate these control signals. The VSM uses a 1-bit select line to select correct values of decoupled signals. The PolyPC framework uses another `reset` signal connecting to the logic that is programmed on the PR region to reset the module in behavior level. To guarantee that the logic runs correctly after loaded into the PR region, each VSM works in the following three states:

1. **Initialize** The group scheduler assigns the memory address along with other information (e.g., size) of the PR bitstream file that is applicable on this virtual socket. The VSM waits for triggers or starts signals to load the PR bitstream file. At this state, the decouple and reset signals remain deasserted.

2. **Load** The VSM asserts the decouple signal to trigger the decoupler to work and notice the fetch patch to bring corresponding PR bitstream file from configuration library to the ICAP module. This state may last for a while depending on the size of the bitstream file and the bandwidth of buses connecting to the memory.

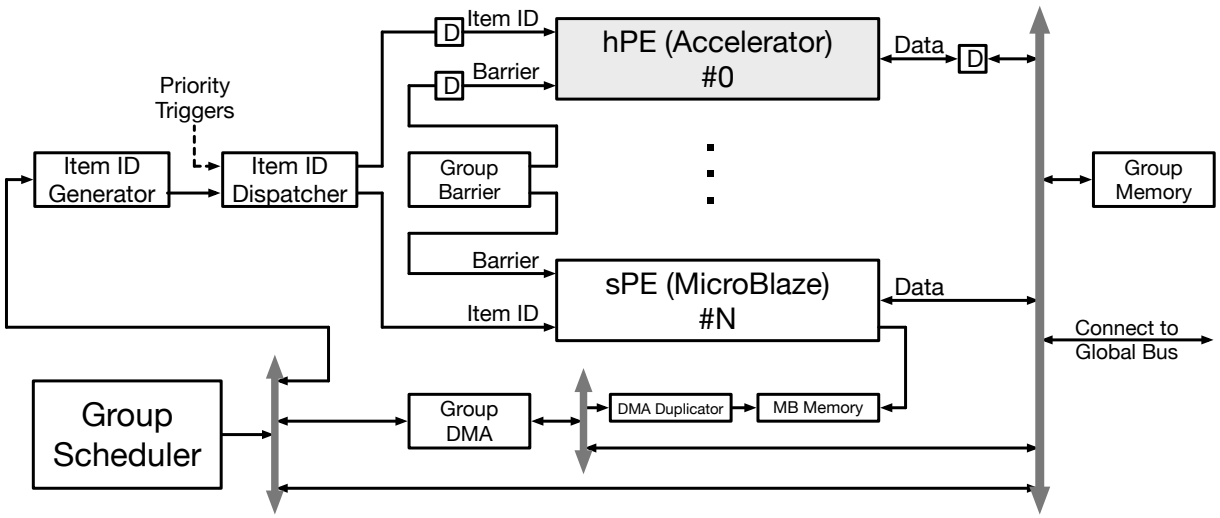

Figure 3.8: Group Architecture

3. **Reset** The VSM deasserts the decoupling signal and assert the reset signal. The duration of the reset signal is defined at the `Initialize` state. After certain clock cycles, the VSM deasserts the reset signal and go back to the `Initialize` step.

The PR controller uses the ICAP IP module to load PR bitstream files to corresponding PR regions. As discussed in section 3.1, the Zynq device has a PCAP module to load bitstream files. Before using ICAP, the PolyPC framework switches from PCAP to ICAP by clearing the `PCAP_PR` bit in the `DEVCFG CTRL` register. In the PolyPC framework, PR bitstream files of hardware threads are stored in the PL DDR memory along with creating the PolyTthread from the host. The `Config Libs` refers to the DDR memory that storing bitstream files. The location of PR bitstream files for each virtual socket are set during runtime by the group scheduler.

3.6 Group Architecture

Groups are computing resources in PolyPC. One group consists of the Group Interface (Group IF) and one or more Processing Elements (PEs). Figure 3.8 shows the overview of detailed group

architecture. Within one group, PEs may consist of hardware accelerators, MicroBlazes, or both of them. Hardware modules to run polymorphic tasks are PEs in two variants: hardware accelerators and general-purpose processor (i.e., MicroBlazes). In other words, one polymorphic task has two execution instances: ELF binaries for MicroBlazes and bitstream files for hardware accelerators. In Figure 3.8, the gray PE block is shown as a PR region. The hardware accelerators can be either placed into PR regions for dynamic scheduling or configrured as non-loadable for performance purposes.

3.6.1 Group Interface

The group IF Group IF has two major components: item ID generator and dispatcher, and the group scheduler. After receiving a trigger command (i.e., the size of a new work group) from the group scheduler, the item ID generator begins to generate item IDs. The item ID is generated one by one clock cycle. The item ID dispatcher will assign one ID to one of the PEs per clock cycle. The item ID generator and dispatcher and PEs are connected by using the AXI-Stream bus. Considering the performance of each PE is different from each other and achieving the best efficiency, `Ready` signals of item ID AXI stream buses from all PEs are wired to the item ID dispatcher as the priority selection input. In this way, one PE can request an ID as soon as it finishes the current thread.

Group Scheduler

The group scheduler decides what application and which work group will run on this hardware group. The group scheduler handles PolyTask context switching, as well. PolyPC supports run-time thread switching for both software and hardware threads. In other words, the new PolyThead applications are programmed off-line and added to the running system without terminating it. Unique PE architectures are designed to support this feature.

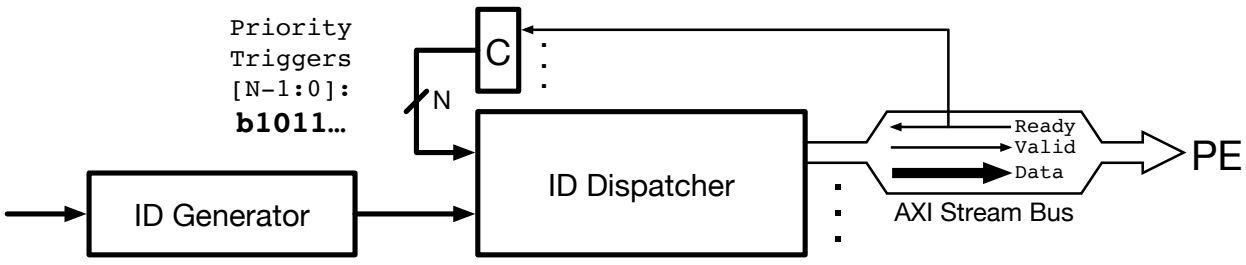

Figure 3.9: ID Generator and Dispatcher

ID Generator and Dispatcher

One ID generator and one ID dispatcher work together to provide item IDs to PEs. Item IDs refer to the global IDs within the problem space (see Figure 3.9). Once requesting a new work group, the group scheduler initializes the ID generator with two arguments: a 2-dimensional global offset of the first position of the new work group and the 2-dimensional size of the new work group. The ID generator then generates each global ID of the new work group one every one clock cycle. The generated ID is passed to the ID dispatcher through the AXI stream bus. In the PolyPC framework, one 2-dimensional ID is composed of two 16-bit width IDs. The total 32-bit wide ID can be delivered and received in one clock on a 32-bit wide AXI stream bus for high throughput. PEs need to know the end of current work group so that one special ID is reserved for the purpose (i.e., `0xFFFFFFFF`). Therefore, the maximum global problem space is $2^{16}-1$ by $2^{16}-1$. The reasons why the ID generator generates the global ID instead of the local ID is because we want to save the computation time for PEs to calculate global IDs. In the current design, for one work group, the group scheduler will help PEs to calculate the global ID of the first position of this work group for only one time. If the ID generator generates the local ID, the maximum problem space will expand to $2^{32}(2^{16}-1)$ by $2^{32}(2^{16}-1)$ from the ID generator side since the group size is 32-bit width. But in this case, every PE will calculate the global ID for every received item ID if necessary, which may waste computation resources.

The ID dispatcher dispatches every ID received from the ID generator to all PEs according to the runtime priority. Every PE uses the AXI stream bus to receive the ID and their `Ready`

signals are concatenated together to work as the priority selection for the ID dispatcher. Once a PE requests an item ID, the `Ready` signal of that AXI stream bus will be asserted, and therefore the corresponding bit within the priority triggers will also be asserted. As the example shown in Figure 3.9, the priority triggers are shown as `b1011...`, and three PEs are requesting IDs at this clock cycle. The more left the bit is; the higher priority the corresponding PE has. The leftmost bit has the top priority and in the next clock cycle, the triggers become `b0011...` once the ID is dispatched to that PE.

3.6.2 Processing Elements

hPE

An hPE can be either unchangeable and re-loadable. An unchangeable hPE is embedded into the system as soon as the system is assembled. The hPE works as a particular benchmark and is represented as an IP module and cannot be changed after the system is generated. A re-loadable hPE is represented as a PR region in the system. The re-loadable hPE can be programmed with any benchmark bitstream file as long as the PR region is large enough to hold that benchmark. Instead of determining which benchmark is programmed into the PR region during the system assembly, the benchmark is loaded into the system during runtime by the PolyTask loading.

sPE

sPEs are MicroBlaze processors in the PolyPC framework. Figure 3.10 demonstrates the sPE architecture. Under normal usage, one MicroBlaze only needs one local BRAM that is connected to the Data Local Memory Bus (DLMB) and Instruction Local Memory Bus(ILMB) as the Harvard architecture. A bootloader existing in the local BRAM performs the context switching of the software threads. In PolyPC system, an extra BRAM called kernel BRAM is added and connected to the DMA duplicator. Since every sPE can execute the same binary of a kernel

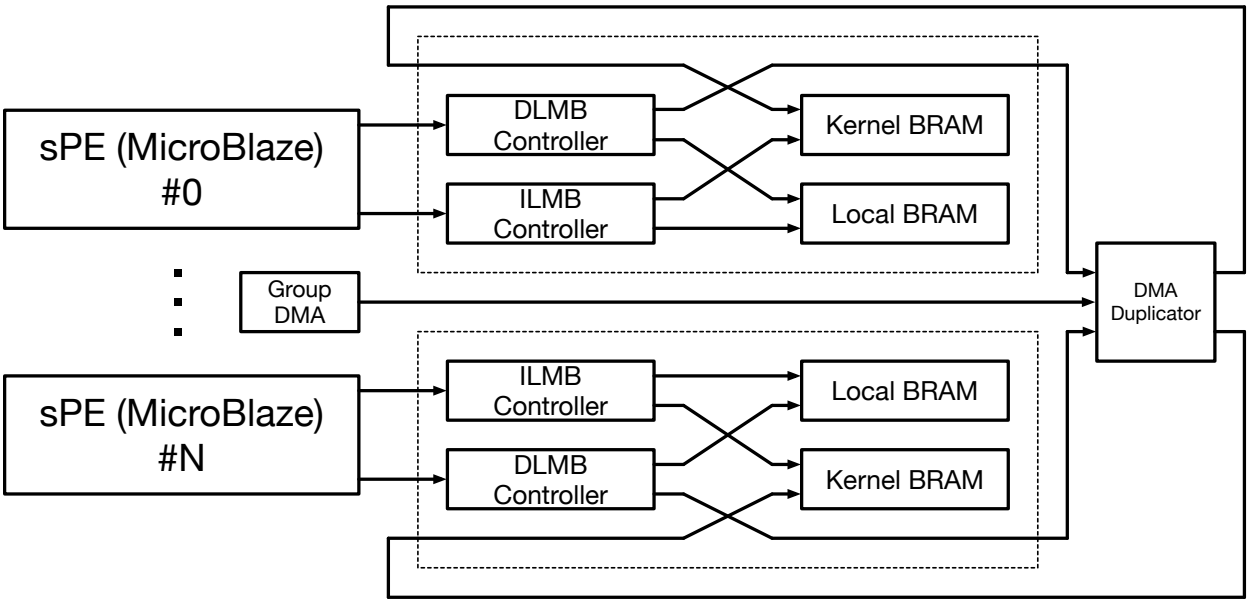

Figure 3.10: MicroBlaze Subsystem Architecture

program, the DMA duplicator is designed to avoid redundant DMA transfers. Within one group, once the group DMA issues an ELF executables transfer to the DMA duplicator, the connection between the DLMB and the kernel BRAM is disconnected by DMA duplicator automatically. The DMA duplicator then dispatches the same ELF executables to each kernel BRAM of the sPE at the same time.

3.6.3 Execution Sequence

Figure 3.11 explains how a group works when receiving a Group ID. As for each PolyTask application, the current group ID is stored in the PolyTask Table that shared by groups and host. The group scheduler has to request a mutex lock for the current group ID, as well as application information. After that, the group scheduler will check if its group has loaded corresponding elf executables and programmed bitstream files according to the type of applications. If not, the group scheduler will issues commands to Group DMA and one VSM of PR Controller to load and program the files, respectively. DMA transferring of ELF executables and bitstream file transferring can happen simultaneously because they use different DMA resources. PR Controller

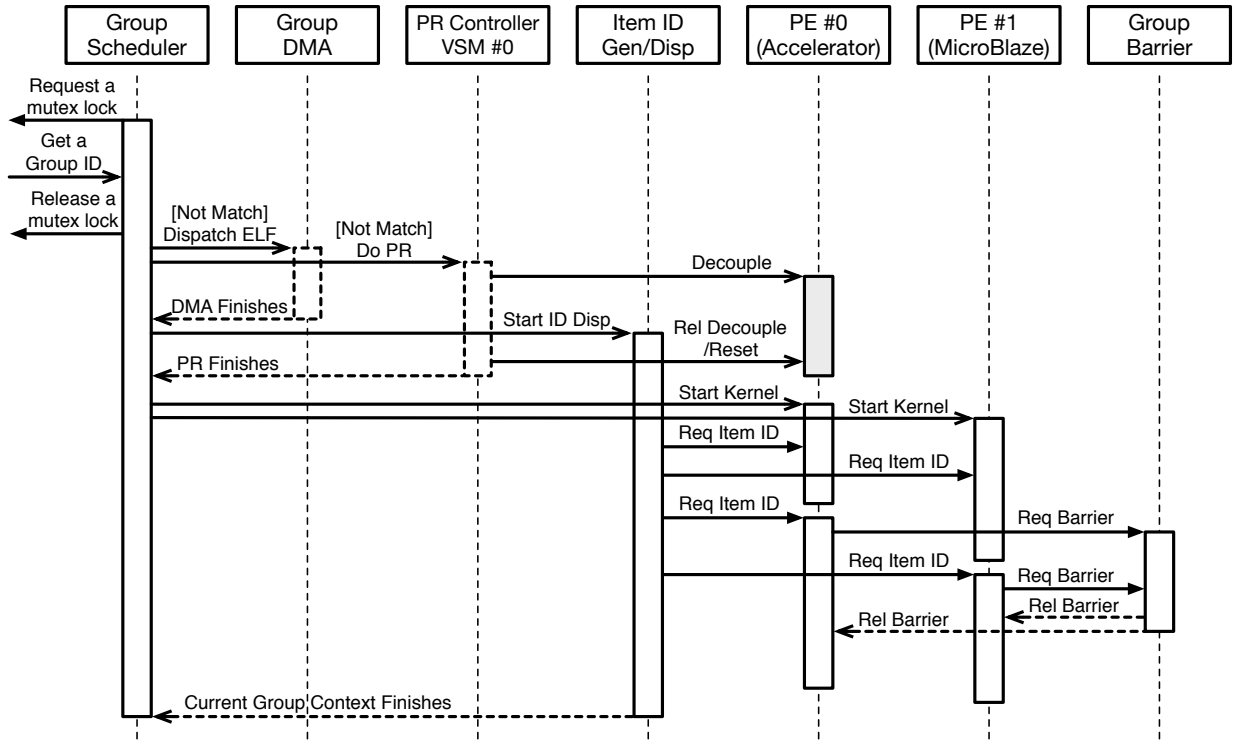

Figure 3.11: Group Execution Sequence

handles the decoupling. Commands to start generating and dispatching are also issued by the group scheduler, which may be overlapped by ELF and bitstream file transferring. This will save preparing time before the group does actual computation.

Once the ELF and bitstream file transferring are finished, the group scheduler notices hPEs and sPEs to start. The kernel threads are executed for one time by receiving an Item ID. During the execution, PEs may request a barrier for synchronization purposes. The barriers will not be released until the Group Barrier receives all barrier requests.

3.7 Trace Subsystem

The trace system is necessary for system evaluation as well as system debug. It records execution information of every work group within one PolyTask. The execution information includes the group index belonging to the hardware group that executes the work group, and three checkpoints

that are listed as follow:

1. After acquiring a work group, the group scheduler writes the first checkpoint before performing the PolyTask loading to the trace system.
2. After the PolyTask loading is finished, the group scheduler writes the second checkpoint before triggering PEs.
3. After the hardware group finishes the current work group, the group scheduler writes the last checkpoint.

The PolyTask is created along with allocating the trace memory for every work group. Therefore, after the PolyTask is finished, we can dump the trace memory to find out which hardware group executes the particular work group and how long the hardware group takes to complete the work group. By examining the checkpoints, we can break down the execution of each work group.

3.8 Evaluation and Results

In this section, we want to examine the potential of the system regarding acceleration performance. The PolyPC framework is configured with unchangeable hPEs. Therefore, there are no overheads of context switching within the system since only one benchmark will run on one system. sPEs are not configured in the system. The system resource utilization is also examined in this part.

Before performing the experiments, we define five kinds of time checkpoints to assist our evaluation. The first three checkpoints will append to every work group during the run-time as the tracing information.

- T_l: It is the time that is before the group scheduler loads ELF executables or bitstream files after acquiring a new work group.

- T_t: It is the time that is before the group scheduler triggers every PEs after the loadings are finished.
- T_f: It is the time that is before the group scheduler requests new work groups after all PEs finish current work group.

Another one is T_s, which is the time that the first work group begins to executed when a specific task or a task sequence is loaded into the system. It is defined as $T_s = min(T_l)$ for all work groups. The last one is T_e, which is the time that the last work group ends. It is defined as $T_e = max(T_f)$ for all work groups.

3.8.1 Benchmarks

We select four benchmarks representing frequently-used applications on reconfigurable systems regarding image processing, signal processing, and web searching. The last benchmark (Memory Operations) is used to evaluate the bandwidth of the system.

Vector Multiplication Two arrays, **A** and **B**, are as for input data. The output array **C** is calculated as $c_i = a_i \cdot b_i$ $(i = 0, ..., n-1)$, where n is the length of arrays **A**, **B**, and **C**.

Finite Impulse Response (FIR) For a discrete-time FIR filter of order N, each output value is calculated as the weighted sum of its most recent input values, which is denoted as $b_i = \sum_{j=0}^{N} a_{i+j} \cdot f_j$ $(i = 0, ..., n-1)$, where **a**, **b**, and **f** are input sequence, output sequence, and filter array, respectively, and n is length of input and output sequences. In our test, we use 4^{th} order (a.k.a 5-tap) FIR filter.

PageRank The PageRank algorithm is used to compute the relative importance of a web page by using the linking information between amount of web pages. The relative importance of N web pages (nodes) is denoted as a PageRank vector **v** with the length of N. For a k^{th} iteration

PageRank algorithm, the PageRank vector is denoted as $\mathbf{v_k} = \mathbf{A}^k\mathbf{v_{k-1}}$, where $\mathbf{v_0} = \{\frac{1}{N}, ..., \frac{1}{N}\}$. The N-by-N matrix $\mathbf{A}$ is a modified adjacency matrix, each element $a_{i,j}$ of which is the normalized importance passing from node j to node i. In our experiments, we test one iteration for this benchmark.

Matrix Multiplication Two squared two-dimensional matrices (**a** and **b**) are as for input data, and one two-dimensional matrix (**c**) is computed as results. The i^{th} raw and j^{th} column element of output matrix is calculated as $c_{i,j} = \sum_{k=0}^{length-1} a_{i,k} \cdot b_{k,j}$ $(i = 0, ..., length-1; j = 0, ..., length-1)$, where $length$ is the dimension of the matrices.

Memory Operations We use three separate benchmarks to evaluate the global memory bandwidth in terms of read-only, write-only, and read-and-write.

3.8.2 Experimental Setup

The PolyPC framework is compared with other two common computing resources on the ZC706 platform: the dual-core ARM processor and the general-purpose processors on FPGA fabric. Table 3.2 lists the detailed configurations of each hardware platform.

PolyPC Framework In this experiment, all PEs are set as hardware accelerators (hPEs). Four benchmarks, `Vector Multiplication`, `FIR`, `PageRank`, and `Matrix Multiplication` are evaluated, respectively. For each benchmark, the configurations of hardware platforms only differ from the number of hardware groups. In other words, to test one benchmark, we generate eight hardware platforms, where the number of hardware groups is from one to eight. The number of hPEs within one hardware group is four for every platform. The global memory uses the PL DDR memory that is connected to the FPGA fabric. The Memory Interface (MIG) module is used to connect the computing logics to PL DDR memory. The MIG is configured as 200MHz to achieve

Table 3.2: Hardware Platform Configurations

Hardware Platform	Items	Descriptions
PolyPC	# Hardware Groups	1, 2, 3, 4, 5, 6, 7, 8
	# PEs	4 within each Hardware Groups
	Memory Size	Global memory: 512 MB
		Local memory: 64 KB
		Private memory: n * 4 * 128 bytes, where n differs from benchmarks
	Memory Frequency	200MHz/64 bits
	PE Frequency	100MHz
ARM	# Cores	2
	Core Frequency	667MHz
	Memory Frequency	533MHz
MicroBlaze	Configuration	Default high performance with 32 KB I/D caches
	Core Frequency	100MHz
	Memory Frequency	200MHz

high memory bandwidth. The maximum sizes of private memory may not be the same, which depends on the types of benchmarks and how the benchmark is optimized. But no matter how many private buffers are, we want to keep each buffer store 128 elements, each of which is usually 4 bytes.

ARM The ZC706 has a hard-core dual-core ARM processor that is called programming system (PS). It runs at a higher frequency than the FPGA fabric and connects to the PS DDR memory that also has a much higher frequency. We want to use both of these cores to compare to the PolyPC framework.

MicroBlaze One MicroBlaze, along with its supporting components in the FPGA fabric is generated for comparison. To maximize the performance, the MicroBlaze is configured as the high-performance mode with 32 KB data and instruction caches, and the cacheline is the size of 16 words. The hardware multiplier and divider are also enabled.

Table 3.3: Benchmark Configurations

Benchmarks	Software(Hardware) Groups	Data Size (Word)	Buffer Size (Word)
Vector Multiplication	1(1), 2(2), 4(3), 4(4), 8(5), 8(6), 8(7), 8(8)	2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K	3 * (8, 16, 32, 64, 128)
FIR	1(1), 2(2), 4(3), 4(4), 8(5), 8(6), 8(7), 8(8)	2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K	2 * (8, 16, 32, 64, 128)
PageRank	1(1), 2(2), 4(3), 4(4), 8(5), 8(6), 8(7), 8(8)	32, 64, 128, 256, 512, 1024	3 * (8, 16, 32, 64, 128)
Matrix Multiplication	1(1), 2(2), 4(3), 4(4), 8(5), 8(6), 8(7), 8(8)	64*64, 128*128, 256*256, 512*512, 1024*1024	3 * 16 * 16

Table 3.3 shows the detailed setup for each benchmark. The number of work groups may not be the same with the number of hardware groups. To reduce the overhead of scheduling when the group scheduler changes between different work groups, for each hardware platform the number of work groups is set as close as the number of its hardware groups, which is the power of 2. The data size describes the output data (i.e., results) without considering input data sizes. For example, The output results of `PageRank` is a vector with sizes from 32 to 1024. However, the input data is one matrix with sizes from 32-by-32 to 1024-by-1024 and a vector with sizes also from 32 to 1024.

In OpenCL-like parallel programming models, optimizing the data locality is one of the most common ways to improve performance. In our experiments, we use private memories to create private buffers working as the closest memories to PEs. Different benchmarks may have different number of buffers with various siezes. For example, `Vector Multiplication` has two input vectors and one output vector. We use three buffers to temporarily store part of these three vectors, respectively. For `Matrix Multiplication`, each of three matrices (two input matrices and one output matrix) has one 16-by-16 buffer. The maximum buffer size is determined by the hardware kernel programs that cannot be changed after the system is generated. But we can still use an arbitrary buffer size that is equal to or smaller than the maximum one from the host programs.

Table 3.4: System Resource Utilization

Items	LUTs (%)	Registers (%)	BRAMs (%)	DSPs (%)
1 Group	32472 (14.90)	33952 (7.77)	62.5 (11.47)	21 (2.33)
2 Groups	47045 (21.52)	51013 (11.67)	84.5 (15.50)	42 (4.67)
3 Groups	61511 (28.14)	67953 (15.54)	106.5 (19.54)	63 (7.00)
4 Groups	75869 (34.71)	84870 (19.41)	128.5 (23.58)	84 (9.33)
5 Groups	90095 (41.21)	101775 (23.28)	150.5 (27.61)	105 (11.67)
6 Groups	104531 (47.82)	118680 (27.15)	172.5 (31.65)	126 (14.00)
7 Groups	118803 (54.35)	135603 (31.02)	194.5 (35.69)	147 (16.33)
8 Groups	133655 (61.14)	152583 (34.90)	216.5 (39.72)	168 (18.67)
HP MicroBlaze	15742 (7.20)	13354 (3.05)	27.5 (5.04)	6 (0.67)

Table 3.5: hPEs Resource Utilization

Benchmark	PrivateMem (Bytes)	LUTs (%)	Registers (%)	BRAMs (%)	DSPs (%)
`FIR`	512 * 2	1213 (0.55)	1596 (0.37)	1.5 (0.28)	2 (0.22)
	0	966 (0.44)	1359 (0.13)	0 (0)	0 (0)
`Matrix Multiplication`	64 * 64 * 3	1348 (0.62)	2233 (0.51)	1.5 (0.28)	7 (0.78)
	0	1160 (0.53)	1707 (0.39)	0 (0)	5 (0.56)
`PageRank`	512 * 3	2010 (0.92)	2961 (0.68)	1.5 (0.28)	7 (0.78)
	0	1343 (0.61)	1936 (0.44)	0 (0)	7 (0.78)
`Vector Multiplication`	512 * 3	1145 (0.52)	1730 (0.40)	1.5 (0.28)	5 (0.56)
	0	877 (0.40)	1436 (0.33)	0 (0)	3 (0.33)

3.8.3 Resource Utilization

As for each hardware platform with the specific number of groups, it has multiple versions with different kinds of benchmarks. We average the resource utilization for each hardware platform and show the results in Table 3.4. The resource utilization of a complete high-performance MicroBlaze is also listed. To compute tasks with large input and output data sizes, the extern DDR memory has to be added, which makes it use more resources than those of a typical MicroBlaze system.

Figure 3.5 shows the resource utilization of each type of hPE. Evey type of hPE has two

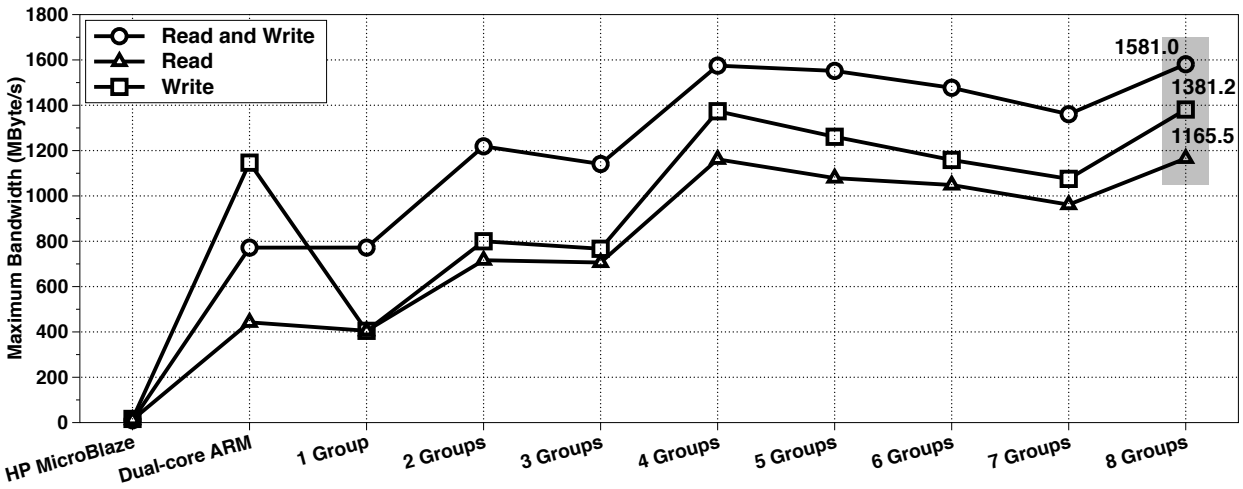

Figure 3.12: Memory Bandwidth

versions: one with private buffers; the other with no buffers. Generally, hPEs with no buffers utilize fewer resources than those with private buffers in terms of every type of hardware resources. Of all four type of benchmarks, `PageRank` consumes the most hardware resources due to its floating point computation. Of the other three integer-based benchmarks, `Matrix Multiplication` does the most complex computation and therefore utilize more resources.

3.8.4 Performance Analysis

As shown in Figure 3.12, we test the memory bandwidth of three types of hardware platforms in Table 3.2. The number of hardware groups significantly affects the memory bandwidth in PolyPC framework. As the number of hardware groups increases, the memory bandwidth of three operations (i.e., read, write, and read/write) roughly increases and achieve the maximum bandwidth when the number of hardware groups is eight. It is noticed that the bandwidth drops compared to the previous one when the numbers of hardware groups are 3, 5, 6, and 7. It is because the mismatch between the number of hardware groups and work groups affect the efficiency of the system. For example, when an application with eight work groups runs on a system with seven hardware groups, after the first seven work groups are finished, there is a high possibility that the last one work group occupies only one hardware group while other six

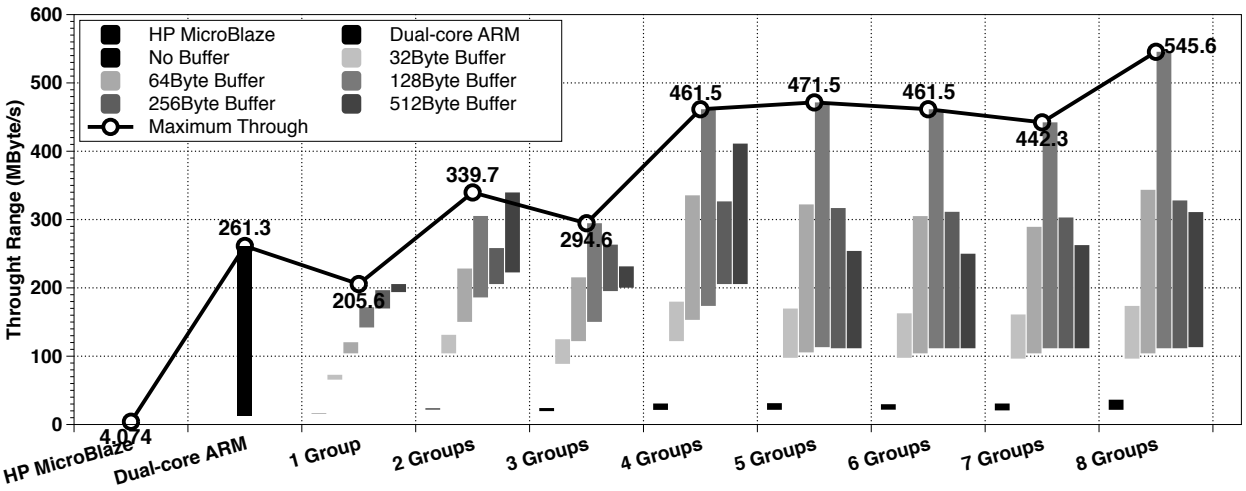

Figure 3.13: Performance Range of Vector Multiplication

hardware groups are idle. This will decrease the efficiency of the system while resulting in a lower bandwidth. We will elaborate this phenomenon in the following experiments. The reason why 1-group and 2-group platforms have lower bandwidth than others is that the total number of hPEs of the two platforms cannot fully utilize the memory bandwidth.

Benefiting from the advanced micro-architecture and higher frequency, the ARM processor achieves a relatively high bandwidth comparing to the PolyPC framework. Due to the cache policy, write bandwidth is higher than the other two operations. Even the frequency of memory interface is the same as the PolyPC's. The high-performance MicroBlaze has no advantages regarding memory bandwidth. In summary, the maximum read/write bandwidth is about $2\times$ and $155\times$ over a dual-core ARM processor and a high-performance MicroBlaze, respectively. The data cache can help general-purpose processors (i.e., ARM processor and HP MicroBlaze) to improve the utilization of AXI buses since the cacheline fetching is under AXI bursting mode. However, to complete a particular task, the general-purpose processors are usually inefficient comparing to hardware accelerators. Both the computation capability and memory bandwidth decide the upper bound of the throughput.

Figure 3.13, 3.14, 3.15, and 3.16 demonstrates the performance of benchmark `Vector Multiplication`, `FIR`, `PageRank`, and `Matrix Multiplication`, respectively. Here the

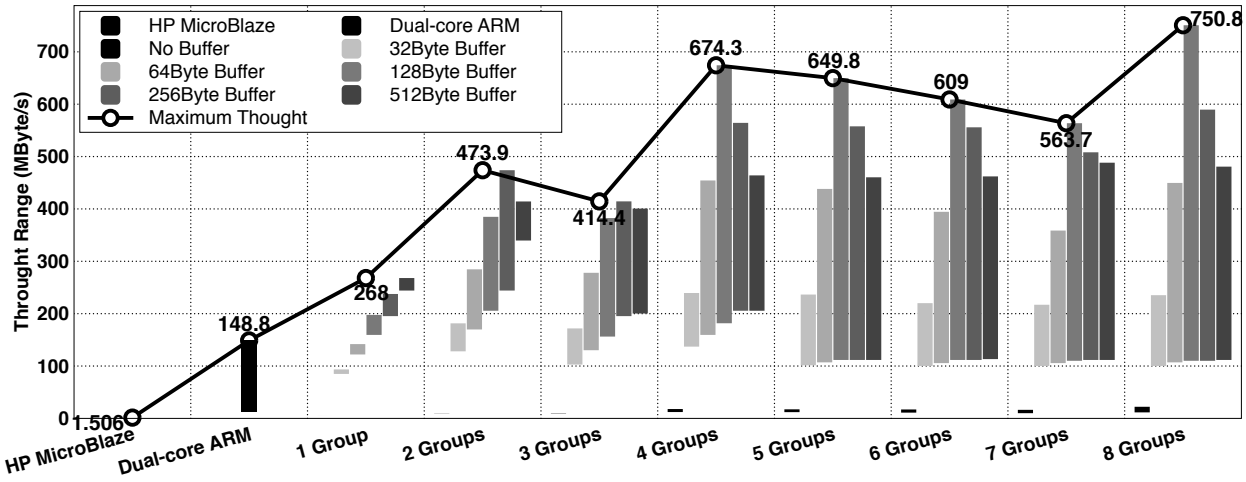

Figure 3.14: Performance Range of FIR

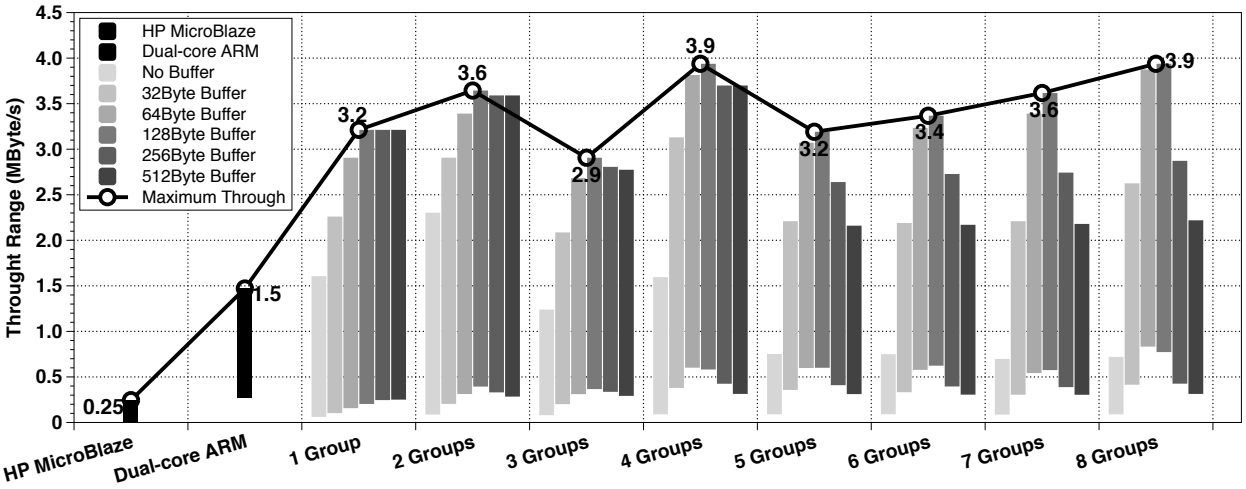

Figure 3.15: Performance Range of PageRank

throughput is defined as how many output data it can process per second. For example, if we say the throughput of `Vector Multiplication` is 500 MB/s, we mean that it can deliver 500 MB output vector per second. For each figure, we measure the lowest throughput and the highest throughput for benchmarks running on a high-performance MicroBlaze, dual-core ARM processor, and PolyPC framework with groups ranging from 1 to 8. For the PolyPC part, we also the range of throughput for each buffer size. The line with the hollow dot marks the highest throughput for each configuration.

From the maximum throughput line for the PolyPC part, we can notice that similar to that of

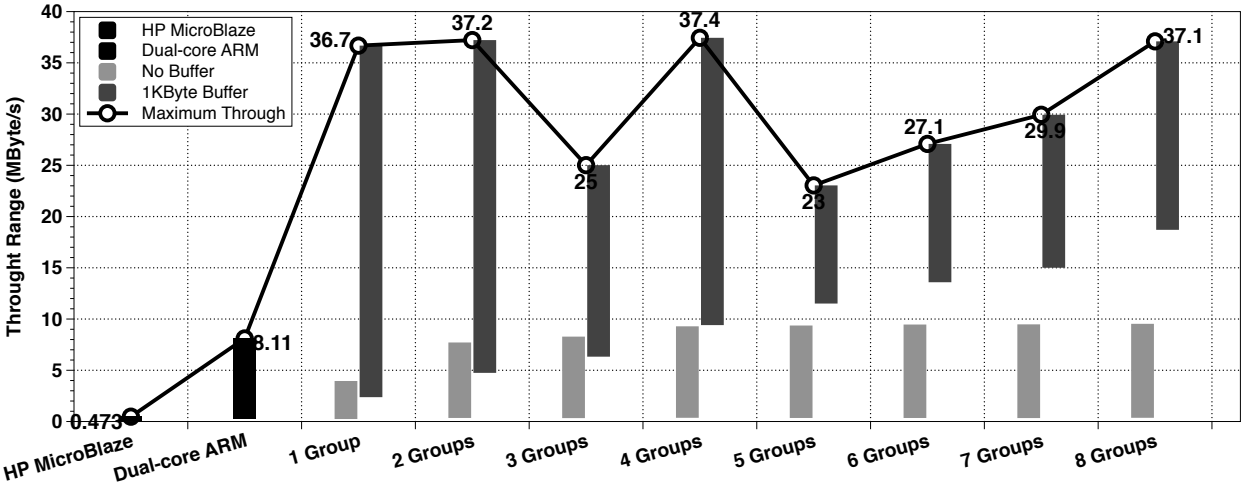

Figure 3.16: Performance Range of Matrix Multiplication

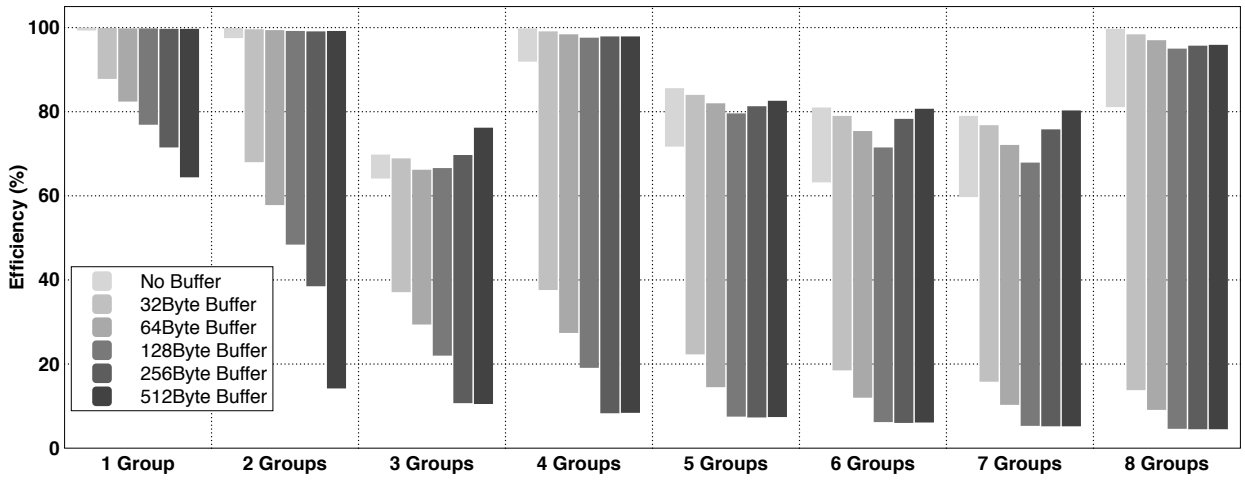

Figure 3.17: System Efficiency Range of FIR

the memory bandwidth (Figure 3.12), the throughput drops when the number of groups is 3, 5, 6, and 7. It is due to the same reason that the number of hardware groups does not match with the number of work groups. To examine this result, we further define another metric called `efficiency`, which is calculated as:

$$efficiency = \frac{\sum_{i=1}^{n} T_{ei}}{T_w * N}$$

where $T_{ei} = T_{ti} - T_{li}$ is the effective computation time of the i^{th} work group, $T_w = T_e - T_s$ is the wall clock time of this task, and N is the number of hardware groups. T_e of each work group is the

time slice between triggering the group and the end of its finishing. Figure 3.17 shows the efficiency of `FIR`. Other benchmarks have the similar results so that we only use `FIR` to demonstrate this result. The maximum efficiencies when the number of groups is 1, 2, 4, and 8 can reach almost 100%. Since as for every buffer size, different data sizes are tested, the efficiency for small data sizes may be low due to the warm-up overheads. When the number of groups is 3, 5, 6, and 7, the maximum efficiencies are from around 70% to 80%. For instance, when 4 work groups running on 3 hardware groups, after the last round of scheduling, only one hardware group is busy running the last work group, and the other 2 hardware groups are idle. We can also notice that when 8 work groups run on 5, 6, and 7 hardware groups, the efficiency decreases as the number of hardware groups increases. This is because that higher ratio of the number of work groups to the number of hardware groups results in higher efficiency.

However, higher efficiency does not inevitably result in higher throughput if we compare Figure 3.13 and Figure 3.17. Instead, more hardware groups usually deliver better throughput. It is because that before reaching the maximum memory bandwidth, more computation resources result in more memory accesses, which increases bus throughputs. Since calculating one output element requires much more computations for benchmark `PageRank` and `Matrix Multiplication` than other benchmarks, they reach the maximum throughputs by using fewer hardware groups. Another important factor that can significantly affect performance is the size of buffers. For these four benchmarks without private buffers, the performance is very low and cannot even beat a dual-core ARM processor. But even if we enable small size buffers, the performance improvement is significant. For the first three benchmarks, as the size increases, the performance reaches its peak when the buffer size is 128 bytes. It drops when the buffer size keeps growing. The lower bound of the throughput results from the small data sizes. In PolyPC framework, how private buffers are used to improve performance can be explained in two aspects: one is to reduce reading redundant data (i.e., `PageRank` and `Matrix Multiplication`); the other is to improve the utilization of the AXI bus by using AXI bursting mode.

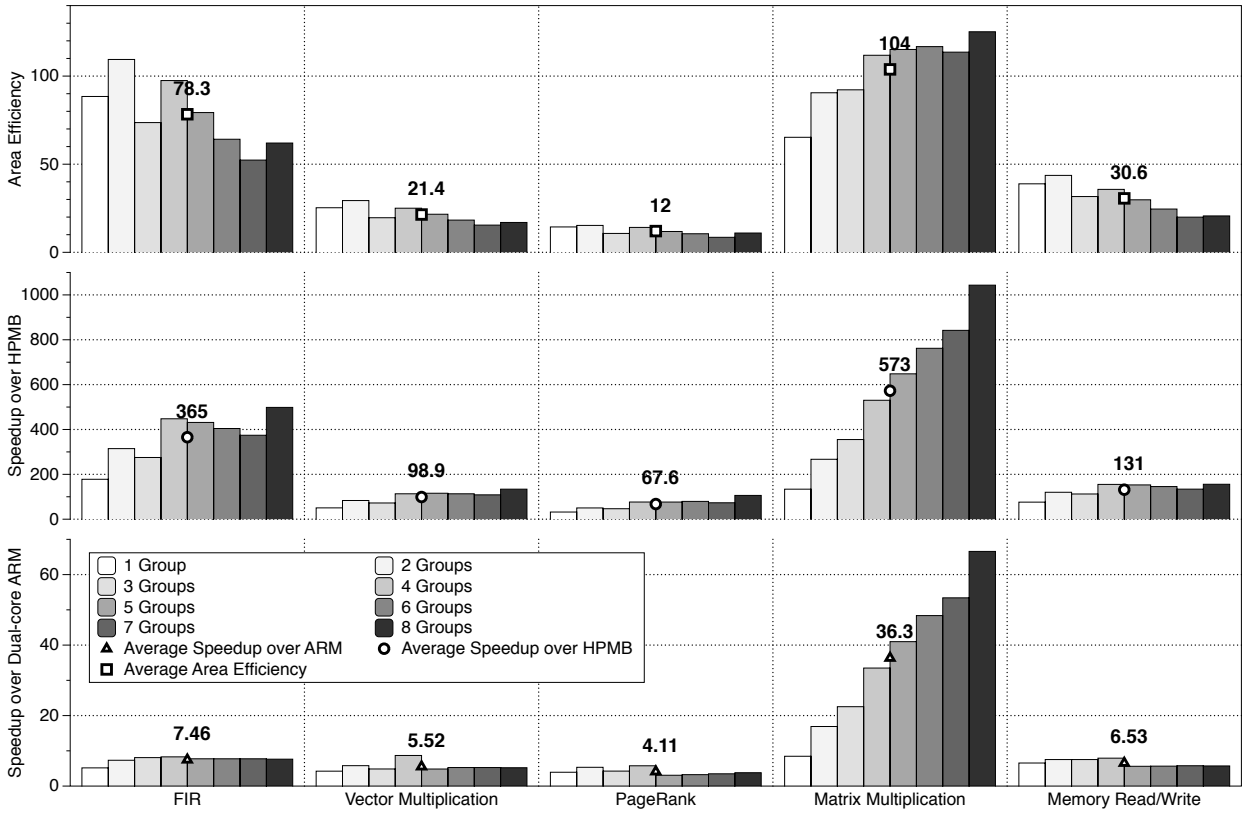

Figure 3.18: Speedup and Area Efficiency over ARM and HP MicroBlaze

Figure 3.18 summaries the speedup of PolyPC framework over the dual-core ARM processor and a high-performance MicroBlaze. The speedup is calculated by comparing the same data size, which may make it look different from previous performance figures. The maximum speedup happens from `Matrix Multiplication`, which is about 66× for ARM processor and 1050× for MicroBlaze. The minimum speedup happens from `PageRank`, which is as low as 3× for ARM processor and 30× for MicroBlaze. The more calculation it needs for one output element, the more speedup PolyPC can get over processors, such as `Matrix Multiplication`. However, it does not happen on `PageRank` that also needs a lot of calculation. It may be due to that `PageRank` calculates on floating point data, at which the FPGA may not be very good. The high-performance MicroBlaze utilize hard DPS cores on FPGA to calculate, which makes it efficient on `PageRank`.

We use the metric of `area efficiency` to measure how effectively the PolyPC framework

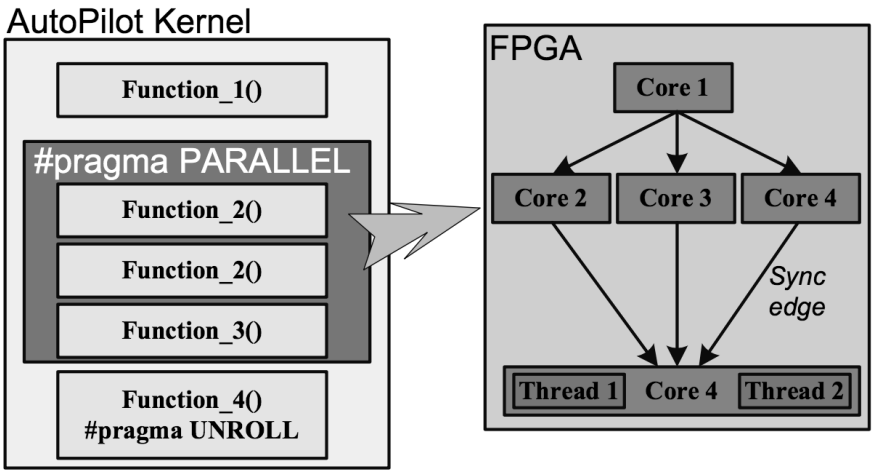

Figure 3.19: AutoPilot C Programming Model from [85]

uses hardware resources. The `area efficiency` is defined as how much speedup the PolyPC can get by using the same amount of hardware resources (i.e., LUTs) comparing to the high-performance MicroBlaze system. The results are shown in Figure 3.18. `Matrix Multiplication` has the best area efficiency; while `PageRank` has the worst one. Usually, computation-intensive benchmarks run more efficiently on PolyPC framework than on a MicroBlaze. However, since FPGAs are not very good at floating point computation, the `PageRank` does not gain too much speedup by using the dedicated hardware accelerators.

3.9 Related Work

3.9.1 HLS-based Systems

There are several research projects and commercial solutions that take OpenCL applications to generate binaries on FPGAs by using HLS approaches.

FCUDA [85] is an FPGA design flow that adapts the CUDA programming model to map the coarse and fine grained parallelism onto the reconfigurable fabric. FCUDA employs AutoPilot [20], an advanced HLS tool, and is highly based on AutoPilot's pragma. Besides AutoPilot, FCUDA builds on a code transformation process, which is divided into three steps.

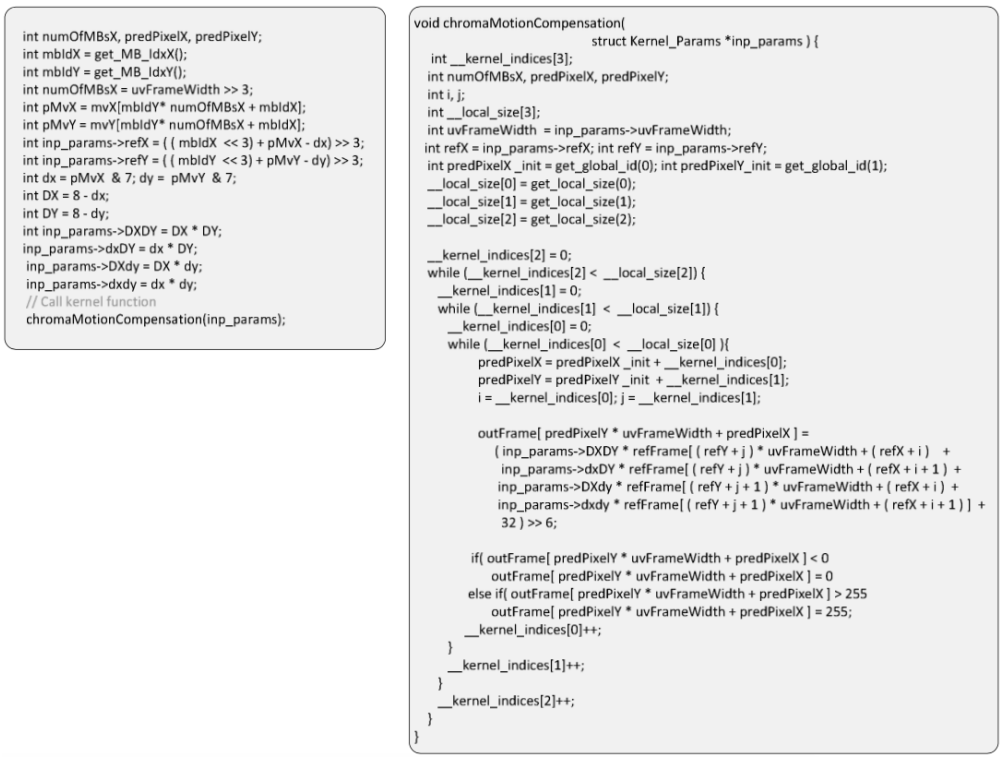

Figure 3.20: SOpenCL Front-end Code Transformation from [83]

Firstly, FCUDA directives are inserted into the CUDA kernel code to generate annotated codes. Then the codes are generated into RTL logics through coarse-grained and fine-grained parallelism extraction. The previous step is expressed explicitly from multiple function calls. These functions are synthesized into FPGA cores via the following step by using AutoPilot (see Figure 3.19). Their subsequent work multilevel granularity parallelism synthesis (ML-GPS) framework [86] employs a space search heuristic to exploit different granularities of parallelism for mapping CUDA kernel onto FPGAs.

Silicon-OpenCL (SOpenCL) [83] is based on the front-end source-to-source transformation and the back-end template-based hardware generation. The front-end transformation includes logical thread serialization that coarsens the OpenCL kernel codes into a series of nested loops and elimination of barriers and variable privatization that transforms OpenCL codes with barriers into fissioned loops [22] (see Figure 3.20). Transformed codes with nested loops are generated into hardware accelerators using a back-end compiler that extends LLVM. Instead of generating

arbitrary logics, SOpenCL puts logics into a hardware accelerator template.

UT-OCL [77] provides an architecture support of OpenCL programming model on embedded reconfigurable systems. The device subsystem comprises several support modules that are implemented using MicroBlazes and compute device that can be generated by HLS or the software compiler as MicroBlazes.

Besides academic works, Xilinx and Intel have launched their solutions based on OpenCL programming model. In Intel's approach [94], OpenCL programs are transformed into dedicated and pipelined hardware circuits that are called PEs on FPGAs by using their SDK tools including the HLS tool. Parallelism is expressed by duplicating PEs. Xilinx's solution, SDAccel [105], is highly relied on their HLS tools. Similar to academic works, the front-end step finishes the source-to-source transformation and the back-end synthesized into hardware logics. SDAccel supports additional complication for new kernel programs without re-generating the whole system. The coarse-grained and fine-grained parallelism is expressed by generated nested-loop codes and its unrolling with directives, which is a combination of FCUDA and SOpenCL. As an example, the following is the HLS-compatible kernel codes with work group size attribute.

```
__kernel __attribute__((reqd_work_group_size(4,4,1)))
void mmult32(global int *A,global int *B,global int *C)
{
  . . .
}
```

This fragment of codes is transformed into nested-loop codes to achieve coarse-grained parallelism.

```
__kernel void mmult32(global int* A,global int* B,global int* C)
{
localid_t id;
int B_local[16*16];
  for(id[2]=0;id[2]<1;id[2]++)
    for(id[1]=0;id[1]<4;id[1]++)
      for(id[0]=0;id[0]<4;id[0]++){
        . . .
      }
    }
  }
}
```

Compared with previous works and industrial solutions, our work is based on a two-step

approach as well: source-to-source transformation and hardware generation with HLS tools. The PolyPC framework requires designers to write PolyPC-specific OpenCL kernel codes, which is similar to write annotate codes in FCUDA. A single PE is generated with a kernel program through the second step. Different from FCUDA, SOpenCL, and SDAccel, the coarse-grained and fine-grained parallelism is expressed through architectural supports in the PolyPC framework instead of explicit expressions within kernel codes. For example, fine-grained parallelism is achieved by unrolling `for` loops to duplicate PEs in SDAccel or parallel function calls in FCUDA. But in our work, PE duplication is independent of source codes and defined as platform specifications. Cooperating with group schedulers and ID dispatchers, the PolyPC framework does not require explicit architecture information (e.g., work group size) within kernel source codes. Besides, PolyPC defines the memory model from OpenCL standard that is missing in previous work (e.g., SOpenCL). A separate barrier module is implemented in our framework so that no further loop fission is needed via source transformation in SOpenCL and SDAccel. In summary, the PolyPC framework provides more conveniences for designers to write kernel programs than previous works through custom architectural design.

3.9.2 Processor Substrate

OpenRCL [68] framework builds on both LLVM compiler infrastructure and OpenCL programming model to employ on a low-power and high-performance many-core-style processor architecture. Coarse-grained parallelism is expressed on an array of PEs that are implemented as many-core processors. The fine-grained parallelism is expressed on a thin, n-stage, n-way multi-threaded simplified MIPS processor that can be extended with custom instructions as well. Other works [2, 28, 67] are based on existing general-purpose processors such as MicroBlaze and LEON. PEs are implemented as processors. The system architecture is highly customized to execute OpenCL programs.

FGPU [3] and FlexGrip [8] focuses on soft-GPU implementation on FPGAs. Different from

HLS-based and general-purpose processor approaches, dedicated soft-GPU designs can have more common architecture features with a commercial GPU but remain abilities to customize system architecture. As an example, FGPU keeps the principle of a commercial GPU architecture but extends the memory subsystem to better fit into FPGAs.

Processor-based GPU solutions for OpenCL programming model have advantages on productivity but may lose performance compared to HLS-based approaches. In general, kernel codes for processor-based GPU do not require source-to-source transformation to generate proper C codes for HLS tools. Besides, architectures can be more flexible. For instance, both OpenRCL and FGPU support the OpenCL memory model with better memory hierarchy and FGPU further designs cache systems for memory accesses from PEs. Heterogeneous architecture design in the PolyPC framework leverage advantages from both HLS-based and processor-based solutions to achieve high performance and flexible architecture design.

Chapter 4

Polymorphic Multitasking

4.1 Polymorphic Tasks

The polymorphic task (PolyTask) is created, launched, and committed by the host program and can only run on the compute device. A PolyTask is associated with a kernel program and consists of multiple threads. A thread associated with a work-item within the kernel program is executed on a PE. As a PE can be either an sPE or an hPE, a thread is called as a software thread or a hardware thread when it is executed on an sPE or an hPE, respectively.

4.1.1 Hardware Threads

A hardware thread executes a work-item on a unchangeable or re-loadable hPE. The hardware thread is initialized once the hPE receives an item ID from the ID dispatcher. Hardware threads retrieve thread arguments (e.g., addresses of input and output data) from the local memory that are passed by the group scheduler in advance. For re-loadable hPEs, they are associated with executables. As hPEs are located on PR regions, an executable (i.e. a PR bitstream file) is loaded into corresponding PR region before hardware thread starts. A running hardware thread is a loaded and powered executable on an hPE associated with a PR region.

4.1.2 Software Threads

A software thread executes a work-item on an sPE. As an sPE is a MicroBlaze in the PolyPC framework, a running software thread is a running program on the MicroBlaze. A software thread is associated with thread arguments, as well as executables. Executables on sPEs are ELF files

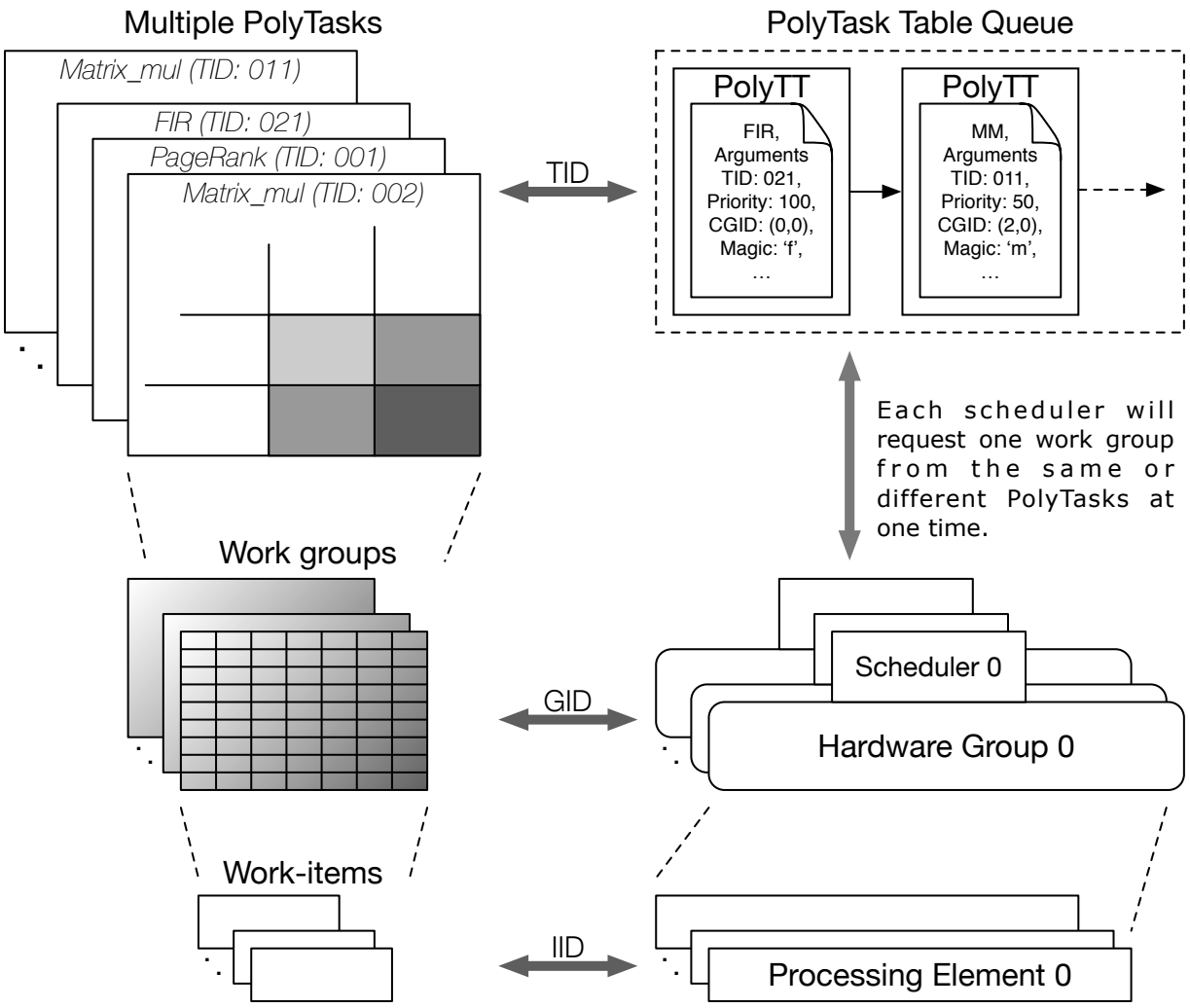

Figure 4.1: PolyTask Mapping with Hardware Resources

compatible with MicroBlazes. Different from hardware threads, a software thread is initialized by the bootloader running on the sPE. The bootloader located on a local BRAM is a program to monitor triggers from the group scheduler and to switch between it and software threads. Before a software thread starts, a corresponding ELF file is loaded into a kernel BRAM.

4.2 Polymorphic Scheduling

Figure 4.1 demonstrates how PolyTasks are mapped on hardware resources, as well as work groups and work-items. A PolyTT in the PolyTT queue is one-to-one mapped to a PolyTask that

is created from the host processor. Their unique PolyTask IDs (TIDs) are used to identify themselves, and the magic number is used to distinguish different types of kernel programs. For one particular PolyTask, the problem space is divided into multiple two-dimensional work groups. All work groups belonging to a PolyTask are of the same size. Work groups from all launched PolyTasks are assigned to hardware groups. In the PolyPC framework, a group-level multitasking is designed. Work groups in a PolyTask are the minimal schedulable units. In other words, a hardware group executing a PolyTask cannot switch out current work group to change in a new group unless the current work finishes execution. The new work group belongs to either the current PolyTask or other PolyTasks. In this way, PolyTasks schedule on hardware groups. Each work group has its group ID (GID) within one PolyTask. Since GIDs are generated by the ID generator and dispatcher in orders, the execution progress can be recorded by using the GID. Then each work group is divided into multiple work-items, which are the minimum processable elements physically deployed on PEs.

The `curr_workgroup` (see Figure 3.5) from the PolyTT queue indicates the next available work group of that PolyTask for execution. The `curr_workgroup` along with other information with the corresponding PolyTT slot is the running context of the PolyTask. The polymorphic multitasking is based on the group-level scheduling and extends to support priority-aware scheduling and PolyTasks. The polymorphic multitasking supports scheduling of PolyTasks and therefore, performs on groups comprised of either sPEs or hPEs.

Figure 4.2 demonstrates the pseudocode of how the group scheduler works. The flow is logically divided into four steps within an infinite loop. They are `scheduling`, `Updating context`, `PolyTask loading`, and `Triggering to finish`. A complete polymorphic multitasking is also comprised of the four steps.

1. **Scheduling:** Before accessing the PolyTT queue, the group scheduler requests a mutex lock. Then the group scheduler will go through every valid slot of this queue. The `searchPolyTTQueue()` function is to choose a new work group from all available

```
isMatch = false;
while true do
    isFound = false;
    requestMutex(groupIndex);
    isFound, isMatch = searchPolyTTQueue();
    if not isFound then
        releaseMutex(groupIndex);
        continue;
    end
    increaseCurrentGroupID();
    releaseMutex(groupIndex);
    passArg();
    if not isMatch then
        if existMB then
            doELFLoading();
        end
        if existPR then
            doPR();
        end
    end
    triggerPEs();
    waittoFinish();
end
```

Figure 4.2: Group Scheduler Pseudocode

PolyTasks within the PolyTT queue. Different algorithms can be implemented in this function. The function will return a PolyTask (i.e. PolyTask table) if there is an available work group belonging to that PolyTask and variable `isFound` is asserted; otherwise, the scheduler will release the mutex lock as other group schedulers obtain changes to access the PolyTT queue. As an example of the priority-aware scheduling, the `searchPolyTTQueue()` function returns a PolyTask with the highest priority or a PolyTask with the highest priority and the same magic number. If the latter one is found, the scheduler prefers the latter one; otherwise, the scheduler chooses the former one. When the magic number of the new PolyTask does not match with the current one, variable `isMatch` is set as `false`; otherwise it is `true`.

2. **Updating context:** If the group scheduler obtains a new PolyTask, it increases the group ID

of the newly-obtained work group by one and updates the `curr_workgroup` with that value. The updated `curr_workgroup` records the next available work group that other group schedulers can execute. If all work groups of this PolyTask is finished, the group scheduler will deasserted the `isValid` value with the PolyTT to tell the host program that the PolyTask is finished; otherwise, the group scheduler releases the mutex lock and copies the kernel arguments to store in the local memory for PEs to access.

3. **PolyTask loading:** The PolyTask loading includes the software thread loading and the hardware thread loading. This step updates executables of sPEs and/or hPEs within the group. Each PolyTask is associated with executables that are stored in the PL DDR. The software thread loading is equivalent to load the corresponding ELF file into sPEs' kernel BRAMs as shown in Figure 3.10; while the hardware thread loading is equivalent to program re-loadable hPEs (i.e. PR regions) within the group by using corresponding PR bitstream files. Whether the PolyTask loading is performed depends on the value of `isMatch`. If the `isMatch` is `true`, the existing executables within the group match with new executables and therefore, this step can be bypassed.

4. **Triggering to finish:** Once the loading is finished, the scheduler triggers all PEs and wait to finish. All PEs within a group shared a local memory that the group scheduler can also access. PEs, as well as the group scheduler, keep polling the local memory to wait for triggers and signals. When all PEs finishes all work-items within the work group, the group scheduler is noticed to finish.

4.2.1 Architectural Supports for Multitasking

Specialized hardware architectures are designed to support both the software thread multitasking and the hardware thread multitasking.

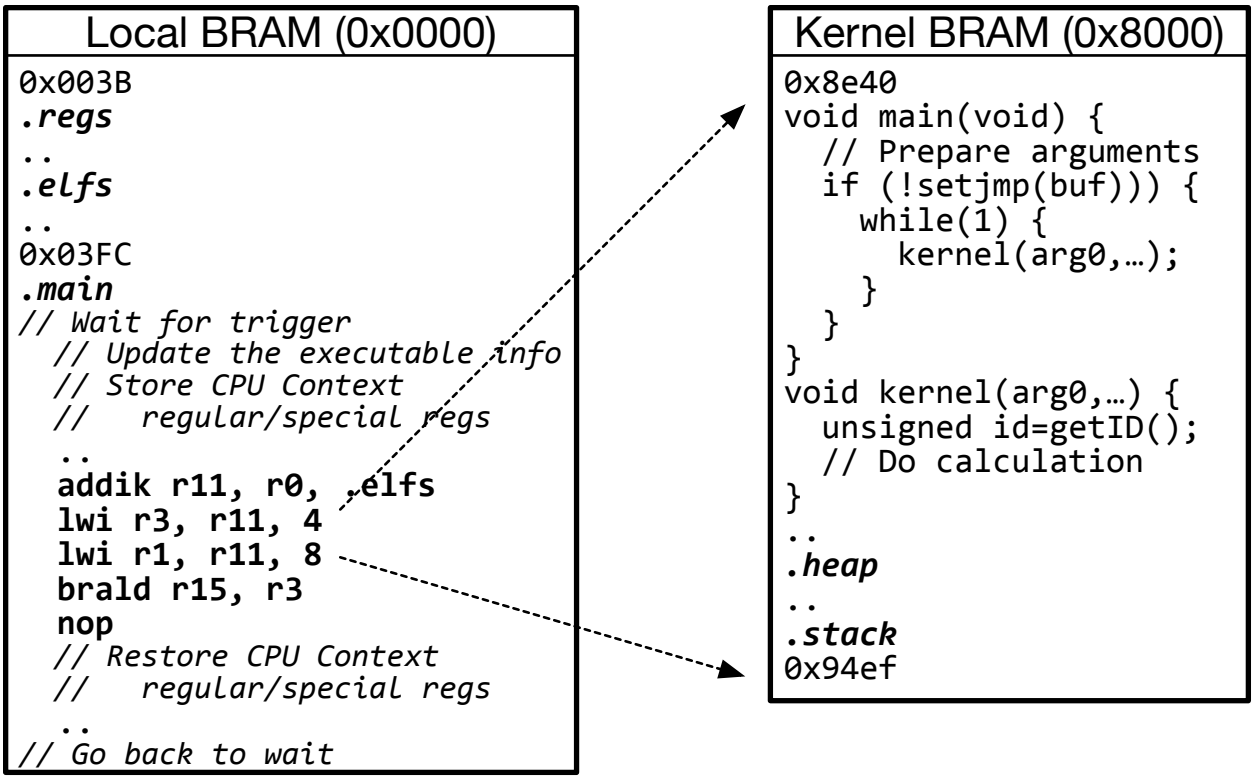

Figure 4.3: Software Thread Multitasking

Software Thread Multitasking

sPEs begin to run a bootloader existing in the Local BRAM once the system starts. The bootloader keeps monitoring triggers from the group scheduler. If it is triggered, the group scheduler has loaded an executable into the kernel BRAM, and the bootloader prepares to switch to the software thread with the executable. Figure 4.3 shows BRAM contents and how sPEs switch between the bootloader and the software thread in the following four steps.

1. Before sending triggers to sPEs, the group scheduler has copied the executable information (i.e. the `main` function entry address and stack address of the software thread) to the local memory. Once receiving the trigger, the bootloader firstly updates the new executable information in the `.elfs` section.

2. Then the bootloader stores values of 32 regular registers and special registers belonging to the MicroBlaze into the `.regs` section. This step saves the execution context of the

bootloader.

3. In MicroBlaze ABI [104], `brald` is an assembly statement to jump to another function. The bootloader uses `brald` to jump to the targeted address that is stored in register `r3`. The new stack address of software thread is passed by using register `r1`.

4. Once the software thread finishes executing all available work-items, the software thread returns to the bootloader. The bootloader then restores the values of regular and special registers from the `.regs` section.

Hardware Thread Multitaksing

The group scheduler cannot perform multitasking for unchangeable hPEs. How the group scheduler does hardware thread multitasking on a re-loadable hPEs is equivalent to use the PR technique to load a PR bitstream file into the corresponding PR region. The gray block showing in Figure 3.8 is the hPE as a PR region to hold a hardware thread executable. The group scheduler manages the context of the PolyTask. Different from the software thread, the hardware thread does not have context information.

All input and output ports of the hPE are connected to a decoupler, and the reset port of hPEs are connected to the PR controller. The group scheduler performs the PR bitstream file loading through a PR controller. Once finishing the PR file loading, the PR controller continues the routine to deassert the decouple and reset the hPE. When the decoupler is deasserted, the hPE can communicate with the outside. The hPE is then powered once it is reset.

4.2.2 Priority-aware Scheduling Policy

The scheduling policy determines how the group scheduler chooses a work group belonging to new PolyTask. This depends on various factors. The priority-aware scheduling policy is based on the priority of each PolyTask, which can be concluded in three steps:

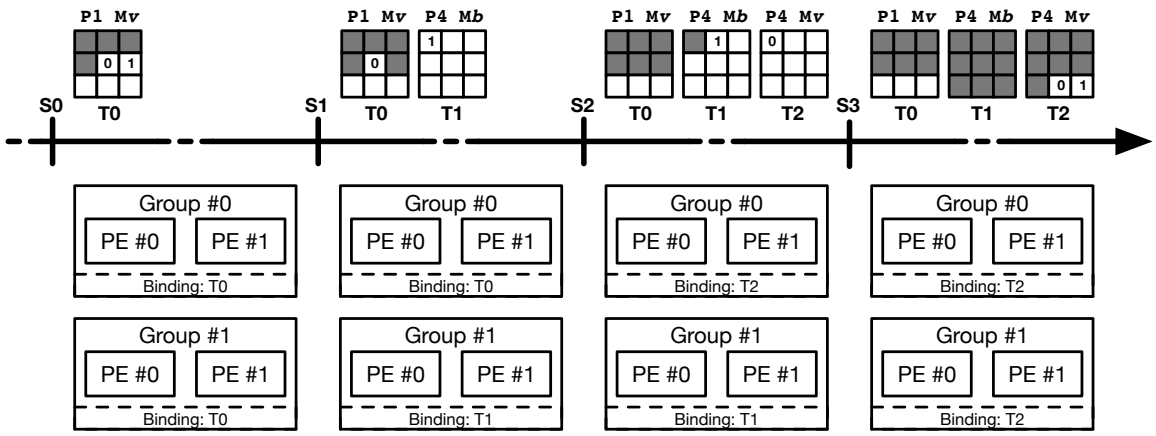

Figure 4.4: Polymorphic Multitasking Demonstration

1. Once a work group is finished by a hardware group, the group scheduler goes through the PolyTT queue to find a valid PolyTask with the highest priority. If this particular PolyTask is found, the group scheduler will choose a work group from this PolyTask and ignore the following step.

2. If there are multiple PolyTasks with the same priority returned from the searching, the group scheduler will choose a PolyTask with the same magic number with the current PolyTask. This reduces the unnecessary overheads of the PolyTask loading. If such a PolyTask exists, the following step is ignored.

3. If these PolyTasks have different magic numbers from the current PolyTask, the group scheduler will choose an arbitrary PolyTask, which is usually the one last visited.

Figure 4.4 gives an example to demonstrate how group-level priority-aware scheduling works. In this example, we have two hardware groups, within each of which there is entirely two PEs. At sometime, PolyTask `T0` with priority 1, and magic number `v` comes into the PolyTT queue.

Time Stamp `S0` There is only on PolyTask in the PolyTT queue. The first four work groups of the PolyTask `T0` have finished. Work group (1, 1) is executed by the hardware group 0, and work

group (1, 2) is by the hardware group 1.

Time Stamp `S1` PolyTask `T1` with priority 4 and magic number `b` comes into the PolyTT queue. At this point, the hardware group 0 has not finished current work group (1, 1) of PolyTask `T0` yet, and the hardware group 1 has finished the work group (1, 2). Then the hardware group 1 chooses to execute new work group (0, 0) from PolyTask `T1` as the PolyTask `T1` has the highest priority 4 within the PolyTT queue.

Time Stamp `S2` Another PolyTask `T2` with priority 4 and magic number `v` comes into the PolyTT queue. The hardware group 1 has finished work group (0, 0) of PolyTask `T1`, and the hardware group 0 has finished work group (1, 1) of PolyTask `T0`. At this point, each hardware group will choose a PolyTask with the highest priority. But there are two PolyTask `T1` and `T2` having the highest priority 4. In order to reduce the overheads of the excessive PolyTask loading, the hardware group 1 will continue on PolyTask `T1` to execute new work group (0, 1). The hardware group 0 can choose a work group from either PolyTask `T1` or `T2`. In this example, the hardware group 0 moves on to the new work group (0, 0) of PolyTask `T2`.

Time Stamp `S3` The hardware group 1 has finished PolyTask `T1`. It chooses the PolyTask with the highest priority, which is `T2`. From this point, both hardware group 0 and 1 will continue to finish PolyTask `T2` and then go back to work on PolyTask `T0` if no other PolyTasks are coming into the PolyTask queue.

4.3 Evaluation and Results

4.3.1 Experimental Setup

In this experiment, we want to examine how polymorphic multitasking works. The experiment has two parts: 1) hardware thread multitasking, and 2) hardware and software thread multitasking.

Table 4.1: Experiment Configurations

Items	Configurations	Descriptions
PolyPC	Hardware Config #1	2G4P0s, 4G2P0s
	Software Config #1	1, 2, 4, 8 (# work groups)
	Hardware Config #2	2G4P4s, 4G4P8s
	Software Config #2	4, 8, 16 (# work groups)
PolyTask Sequence	# Types	4
	# PolyTasks	4
	# PolyTasks per Type	4
	Total # PolyTasks	16
`Vector Multiplication` `FIR`	Data Size #1	256K * 4 bytes
	Buffer Size #1	32 * 4 bytes
	Data Size #2	32K * 4 bytes
	Buffer Size #2	0
`PageRank`	Data Size #1	512 * 4 bytes
	Buffer Size #1	32 * 4 bytes
	Data Size #2	1024 * 4 bytes
	Buffer Size #2	0
`Matrix Multiplication`	Data Size #1	32 * 4 bytes
	Buffer Size #1	0
	Data Size #2	256 * 4 byte
	Buffer Size #2	0

In the first part, only the hardware thread multitasking is examined on re-loadable hPEs. Since the maximum number of PR regions cannot exceed eight in our experiment, only two hardware platforms are generated as the `hardware config #1` shown in Table 4.1. One platform has two hardware groups (`2G`), each of which contains four PEs (`4P`). The other platform has four hardware groups (`4G`) and each one contains two PEs (`2P`). Both platforms have no sPEs (`0s`), which means all PEs are re-loadable hPEs. The total number of hPEs are eight for both of the two platforms, which is the same as the number of total PR regions. Different numbers of work groups are tested on both platforms.

In the second part, hardware and software thread multitasking are examined on two

platforms with both re-loadable hPEs and sPEs. The configurations are shown in Table 4.1 as `hardware config #2`. A Larger number of work groups than that of the first part is tested in this part.

In this experiment, a PolyTask sequence that consists of total 16 PolyTasks is generated randomly as the synthetic benchmark. The sequence comprises four types of benchmarks, each of which has four identical PolyTask in terms of data size and buffer size. The details of the four type of benchmarks are shown in Table 4.1. For the first part, considering we want the execution time of each PolyTask to be roughly the same, the data size and buffer size of each benchmark is carefully chosen. For the second part, we want to examine the breakdown of the execution of each hardware group, the data size and buffer size are different from the first part.

In order to examine how the priority-based scheduling policy works, two other scheduling policies are set as baselines. All the three policies use the same synthetic benchmark.

Baseline Every PolyTask is launched on the system one-by-one, which means the PolyTask will not be executed unless the previous one is finished. After one PolyTask is finished, all group schedulers will continue on the next PolyTask by performing the PolyTask loading. The test is implemented by setting the same priority and different magic numbers for each PolyTask within the synthetic benchmark.

Baseline+ Baseline+ is an improved `Baseline` scheduling policy. Group schedulers are aware of the PolyTask sequence in advance. After finishing one work group, the group scheduler will prefer to switch in next work group with the same magic number; otherwise, the group scheduler moves to a new group by performing the PolyThead loading. In this test, every PolyTask has the same priority. But the magic number is set correctly according to their benchmark types.

Priority This scheduling policy is the default one in our system. Different from `Baseline+`, the priority of each PolyTask in the sequence are not the same. After finishing one work group, the

group scheduler will choose a PolyTask with the highest priority from the PolyTT queue. If there are multiple PolyTasks with the same top priorities, the group scheduler will choose the one with the same magic number. In the test, two of the four PolyTasks with the same benchmark type have the priority of 1, and the remaining two have the priority of 0. Therefore, there are totally eight PolyTasks with the priority 1 and the other eight with 0.

We use two kinds of metrics to evaluate the first part of this experiment: average response time (ART) and average execution time (AET). Average response time is defined as:

$$ART = \frac{\sum_{i=1}^{N} ART_i}{N}$$

where N is the number of PolyTasks within the sequence, and ART_i is the average response time of the i^{th} PolyTask. And ART_i is defined as:

$$ART_i = \min T_{lj}, (1 \leq j \leq n)$$

where n the number of work groups of the i^{th} PolyTask. The average response time is used to evaluate how fast that a PolyTask begins to execute for different scheduling policies and different priorities.

Average execution time is defined as:

$$AET = \frac{\sum_{i=1}^{N} \sum_{j=1}^{n} (T_{fj}^{i} - T_{lj}^{i})}{N}$$

where N is the number of PolyTasks within the sequence, and n is the number of work groups of the i^{th} PolyTask. T_{fj}^{i} and T_{lj}^{i} belong to the j^{th} work group of the i^{th} PolyTask. The average execution time describes the averaged time (including all overheads and the effective execution) for all PolyTasks within the sequence.

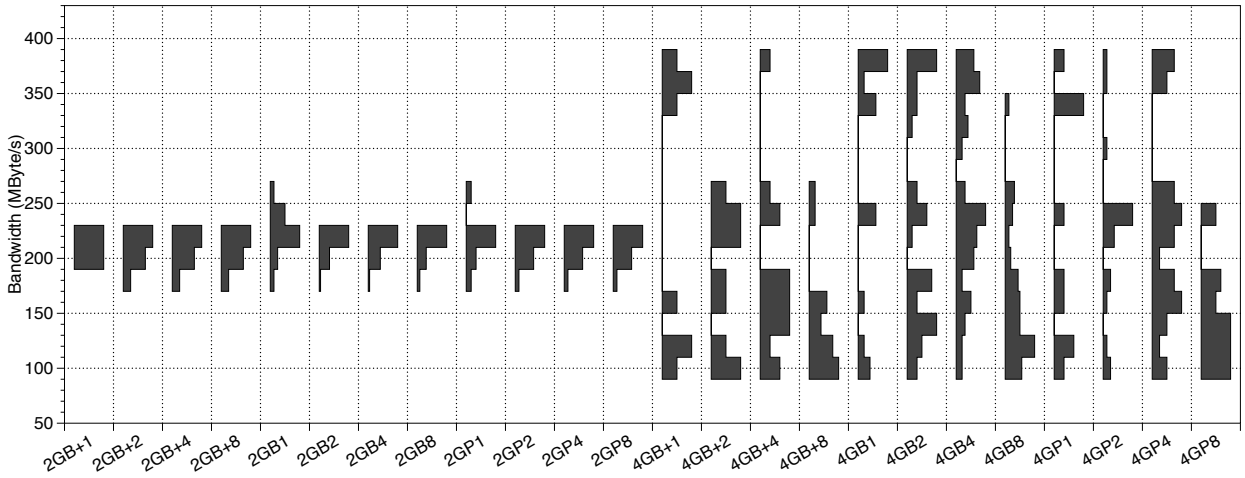

Figure 4.5: Partial Reconfiguration Bandwidth

Table 4.2: Partial Reconfiguration Bandwidth (unit: MB/s)

	2 Hardware Groups					4 Hardware Groups				
	1 WG	2 WG	4 WG	8 WG	Ave	1 WG	2 WG	4 WG	8 WG	Ave
PR Bandwidth	216.6	209.6	209.6	209.0	201.5	255.7	222.1	235.2	156.0	207.5

4.3.2 Result Analysis

Analysis for the Part 1

Figure 4.5 describes the bandwidth to load PR bitstream files for each scenario. Each x-axis label is denoted as the number of hardware groups (`2G`, and `4G`), scheduling policies (`B`, `B+`, and `P`), and the number of work groups (`1`, `2`, `4`, `8`). The serrated bars is the possibility distributions of bandwidth ranges. The maximum PR speed can reach almost 400 MB/s for all scenarios. The PR bandwidth for 2-group and 4-group hardware platforms are very different. Bandwidths of the 2-group one gather within a range from 200 MB/s to 230 MB/s; while bandwidths of the 4-group one have a wider range of 90 MB/s to 400 MB/s. The reason is that PR bitstream files are stored in the DDR memory that also stores the input and output data. When all PEs are fetching data elements under the burst mode, the PR loading and the data fetching may interfere with each other. For the 4-group platform, the data fetching and the PR loading have more changes to avoid

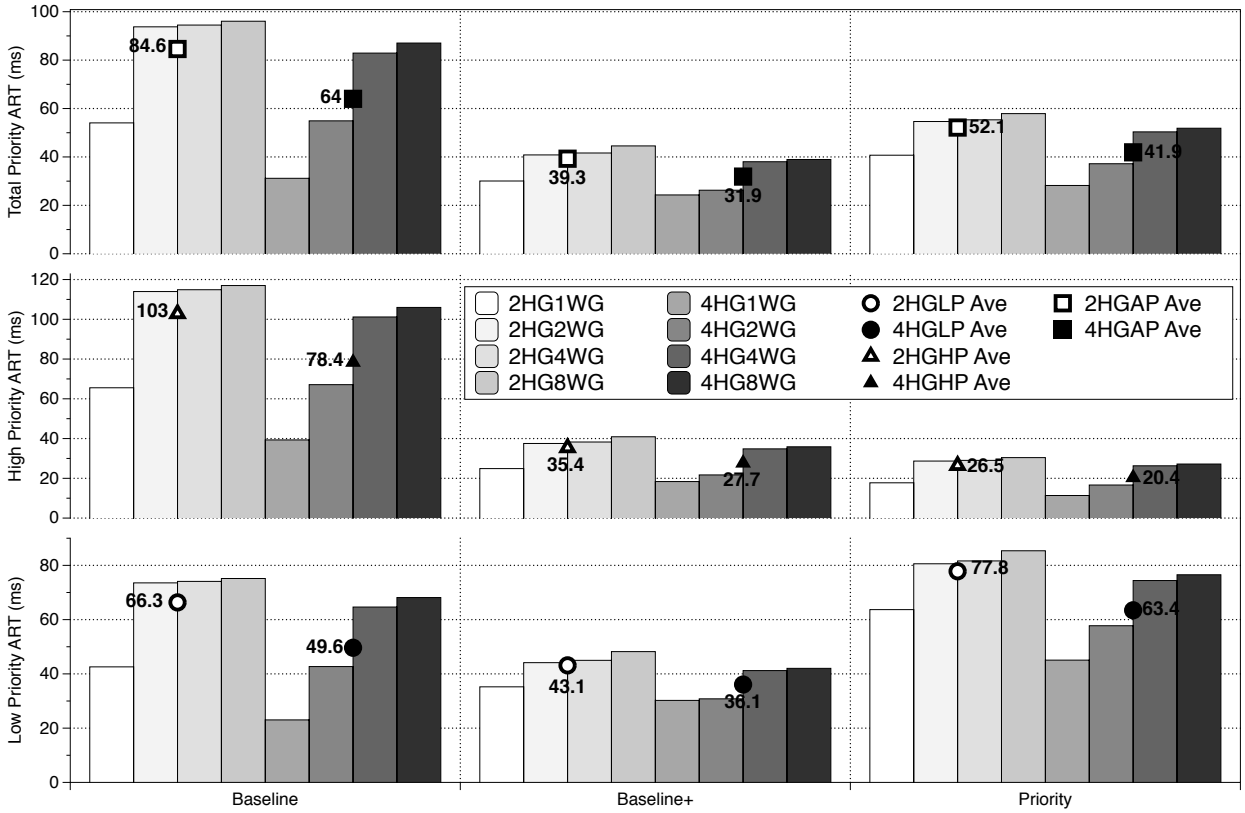

Figure 4.6: Average Response Time (Synthetic Benchmark with 16 PolyTasks)

interferences.

However, if we further examine the average bandwidth grouped by the number of work groups from Table 4.2, the PR bandwidth of 2-group and 4-group hardware platforms is almost the same (201.5 MB/s vs. 207.5 MB/s). As more work groups are used, the PR bandwidth decreases. This results from more interferences involved as the PolyTask is divided into more work groups.

The average response time of all the hardware configurations of part 1 is shown in Figure 4.6. For each scheduling policy, PolyTasks' ART with low priority (0), PolyTasks' ART with highest priority (1), and all PolyTasks' ART are demonstrated. From all scheduling policies, `Baseline` has the worst ART no matter it comes to low or highest priority PolyTasks. It's easy to explain that the system wastes too much time on redundant PR loadings. If the performance of 2-group hardware platform and that of 4-group hardware platform are compared, it is noticed that

more hardware groups can reduce the ART in general, which is good. It is because each group scheduler works independently, and more hardware groups provide more changes for more PolyTasks running at the same time, which may decrease the response time. Also, when the number of software of each PolyTask increases, the 4-group hardware platform reaches its convergence slower than the 2-group one for all three scheduling policies.

When `Baseline+` is used, all three kinds of ARTs decrease dramatically. The improvement comes from the redundant PR loadings are eliminated. When it comes to the `Priority`, the high priority ART decreases compared to the one from `Baseline+`. The low priority ART, and the combined ART increase. The `Priority` scheduling policy works in the way that when higher priority PolyTask is coming into the system, the current hardware groups will switch to that one, which may increase overheads of PR loadings, and be harmful to the ART of low priority PolyTasks. Therefore, the `Baseline+` can achieve the most efficient ART for overall PolyTasks; while the `Priority` will result in the least ART for high priority PolyTasks.

Figure 4.7 shows the average execution time of all the hardware configurations from the part 1. Similar to the ART from Figure 4.6, `Baseline` get the worst results due to the redundant PR loadings. Different from results of the ART, the 4-group hardware platform has worse AET than 2-group one. Although both hardware platforms have the same number of PEs entirely, more hardware groups may result in more scheduling overheads. When the number of work groups increases, the AETs decrease especially for 4-group hardware platform. When each PolyTask uses a small number of work groups, the hardware platform cannot be thoroughly used up, where some hardware resources may be wasted.

When `Baseline+` is enabled, the AET drops. Similar to how the ART drops, the improvement comes from reducing the redundant PR loadings, as well. However, the high priority AET does not benefit from the `Priority` scheduling policy, as well other kinds of AET. All kinds of AETs are slightly greater than those of `Baseline+` since the overheads of additional PR loadings.

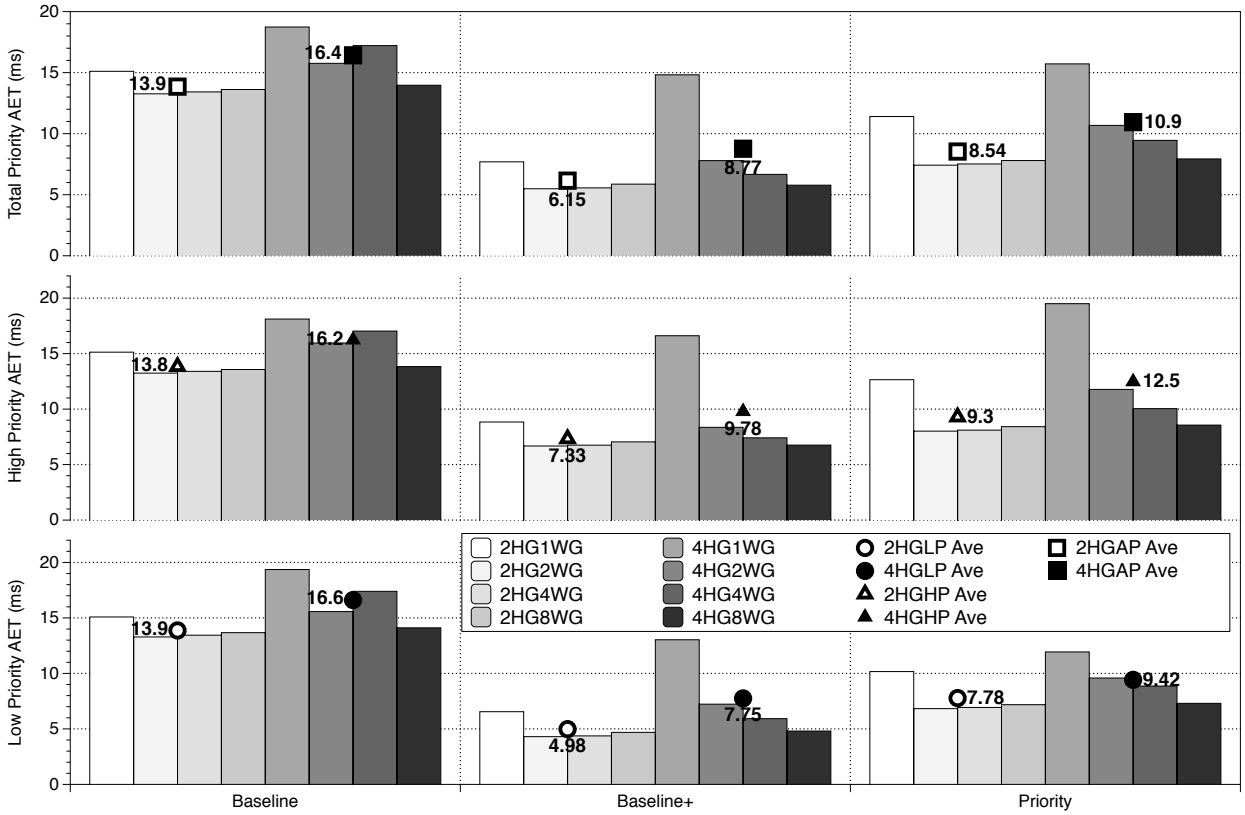

Figure 4.7: Average Execution Time (Synthetic Benchmark with 16 PolyTasks)

Analysis for the Part 2

Figure 4.8 demonstrates the bandwidth of the PolyTask loading for both hardware and software threads. The PR loading bandwidth is higher than the ELF loading for both 2-group and 4-group hardware platforms. However, more hardware groups have a negative influence on the PR loading bandwidth, but a slight influence on the ELF loading bandwidth. The PR controller performs the PR loading through one ICAP module. As the ICAP module can only handle one PR bitstream file at one time, PR requests from all group schedulers have to queue in line, which may result in lower bandwidth. The ELF loading is performed by the group DMA that every hardware group has. The bandwidth is more limited by the memory bandwidth instead of the structure hazard as there are multiple group DMAs working simultaneously.

If examining the bus hierarchy from Figure 3.4, we can notice that the data path for the PR

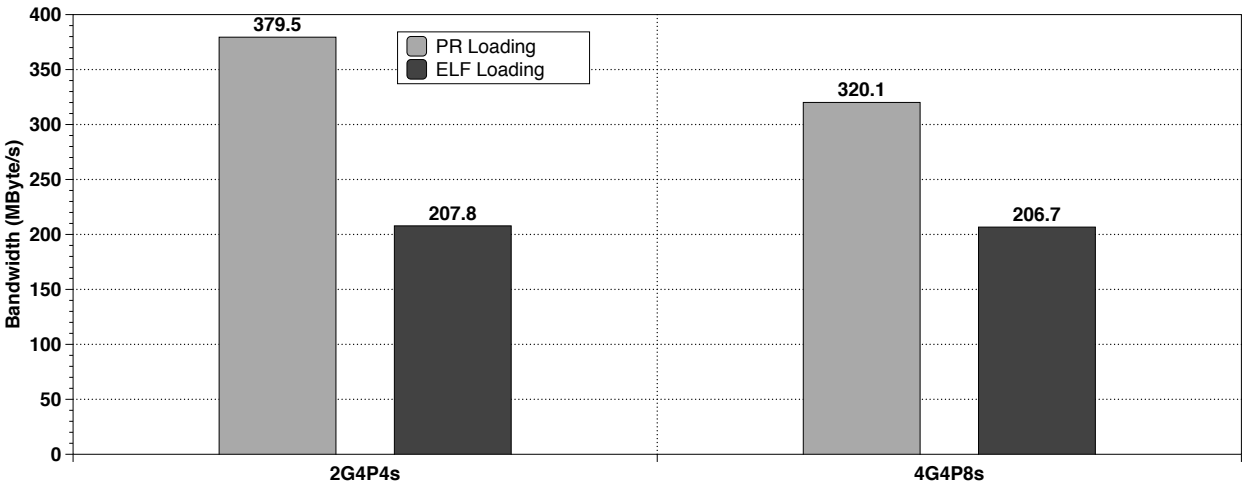

Figure 4.8: Bandwidth of the PolyTask Loading

Table 4.3: Numbers of Work Groups that Hardware Groups Execute

		G0 w/ L	G0 total	G1 w/ L	G1 total	G2 w/ L	G2 total	G3 w/ L	G3 total	Total # loadings
2G	B	16	1057	10	159	-	-	-	-	26
4P	B+	4	1047	4	169	-	-	-	-	8
4s	P	7	1114	3	102	-	-	-	-	10
4G	B	16	522	16	523	11	85	11	86	54
4P	B+	4	530	4	525	3	80	3	81	14
8s	P	7	552	7	512	3	76	3	76	20

controller to load PR bitstream files from the PL DDR memory into ICAP module is different from that for group DMAs to load ELF files. PR files from the PL DDR to ICAP go through one interconnect and while ELF files go through three interconnects. The longer data path may have lower data transfer bandwidth that of the shorter data path. This explains why the ELF loading bandwidth is lower than that of the PR loading bandwidth.

We calculate how many work groups are executed by hardware groups for the whole synthetic benchmark. Table 4.3 illustrates the results for both hardware platforms when the number of work groups is set as 16 for each PolyTask. For the 2-group hardware platform, hardware group 0 contains re-loadable hPEs and hardware group 1 contains sPEs. For the 4-group one, the first two hardware groups hold re-loadable hPEs, and the last two have sPEs. Three scheduling policies are tested for each platform, and each test executes totally 1216 work groups

Table 4.4: Overhead Breakdown of Each Hardware Group (unit: ms)

		Group 0				Group 1				Group 2				Group 3				Wall-clock
		1*	2†	3‡	4§	1	2	3	4	1	2	3	4	1	2	3	4	
2G	B	33.5	12.2	79.4	8258.1	4.9	1.8	0.3	8406.7	-	-	-	-	-	-	-	-	8413.7
4P	B+	30.5	12.1	19.9	8272.6	4.9	1.9	0.1	8328.0	-	-	-	-	-	-	-	-	8335.0
4s	P	31.7	12.8	34.7	8573.7	2.9	1.2	0.1	8667.4	-	-	-	-	-	-	-	-	8671.6
4G	B	16.5	6.0	106.4	6798.1	16.4	6.0	113	6786.3	2.7	1.0	0.3	7140.5	2.7	1.0	0.3	7140.4	7144.6
4P	B+	15.6	6.1	24.7	6813.9	15.2	6.1	23.5	6815.9	2.3	0.9	0.1	6858.8	2.3	0.9	0.1	6858.5	6862.1
8s	P	15.7	6.4	48.2	6816.8	14.6	5.9	47.0	6820.0	2.1	0.9	0.1	6943.5	2.1	0.9	0.1	6943.0	6946.5

*Overheads of scheduling. In this step, the group scheduler keeps going through the PolyTT queue until it picks up an available PolyTask.

†Overheads of updating context and the PolyTask loading without the DMA transferring. This is the route that the group scheduler updates the context and check if the DMA transferring needs to perform or not.

‡Overheads of the DMA transferring. Is is the time of the DMA transferring of executables for sPEs and/or hPEs. If no DMA transferring is performed, this time is zero.

§PE execution time. It is the time slice PEs finish all work-item within the work group. The group scheduler records the time stamp from triggering to receiving finish signals from all PEs.

by a run of the synthetic benchmark.

Similar to the results from the part 1, the scheduling policy affects the number of total loading (i.e. PR loading and ELF loading). The `Baseline` policy performs the most loadings while the `Baseline+` carries out the least. However, hPE groups execute much more work groups than sPE groups, which is about 7.5× for the 2-group platform, and about 6.5× for the 4-group one. The result can be compared with the execution time from Table 4.4. The execution time of hPE groups is pretty much the same with that of sPE groups. The speedup gains come from dedicated hardware accelerators over the general-purpose processors.

Table 4.4 demonstrates the overhead breakdown by using the trace subsystem through group schedulers. The 4-group hardware platform is about 1.18× - 1.25× faster over the 2-group hardware platform by comparing the wall-clock time. The speedup from the unchangeable hPE experiment (see Subsection 3.8) is higher than this one. The speedup is affected not only by the types of the benchmark, data size, and buffer size, but also by multitasking overheads. Overheads of scheduling and updating context and the PolyTask loading of the 2-group hardware platform

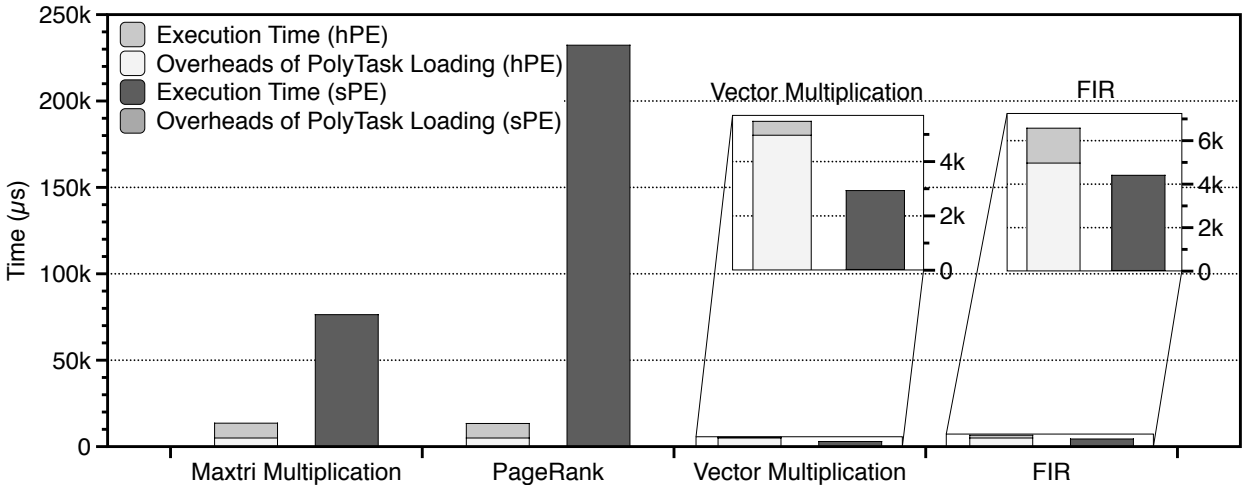

Figure 4.9: Overhead Breakdown of PolyTask Execution

are about twice as the 4-group hardware platform as the 2-group hardware platform executes twice work groups as the 4-group one. Although the 2-group platform executes the same number of work groups as that of the 4-group one, the 4-group spends more time to do the DMA transferring, which is explained from Figure 4.5. Overheads of the ELF loading is much less than overheads of the PR loading even the number of loadings are the same.

We further break down the PolyTask execution on the `2G4P4s` platform (see Figure 4.9) in terms of benchmark types. Although according to our experiments, the averaged bandwidth to load PR files is higher than that to load ELF files (i.e. 379.5 MB/s vs. 210.4 MB/s), overheads to re-load hPEs are significantly higher than those to re-load sPEs within a group, which is around 134× longer. The main reason is that one PR file for one hPE is usually larger than the ELF file for one sPE and ELF loadings benefit from the DMA duplicator while one PR file for one hPE cannot be used for other hPEs. However, sizes of PR files and ELF files may change depending on different types of applications. The averaged speedup that hPEs gain over sPEs is about 16×. Therefore, we can conclude from Figure 4.9 that sPEs are suitable for applications with small data size and short execution time due to its low overheads of the PolyTask loading. hPEs have advantages in large applications with long execution time.

Table 4.5: Reconfigurable System Resource Utilization

Items	LUTs (%)	Registers (%)	BRAMs (%)	DSPs (%)
One PR Region	3200 (1.47)	6400 (1.47)	10 (1.83)	20 (2.22)
Static 4G2P0s	54501 (24.93)	53618 (12.26)	104.5 (19.17)	0 (0)
Static 2G4P0s	42430 (19.41)	42079 (9.63)	72.5 (13.30)	0 (0)
Static 2G4P4s	45311 (20.73)	42292 (9.67)	104.5 (19.17)	0 (0)
Static 4G4P8s	70411 (32.21)	65486 (14.98)	168.5 (30.92)	0 (0)

4.3.3 Resource Utilization

Figure 4.5 demonstrates the resource utilization of four platforms as well as the PR region. In this experiment, one PR region has to be large enough to hold all kinds of benchmarks on hPEs. Resources of eight PR regions are excluded from resources of first static platforms (i.e. `4G2P0s`, and `2G4P0s`) so that the static systems do not use DSPs at all. `2G4P4s` utilizes more resources than `2g4P0s` since the former one has one group with four re-loadable sPEs. If we further compare the resource utilization of `2G4P4s`, and `2g4P0s` with the multitasking experiment from Figure 4.9, extra sPEs will bring more flexibilities on the polymorphic multitasking with slight overheads of resources.

4.4 Related Work

4.4.1 Reconfigurable Systems

To the best of our knowledge, this is the first time that the polymorphic multitasking is enabled on a heterogeneous system with the OpenCL programming model. Several previous works are focusing on HW/SW tasks on FPGA platforms. Most of them refer to operating system extension to manage accelerator resources. They are still highly related to our work since we also provide the intermediate framework to manage hardware resources. Some works provide a method to manage HW accelerators during runtime [4, 58, 92, 100, 107]; while others support HW tasks

with dedicated OS interfaces [6, 69, 70, 96].

In FUSE [58] the custom hardware circuits are integrated into a SoC as a memory mapped IO device peripheral. This is achieved via a customized hardware interface abstracted away by a corresponding loadable kernel module (LKM) in the kernel space. Transparent to the user, *thread_create()* will run the entire thread in either SW or HW based on resource availability. In other words, the operation system checks to see if there is a free accelerator for that function. If so, it runs in HW. The overhead of loading/unloading the LKM as well as calling OS services to communicate with the HW accelerators are among the drawbacks of this approach. Plus, there is no fine grained HW/SW dynamic partitioning. In other words, the entire thread is either running on SW or HW.

In BORPH [96], the hardware abstraction is achieved by adopting OS threads, which extends Linux targeting multi-FPGA platforms. Each HW task is implemented on a separate FPGA.

ReconOS [69, 72] targets dynamic partial reconfiguration (DPR) on FPGA. Hardware tasks are implemented as reconfigurable modules on PR regions. A set of unified POSIX-compliant APIs is provided for both hardware and software threads. They further support part of virtual memory management by providing memory management unit (MMU) for hardware threads. [1] also presents memory virtualization for reconfigurable hardware. Recent updates on ReconOS involve the preemptive hardware multitasking supports [45], where no deep modifications on source codes of hardware threads are required. It is implemented via parsing and restoring bitstream files.

Hthreads [6] allows hardware and software threads to co-exist and seamlessly be scheduled and synchronized under a unified programming model (i.e., POSIX). These approaches provided performance increases through the use of standard programming models that allowed the parallelism within the application to increase the scalable numbers of processors that could be mapped into each new generation of FPGA's. Similar to our work, Hthreads gives HW/SW

threads the support of dynamic memory allocation and recursive execution of functions.

In [41], authors present a runtime adaptive OS referred to as Configuration Access Port OS (CAP-OS). They argue that the traditional software-tasks scheduling based on resource (processor) availability and priority does not suffice for systems with runtime reconfigurable hardware. Their work addresses task scheduling onto reconfigurable hardware consisting of processors, co-processors, and accelerators. During design time, applications are defined as a collection of tasks whereby each is described through a control data flow graph (CDFG). These applications are profiled off-line for execution time, and possible suggestions for hardware implementation opportunities are presented to the user. The user can choose to provide hardware implementations of suggested code blocks enabling the runtime system for alternative task placement and scheduling. Similar to our work, this work classifies tasks into three categories according to where execution occurs: software tasks (processors), co-design tasks (processor+hardware accelerator), and hardware tasks (hardware accelerator).

4.4.2 Task Scheduling and Placement

Several works were devoted to scheduling and placement of task on reconfigurable systems. In [18], authors present a mixed off-line/on-line scheduling strategy, where the CDFGs are analyzed off-line via customized list scheduling techniques. The extracted parameters are used to optimized the scheduling results during run-time. The on-line scheduling is implemented in hardware as dedicated logics. Similar to this work, CAP-OS [41] requires user profiling to build up a flow graph (CDFG) that is used for run-time scheduling. Scheduling of all such tasks occurs through the main processor where CAP-OS executes. Similarly, OS services such as additional hardware resources occur through this single processor. Another work [14] extending Hthreads provides the run-time tuning for applications with different types and data sizes. Information that is needed for scheduling is extracted off-line presenting within a resource table.

In [71], authors extend ReconOS to support cooperative scheduling techniques. The source

codes of hardware threads require modification to support the cooperative scheduling. Hardware threads can be preempted at certain points of execution. Our work is similar to this work in terms of the 'semi-preemptive' scheduling. But source codes of hardware codes does not need special modifications. The follow-up work [45] on ReconOS supports preemptive scheduling for hardware threads. They leverage the 'reverse engineering' on PR bitstream files to resolve checkpoints during run-time. However, this technique highly relies on vendor specifications and may not be used on the newest FPGA devices (e.g., Zynq devices). Works in [34, 35, 89] execute hardware tasks in non-preemptive scheduling. They divide the reconfiguration areas into fixed regions.

We compare our work with the previous work above and would like to highlight the following aspects:

- The PolyPC framework is designed to parallelize applications with multitasking supports via parallel programming models. It is different from the previous works that are implemented for sequential applications or multi-threaded applications without significant performance improvement.
- The PolyPC framework is fully implemented on existing FPGA devices. However, some previous works are not yet fully implemented and they are at the theoretical stage.
- The decision of scheduling tasks on either hardware and software resources appears to be limited to the main processor (e.g., CAP-OS), which may be the bottleneck increasing scheduling overheads. The PolyPC framework allows decision making on top of multiple independent group schedulers.
- Both hardware threads and software threads are not locked to particular PEs during system generation stage. They are allowed to run on any PEs during run-time.
- Although our PolyTasks do not support fully preemptive scheduling, source codes of PolyTasks does not require specific modifications to support multitasking; while other

previous work may need modifications on their source codes [71].

- In our PolyPC framework, we can achieve up to 400 MB/s to switch out and in both hardware and software threads. Previous work (e.g., CAP-OS) can only achieve about 13 MB/s to 28 MB/s

Chapter 5

Design Flow

In early times, system designers are able to build a reconfigurable embedded system manually with few processors, buses and other peripheral components by using the vendor-provided EDA tools. However, densities of current platform FPGAs are continuing to grow in recent decades. When FPGAs have reached the million-LUT level, it is possible to construct a complex system composed of hundreds of components on a single chip. Traditional GUI-based methods are certainly no longer sufficient for handling this complexity. FPGA vendors continue to launch tools with text-based configuration files (e.g., Xilinx MHS) and TCL-based scripts (e.g., Vivado TCL) for automatic generation of multiprocessor systems. But these tools are not designed for generating framework such as the PolyPC framework.

The PolyPC framework requires extensions on existing vendor-provided tools. The challenge is to create a design flow that allows users to work from software source codes and platform specifications to executable binaries.

5.1 Platform Generation

The PolyPC framework extends the existing process to build a reconfigurable embedded system by using the vendor-provided tools. The process involves both hardware and software generation tools. The framework mainly utilizes the `bash` scripts, and `Tcl` scripts that Vivado tool sets (including `Vivado`, `Vivado HLS`, `MicroBlaze SDK`, `Petalinux SDK`, and `XSDB` debug tools) use to work automatically without invoking GUI-based tools.

Figure 5.1 shows the entire generation flow from source codes to executable binaries on the SD cards. What designers need to prepare is source codes regarding hardware platform

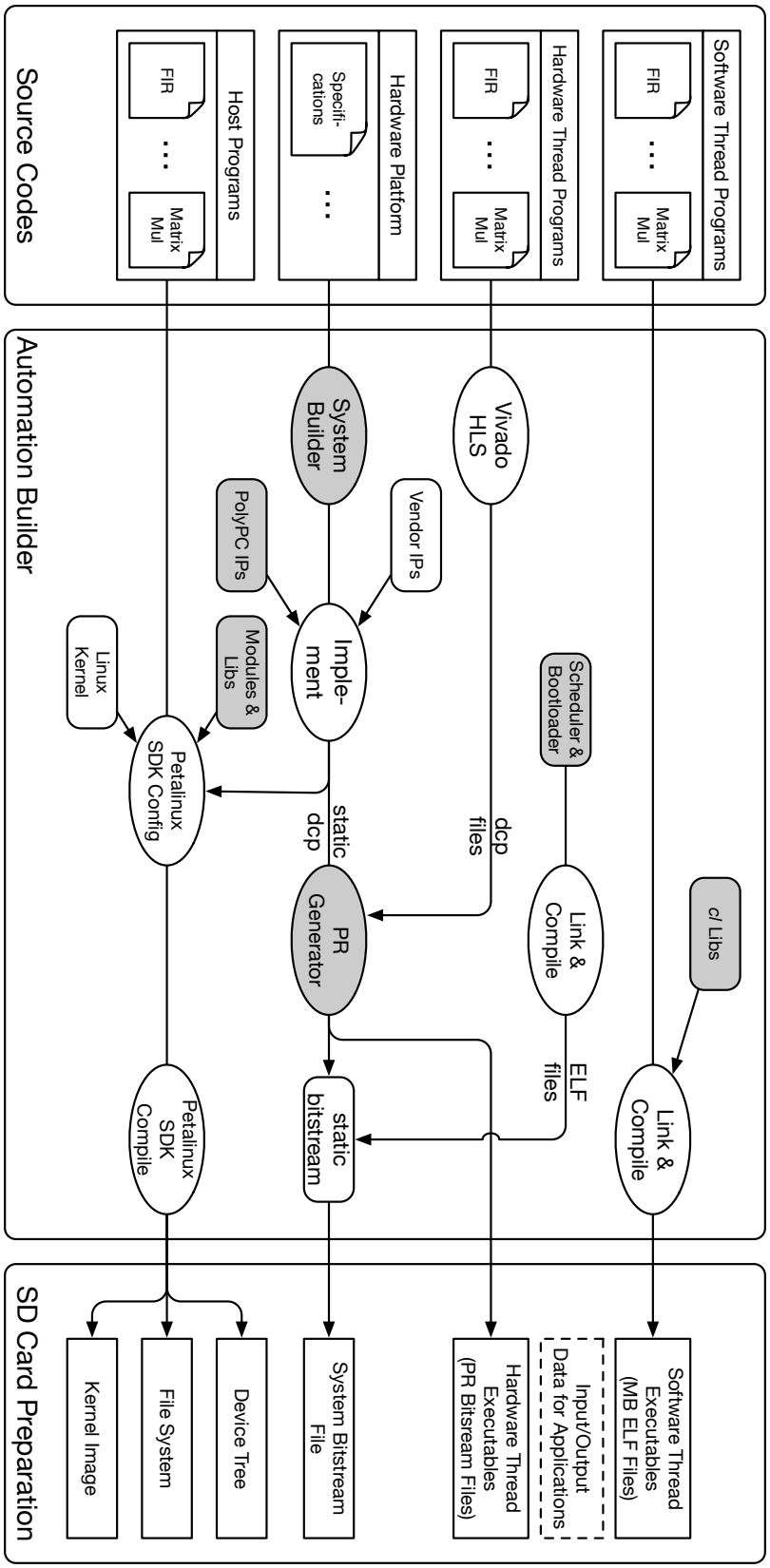

Figure 5.1: Platform Generation Flow

specifications and software programs including host programs and PolyTask kernel programs. The hardware platform specifications are comprised of the number of hardware groups, the number of PEs per group and other configurations (e.g., trace and debug systems). If users want to use unchangeable hPEs, the number of and the name of the hPE are also attached to the specifications. Hardware specifications do not need to be associated with software programs. In other words, no matter how many groups are set in software programs, they can run on hardware platforms with any number of hardware groups.

Software programs consist of host programs and PolyTask kernel programs. Host programs will initialize input and output data, create PolyTasks and launch PolyTasks into the system. Kernel programs have two forms: software thread programs for sPEs and hardware thread programs for HLS to compile to hardware accelerators as either unchangeable hPEs and re-loadable hPEs. The host program does not necessarily have a one-to-one relationship with the PolyTask kernel program. For instance, one host program can launch multiple PolyTask kernels, and one PolyTask kernel can be launched by multiple host programs.

The `Automation Builder` is comprised of several `Tcl` scripts that can take specifications and source codes and runs on `Vivado` tool sets. The `Automation Builder` assembles every hardware components (including vendor-provided IPs and custom PolyPC IPs) into a block design and connects them together. After the implementation of the block design, it will generate a static dcp file that has empty PR regions for hPEs. The system configuration file will be generated, as well. `PR Generator` applies every dcp file that is synthesized by `Vivdao HLS` from hardware thread programs onto the static dcp file to generate a static bitstream file and PR bitstream files. The group scheduler program along with the bootloader program of sPEs are compiled to executable ELF files. The static bitstream file is updated with them to generate the final system bitstream file. The system configuration file, PolyPC drivers, libraries, and Linux kernel source codes are linked and compiled by the `Petalinux SDK` to configure the Linux system running on the ARM processor. After compilation, necessary bootable file (e.g., device tree, file system, and

kernel image) are generated. Software thread programs are compiled into ELF files that can run on sPEs. All generated files along with data that is about to process will be stored on an SD card.

5.1.1 Automation Building Flow

As shown in Figure 5.1, hardware platform specifications, PolyTask kernel programs and host programs are given to the automation builder as input. Hardware platform specifications are used to build the hardware platform. PolyTask kernel programs can be compiled into either sPE executables or hPE executables. Host programs are associated with PolyTasks running on the Linux system. We create multiple `Tcl` and `bash` scripts (including hardware platform building tool, hardware thread dcp generation tool, SDK automation tool, and Petalinux configuration tool) to connect different parts as a complete automatic design flow. The following shows an example to demonstrate how our automation builder works.

Hardware Thread Kernel Compilation First of all, if we want to have hardware thread kernels running on hPEs in the system, we need to generate the hardware thread kernel IPs and dcp files from kernel programs that are written by designers. If we do not want to add hPEs to the system this step can be ignored.

```
vivado_hls -f hls_gen.tcl <Options>
```

This step invokes the Vivado HLS tool to run the script called `hls_gen.tcl`. `Option` can be either a name of a hardware thread kernel name that is the same as the top function name in HLS source codes or `all`. If a particular kernel name is given, this step will generate or update this hardware kernel; otherwise, all hardware thread kernel programs will be compiled or updated.

Hardware System Generation The system can be generated by using the following command.

```
vivado -mode batch -source gen.tcl -tclargs <Number of Groups>
                                            <Number of PEs per Group>
```

```
                                    <Total Number of hPEs>
                                    [Name of the hPE]
```

This command invokes Vivado to finish system assembly, implementation and PR generation. The first three arguments are required to build the system. The last argument is optional. If it is set as a platform with specific unchangeable hPEs, the system will not have PR regions, and all hPEs are associated with that hardware thread kernel. In this configuration, the name of the hPE has to be specified; otherwise, all hPEs are re-loadable. The total number of re-loadable hPEs cannot exceed the maximum number of PR regions that is eight in the PolyPC framework. In this configuration, PR regions will be created, and PR bitstream files are generated in the final step.

The `PR Generator` can work off-line without going through the automation builder from the beginning. When new hardware thread kernels are added to the framework, the `PR Generator` compiled new kernel programs with system static dcp file to generate PR bitstream files.

SDK Project Generation Next, we need to create all MicroBlaze SDK projects and compile them for sPEs and group schedulers.

```
xsdk -batch -source sdk_config.tcl <Number of Groups>
                                   <Number of PEs per Group>
                                   <Total Number of hPEs>
```

After this command is executed, the software kernel projects are created and compiled. Executable binary files of group schedulers and bootloader for sPEs are compiled, as well. Two *lscript.ld* files for sPE MicroBlazes are modified to link bootloader and software thread kernel programs, respectively as the bootloader and software thread executables locate in different BRAMs with different addresses. Since sPEs within a group are identical to each other, we can either build only one ELF file that runs on sPEs or different ELF files for sPEs. Similar to the `PR Generator`, software thread kernels can be compiled off-line, as well.

Table 5.1: Intermediate Libraries of PolyPC Framework

Libraries	Scope	Functionalities
sche	Scheduler	ELF loading, mutex, PRC controlling, and many others.
cl	Kernel	OpenCL-like kernel functions (e.g., getID()) and supported functions.
mem_man	Host	The dynamic memory manager for allocating and freeing.
exe_loader		The dynamic executable loader.
mutex		Handle mutex-related functions, such as requesting and releasing.
register		Handle PolyTask-related functions such as lunching and cleaning up.
trace		Tracing-related functions, such as creating trace space and dumping.

Petalinux Project Generation The last step is to create and compile Petalinux projects by using Petalinux SDK.

```
./petalinux_config.sh <Project Name>
```

The PolyPC framework provides necessary modules and libraries for embedded Linux system. Petalinux SDK takes these drivers and system configuration to generate a Petalinux project. After compiling, it delivers files that the system need to boot from SD card. Host programs, as well as other user programs, are linked and compiled in this step.

5.2 Intermediate Libraries

Intermediate libraries are between the hardware platform and software programs that include host programs and kernel programs. Table 5.1 demonstrates the most important libraries that are used to create programs for group schedulers, PolyTasks, and host programs. `sche` is specially used for group schedulers. It includes drivers to load executables of polymorphic tasks and other functions, such as searching the PolyTT Queue and requesting/releasing mutex locks for group schedulers.

5.2.1 Memory Management

The PolyPC framework manages memory resources in multiple levels. Private memories of sPEs and hPEs are managed in different ways. Programs on sPEs manage private memories through an assembly linkage and compilation standard that is executable and linkable format (ELF). Users can use the C standard library to allocate and free memory spaces within the `.heap` section of ELF files dynamically. Alternatively, static allocation is available on other sections (e.g., `.bss` and `.data`). Private memories of hPEs are allocated statically and implemented on memory resources (i.e., registers and BRAMs) according to actual usage.

The host memory is accessible to the host processor, and therefore the memory is managed by the Linux OS. The global memory is managed by the host process, as well. But this memory space span is out of the virtual memory system under Linux OS. The host processor is always able to use this memory manually by using its physical address. This is not an efficient way to utilize memory resources. In the PolyPC framework, the intermediate library provides a dynamic memory manager for the host processor to manage the global memory. Although PEs are accessible to the global memory, they cannot manage the global memory.

The challenge of the dynamic memory manager is to maximize utilization of the global memory while to minimize overheads. The former one requires fine-grained allocation, but the latter one requires coarse-grained allocation. The dynamic memory manager within the PolyPC framework manages a doubly-linked queue in the Linux kernel space. Each node of the queue contains information (i.e., entry address and size) about a contiguous memory space that is either free or allocated. Considering the 4K boundary for DMA transfer, start addresses of allocated memory spaces are under 4K boundary, which means the minimum size of an allocated memory space is 4K bytes How it works is described in two aspects:

- **Allocation** The manager goes through nodes of the queue from the beginning until it encounters a node that is free and equal to or larger than the size requested. Then the

manager attaches a new node to the node found if there is remaining space after allocating for the node found. If There are no more space for the PL DDR memory, the manager will return an error. Initially, there is one node existing in the queue. The node is free and indicates the whole free PL DDR memory.

- **Free** The manager goes through nodes of the queue from the beginning until it encounters the node that matches the entry address we want to free; otherwise, an error is returned. Then the manager marks the node as free or merges with the previous and later node. If the previous and following nodes are allocated, the targeted node is marked as free. If either the previous or the following node is free, the targeted node is merged with that one to extend to have a larger free memory space.

5.2.2 Dynamic Executable Loader

The sPE executable files are flattened to preserve the memory layout of instructions and data. It is required as memory addresses and file offsets of sections in an ELF file may not be the same [19]. In other words, sections are not contiguous in the memory. Instead of loading ELF files into memory directly, an executable loader is needed to analyze the structure of ELF files and load them correctly. The hPE executable files are PR bitstream files. They are binary files that are generated from Vivado tool sets.

The PolyPC framework provides a dynamic executable loader for loading executables of both hPEs and sPEs.

hPE Executable Loader

Original `.bit` bitstream files cannot be use to load into the PR controller directly. Through the Vivado tools, the `.bit` files are converted into `.bin` files to handle the endianness and bitswap problems. The PolyPC framework uses a little-endian system with 32-bit wide buses. The `.bit`

Table 5.2: Program Header within an ELF File

Index	Type	File offset	File size	Virtual address	Memory size
0	LOAD	0x00000094	0x00000028	0x00000000	0x00000028
1	LOAD	0x000000BC	0x00000A58	0xC0000000	0x00000A58
2	LOAD	0x00000B14	0x00000128	0xC0008000	0x000009B0

Table 5.3: Sections within an ELF File

Index	Name	File offset	Virtual address	Memory size
3	.vectors.hw_exception	0x000000B4	0x00000020	0x00000008
4	.text	0x000000BC	0xC0000000	0x000009FC
14	.head	0x00000C3C	0x000081B0	0x00000400
15	.stack	0x00000C3C	0x000085B0	0x00000400

files are converted to little-endian files, as well as indicating a 32bit bus compatible. The PR bitstream files have a particular bitswap format that is required by the ICAP module. But in our framework, the PR controller will handle the bitswap.

Since every `.bit` file has the same size, the executable loader combines every single file into one file for each hPE group. The entry offset of each PR file within the big one file is calculated by using the number of hPEs within a group and the size of PR files from each group scheduler.

sPE Executable Loader

The dynamic executable loader for sPEs can analyze and load ELF files during runtime. The `program header` within the executable ELF file describes a `segment` that is needed to prepare the program for execution. A `segment` contains multiple `sections`. Table 5.2 shows the loadable segments that are necessary for the program to execute, which means these segments are required to load into memory for execution. File offsets are not equal to the virtual addresses as well as file sizes and memory sizes. To load segments correctly, segment 1 and 2 are loaded into their virtual addresses from their file offsets, respectively. Segment 2 requires memory padding as well as its file size is not the same as its memory size.

As an example, consider a portion of the ELF file shown in Table 5.3. Section 3 and 4 are contiguous within the ELF file but are non-contiguous for the memory layout. So that these two sections need to be loaded into their addresses separately. Although file sizes of section 14 and 15 are zero as they have the same file offset, the memory size is not equal to zero. This is because `.head` and `.stack` do not have static contents within the file but are allocatable during execution. These two sections need memory padding when loaded into the memory.

According to the PolyTask information within the PolyTT (see Figure 3.5), symbol information including entry addresses of the `main` function and the `.stack` section must be extracted as the bootloader requires this information to jump to execute the new thread. We cannot load only the `main` function as the MicroBlaze bare-metal program assembles functions, such as exception handles and initialization, which are necessary for execution. The new thread also needs a stack space with the address for recursive functions. The `DMA_size` is the size of the maximum memory footprint that has instruction and data contents. The `thread_size` is usually larger than the `DMA_size` as the thread size includes allocatable sections such as `.head` and `.stack`. The MicroBlaze linker uses a `linkscript` file to change the entry address of the PolyTask to 0x00008000, which is the physical address of the kernel BRAM shown in Figure 3.8.

5.3 Programming OpenCL Applications

The PolyPC framework requires host programs and kernel programs that include software thread kernel programs and hardware thread kernel programs.

5.3.1 Execution Model

The execution model refers to how to write a pogrom with the OpenCL model and how to map the divided work groups on hardware groups. Figure 5.2 shows how the problem space of a 2-dimensional problem (i.e., `matrix multiplication`) is divided and mapped to each hardware

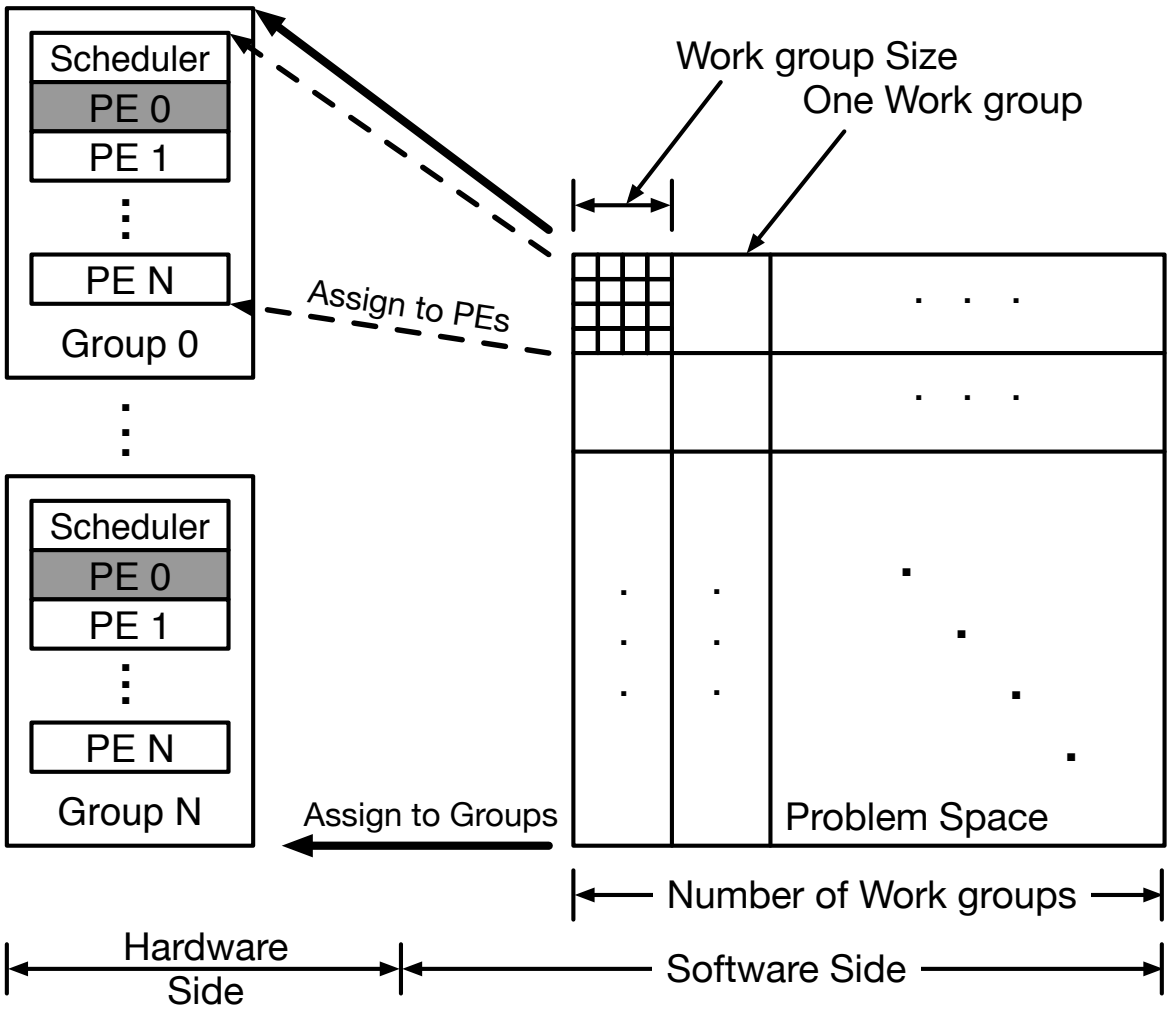

Figure 5.2: Problem Space Mapping in Hardware Side

group. The whole problem space is divided into multiple work groups. The number of work groups of the two dimensions is called `group numbers` that is indicated by a pair of numbers. The number of work items of two dimensions within each work group is called `group size`. The current PolyPC framework supports up to two dimensions. The maximum number of work groups and work group size is limited by the ID data length that is illustrated in Section 3.6.1.

The intermediate library also provides supports for both host and kernel programs. As shown in Table 5.1, besides libraries for group schedulers others are for host and kernel programs. Since the OpenCL language is library-base programming model, PolyPC provides `cl` to write

software and hardware thread kernel programs. It is equivalent to the `CL` library from the OpenCL standard library. Other libraries are used in host programs. They adapt a broad range of low-level drivers to high-level user interfaces.

- `mem_mam` provides user functions `ddr_malloc` and `ddr_free` that are similar to `malloc` and `free` functions in C libraries, which is the user interface of the dynamic memory manager.
- `exe_loader` is for the dynamic executable loader. It provides two user interfaces: `elf_loader` for sPE executables and `pr_loader` for hPE executables.
- `mutex` provides functions to request and release mutex locks from the mutex manager as well as Linux synchronization subsystem. This library is implemented in the Linux kernel space and does not provide user interfaces. Libraries such as `register` call this library for shared resources.
- By using `register`, host programs can launch and delete a PolyTask from the PolyTT queue. It provides user interfaces as `reg_add` and `reg_del`.
- `trace` subsystem can work as either a debug system or an evaluation system that can help developers further examine the system. User interfaces `trace_alloc` and `dump_trace` allocate a trace memory and dump the trace data, respectively.

5.3.2 Host Programs

Listing 5.1 Host Program of PageRank

```
typedef float d_type;

int main(int argc, char *argv[])
{
    // Preparation
    reg_clr();
    trace_clr();
    int ret         = 0;
    int SIZE_PER_PE = 2;
    int num_group   = 4;
    int data_size   = 1024;
    int buf_length  = 32;
```

Listing 5.1 (Cont.): Host Program of PageRank

```
    int group_size  = data_size / num_group / SIZE_PER_PE;                    13
                                                                              14
    // Construct PolyTask structure                                           15
    struct PolyTask_struct PolyTask;                                          16
    PolyTask.argv[0] = ddr_malloc(data_size * data_size * sizeof(d_type));    17
    PolyTask.argv[1] = ddr_malloc(data_size * sizeof(d_type));                18
    PolyTask.argv[2] = ddr_malloc(data_size * sizeof(d_type));                19
    PolyTask.argv[3] = data_size;                                             20
    PolyTask.argv[4] = buf_length;                                            21
    PolyTask.kernel_magic     = 'p';                                          22
    PolyTask.cur_group_id.id0 = 0;                                            23
    PolyTask.cur_group_id.id1 = 0;                                            24
    PolyTask.group_size.id0   = 1;                                            25
    PolyTask.group_size.id1   = group_size;                                   26
    PolyTask.group_num.id0    = 1;                                            27
    PolyTask.group_num.id1    = num_group;                                    28
    PolyTask.trace_ram_off    = trace_alloc(PolyTask.group_num) /             29
                                sizeof(struct hapara_trace_struct);           30
                                                                              31
    // Load ELF and PR files into memory                                      32
    ret = elf_loader(ELF_FILE_NAME, ELF_START_ADDR, &PolyTask.elf_info);      33
    ret = pr_loader(PR_FILE_NAME, &PolyTask.pr_info);                         34
                                                                              35
    // Add PolyTask structure to PolyTask Queue                               36
    ret = reg_add(&PolyTask);                                                 37
                                                                              38
    // Dump trace file and wait for this thread to terminate                  39
    dump_trace(0, TRACE_FILE_NAME);                                           40
                                                                              41
    // Free all allocated memory space                                        42
    ret = reg_del(ret);                                                       43
    ret = ddr_free(PolyTask.elf_info.ddr_addr);                               44
    ret = ddr_free(PolyTask.pr_info.ddr_addr);                                45
    ret = ddr_free(PolyTask.argv[0]);                                         46
    ret = ddr_free(PolyTask.argv[1]);                                         47
    ret = ddr_free(PolyTask.argv[2]);                                         48
    return 0;                                                                 49
}                                                                             50
```

Host programs running on the host processor are mainly responsible for initializing data, creating, and launching PolyTasks into the system. Listing 5.1 shows a portion of a workable host program associated with the `PageRank` benchmark. The host program is programmed and executed in orders, which is demonstrated in the following.

1. **Create a PolyTask instance:** Line 16 create a PolyTask instance in the host memory. Creating a PolyTask is equivalent to create a PolyTask structure. The `PolyTask_struct` structure is associated with the data structure storing in the PolyTT queue.

2. **Create Memory Objects:** The dynamic memory manager is used to create memory objects in the global memory. Line 17-19 creates three memory objects by using `ddr_malloc`. Similar to the `malloc` function from C standard library, this function returns a physical

address of the global memory given a size that users want to allocate.

3. **Initialize Memory Objects:** In this step, users initialize the allocated memory objects. Since the host processor can access the global memory by mapping the physical memory address to its virtual memory space, users can initialize the allocated memory objects from the host program. Alternatively, users can always copy data from extern storage (e.g., SD card) to the memory objects. Figure 5.1 does not show this part.

4. **Initialize PolyTask:** Initializing a PolyTask includes two parts: 1) arguments that are passed to kernel programs; 2) runtime information that is necessary for group schedulers. In this example, totally five arguments are passed for the PolyTask including three memory object addresses of input and output data elements and the other two are required particularly by the benchmark itself (see Line 17-21). Runtime information includes the `kernel_magic` as well as how the problem space is divided such as `cur_group_id`, `group_size`, and `group_num`. The entry of the trace ram can also be attached to the PolyTask by allocating the space for the trace data.

5. **Load Executables:** The dynamic executable loader loads sPE and hPE executables from the extern storage into the global memory. It calls the dynamic memory manager to allocate global memory for executable files. The information of executables is updated in the PolyTask as well. At this point, the PolyTask is created and up-to-date with necessary information for launching.

6. **Launch the PolyTask:** Then the host program calls `reg_add` to add `PolyTask` into PolyTT queue by copying the PolyTask structure to the PolyTT queue. After this step, the PolyTask is launched successfully, and each group scheduler can access this PolyTask from the PolyTT queue.

7. **Dump Trace Data:** Trace data can be read from the trace memory and stored into either the extern storage or the host memory. Once the PolyPC framework finishes the PolyTask, the host program will return from the `dump_trace` function.

8. **Clean Up:** Firstly, the PolyTask is unregistered from the PolyTT queue to save spaces. Then the dynamic memory manager frees all allocated memory objects including spaces for executables.

5.3.3 Kernel Programs

Kernel programs are compiled to executables to run on PEs conducting the computation tasks. Kernel programs are comprised of software and hardware thread kernel programs. Software thread kernel programs are compiled to ELF files to run on sPEs. Hardware thread kernel programs are compiled by the HLS tool to generate two types of files: dcp files for re-loadable hPEs and IP modules for unchangeable hPEs.

Hardware Thread Kernel Programs

A complete hardware thread kernel program that is written in C for the Vivado HLS is decomposed into three parts: application main function, kernel function, and definitions.

Listing 5.2 `pagerank` Function of PageRank for hPEs

```
void pagerank(volatile unsigned int *id,                                    1
              volatile unsigned int *barrier,                               2
              volatile unsigned int *barrier_rel,                           3
              d_type *data,                                                 4
              char htID) {                                                  5
#pragma HLS INTERFACE ap_ctrl_none port=return                              6
#pragma HLS INTERFACE m_axi depth=16 port=data                              7
#pragma HLS INTERFACE axis port=id                                          8
#pragma HLS INTERFACE axis port=barrier                                     9
#pragma HLS INTERFACE axis port=barrier_rel                                 10
    unsigned int internal_id;                                               11
    unsigned int id0, id1;                                                  12
    unsigned int trigger;                                                   13
    unsigned int zero = 0;                                                  14
    d_type trigger_d_type;                                                  15
    while (1) {                                                             16
        trigger_d_type = data[SCHE_SLAVE_TRIGGER_BASE + htID];              17
        trigger = *((unsigned int *)&trigger_d_type);                       18
        if (trigger == 2) *barrier = 2;                                     19
        if (trigger == 1) {                                                 20
            data[SCHE_SLAVE_TRIGGER_BASE + htID] = *((d_type *)&zero);      21
            d_type arg0_d_type = data[SCHE_SLAVE_ARGV_BASE];                22
            unsigned int arg0  = *((unsigned int *)&arg0_d_type);           23
            d_type arg1_d_type = data[SCHE_SLAVE_ARGV_BASE + 1];            24
```

Listing 5.2 (Cont.): `pagerank` Function of PageRank for hPEs

```
            unsigned int arg1  = *((unsigned int *)&arg1_d_type);                    25
            d_type arg2_d_type = data[SCHE_SLAVE_ARGV_BASE + 2];                     26
            unsigned int arg2  = *((unsigned int *)&arg2_d_type);                    27
            d_type arg3_d_type = data[SCHE_SLAVE_ARGV_BASE + 3];                     28
            unsigned int arg3  = *((unsigned int *)&arg3_d_type);                    29
            d_type arg4_d_type = data[SCHE_SLAVE_ARGV_BASE + 4];                     30
            unsigned int arg4  = *((unsigned int *)&arg4_d_type);                    31
            internal_id = *id;                                                       32
            while (internal_id != 0xFFFFFFFF) {                                      33
                id0 = internal_id >> 16;                                             34
                id1 = internal_id & 0x0000FFFF;                                      35
                kernel(arg0, arg1, arg2, arg3, arg4, id0, id1, data);                36
                internal_id = *id;                                                   37
                if (internal_id == 0xFFFFFFFF) {                                     38
                    *barrier = 1;                                                    39
                }                                                                    40
            }                                                                        41
        }                                                                            42
    }                                                                                43
}                                                                                    44
```

The `pagerank` function (see Listing 5.2) is the entry function for a hardware thread kernel program. When users write different hardware thread kernel programs, the differences lie in the arguments passing. Arguments that are needed to pass to the kernel function are fetched from the local memory via codes from Line 22-31. In this PageRank example, there are totally five arguments. They are initialized within the host program (see Listing 5.1) For other applications, designers can only change these codes to adjust the number of arguments they want to pass. Correspondingly, the number of arguments that are passed to kernel functions needs to change to the caller on Line 36.

Interfaces of hPEs are defined as arguments from the `pagerank` function. There are four kinds of interfaces to this module: `id` port, `barrier` ports, `data` port, and `htID` port, each of which is associated with a bus protocol, which is defined from Line 6-10.

- `id` is implemented in the slave AXI-Stream protocol, which is used to get work group IDs from the ID dispatcher.
- A pair of `barrier` ports are implemented in master and slave AXIS-Stream protocols, respectively. They are used in `pagerank` and `kernel` functions to synchronize with other PEs within a group.

- `data` uses a master AXI memory-mapped protocol that is used for reading and writing data from the global memory and local memory.
- `htID` is assigned a unique number to identify each PE within a group. It is wired to the hPE without special standard protocol.

Line 38-40 in this example shows that a synchronization occurs when all item IDs within this group finish dispatching. The `pagerank` function is to keep polling the trigger that is set by the group scheduler from the local memory. Once the trigger is set, the sPE will fetch the new arguments that are updated by the group scheduler. Then the hPE keeps fetching item IDs from the `id` port until it occurs a `0xFFFFFFFF` that indicates the termination . The 32-bit ID is divided into two 16-bit IDs that stand for a pair of two-dimensional IDs. The two IDs will be passed to the `kernel` function eventually.

Listing 5.3 `kernel` Function of PageRank for hPEs

```
void kernel(unsigned int a_addr,                                                    1
            unsigned int b_addr,                                                    2
            unsigned int c_addr,                                                    3
            unsigned int num_nodes,                                                 4
            unsigned int buf_size,                                                  5
            unsigned int id0,                                                       6
            unsigned int id1,                                                       7
            d_type *data) {                                                         8
    unsigned int a = a_addr >> 2;                                                   9
    unsigned int b = b_addr >> 2;                                                   10
    unsigned int c = c_addr >> 2;                                                   11
                                                                                    12
    unsigned int line_off = id1 * SIZE_PER_PE * num_nodes;                          13
    unsigned int vector_off = id1 * SIZE_PER_PE;                                    14
    int i, j;                                                                       15
    int num_buf_chunk = num_nodes / buf_size;                                       16
    d_type sum[SIZE_PER_PE] = {0.0f, 0.0f};                                         17
    for (i = 0; i < num_buf_chunk; i++) {                                           18
        unsigned int chunk_off = i * buf_size;                                      19
        d_type *ina0 = &(data[a + line_off + chunk_off]);                           20
        d_type *ina1 = &(data[a + line_off + num_nodes + chunk_off]);               21
        d_type *inb  = &(data[b + chunk_off]);                                      22
        memcpy((d_type *)a_buffer0, (const d_type *)ina0, sizeof(d_type) * buf_size);   23
        memcpy((d_type *)a_buffer1, (const d_type *)ina1, sizeof(d_type) * buf_size);   24
        memcpy((d_type *)b_buffer,  (const d_type *)inb,  sizeof(d_type) * buf_size);   25
        for (j = 0; j < buf_size; j++) {                                            26
#pragma HLS PIPELINE                                                                27
            sum[0] += a_buffer0[j] * b_buffer[j];                                   28
            sum[1] += a_buffer1[j] * b_buffer[j];                                   29
            if ((i == num_buf_chunk - 1) && (j == buf_size - 1)) {                  30
                data[c + vector_off] = sum[0];                                      31
                data[c + vector_off + 1] = sum[1];                                  32
            }                                                                       33
```

Listing 5.3 (Cont.): `kernel` Function of PageRank for hPEs

```
        }                                                                    34
    }                                                                        35
}                                                                            36
```

Listing 5.3 shows the `kernel` function of PageRank. The contents of `kernel` functions may differ from one application to another. But the interfaces are similar regarding functionalities. In this example, the first three arguments are the two input data addresses and one output data address, respectively. The next two arguments are the two 16-bit two-dimensional item IDs. This particular application uses the remaining two arguments.

It is noticed that there are optimization techniques that can be used to program hPEs. One way to do this is to cache data. When using this method, the hPE process multiple data elements with one item ID. Line 23-25 uses `memcpy` to burst multiple data elements per transfer. Data elements are stored temporally in the private memories. The other way is to pipeline the execution. The hardware thread kernel program uses a parameter in the source codes to make the HLS tool optimize compilation. This technique is usually to pipeline a `for` loop as shown in Line 27.

Listing 5.4 Definitions of PageRank for hPEs

```
#define SCHE_SLAVE_ARGV_BASE      0x30000020                                 1
#define SCHE_SLAVE_TRIGGER_BASE   0x30000040                                 2
typedef float d_type;                                                        3
#define N               128                                                  4
#define SIZE_PER_PE     2                                                    5
#include "string.h"                                                          6
d_type a_buffer0[N];                                                         7
d_type a_buffer1[N];                                                         8
d_type b_buffer[N];                                                          9
```

Listing 5.4 shows the definition part. Data type is defined as `d_type` that is `float`. `N` is maximum buffer size. In order to reduce redundant data transferring, we process two elements from one item ID so that the `SIZE_PER_PE` is defined as 2, and we defined two buffers to temporarily store matrix input data as `a_buffer`. The `b_buffer` is for the vector input data. These arrays are implemented as private memories for hPEs.

Software Thread Kernel Programs

Listing 5.5 `main` and `kernel` Functions of PageRank for sPEs

```
#include "../../generic/CL/cl.h"                                                    1
#define SIZE_PER_PE     2                                                           2
                                                                                    3
void kernel(                                                                        4
            float *A,                                                               5
            float *B,                                                               6
            float *C,                                                               7
            int num_nodes,                                                          8
            int buf_size) {                                                         9
    unsigned int id0 = getGlobalID(0);                                              10
    unsigned int id1 = getGlobalID(1);                                              11
    unsigned int off  = id1 * SIZE_PER_PE;                                          12
    unsigned int off0 = (off + 0) * num_nodes;                                      13
    unsigned int off1 = (off + 1) * num_nodes;                                      14
    int i;                                                                          15
    float sum = 0;                                                                  16
    for (i = 0; i < num_nodes; i++) {                                               17
        sum += A[id0 + off0 + i] * B[id0 + i];                                      18
    }                                                                               19
    C[id0 + off] = sum;                                                             20
    sum = 0;                                                                        21
    for (i = 0; i < num_nodes; i++) {                                               22
        sum += A[id0 + off1 + i] * B[id0 + i];                                      23
    }                                                                               24
    C[id0 + off + 1] = sum;                                                         25
}                                                                                   26
                                                                                    27
int main() {                                                                        28
    setArgv(0, arg0, float *);                                                      29
    setArgv(1, arg1, float *);                                                      30
    setArgv(2, arg2, float *);                                                      31
    setArgv(3, num_nodes, int);                                                     32
    setArgv(4, buf_size, int);                                                      33
    if (!setjmp(buf)) {                                                             34
        while (1) {                                                                 35
            kernel(arg0, arg1, arg2, num_nodes, buf_size);                          36
        }                                                                           37
    }                                                                               38
    clean_up();                                                                     39
    return 0;                                                                       40
}                                                                                   41
```

Listing 5.5 shows the `main` and `kernel` functions for sPEs. The `main` function is different from that of the hardware kernel program in terms of detailed implementation but shares the same execution flow. sPEs fetch the updated arguments by using a micro `setArgv` that is defined in `cl` library, which is shown from Line 29-33. The micro defined a variable with the given type and assigned the argument to the variable according to the index. Different applications may require a different number of arguments. Users can change the codes on demands. If we compare with the

codes from the hardware thread kernel program, the software thread kernel program does not have a trigger part. This part is implemented by the bootloader instead.

Then the `kernel` function will be executed once it gets an item ID as shown in Line 10-11. Once the sPE receives the termination signal `0xFFFFFFFF`, the `getGlobalID` function will throw an exception that can be caught by `setjmp` in Line 34. Software thread kernel programs are easier to program than hardware ones in terms of footprints. This provides more flexibility to the users. However, hPEs offer more changes for the high performance and high throughputs.

5.4 Related Work

5.4.1 Executables for PolyTasks

For a single type of general-purpose processors (e.g., sPEs in our work), traditional compilation processes generate an executable file that is a binary representation of the instruction stream for this type of processor. Each instruction has an ISA-specific encoding that perfectly matches a decode stage in a given processor, allowing a single compiler to hide ISA-level intricacies from the programmer. In heterogeneous general-purpose systems, compilation flows are required to produce executables that contain multiple instruction streams, each of which is encoded in a particular ISA [59]. Additionally, the distinct instruction streams have to be linked so that references to the same data object refer to the same address in memory. Unfortunately, most executable formats do not specify how to resolve references for common data and code objects between different processor architectures [54].

To overcome this limitation, many previous heterogeneous design flows create augmented compilation tools that are able to handle cross-linking heterogeneous binaries [53, 59, 81, 98, 102] for general-purpose processors. However, modifying modern compilers and linking utilities is an exceedingly difficult process; as compilation frameworks are extremely large and complex. Additionally, maintenance and extension of heterogeneous systems not only implies but requires

the modification and maintenance of multiple compilers and binary utilities. As an example, previous work creates customized extensions to standard executable-linkable file formats (ELFs) to allow for cross-architecture linking and embedding of binary files [53, 59, 102]. These extensions require the modification of compilers to produce ELF files that conform to the new extensions. The extensions defined by both IBM's CESOF [53, 54] and Joglar's HELF [59] aim to create standardized executable formats that enable heterogeneous compilation flows while simultaneously preserving backward compatibility. Other heterogeneous compilation flows prefer to leave compilation tools as is to create new helper utilities that can link and embed heterogeneous binaries by using intermediate files that are encoded in high-level languages (e.g., C). As an example, both NVIDIA's heterogeneous compilation drivers (nvcc) [81] and Arachne's preprocessing compiler [26] produce intermediate files that are encoded in traditional library header files that can be readily linked and embedded using normal compilation tools.

In the PolyPC framework, none of these approaches are applicable. For a PolyPC system with both sPEs and hPEs, it is even harder to have a unified executable format to combine and link ELF files and PR bitstream files, not to mention having custom compilers and linkers to put them together. The helper utilities to embed heterogeneous binaries into a single executable (e.g., for host processor) seem to work at first. But this approach may suffer from low extensibility. All executables have to be embedded into a source code to compile and link with that source code together. When the number of applications increases, it may generate extremely large binaries by using this approach. Furthermore, this approach can only work off-line during the compiling and linking stages. In our framework, we introduce the dynamic executable loader to address issues raised above. PolyTask executables are compiled, loaded and stored separately. This enables off-line generation and run-time loading for all executables.

Table 5.4: Overview of Selected High-level Synthesis Tools

Tools	Owner	License	Input	Output	Year
MaxCompiler	Maxeler	Commercial	MaxJ	RTL	2010
Bluespec	BlueSpec Inc.	Commercial	BSV	System Verilog	2004
Vivado HLS	Xilinx	Commercial	C/C++/SystemC	VHDL/Verilog/SysmteC	2013
Bambu	PoliMi	Academic	C	Verilog	2012
LegUp	U. Toronto	Academic	C	Verilog	2011
DWARV	TU. Delft	Academic	C subset	VHDL	2012

5.4.2 High-level Synthesis Approaches

Hardware description languages (HDLs, such as Verilog and VHDL) are used to design digital system for decades on register transfer level (RTL) of FPGAs and are synthesized into targeted logics. In contrary, high-level synthesis (HLS) tools are expected to convert high-level languages (e.g., C language) to either HDLs or bitstream files. Although HLS approaches for digital design have drawn attentions from both academia and industry for decades, designers are wondering if traditional HDLs can be replaced by them to increase productivity without performance loss. But one thing for sure is that HLS solutions are increasing their shares on commercial markets [74]. One important contribution to this trend is the failure of Moore's law. Designers tend to integrate application-specific hardware accelerators into their processors [10] or to have custom hardware accelerators instead. But designing specialized hardware requires excellent skills in hardware knowledge, which significantly bars the widespread use of custom accelerators. HLS tools provide a promising alternative to this issue.

There are many HLS tools created, some of which are in use and others are abandoned. Tables 5.4 is extracted from [79] to list few HLS tools that are in use.

- **MaxCompiler:** It is an HLS tool especially aiming data flow optimization. The input language MaxJ is a Java-based language. It delivers synthesized logics targeting on hardware platforms from Maxeler.

- **Bluespec:** Bluespec compiler uses Bluespec SystemVerilog (BSV) as input language. BSV is based on System Verilog and is originally from Haskell [80]. BSV is provided to express a circuit: behavior and structure. The behavior is demonstrated by using *Guarded Atomic Actions*, and is based on *Atomic Rules* and *Interfaces*. The guard is a statement that evaluates to a Boolean and if true, the rule fires and executes until completion. BSV is an invented language that is different from transitional hardware and software languages. Therefore, users are required to have experience in BSV.

- **Vivado HLS:** It originates from AutoESL's AutoPilot [20], which was acquired by Xilinx. Vivado HLS was released in 2013 along with the Vivado design suite. It accepts C, C++, and System C as source codes and generates either IP modules that consist of Verilog/VHDL modules or pre-synthesized `dcp` file on targeted Xilinx devices. Vivado HLS is based on module design. The generated modules are embedded into Vivado to comprise a complete system. Vivado HLS provides abundant optimization options including resources, performance, bandwidth, and interfaces. Usually, options are combined with considerations of downstream and upstream modules to deliver optimized circuits.

- **Bambu:** The *bambu* framework [87] is modular and can be extended with new HLS algorithms for research purposes. It requires the C specification and an XML configuration file as input. It produces the HDL description of the hardware implementation and the scripts for logic synthesis.

- **LegUp:** The LegUp compiler [12] is built within LLVM compiler framework [64]. It accepts C codes to synthesize to either hardware logics or hardware accelerators that cooperate with a general-purpose processor communicating via memory-mapped interfaces targeting on Altera devices. LegUp supports Pthreads and OpenMP to synthesize into parallel hardware logics automatically.

- **DWARV:** The DWARV compiler [78] accepts universal C codes as input to generate VHDL codes that can be synthesized on reconfigurable architectures. This work is based on the

CoSy commercial compiler. The compiler can be extended by including standard CoSy or custom engines.

The generation of hPEs within the PolyPC framework is built on Xilinx Vivado HLS tools. However, Vivado HLS does not take OpenCL kernel programs as input. Besides, it is not suitable to design a complete system without other IP modules. Thereby, our framework is intended to cooperate with Vivado HLS. It is the similar approach to Xilinx's OpenCL tool SDAccel [105] that is also built on Vivado HLS. But there are many differences that will affect the performance and productivity. The architectural parallelism within SDAccel is revealed by unrolling `for` loops. Particularly, a compute unit (a group in our implementation) is generated from a single kernel program. In our approach, an hPE is generated from a single kernel program. The source codes and hardware architecture are not independent of each other in Xilinx's approach, which may suffer from resource limitations when implementing a large design.

5.4.3 Automatic System Builder

Several works are focusing on automating system generation on multiprocessor system-on-chip (MPSoC) [30]. In [13], authors present a prototype design flow and set of automated tools that enable designers to construct MPSoC architectures on FPGAs automatically. The tool targets systems with tens of processors, and exists at an abstraction level above vendor-provided EAD tools. ReconOS [72] provides a tool flow to assist their design, as well. Their flow takes source codes of hardware and software programs as input to generate executable binaries, which is very similar to our approaches. But in our design flow, a very important part is the system assembly, which is more flexible than other works. This tool targets heterogeneous systems based on the OpenCL framework. It takes any specifications from users and assembles every component into a system. Other works [33, 73] focus on system automation of general bus-based MPSoC architectures. They take the vendor-provided scripts as an intermediate implementation, on top of which they create an abstraction layer by using high-level languages (e.g., Java). For instance, in

[33] components existing within a system are represented as objects that can be connected to a bus. In this way, the system is constructed in a clear and easy way as designers can focus on system hierarchy instead of implementation details.

Chapter 6

Conclusions

This work explores challenges and solutions of FPGAs as a candidate within heterogeneous systems beyond the multi-core era. FPGAs have advantages of performance over general-purpose processors and productivities over ASIC designs. FPGAs have been playing as a mid-point between processors and dedicated circuits. As the failure of Moore's law, both academia and industry pay more attentions on heterogeneous solutions on FPGAs. Various frameworks on FPGAs have been proposed for designers to use FPGAs easily. The launch of high-level synthesis tools provides an efficient way to increase productivity by compiling high-level languages into RTL logics. However, HLS tools are required to cooperate with proper frameworks supporting parallel programming models to achieve performance improvement along with productivity.

In this work, we propose the PolyPC (**Poly**morphic **P**arallel **C**omputing) framework that targarts parallelizing applications on FPGAs. The PolyPC framework supports the OpenCL parallel programming models by providing a custom hardware platform, software intermediate libraries, and automatic system builders. The OpenCL application programs are converted into executables through the front-end source-to-source transformation and back-end synthesis/compilation to execute on PEs. Coarse-grained parallelism is expressed through architectural supports. Fine-grained parallelism is expressed by PE duplications and loop unrolling within HLS kernel codes. The automatic system builder takes the specifications to generate the hardware platform. The PolyPC framework enables partially reloaded processing elements (PEs) for both sPEs and hPEs. On the one hand, PEs can be generated off-line without changing the static system and loaded into the system during run-time. On the other hand, benefiting from the flexibility of re-loadable PEs, the priority-aware scheduling is enabled to provide designers more flexibilities.

The PolyPC framework is evaluated regarding performance, area efficiency, and multitasking. The results show a maximum 66× speedup over a dual-core ARM processor, and 1043× speedup over a high-performance MicroBlaze with 125× of area efficiency. It delivers a significant improvement in response time to high-priority tasks with the priority-aware scheduling. Overheads of multitasking are evaluated to analyze trade-offs.

In summary, this work investigates and answers questions raised in Chapter 1.

- *Can platform specification details be removed from users' codes and be encapsulated within an automatic system builder?* Configurations of hardware platforms, such as the number of groups, are transparent to users' source codes during run-time. This is quite different from current HLS-based approaches, where hardware configurations are embedded into kernel source codes. The automatic system builder helps to generate the hardware framework from users' specifications. Therefore, the answer to this question is yes.

- *Can the complexity of designing parallel applications be reduced by applying standard parallel programming models on FPGAs?* Yes. The ultimate goal of this framework is to make writing OpenCL programs on FPGAs as quickly as that on GPGPUs. In this work, the complexity is reduced in two ways. Firstly, architectural components help with parallelism and implementation of the OpenCL programming model. Then HLS tools improve design productivity from source codes to RTL circuits through front-end and back-end compilation.

- *What are the benefits when heterogeneous computing resources are used on FPGAs?* Heterogeneous solutions on FPGAs aim to increase productivity without performance loss. hPEs can provide significant performance improvement over soft processors. sPEs have much lower multitasking overheads compared to hPEs. Heterogeneous solutions provide more flexibilities for users to trade-off their different designs.

- *Can the system efficiency be increased by introducing the priority-aware multitasking?* Generally speaking, yes. When multiple kernels are launched, groups keep busy to avoid

idle states. In contrary, when only one kernel is launched, some groups may become inactive while others keep busy as this kernel is about to finish. However, multitasking has its costs. The efficiency may decrease especially when multitasking overheads dominate the wall clock execution time. Therefore, this may require designers to trade-off between flexibility and overheads carefully.

6.1 Future Work

6.1.1 Adoption of More OpenCL Features

In this work, we have implemented basic functionalities of the OpenCL standard. However, some very exciting features from the OpenCL standard have not been implemented yet. These features may improve productivity and increase performance for some applications. For instance, pipes appear from the OpenCL 2.0 specification. It enables direct communication between executing kernel instances. Before pipes were introduced, memory objects are transferred from one kernel instance to another through the host memory. Pipes provide a logic link between kernels for memory objects.

Other features that can be added to our implementation are not from the OpenCL standard. Shared memory between host and compute devices is useful since it avoids potential overheads of data transferring between global memory and host memory. Besides, PEs can keep coherence and consistency of memory objects via host's cache controller to access data from the host memory directly. Moreover, a global TLB or OS page tables can be shared by PEs and host. This further provides a virtualization layer for compute devices on FPGAs.

Further works on the OpenCL standard may include more unified kernel codes for sPEs and hPEs. The ultimate purpose is to make the platform details transparent to users, where designers can use the same kernel codes for one application. This requires further investigations on the heterogeneous compilation flow of general-purpose processors and hardware circuits.

6.1.2 Intelligent Scheduling

A lot of interesting works can be researched for the scheduling part. Two of them are discussed here: multilevel granularity scheduling and run-time adaptive scheduling. Currently, the framework supports coarse-grained multitasking scheduling. The minimal units that can be scheduled are work groups from multiple PolyTheads. Future works can involve fine-grained scheduling within group execution. PEs can switch between different work-items from multiple PolyTheads, which may further improve the efficiency of the whole system.

Another important part in the future work is the run-time adaptive scheduling. The group scheduler can consider more factors when deciding to switch between PolyTasks. More flexible scheduling algorithms can be introduced to trade-off between these factors. Furthermore, instead of off-line profiling, on-line profiling can be implemented in the future. A run-time profiler can monitor and analyze execution information to provide to group schedulers for decisions.

References

[1] Andreas Agne, Marco Platzner, and Enno Lubbers. Memory virtualization for multithreaded reconfigurable hardware. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 185–188. IEEE, 2011.

[2] Abdullah Al-Dujaili, Florian Deragisch, Andrei Hagiescu, and Weng-Fai Wong. Guppy: A gpu-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60. IEEE, 2012.

[3] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. Fgpu: An simt-architecture for fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 254–263. ACM, 2016.

[4] Ali Ali, Mohamad Jomaa, Bashar Romanous, Mageda Sharafeddine, Mazen AR Saghir, Haitham Akkary, Hassan Artail, Mariette Awad, and Hazem Hajj. An operating system for a reconfigurable active ssd processing node. In *Telecommunications (ICT), 2012 19th International Conference on*, pages 1–6. IEEE, 2012.

[5] AMD AMD. Accelerated parallel processing: Opencl programming guide. *URL http://developer.amd.com/sdks/AMDAPPSDK/documentation*, 2011.

[6] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving programming model abstractions for reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):34–44, 2008.

[7] Gregory R Andrews. Foundations of multithreaded, parallel, and distributed programming. 2000. *Wesley, University of Arizona, USA*.

[8] Kevin Andryc, Murtaza Merchant, and Russell Tessier. Flexgrip: A soft gpgpu for fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 230–237. IEEE, 2013.

[9] AMBA Arm. Axi protocol specification, 2003.

[10] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[11] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[13] Eugene Cartwright, Azad Fahkari, Sen Ma, Christina Smith, Miaoqing Huang, D Andrews, and Jason Agron. Automating the design of mlut mpsopc fpgas in the cloud. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 231–236. IEEE, 2012.

[14] Eugene Cartwright, Alborz Sadeghian, Sen Ma, and David Andrews. Achieving portability and efficiency over chip heterogeneous multiprocessor systems. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[15] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[16] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[17] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

[18] Juan Antonio Clemente, Javier Resano, Carlos González, and Daniel Mozos. A hardware implementation of a run-time scheduler for reconfigurable systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(7):1263–1276, 2011.

[19] Tool Interface Standards Committee et al. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 2001.

[20] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[21] Nvidia Coporation. Opencl programming guide for the cuda architecture, version 3.2. *CUDA SDK*, 3, 2010.

[22] Konstantis Daloukas, Christos D Antonopoulos, and Nikolaos Bellas. Glopencl: Opencl support on hardware-and software-managed cache multicores. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 15–24. ACM, 2011.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[25] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, 2012.

[26] Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, 1998.

[27] Hongyuan Ding and Miaoqing Huang. Improve memory access for achieving both performance and energy efficiencies on heterogeneous systems. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 91–98. IEEE, 2014.

[28] Hongyuan Ding and Miaoqing Huang. A unified opencl-flavor programming model with scalable hybrid hardware platform on fpgas. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–7. IEEE, 2014.

[29] Hongyuan Ding and Miaoqing Huang. Achieving energy-efficiency on mpsocs: performance and power optimizations. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, pages 1–7. IEEE, 2015.

[30] Hongyuan Ding and Miaoqing Huang. An automatic design flow for hybrid parallel computing on mpsocs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 275–275. ACM, 2015.

[31] Hongyuan Ding and Miaoqing Huang. Exploiting hardware abstraction for hybrid parallel computing framework. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, pages 1–7. IEEE, 2015.

[32] Hongyuan Ding and Miaoqing Huang. Performance and energy optimization on mpsocs by enabling stt-mram luts. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 35–35. IEEE, 2015.

[33] Hongyuan Ding, Sen Ma, Miaoqing Huang, and David Andrews. Oogen: An automated generation tool for custom mpsoc architectures based on object-oriented programming methods. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 233–240. IEEE, 2016.

[34] Florian Dittmann and Stefan Frank. Caching in real-time reconfiguration port scheduling. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 740–744. IEEE, 2007.

[35] Florian Dittmann and Stefan Frank. Hard real-time reconfiguration port scheduling. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[36] Hadi Esmaeilzadeh, Emily Blem, Renée St Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56(2):93–102, 2013.

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

[38] Samuel H Fuller and Lynette I Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[39] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[40] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal*, 11(4), 2007.

[41] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, and Jürgen Becker. Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[42] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[43] Khronos OpenCL Working Group et al. The opencl specification version 2.2, 2015.

[44] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of ineffciency in general-purpose chips. *Communications of the ACM*, 54(10):85–93, 2011.

[45] Markus Happe, Andreas Traber, and Ariane Keller. Preemptive hardware multitasking in reconos. In *International Symposium on Applied Reconfigurable Computing*, pages 79–90. Springer, 2015.

[46] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[47] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[48] Jim Holt, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister. Software standards for the multicore era. *IEEE micro*, 29(3), 2009.

[49] http://developer.nvidia.com. *Nvidia Developer Zone*, August 2015.

[50] http://msdn.microsoft.com/en-us/directx/default. *IMicrosoft DirectX Developer Center*, August 2015.

[51] http://msdn.microsoft.com/en-us/library/ff471356(v = vs.85).aspx. *Shader Model 5 (Microsoft MSDN)*, August 2015.

[52] http://www.nvidia.com/object/cuda_home_new.html. *CUDA Zone*, August 2015.

[53] IBM. Cell BE Linux Reference Implementation ABI Specification. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/44DA30A1555CBB73872570B20057D5C8. Last accessed May 3, 2017.

[54] IBM. Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D. Last accessed May 3, 2017.

[55] IEEE. *IEEE P1003.1c/D10: Draft Standard for Information Technology - Portable Operating Systems Interface (POSIX)*. IEEE, 1994.

[56] Intel. *Sophisticated Library for Vector Parallelism: Intel Array Building Blocks*. http://software.intel.com/en-us/articles/intel-array-building-blocks.

[57] Intel. *Intel Threading Building Blocks*. http://www.threadingbuildingblocks.org, October 2011.

[58] Aws Ismail and Lesley Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 170–177. IEEE, 2011.

[59] Xavi Joglar, Judit Planas Carbonell, and Marisa Gil. Os paradigms adaptation to fit new architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2008)*, 2008.

[60] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Network and Parallel Computing*, pages 266–275. Springer, 2008.

[61] Wooyoung Kim and Michael Voss. Multicore desktop programming with intel threading building blocks. *IEEE software*, (1):23–31, 2011.

[62] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[63] Laszlo B Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3):144–149, 2002.

[64] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[65] Ilia Lebedev, Shaoyi Cheng, Austin Doupnik, James Martin, Christopher Fletcher, Daniel Burke, Mingjie Lin, and John Wawrzynek. Marc: A many-core approach to reconfigurable computing. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 7–12. IEEE, 2010.

[66] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 1–14. ACM, 2016.

[67] Shiming Li, Miaoqing Huang, Hongyuan Ding, and Sen Ma. A hierarchical memory architecture with noc support for mpsoc on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 173–173. IEEE, 2014.

[68] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. Openrcl: low-power high-performance computing with reconfigurable devices. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 458–463. IEEE, 2010.

[69] E Lubbers and Marco Platzner. Reconos: An rtos supporting hard-and software threads. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 441–446. IEEE, 2007.

[70] Enno Lubbers and Marco Platzner. A portable abstraction layer for hardware threads. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 17–22. IEEE, 2008.

[71] Enno Lubbers and Marco Platzner. Cooperative multithreading in dynamically reconfigurable systems. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 551–554. IEEE, 2009.

[72] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, 2009.

[73] Sen Ma, Hongyuan Ding, Miaoqing Huang, and David Andrews. Archborn: an open source tool for automated generation of chip heterogeneous multiprocessor architectures. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, pages 1–6. IEEE, 2015.

[74] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[75] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[76] J Quirm Michael. Parallel programming in c with mpi and openmp. *Dubuque, IA: McGraw-Hill*, 2004.

[77] V. Mirian and P. Chow. Ut-ocl: an opencl framework for embedded systems using xilinx fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[78] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 619–622. IEEE, 2012.

[79] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[80] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[81] NVIDIA. CUDA Compiler Driver NVCC. http://www.nvidia.com/object/cuda_documentation_stage.html. Last accessed May 3, 2017.

[82] OpenMP. *API Specification for Parallel Programming*. http://openmp.org/wp/openmp-specifications, October 2011.

[83] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[84] Peter S Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[85] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pages 35–42. IEEE, 2009.

[86] Alexandros Papakonstantinou, Yun Liang, John A Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W Hwu, and Jason Cong. Multilevel granularity parallelism synthesis on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 178–185. IEEE, 2011.

[87] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.

[88] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[89] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. Improving the efficiency of run time reconfigurable devices by configuration locking. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 264–267. IEEE, 2008.

[90] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[91] Rafat Rashid, J Gregory Steffan, and Vaughn Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 20–27. IEEE, 2014.

[92] Marco D Santambrogio, Vincenzo Rana, and Donatella Sciuto. Operating system support for online partial dynamic reconfiguration management. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 455–458. IEEE, 2008.

[93] K Schloegel, G Karypis, and V Kumar. The sourcebook of parallel computing. *San Francisco, CA, USA: Morgan Kaufmann Publishers Inc*, pages 491–541, 2003.

[94] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[95] Kirk Skaugen. Petascale to exascale: Extending intels hpc commitment. In *International Supercomputer Conference. Available online at http://download. intel. com/pressroom/archive/reference/ISC_2010_Skaugen_keynote. pdf*, 2010.

[96] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):14, 2008.

[97] Apache Spark. Apache spark: Lightning-fast cluster computing, 2015.

[98] Bjarne Steensgaard and Eric Jul. *Object and native code thread mobility among heterogeneous computers (includes sources)*, volume 29. ACM, 1995.

[99] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[100] Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. An fpga run-time system for dynamical on-demand reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 135. IEEE, 2004.

[101] Steve Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine*, 35(2):46–55, 1997.

[102] Perry H Wang, Jamison D Collins, Gautham N Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *ACM SIGPLAN Notices*, volume 42, pages 156–166. ACM, 2007.

[103] Michael Wolfe. Compilers and more: Knights ferry versus fermi. *HPCwire, Aug*, 2010.

[104] Xilinx. Microblaze processor reference guide. *reference manual*, 23, 2006.

[105] Xilinx. Sdaccel environment user guide, v2016.3. 2016.

[106] AXI Xilinx. Reference guide. *Xilinx Inc*, 1999.

[107] Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Rapid application development on multi-processor reconfigurable systems. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 285–290. IEEE, 2010.