12-2014

Optimizing Performance and Scalability on Hybrid MPSoCs

Hongyuan Ding
University of Arkansas, Fayetteville

Optimizing Performance and Scalability on Hybrid MPSoCs

Optimizing Performance and Scalability on Hybrid MPSoCs

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Hongyuan Ding
Shandong University
Bachelor of Science in Electrical Engineering, 2012

December 2014
University of Arkansas

This thesis is approved for recommendation to the Graduate Council

_____

Dr. Miaoqing Huang, Ph.D.
Thesis Director

_____

Dr. David Andrews, Ph.D.
Committee Member

_____

Dr. Christophe Bobda, Ph.D.
Committee Member

Abstract

Hardware accelerators are capable of achieving significant performance improvement. But designing hardware accelerators lacks the flexibility and the productivity. Combining hardware accelerators with multiprocessor system-on-chip (MPSoC) is an alternative way to balance the flexibility, the productivity, and the performance. However, without appropriate programming model it is still a challenge to achieve parallelism on a hybrid (MPSoC) with with both general-purpose processors and dedicated accelerators. Besides, increasing computation demands with limited power budget require more energy-efficient design without performance degradation in embedded systems and mobile computing platforms. Reconfigurable computing with emerging storage technologies is an alternative to enable the optimization of both performance and power consumption.

In this work, we present a hybrid OpenCL-like (HOpenCL) parallel computing framework on FPGAs. The hybrid hardware platform as well as both the hardware and software kernels can be generated through this an automatic design flow. In addition, the OpenCL-like programming model is exploited to combine software and hardware kernels running on the unified hardware platform. By using the partial reconfiguration technique, a dynamic reconfiguration scheme is presented to optimize performance without losing the programmable flexibility.

Our results show that our automatic design flow can not only significantly minimize the development time, but also gain about 11 times speedup compared with pure software parallel implementation. When partial reconfiguration is enable to conduct dynamic scheduling, the overall performance speedup of our mixed micro benchmarks is around 5.2 times.

Acknowledgment

I would like to thank my advisor Dr. Miaoqing Huang for offering me the opportunity to work with him, and for his patient guidance and generous support. I also would like to thank Dr. David Andrews, and Dr. Christophe Bobda for serving as my committee members, and broadening my knowledge. Finally, I would like to thank my lab mates for their kind help during the last three semesters.

Table of Contents

Chapter 1

Introduction

As mobile devices (e.g., mobile phones, tablets, and wearable computers) become pervasive, new demands and challenges emerge in this field [15]. On the one hand, most mobile devices are powered by batteries, which provide a quite tight power budget for carrying out the computation on them. On the other hand, more and more highly computation-intensive applications (e.g., image processing, high-definition games) are deployed on mobile devices. It becomes critical to provide a solution that can achieve both high performance and energy efficient for mobile devices, and in a broader range, the embedded systems.

On one hand, state-of-the-art Field-Programmable Gate Arrays (FPGAs) are capable of hosting multiple general-purpose processors as well as specified hardware logic on a single chip, which makes it a promising alternative for or an addition to mobile computing. With the help of advanced EDK tools, software designers without rich experience on hardware design can build a workable embedded system-on-chip (SoC) with either single or multiple processors on an FPGA board within minutes. But the potential of hardware acceleration is far from being fully utilized because the performance of general-purpose processors is typically much worse than the hardware acceleration logic. Although hardware designers can design dedicated hardware accelerators by using hardware description languages (HDLs), designing hardware accelerators is still full of challenges when it comes to debugging, optimizing, and integrating. High-level synthesis (HLS) [17] is an alternative way to achieve the balance between costs and performance gains. C/C++ codes with necessary optimization and constrain parameters can be directly synthesized into RTL codes. In addition, appropriate interfaces are automatically generated to connect with external buses. However, there is still a huge gap between using hardware accelerators and software programs when putting them together into a unified framework by leveraging appropriate programming models and system structures.

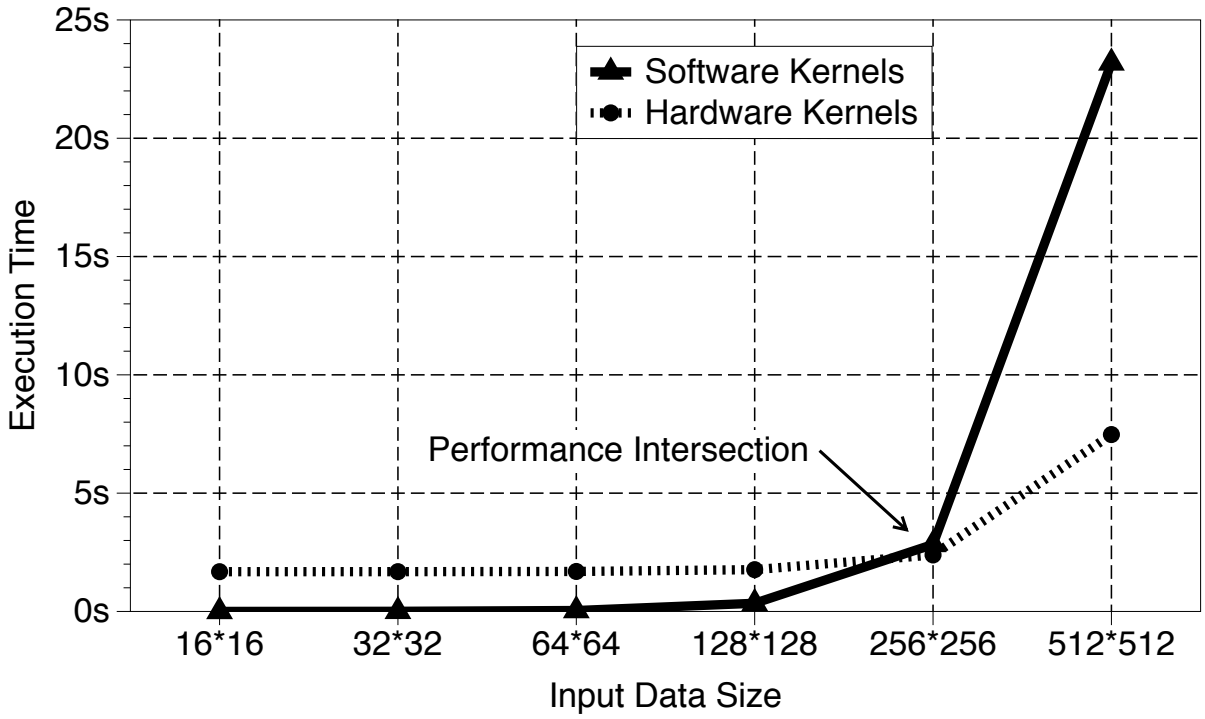


Figure 1.1: Execution time of matrix multiplication by using software and hardware kernels for different input sizes.

On the other hand, software running as executable files can be re-loaded between memories without disturb the current system. However, hardware running on FPGAs are pre-programmed into SRAM storing the bitstream files. Usually, changing what runs on an FPGA will results in the terminating of current system. Thanks to the recent partial reconfiguration techniques in modern FPGAs, parts of FPGA logics can be re-programmed by loading corresponding bitstream files to these parts. In this way, the FPGA can be partially re-programmed without reseting the whole chip. By using partial reconfiguration techniques, designers can dynamically load different hardware accelerators that co-operating with software running on the same chip.

In this work, we propose our hybrid parallel computing framework [5] to support dynamic task-level scheduling between general-purpose processors and dedicated hardware accelerators to achieve the optimization of both performance and scalability consumption without losing the programmable flexibility [6]. As a motivating example, we examine the application of matrix multiplication running on our hybrid platform. In the matrix multiplication $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, both $\mathcal{A}$ and $\mathcal{B}$ are

square matrices and of the same size. Each matrix element is a 32-bit floating-point number. Hardware implementation with partial reconfiguration enabled and software implementation are tested, respectively. Execution time of two implementations varies among different input sizes as shown in Figure 1.1. When dealing with small input sizes, the advantages of hardware acceleration are hidden due to the partial reconfiguration overhead. However, as the input size increases, hardware implementation starts outperforming software. Regarding the performance, this observation provides the trade-off between hardware and software implementations when both implementations are available for scheduling.

1.1 Thesis Contributions and Organization

This thesis explores the methodologies to optimize both performance and scalability of a hybrid parallel computing framework. Detailed implementation of this framework is discussed. An automatic design flow and hybrid parallel programming model are proposed for easily using this framework. We examined the performance and scalability of our framework. Furthermore, by conducting different micro benchmarks, partial reconfiguration techniques and task-level scheduling algorithms are exploited to optimize both performance and scalability.

The major contributions of this work are as follows:

- A hybrid parallel computing framework is proposed. The parallel programming model on the framework leverages many features from OpenCL. The kernel programs can be compiled into software kernels and hardware ones with heterogeneous computation resources.
- An automatic generation flows to help designer build the customized hybrid hardware platform. A unified design flow to automatically generate both software and hardware kernels running on the same hardware platform. In addition, the corresponding SDK projects are built and compiled.

- An OpenCL-like programming model to unify hybrid kernels (i.e., hardware kernels and software kernels) to carry out execution under a specially designed multiprocessor system-on-chip (MPSoC).

- Extending partial reconfiguration techniques on this co-design platform, hardware kernels for different applications can be re-loaded into FPGA by scheduling algorithms to achieve the optimization of performance and flexibility.

This thesis fully demonstrates the hardware architecture, and programming model of a hybrid parallel computing framework. Methodologies and experimental results are discussed to show the optimization of both performance and scalability on our system. Chapter 2 provides some background information and discusses related work on OpenCL-like programming model for Multiprocessor system-on-chips (MPSoCs). In Chapter 3, system design will be introduced with details into four different aspects: memory model, hardware architecture, an automatic design flow, and corresponding parallel programming model. Chapter 4 discusses the experimental methodologies and results. Firstly, the scalability and performance of our hybrid parallel computing framework is tested and discussed. Then, we explore different micro benchmarks running on our hybrid parallel computation system. Partial reconfiguration techniques, as well as the dynamic scheduling methods, is enabled to further improve the performance and flexibility. Finally, chapter 5 provides conclusions and directions for future work.

Chapter 2

Background and Related Work

2.1 OpenCL on FPGAs

In our work, we leverage an OpenCL-like programming model for high-level software and hardware kernels design. OpenCL [9] is a framework to design parallel applications on various computation resources (CPU, GPU, and FPGA). Programming using OpenCL consists of two steps. The first step is to define a computing platform on which the application will execute. In OpenCL's term a platform consists of one host processor and multiple compute devices. The second step is to assign the computation tasks to each compute device and specify the dependencies among them through the explicit data transfer between these tasks. The information of both platform and the corresponding data affinity and parallelism is explicitly presented and easily extracted in OpenCL framework.

Due to the appealing feature of OpenCL in terms of architecture representation, it has been adopted in many related work to define the multicore architecture. In [11] a direct implementation of OpenCL framework on Xilinx FPGA is presented. The OpenRCL machine consists of an array or processing elements, their on-board local memory, and an off-chip global memory. This work is generalized in the following work "MARC" [10]. In MARC (Many-core Approach to Reconfigurable Computing) an application is mapped to the MARC template, which consists of one control processor and 48 algorithmic processing cores. These 48 processing cores can be parameterized to fit the application requirements. The MARC approach is similar to the approach presented in [16], in which the authors develop a tool kit for embedded designers, including compiler, mapper, designers. The FlexCL approach proposed in [4] is used to configure the parameters of the open-source MicroBlaze-Lite software processor based on the application description. Chin and Chow introduced a memory infrastructure for FPGAs designed for OpenCL style computation [3]. An Aggregating Memory Controller is implemented in hardware and aims to maximize bandwidth to

external, large, high-latency, high-bandwidth memories by finding the minimal number of external memory burst requests from a vector of requests.

RAMPSoC is a framework for generating an MPSoC system composed of multiple microprocessors and hardware accelerators for executing an algorithm [7]. An alternative to RAMPSoC is introduced in [12], which allows the runtime reconfiguration of heterogeneous processor cores with a finer granularity. In [8] OpenCL is used to design application-specific processors. Given an application written in OpenCL, an application-specific processor is generated to execute the application. In [14], the SOpenCL architectural synthesis tool is presented. The SOpenCL tool takes an OpenCL application and maps it to a custom designed hardware circuit. In this sense, it is still one variant of C-to-gate compiler, which is not the goal of this work. This approach is generalized in [13]. A similar approach is proposed in [2] in which OpenCL kernels are translated into CatapultC code for high-level synthesis. Altera has introduced its own OpenCL SDK [1]. Starting with OpenCL as an input, the SDK generates the host executable and the hardware accelerators that carry out the computation.

Chapter 3

Hybrid System Design

We exploit a hybrid parallel programming model that is similar with the OpenCL. In this work, we call our hybrid system HOpenCL (Hybrid OpenCL). In this chapter, system design will be introduced with details into four different aspects: memory model, hardware architecture, an automatic design flow, and corresponding parallel programming model. A two-level memory hierarchy with both distributed and shared memories is leveraged by HOpenCL. Besides, dedicated DMA modules are used for fast data movement. In hardware architecture part, three different approaches are discussed. Using the automatic tools following the design flow, both hardware platform architecture, and software executable files can be generated to run on FPGAs. At last, designers can use the hybrid parallel programming model to write both software and hardware kernels.

3.1 Memory Model

The current OpenCL specification is heavily influenced by GPU programming. In OpenCL a platform consists of one host processor and several compute devices, each of which contains multiple compute units. A single compute unit is comprised of multiple processing elements. An OpenCL function, called "kernel", is assigned to one compute device during the runtime. A kernel function is implemented as a grid of work-items, each of which can be considered as a thread. The work-items in a grid are broken into work-groups, each of which is scheduled to execute on a compute unit. Every work-item is physically executed on a processing element. Through this mapping of compute device↔kernel, compute unit↔work-group, processing element↔work-item, the data parallelism in an application is explicitly expressed. On the HOpenCL platform, a kernel can run on either general-purpose processors or hardware accelerators. Then the kernel is called software kernel or hardware kernel, respectively.

Figure 3.1 demonstrates the memory model in HOpenCL in which all Group Computation

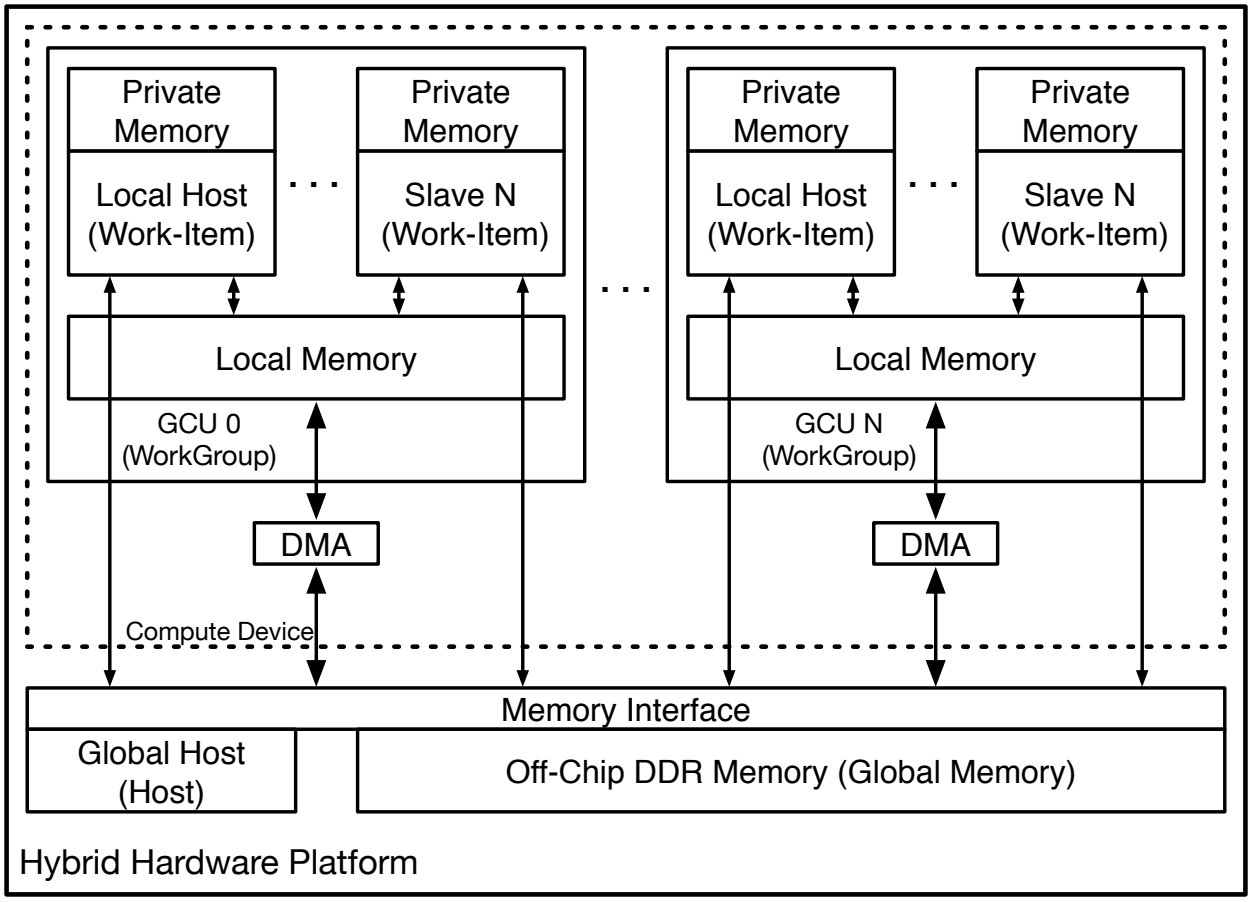


Figure 3.1: Memory model in HOpenCL.

Units (GCUs) combiningly form a compute device. Similar to OpenCL, each processor in the GCU has its own private memory. All processors in one GCU share the same local memory. Global memory is visible to all GCUs. Global host processor is used to coordinate kernel execution, data movement, and arguments passing. Different from OpenCL, a specified local host processor in each GCU is assigned to coordinate data movement and work-group execution. Each GCU is also assigned with a local direct memory access (DMA), which is in charge of transferring block data from the local memory to the global memory and vice versa to avoid frequent bus requests to the global memory. DMA can be called only once by the local host at the beginning of the execution of each work-group of work-items to avoid redundant data movement. In stead of having two separate global memories for the host and the compute device respectively in OpenCL, there is a union global memory shared by both the global host and all GCUs in HOpenCL. The advantage

of a single global memory is to eliminate memory copies between the two global memories in OpenCL. Data requests to global memory are further optimized using cache.

3.2 Platform Architecture

From the first approach to the last one, new features are added to improve the performance and productivity of HOpenCL system. In the first approach, all computation resources consist of general-purpose processors. Software kernels running on processors are executable files generated from kernel programs. In the second approach, besides general-purpose processors, dedicated hardware accelerators (namely, hardware kernels) are also responsible for computation tasks. In the last approach, feature of partial reconfiguration (PR) in FPGAs are used for dynamically scheduling. PR regions can be downloaded with different hardware kernels.

Figure 3.2 shows the structure of the basic HOpenCL hardware platform and the components inside each GCU, respectively. There are two levels of AXI buses, local bus and global bus, connecting local devices and global devices, respectively. Besides AXI buses, AXI-Stream is used to conduct operations that are not memory-mapped. Since AXI-Stream is a simple point-to-point connection, it does not waste AXI interconnection resources and increases no arbitration time. Inside each GCU, the local host is a general-purpose processor. Other slaves can be configured as either hardware accelerators for hardware kernels or general-purpose processors running software kernels.

3.2.1 First Approach: General-purpose Processors

In the first approach, we only have general-purpose processors as computation resources. In other words, there are no hardware kernels assigned into GCUs. Including the local host, software kernels are running on general-purpose processors as executable files. There are two types of hosts: global host and local host. When receiving a task, global host divided the problem space into small size of chunks. Each chunk will be allocated to each GCU. After gaining necessary information from global host, each local will coordinating detailed execution inside each GCU. Smaller data

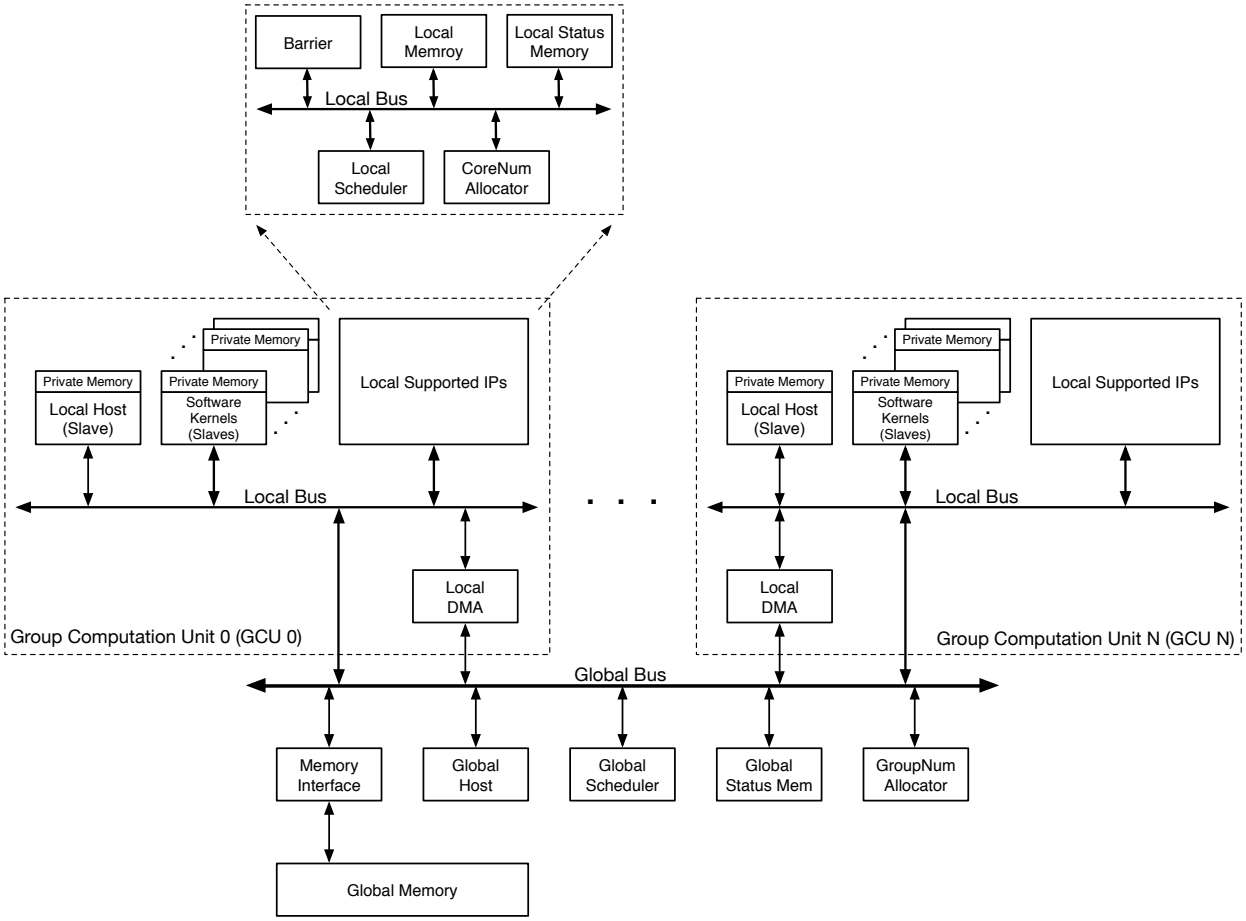


Figure 3.2: System Architecture with General-purpose Processors.

items are dispatched to each slaves inside GCUs. Besides scheduling execution, local host also take part in computation. In other words, global host is responsible for coordinating execution across GCUs, and local host for dispatching data items inside GCUs. We have local supported IPs inside each GCU to provide accurate functionality. Important components with corresponding principles are discussed as follows.

Global Scheduler: Global scheduler provides a pair of numbers to map tasks on each GCU. Similar to OpenCL where a problem space is divided into many N-dimension work-groups, HOpenCL can support up to 2-D group size. Before proceeding to a new group, the local host will request a new group ID from the group ID generator, and assign it to each slave in its GCU. Since the number of GCUs is limited, the number of groups may exceed the number of GCUs when a large problem space is divided into many groups. Global scheduler will assign every group to each GCU

following the principle of first-come, first-served (FCFS).

Group Number Allocator: In application developers' view, all GCUs are symmetrical. Every GCU runs the same kernel program. However, in terms of hardware abstraction, every GCU needs to identify itself. In this case, a unique group number will be assigned by the group number allocator to each GCU. This number is received by the local host to achieve two objectives. The first one is to identify local-owned devices including DMA, local memory, and others that cannot be shared by other GCUs. The second one is to switch local daemon programs when there are hardware kernels running in the current group.

Global Status Memory: Global status memory (GSMem) stores a few synchronized signals and global shared information. Specifically, when trying to notify GCUs to start running kernels, the global host will write a trigger value to the associated locations in GSMem. Daemon programs running on local hosts will be polling these signals before executing kernels. In addition the global host will write kernel arguments, which need to be passed to each GCU, into the GSMem.

Local Scheduler: Within every GCU, each slave processor can request a 2-D local ID from the local scheduler after a task is assigned to a slave. HOpenCL supports up to two-dimensional problem size. Every ID request will be queued into the local scheduler with the principle of FCFS. In other word, every slave in one GCU has the equal opportunity to get a local ID. Those slaves running faster will get more IDs than those slower slaves.

Figure 3.3 shows the detailed implementation of schedulers (including both local and global schedulers). The inputs of schedulers are connected with different PEs through AXI-Stream bus. For local schedulers, PEs are considered as all slaves in each GCUs. Besides, all local hosts among GCUs are required to connect with global scheduler. PEs will send their own core IDs to schedulers. The round-robin arbitration is done inside AXI-Stream interconnection. Based on the order of core IDs storing in FIFO. ID Generator generates pairs of IDs to associating PEs.

Core Number Allocator: Since the local host does more jobs than other slaves, the local host needs to identify itself from others. Furthermore, once there are hardware kernels running in the current group, hardware accelerators and general-purpose processors will be identified with the

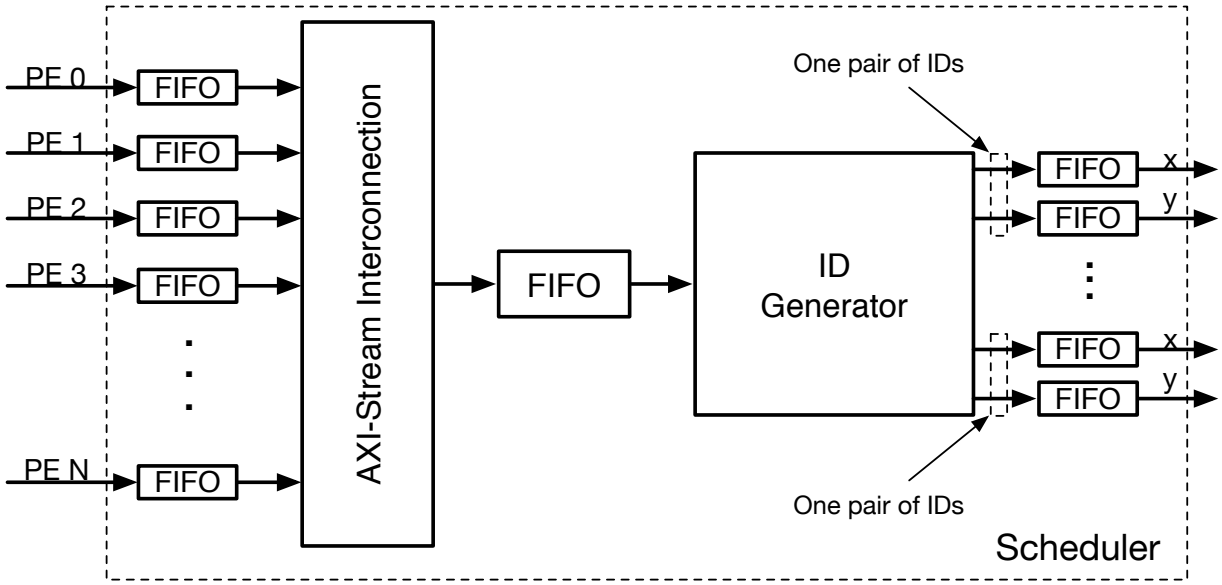


Figure 3.3: Detailed Structures of Scheduler.

core number allocated by the core number allocator.

Local Status Memory: Local status memory (LSMem) stores group running information. Since each DMA operation can only be carried out once for each work-group of work-items, after each DMA operation the local host will register this operation in LSMem to avoid executing it again. When noticing that the group ID generator FIFO is empty, the local host will write a value in LSMem to notify other slaves that all work-items in the current work-group have been finished.

Barrier: Barrier provides a hardware synchronization mechanism within each GCU. When a barrier request is received by a barrier from any work-item in one group, other work-items cannot get the released signals from this barrier until they all send the barrier requests to the barrier. Once the released signals are obtained by work-items, they will continue proceeding.

3.2.2 Second Approach: Hybrid Accelerators

Although programming with general-purpose processors are easy, in most cases, the demands for high performance are not met by using general-purpose processors. In this approach, in order to improve the overall performance, we added dedicated hardware kernels co-operating with software kernels. Figure 3.4 demonstrated how hybrid accelerators are organized with original design.

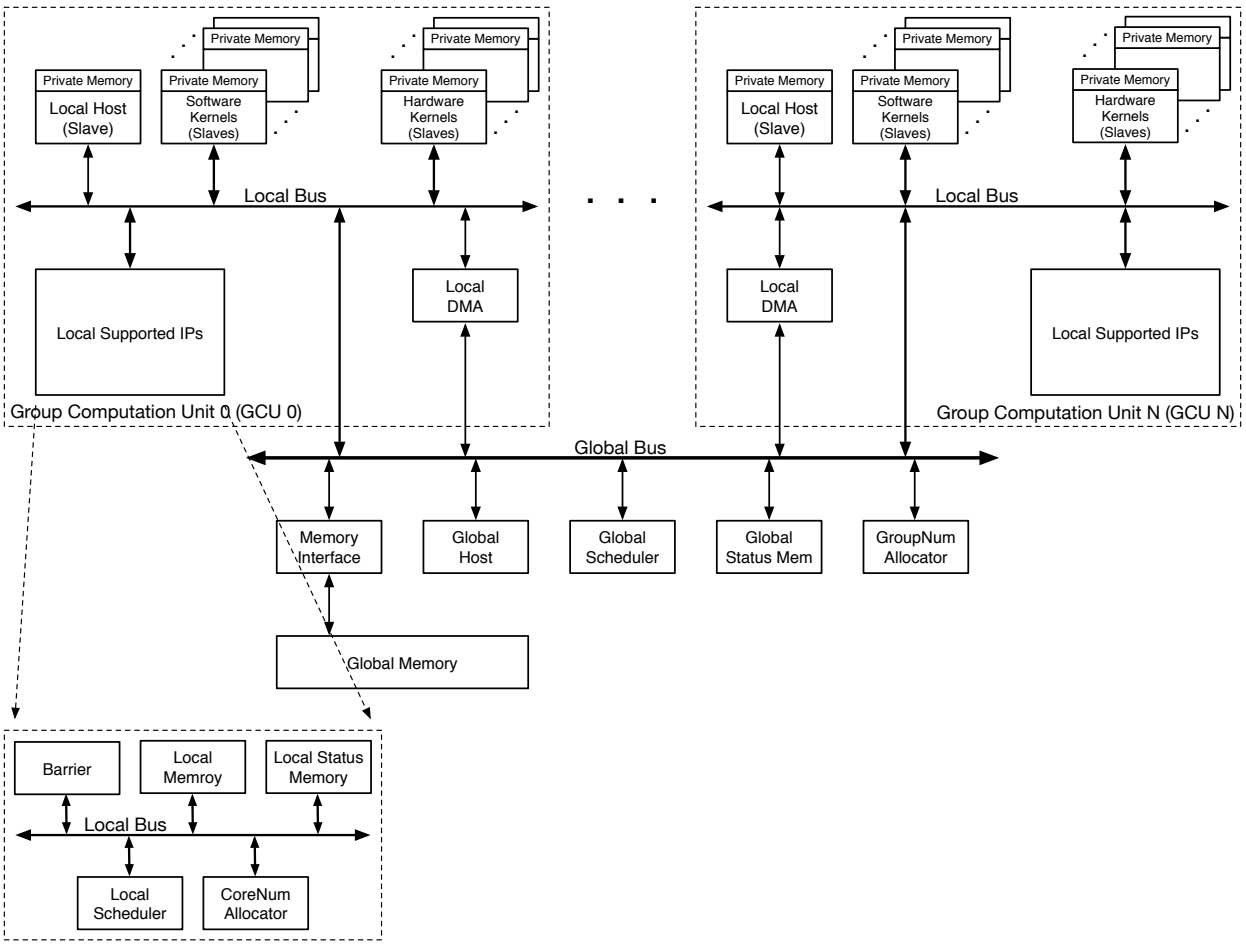


Figure 3.4: System Architecture Adding Hardware Kernels.

Besides hardware kernels, functionalities of all components in the first approach are the same in the second one. In this way, input and output ports of hardware kernels are designed to keep compatible with general-purpose processors in the first design.

Table 3.1 lists all the port on hardware kernels. Port *data* is used to store computation outputs of hardware kernels to memory, and retrieve data from memory. It is connected with local bus. Other ports are connected with local supported IPs to provide functionalities of software APIs.

3.2.3 Third Approach: Enabling Partial Reconfiguration

Partial reconfiguration in FPGAs is used to dynamically downloading bitstream files for parts of areas on FPGA without reseting the whole FPGA chip. In this work, in order to provide flexibility

Table 3.1: Input and Output Ports of Hardware Kernels.

Ports	Type	Direction
data	AXI-Full Master	Input & Output
barrierOut	AXI-Stream	Output
barrierIn	AXI-Stream	Input
coreNum	AXI-Stream	Input
localID0	AXI-Stream	Input
localID1	AXI-Stream	Input
groupNum	AXI-Stream	Input

of running different hardware kernels during run-time without programming FPGA chip again, PR is enabled with supported software and hardware features. As figure 3.5 shows, gray color block are configured as PR regions, as well as hardware kernels. Since software kernels run on executable files that can redirected to different memory locations, there is no need to add PR features on software kernels. Every hardware kernel in all GCUs has its own specified PR region on FPGAs. In other word, each hardware kernel of every location will have its own downloadable bitstream file.

The third approach is the extended version of the second one. Besides PR regions for hardware kernels, a separate *ICAP* module is connected to the global bus. This IP is vendor-provided to retrieve bitstream data from AXI bus to partially program FPGAs. Global host will be responsible for scheduling which bitstream is used to download for current task.

3.3 Automatic Design Flow

Figure 3.6 demonstrates hardware design and software design flow, respectively. Currently, automatic design flow can only support approach one. Hardware configuration, including the number of GCUs, the size of local memory, and the configuration of hardware kernels, will be given as the input of the TCL scrip generator. Hardware platform is independent with software programming. In other words, how you configure the hardware platform mainly depends on the available hardware resources on the target FPGAs. Once the hardware platform is generated, the local scheduler

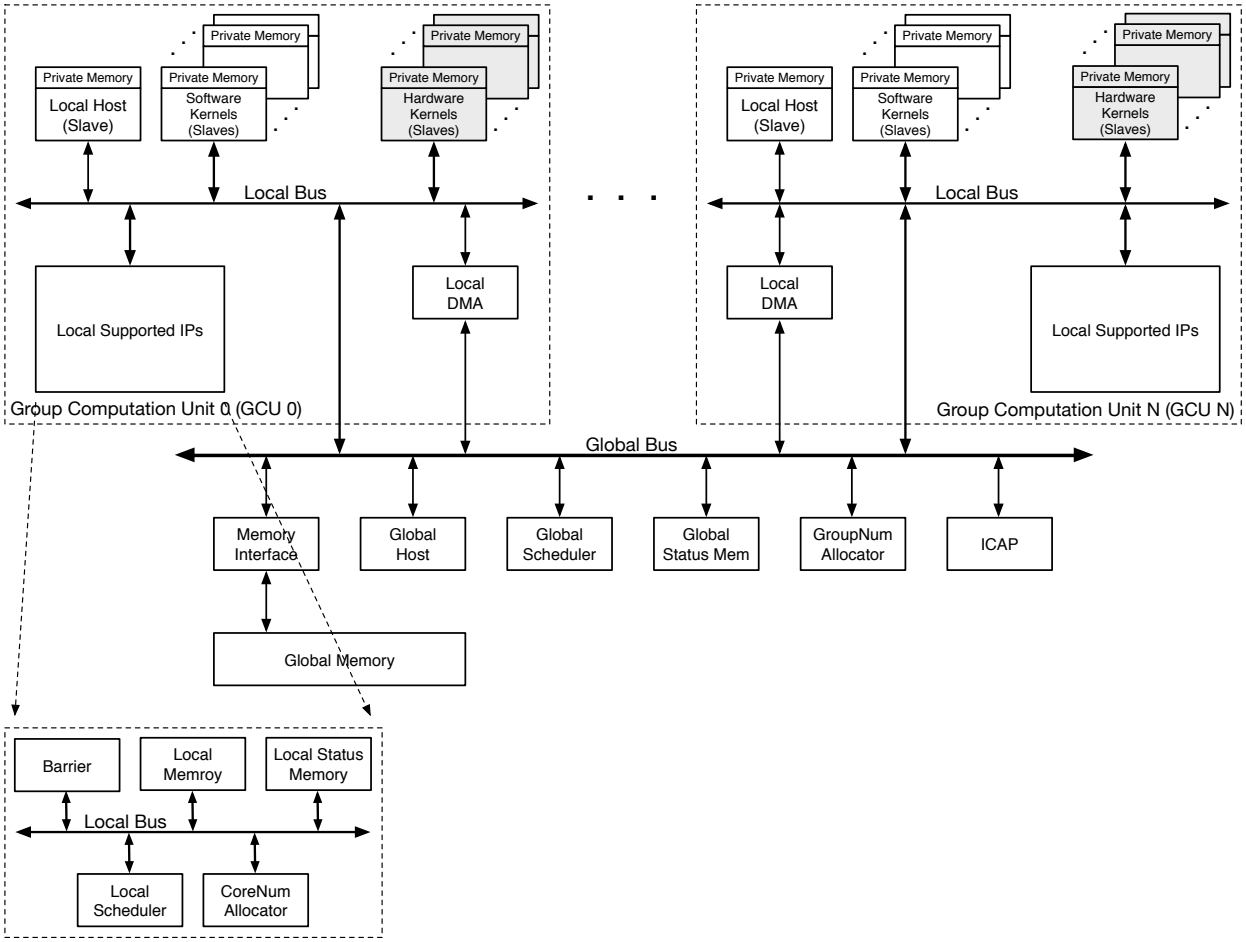


Figure 3.5: System Architecture Enabling Partial Reconfiguration.

as well as the global scheduler will try their best to maximize the performance of the applications running on the platform. Combining hardware platform, HOpenCL hardware IPs, and hardware kernels, the platform bitstream will be generated.

Kernel programs can be either translated into hardware kernel IP by using Vivado HLS tools, or into ELF files by using MicroBlaza compiler. Configuration headers include all configurations for both hardware and software libraries. Memory offsets of polling signals, mutex variables, and shared configurable values storing in LSMem and GSMem will be listed into configuration headers. Although hardware and software libraries showing in Table 3.2 are the same, the implementation of hardware libraries in HLS source codes will be much different from software libraries. Figure 3.7 shows how barriers are implemented in hardware kernel programs. Since there are no buffers

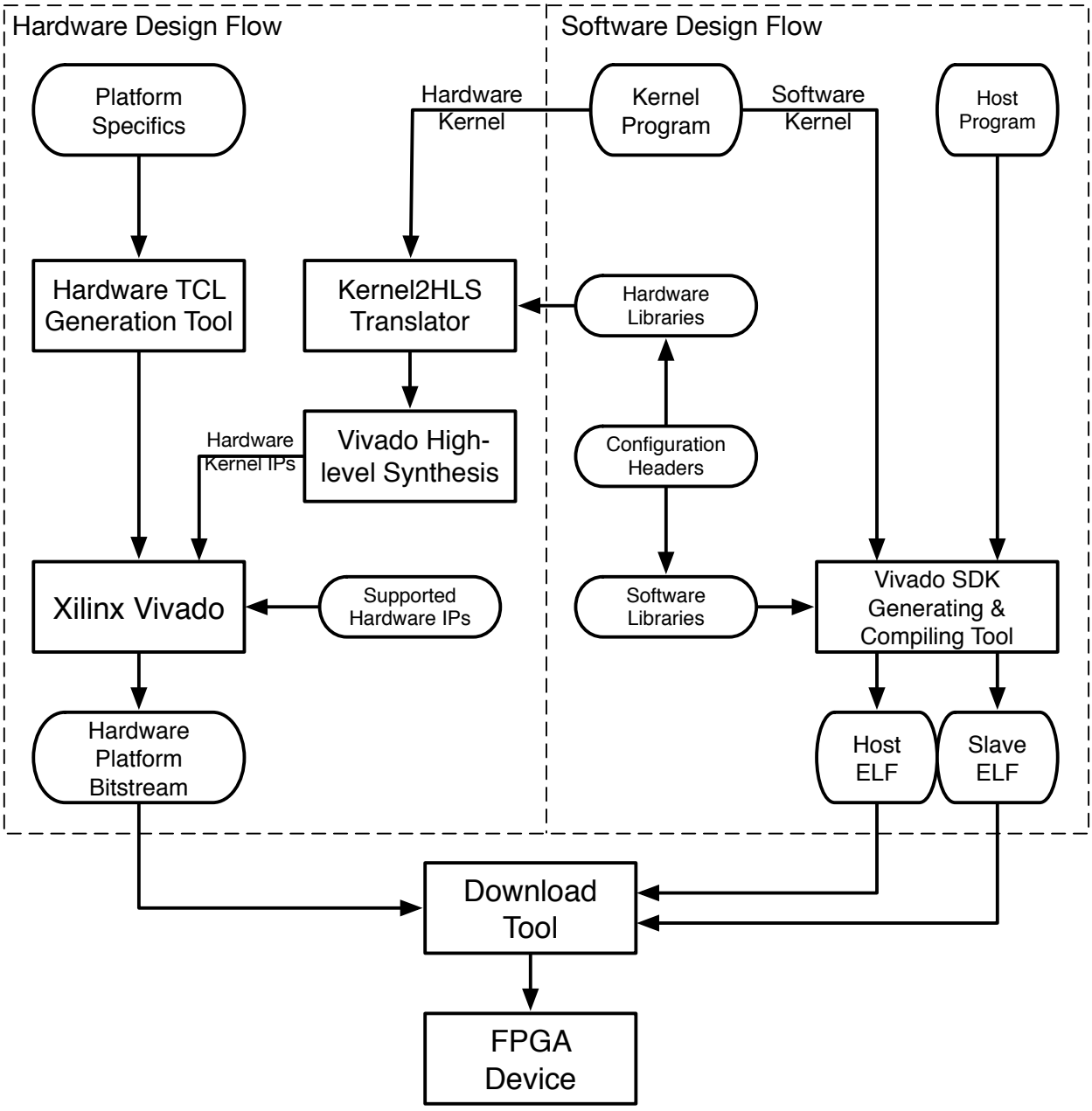


Figure 3.6: Automatic Design Flow for Both Hardware and Software Sides.

Table 3.2: Our software and hardware libraries compared to OpenCL ones.

Execution Scope	OpenCL	Our Versions	
		Software Libs	Hardware Libs
Kernel	get_local_id()	getLocalID()	getLocalID()
	get_gourp_id()	getGroupID()	getGroupID()
	get_global_id()	getGlobalID()	getGlobalID()
	get_local_size()	getLocalSize()	getLocalSize()
	–	getKernelArg()	getKernelArg()
	barrier()	barrier()	barrier()
	–	simpleDMA()	simpleDMA()
Host	–	setLocalSize()	–
	–	setGroupSize()	–
	clSetKernelArg()	setKernelArg()	–
	–	KernelStart()	–
	–	KernelFinish()	–

```
unsigned int bIn;
*barrierOut = BarrierID;
bIn = *barrierIn;
if (bIn == BarrierID) {
  //Execution body after barrier operation;
}
```

Figure 3.7: Barrier implementation in hardware kernel.

between barrier IP and hardware kernels, reading data from AXI-stream will be blocked until the barrier IP releases the current barrier. The *IF* statement guarantees that HLS will not optimize the following statements; otherwise, barrier operations will not work correctly. The implicit translation is done by *Kernel*2*HLS Translator*.

3.3.1 Automatic Tools

With the help of our automatic design tools, by only giving system specifics, and corresponding kernel and host programs, designers can (1) generate hardware platform, (2) generate software kernels by building and compiling SDK projects, and (3) generate hardware kernels by converting kernel programs into HLS source files. Xilinx 7-series FPGA devices are supported in our tools.

As shown at the top of Figure 3.6, platform specifics, kernel programs, and host programs are

given to our automatic design flow as input. Platform specifics are used to build the hybrid hardware platform. Kernel programs can be compiled into either hardware kernel IPs (i.e., hardware accelerators) or software kernels running on MicroBlazes. Host program is compiled to ELF file running on the global host processor. We write four scripts (including hardware platform building tool, hardware kernel IP generation tool, SDK generation tool, and download tool) to link different parts as a complete automatic design flow. The following shows an example to demonstrate how our automatic design flow works.

(1) First of all, if we want to add hardware kernels to the platform, we need to generate the hardware kernel IP from the kernel program that is written by designers in software style. If we do not want to add hardware kernels this step can be ignored.

```
./kernel_generate.py Convolution
```

Convolution is the kernel name that is the same with the top function name in HLS tools. As showing in Figure 3.6 hardware libraries will be added into kernel program. Table 3.2 shows some of our core library functions for both hardware and software sides. (2) Then we can build the hybrid platform.

```
./platform_build.py Convolution 5 4 3
```

Hardware kernel IP of *Convolution* will be added into the hybrid platform. The arguments are the number of GCUs, the number of MicroBlazes per GCU, and the number of hardware kernels per GCU, respectively. A hybrid hardware platform that is similar to Figure 3.5, as well as the bitstream file, will be built after running the above script. (3) Next, we need to build SDK projects and compile them for software kernels and host programs.

```
./SDK_compile.py 0 Convolution 5 4
```

In this step, *lscript.ld* files of all MicroBlazes (including global host, and local slaves) will be modified to correct memory locations for each section in ELF files. (4) Finally, platform bitstream and ELF files will be combined together by the following command to be downloaded to FPGA devices.

```
./download.py 0 Convolution 5 4
```

Since all slaves (including local hosts) in each GCU are identical to each other, we can either build only one ELF file that runs on all slaves or different ELF files for different slaves. Argument *0* in the above two commands stands for the former option and *1* for the later one.

3.4 Hybrid Parallel Programming Model

In this section, the hybrid parallel programming model will be introduced. We exploit the similar way as OpenCL does to dispatch and divide problem spaces into fine-grained data elements. As discussed in Section 3.1, In OpenCL a platform consists of one host processor and several compute devices, each of which contains multiple compute units. A single compute unit is comprised of multiple processing elements. An OpenCL function, called "kernel", is assigned to compute devices during the runtime. A kernel function be considered as a thread running on each processing element. The work-items in a grid are broken into work-groups, each of which is scheduled to execute on a compute unit. Every work-item is physically executed on a processing element. The data parallelism in an application is explicitly expressed.

On the software side, users need to write OpenCL-flavor kernels and host programs with the associated APIs shown in Table 3.2. HOpenCL provides essential functions inherited from OpenCL. Different from OpenCL that provides an explicit method to manage kernel queue and context, HOpenCL users have to arrange kernel execution manually.

3.4.1 Problem Mapping

Figure 3.8 shows how the problem space of the matrix multiplication is divided into groups and scheduled to GCUs. The whole problem space (i.e., the 2-D output array) is divided into groups. The numbers of groups along the two dimensions are called the group size. The numbers of elements along the two dimensions of the group are called the local size. In this way, each item in a group stands for one element of the output matrix. Once a group is assigned to one GCU, all slave processors work together to compute all element in the group.

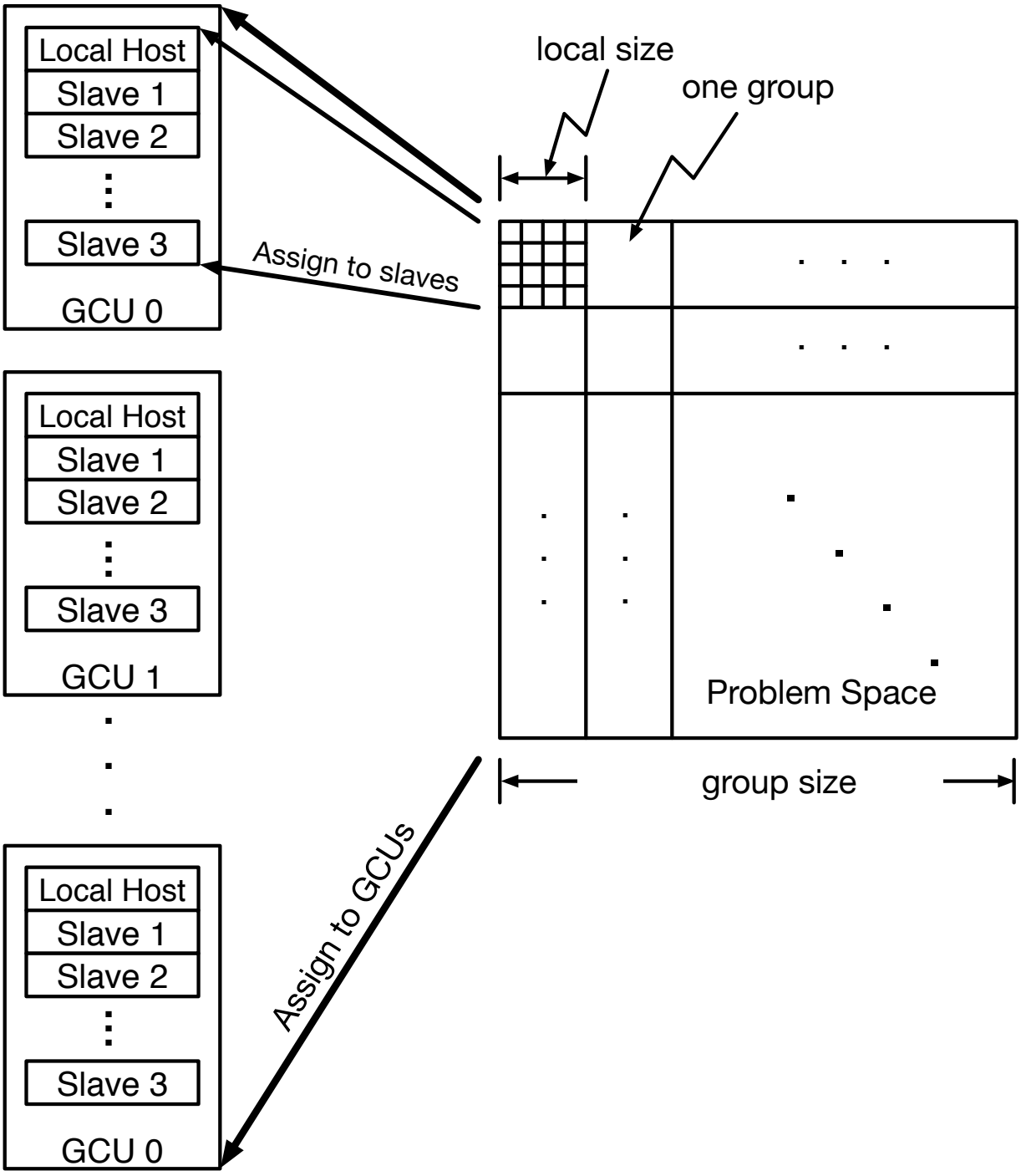


Figure 3.8: Problem space mapping in computation resources.

```
void singel_hardware_kernel(
        volatile unsigned int *data,
        volatile unsigned int *barrierIn,
        volatile unsigned int *barrierOut,
        volatile unsigned int *coreNum,
        volatile unsigned int *localID0,
        volatile unsigned int *localID1,
        volatile unsigned int *groupNum
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE m_axi port=data
#pragma HLS INTERFACE axis port=barrierIn
#pragma HLS INTERFACE axis port=barrierOut
#pragma HLS INTERFACE axis port=coreNum
#pragma HLS INTERFACE axis port=localID0
#pragma HLS INTERFACE axis port=localID1
#pragma HLS INTERFACE axis port=groupNum
}
```

Figure 3.9: Hardware kernel interface.

3.4.2 Kernel Programming

HOpenCL also supports hardware kernel design in Vivado HLS with the associated HOpenCL hardware libraries. Hardware kernels are executed on hardware accelerators. Kernel program similar to software kernels can be converted into hardware design by HLS tool. The differences between software kernels and hardware kernels lie in the hardware interfaces and the HLS design principles. Figure 3.9 shows the compatible hardware interfaces with HOpenCL platform defining in Vivado HLS. An AXI master interface is used to request data from buses. Six AXI-stream interfaces are included to communicate with the local host and other functional HOpenCL hardware IPs (i.e., barrier, local scheduler, and core number allocator). When applying hardware kernels, at least one general-purpose processor will serve as the local host in each GCU to coordinate group execution. Since HLS intends to unitize parallelism features to optimize kernel program, sequential statements need to be handled carefully.

Figure 3.10 demonstrates a kernel program of matrix multiplication with the option to enable DMA. Without DMA, requests of data read and write will be arbitrated through two levels of buses: the local bus and the global bus. When DMA is enabled, data that are consecutive in global memory and are accessed by the processors in each GCU can be moved to local memory to reduce bus requests. There are two implicit barrier operations in $isCurrKernelFinish()$ and

```
#define _SLAVE_
#define DMA
#include "../HCL/hcl.h"
int main () {
  initCore();
  float *arrayA           = (float *)getKernelArg(0);
  float *arrayB           = (float *)getKernelArg(1);
  float *arrayC           = (float *)getKernelArg(2);
  unsigned int A1         = getKernelArg(3);
  unsigned int B1         = getKernelArg(4);
  unsigned int localSize0 = getLocalSize(0);
  while (!isCurrKernelFinish()) {
    unsigned int groupID0 = getGroupID(0);
    unsigned int groupID1 = getGroupID(1);
    while (!isCurrGroupFinish()) {
      unsigned int localID0  = getLocalID(0);
      unsigned int localID1  = getLocalID(1);
      unsigned int globalID0 = getGlobalID(0);
      unsigned int globalID1 = getGlobalID(1);
      int i;
      float sum = 0f;
#ifdef DMA
      float *localArrayA=(float *)LocalMem.BaseAddress;
      simpleDMA (0,
                 (u32)((u32 *)arrayA+globalID0*A1),
                 (u32)((u32 *)localArrayA),
                 localSize0*A1);
      //Execution body with local memory
#else
      //Execution body without local memory
#endif
      arrayC[B1*globalID0+globalID1]=sum;
    }
  }
  cleanupSlave();
  return 0;
}
```

Figure 3.10: Sample kernel code with DMA mode.

```
#define _HOST_
#include "../HCL/hcl.h"
#include "malloc.h"
int main () {
  initCore();
  int A0  = 512;
  int A1  = 256;
  int B0  = 256;
  int B1  = 512;
  float *arrayA = (float *)malloc(A0*A1*sizeof(float));
  float *arrayB = (float *)malloc(B0*B1*sizeof(float));
  float *arrayC = (float *)malloc(A0*B1*sizeof(float));
  setLocalSize(16,16);
  setGroupSize(32,32);
  setKernelArg(0,(unsigned int)arrayA);
  setKernelArg(1,(unsigned int)arrayB);
  setKernelArg(2,(unsigned int)arrayC);
  setKernelArg(3,(unsigned int)A1);
  setKernelArg(4,(unsigned int)B1);
  kernelStart();
  kernelFinish();
  return 0;
}
```

Figure 3.11: Sample host code.

simpleDMA(). Before proceeding to get new group IDs, the local host obtains the group IDs from the group scheduler, and then assigns them to other slaves later, before every slave in one group can continue running. In *simpleDMA*(), firstly, the local host registers DMA number in LSMem to guarantee the one-time DMA operation in the same work-group. This will avoid redundant data movement when the slaves try to get new local IDs. Before the local host finishes DMA operations, all slaves cannot release barriers.

3.4.3 Host Programming

Figure 3.11 shows a sample of how host transfers global shared data to GCUs and manages kernel execution. At the beginning, the global host needs to request a block of memory in the global memory. Since in HOpenCL the compute device and the global host share the same memory space, shared data (i.e., *arrayA*, and *arrayB*) will not be moved between memories. The addresses of shared data is passed to GCUs through *setKernelArg*(), and *getKernelArg*(). At last, the global host will trigger all GCUs to start and wait to return until all GCUs finish the current kernel.

Chapter 4

Experiments and Results

In this chapter, we discuss the experimental methodologies and results. In Phase One, the scalability and performance of our hybrid parallel computing framework without partial reconfiguration is tested and discussed. Then, we enable the partial reconfiguration techniques, and explore the approaches to dynamically schedule the hardware kernels between different micro benchmarks.

4.1 Phase One: Performance and Scalability

In this section, we use the matrix multiplication for experimental analysis to demonstrate the potential of our HOpenCL platform. We take the hardware architectures from the first and the second approaches describing in Section 3.2.1, and Section 3.2.2. Experiments are conducted by using Vivado 2014.2 with the corresponding Vivado HLS on Xilinx ZC705 board. Hardware platform configuration is shown in Table 4.1. Through the AXI bus, the global bus is connected to the global memory that runs at 533 MHz. The hardware platform itself is driven by a 100 MHz clock. The inputs of our benchmark are two 512×512 matrices. Table 4.2 lists the configurations of the host and the kernel programs. Five different local sizes, i.e., 2×2, 4×4, 8×8, 16×16, and 32×32, are tested. We also test the effect of the DMA by enabling or disabling it. Inside each GCU, the computation can be handled by either general-purpose processors or hardware accelerators. Further the number of GCUs can vary from 1 to 6. With the different number of GCUs, local sizes, DMA modes, and kernel types, we conduct the tests on total 120 combinations.

Figure 4.1 shows the FPGA resource utilization under 12 configurations. *A* and *B* stands for using MicroBlazes and hardware kernels as slave processors, respectively. Compared with MicroBlazes, hardware kernels consume slightly fewer registers, LUTs, and BRAMs than MicroBlazes. Since hardware kernels are fully customized accelerators, more DSPs are used in order to maximize the performance by parallelizing computation.

Table 4.1: HOpenCL Hardware platform configurations

Platform		GCUs		Slaves	
Global Mem	512MB	Local Mem	64KB	Private Mem	16KB*
GSMem	8KB	LSMem	8KB	Types	MicroBlaze or
# of GCUs	1 to 6	# of processors†	4		Hardware kernels
Scheduler Policy	FCFS	Scheduler Policy	FCFS	Connection	AXI Master and
Host Type	MicroBlaze	ID FIFO Depth	64	Ports	AXI Stream‡

*When using MicroBlaze, private memory is shared by data and instructions. When using hardware kernels, private memory is implicit since HLS will allocate storage space based on the source code of hardware kernels.

†Including the local host the slaves. The slave can be either general-purpose processor (for software kernel) or hardware accelerator (for hardware kernel).

‡Total eleven AXI-Stream ports are configured on the local host processor, including seven slave ports and 4 master ports. A pair of slave and master AXI-Stream ports are connected to the hardware barrier, three master ports are used to transfer group number to the other three slaves in current GCU. Severn slave ports are connected to hardware IPs. Since other slaves are not connected with group scheduler, they only have five slave ports and one master port.

The scalability of our HOpenCL platform is expressed in Figure 4.2. For every number of GCUs, we measure the speedup for all five different local sizes with DMA enabled. The slave processors can be either MicroBlazes or hardware accelerators. When general-purpose processors are used as slaves, all 3 slaves plus the local host carry out the computation. On the other hand, when the hardware accelerators are implemented as the slaves, only the 3 hardware accelerators carry out the computation because they are much faster than the local host for the matrix multiplication. When the number of GCUs is fewer than 4, the speedup of both software and hardware kernels grows linearly as the number of GCUs increases. When the number of GCUs reaches 4 and above, the performance gains deviate from the linear projection. This trend is more obvious for hardware kernels. This deviation is due to the change of dominant factors that decide the performance of the system. The total execution time of the matrix multiplication benchmark consists of the computation time, the data movement overheads (when DMA is enabled), and the delay of bus requests. When there are less than 4 GCUs, the dominant factor of the total performance is the computation time spent by the processors. Since we use two-level buses with multiple memory hierarchies, the number of memory requests to the global memory are decreased by two AXI interconnections. Besides, the frequency of programmable logic (i.e., 100 MHz) is configured much lower

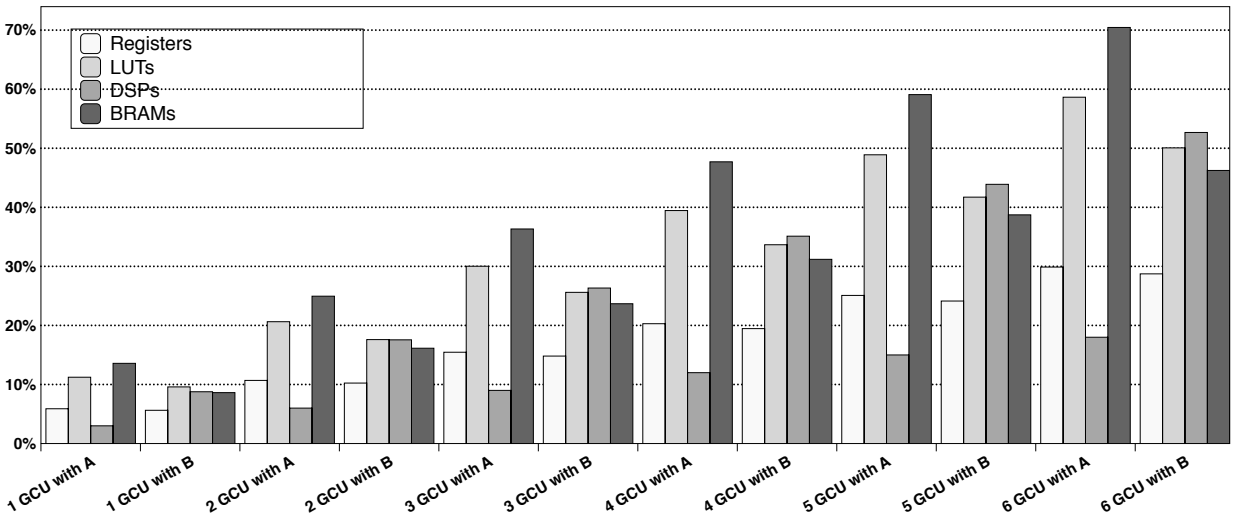


Figure 4.1: Programmable resource utilization under different configurations

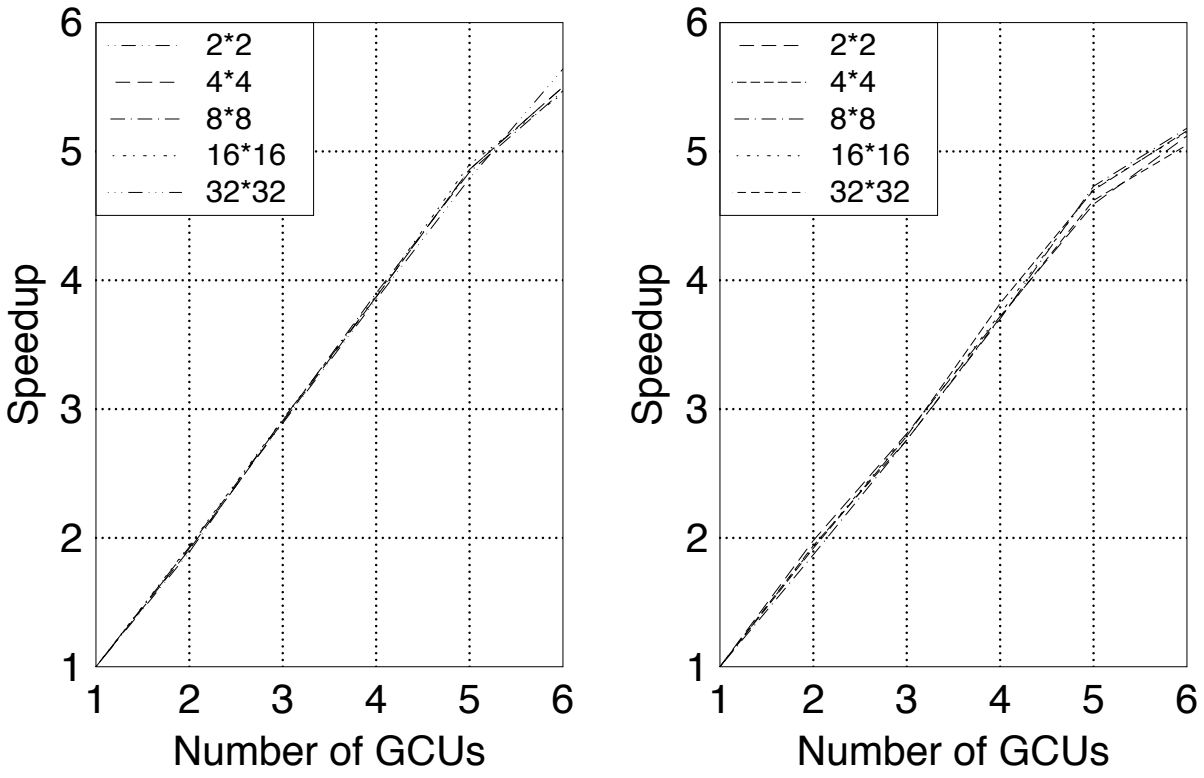
Table 4.2: Software configurations in the matrix multiplication benchmark

Configure No.*	Host Program Configurations		Kernel Program Configurations	
	Local Size	Group Size	DMA Modes	Kernel Types
1	2×2	256×256	Enabled or Disabled	Hardware or Software
2	4×4	128×128	Enabled or Disabled	Hardware or Software
3	8×8	64×64	Enabled or Disabled	Hardware or Software
4	16×16	32×32	Enabled or Disabled	Hardware or Software
5	32×32	16×16	Enabled or Disabled	Hardware or Software

*Every configuration has one local size with the corresponding group size, as well as two possible DMA modes and two possible kernel types. We test every configuration on 6 different GCU numbers ranging from 1 to 6. Therefore, 2×2×5×6=120 combinations are tested.

than that of the DDR interface (i.e., 533 MHz). It is difficult for the system to fully unitize the memory bandwidth when there are a few GCUs. However, when the number of GCUs increases, the dominant factor becomes the delay of bus requests. Hardware kernels are more likely to reach the maximum memory bandwidth. This is because dedicate hardware accelerators are better optimized for the data flows in the given tasks, and thus have more intensive memory accesses than the general-purpose processors.

Table 4.3 extracts parts of results from Figure 4.2 where the local size is configured as 16×16 to demonstrate the speedup by enabling DMA with hardware kernels. Before starting new group



(a) MicroBlazes as slaves.

(b) Hardware accelerators as slaves.

Figure 4.2: Scalability by using MicroBlazes and hardware accelerators.

execution, a block of consecutive data with the size of 512×16 from the first matrix will be fetched into the local memory from the global memory. No matter DMA is enabled or not, speedup by using hardware kernels is around 11 times. Speedup by using DMA in hardware kernels is slightly higher than that in software kernels. Hardware kernels are more sensitive with the distance of targeted memory than general-purpose processors since they usually have more intensive memory requests.

In order to further compare the performance potential of our hybrid platform, we conduct the same experiment of matrix multiplication on one of hard ARM cores built on Zynq device. The ARM core is connected to the on-chip memory (OCM) through two-level caches that are enabled in our experiment and runs at 667 MHz. The execution time is $15.24s$. Since inputs and outputs are all located in OCM, there is no further optimization for memory accesses. Comparing with DMA enabled software kernel implementation from Table 4.3, we can conclude that the

Table 4.3: Performance results with the local size of 16×16 (unit: *s*)

# of	Software Kernel			Hardware Kernel			Hardware Speedup	
GCUs	w/o DMA	w/ DMA	Speedup	w/o DMA	w/ DMA	Speedup	w/o DMA	w/ DMA
1	44.32	36.12	1.23	3.94	3.07	1.28	11.25	11.75
2	23.20	18.54	1.25	2.02	1.60	1.26	11.49	11.58
3	14.91	12.33	1.21	1.35	1.07	1.26	11.04	11.52
4	11.87	9.30	1.28	1.08	0.83	1.30	10.99	11.20
5	9.12	7.28	1.25	0.83	0.65	1.27	10.99	11.20
6	8.02	6.64	1.21	0.74	0.57	1.29	10.84	11.65

performance of one ARM core is equivalent to 2 and 3 GCUs. When comparing with hardware kernel implementation, one ARM core performs much worse than a single GCU.

4.2 Phase Two: Multiple Kernel Scheduling

In Phase two, we propose a scheduling algorithm to dynamically reconfigure hardware kernels. As the motivation discussed in Chapter 1, one application with different input sizes running on either software kernels or hardware kernels will have different performance. When input sizes are small, the performance of hardware kernels will be hidden from the overheads of partial reconfiguration.

4.2.1 Dynamic Partial Reconfiguration

4.2.1.1 Profiling

The scheduling program runs on the global host. Based on the given application and its input sizes, the global host decides which scheme is used to implement this application. We propose two different schemes including:

1. Pure Software Kernels (SW): Application runs on general-purpose processors without any acceleration from hardware.

2. Pure Hardware Kernels (HK): Application runs on hardware kernels after its bitstream is downloaded into corresponding PR regions.

```
#define _HOST_
//Include software libraries
#include "../HCL/hcl.h"
void app_implementation(type scheme) {
  if (isPR(scheme))
    do_PR(app_type);
  {
    //Allocate global memory
  }
  setLocalSize(...);
  setGroupSize(...);
  setKernelArg(0,...);
  kernelStart();
  kernelFinish();
}
int main () {
  //Initial global host
  initCore();
  while (!isFinish()) {
    get_app_task(app_type,app_size);
    scheme = check_lookup_table(app_type,app_size);
    app_implementation(scheme);
  }
  return 0;
}
```

Figure 4.3: Pseudocode of the basic scheduling algorithm.

Downloading hardware kernels into PR regions is handled by the global host. In our current work, the execution time cannot be fetched during runtime. Therefore, an off-line evaluation of each application with different input sizes is performed. A lookup table saved on the global host will record the execution time with its corresponding scheme of each benchmark with different input sizes after the off-line evaluation. We propose two scheduling algorithms as follows.

4.2.1.2 Basic Scheduling (BS)

Pseudocode of basic scheduling is provided in Figure 4.3. Before continuing to the next task, application type and input sizes are fetched from execution queue. Lookup table stores the execution time resulted in from one of the following three scenarios: (1) Running on general-purpose processors as SW scheme; (2) Reconfiguring PR regions with corresponding hardware kernels, and running as HK scheme; the execution time (including PR time and kernel execution time) is calculated by running different schemes. After checking with lookup table, the scenario resulting in the minimum execution time will be chosen to load.

```
check_lookup_table() {
  if (isMatchCurrApp()) {
    //Get minimal value from scheme 1,2,3
  }
  else {
    //Get minimal value from scheme 1,2
  }
}
```

Figure 4.4: Pseudocode of *check_lookup_table* in the enhanced scheduling algorithm.

4.2.1.3 Enhanced Scheduling (EH)

In basic scheduling, when two adjacent kernel programs are running, the corresponding PR regions will be downloaded twice. With enhanced scheduling, some PR overheads can be avoided. In case where the next task and the previous task share the same application and hardware kernel scheme (HK), there is no need to perform the PR again. The current task can run as hardware kernel without reconfiguring the PR region. In enhanced scheduling, besides the two scenarios in basic scheduling, another scenario should be added to achieve the minimum execution time: kernels running as (3) HK without reconfiguring PR regions. Function *check_lookup_table* need to be modified as Figure 4.4 shows.

4.2.2 Micro Benchmarks

We evaluate three micro benchmarks listed as follows:

Matrix Multiplication: Two two-dimensional floating-point square matrices are used as the input data, and one 2D matrix will be computed as the output result. Different input sizes are examined.

Convolution: We preform a one-dimensional convolution. The input data is an integer array with different lengths. The filter size is always of 128 elements.

Pre-scan: Addition is used as the operation. Input linear arrays are tested with different sizes. The output array has the same length as the input array. Each element of the output array is the sum of previous elements of the input array.

Each benchmark has two versions: the software one, the hardware one. Experiments are con-

Table 4.4: Hybrid hardware platform configurations

Platform		GCUs		Slaves	
GlobalMem	1GB	LocalMem	64KB	PrivateMem	16KB*
PR Enabled	Yes	DMA Enabled	Yes	Types	MicroBlaze and Hardware kernels
# of GCUs	4	# of processors†	4		
Scheduler Policy	FCFS	Scheduler Policy	FCFS	PR Features	Three PR regions
Frequency	100MHz	# of PR Regions	6		

*When using MicroBlaze, private memory is shared by data and instructions. When using hardware kernels, private memory is implicit since HLS will allocate storage space based on the source code of hardware kernels.

†Including the local host and the slaves. The slave can be either general-purpose processor (for software kernel) or hardware accelerator (for hardware kernel).

Table 4.5: Hardware resource utilization of the default platform (4 MicoBlazes in one GCU, and totally 4 GCUs)

Resources	Usage	Available	Percentage of Utilization
Slice LUTs	101058	203800	49.59%
Slice Register	84830	407600	20.81%
Memory	219	445	49.21%
DSPs	111	840	13.21%

ducted by using Vivado 2014.2 with the corresponding Vivado HLS tools. Xilinx Kintex-7 is chosen as the FPGA platform. Hardware platform configuration is shown in Table 4.4. The default hardware configuration consists of 4 GCUs. Each GCU has four MicroBlazes, of which one is used as local host, and the other three are normal slaves. In addition, there are totally three PR regions assembled into each GCU. The resource utilization of default platform without any PR regions is shown in Table 4.5.

There are 3 physical PR regions in one GCU on the platform we use to compare various scheduling algorithms in later text. Totally there are 12 PR regions for 4 GCUs. Table 4.6 shows the size of one PR region, which is larger than the actual need of three hardware kernels. The extra resources in PR regions provide room for future extension. The size of bitstream for one PR region is 378 KB. PR is performed by the ICAP IP core and the Global Host processor through global bus. The time to implement one PR region is 0.14 s. In other words, the overhead to implement 12 PR regions is $0.14 \times 12 = 1.68$ s.

Table 4.6: Resource utilization of Partial reconfiguration and hardware kernels

	Slice LUTs	%	Slice Registers	%	DSPs	%	Memory	%
Matrix	1393	40.97%	1506	31.38%	20	50%	0	0%
Prescan	1205	35.44%	1062	22.13%	4	10%	0	0%
Convolution	1083	31.86%	978	20.38%	6	15%	0	0%
PR Region	3400*	–	4800	–	40	–	0	–

*The total number consists of 350 SLICELs, and 250 SLICEMs with 2400 LUTs as logic and 1000 LUTs as memory, respectively.

Table 4.7: Configurations of execution queue with micro benchmarks.

Micro Benchmarks	# of Input Sizes	# of Tasks	# of Task per Input Size
Matrix Multiplication	6*	60	10
Prescan	6†	60	10
Convolution	6‡	60	10

*6 input sizes are 16*16, 32*32, 64*64, 128*128, 256*256, and 512*512.
†6 input sizes are 1K, 2K, 4K, 8K, 16K, and 32K.
‡6 input sizes are 64K, 128K, 256K, 512K, 1M, and 2M.

4.2.3 Evaluation Methodology

For the micro benchmarks discussed in Section 4.2.2, we generate an execution queue with size of 180, i.e., using 180 tasks to evaluate the performance of the proposed scheduling algorithms on our system. As Table 4.7 shows, each of three micro benchmarks own 60 tasks in the execution queue, and every input size has 10 tasks. The order of tasks in the execution queue is generated randomly. We present static and dynamic scheduling techniques to show the optimization of energy and performance of our dynamic partial reconfiguration algorithms. In static scheduling, all tasks in the execution queue run on only SW, and HK schemes as the evaluation baselines. When tasks running on hardware kernels, the RP is always carried out by default. The results using basic scheduling (BS) and enhanced scheduling (EH) are compared with the results using static scheduling. In the *Ideal* scheduling, all PR overheads are ignored when different applications are switched. We use execution time in Section 4.2.1 as the metric to evaluate the overall performance after all tasks are finished.

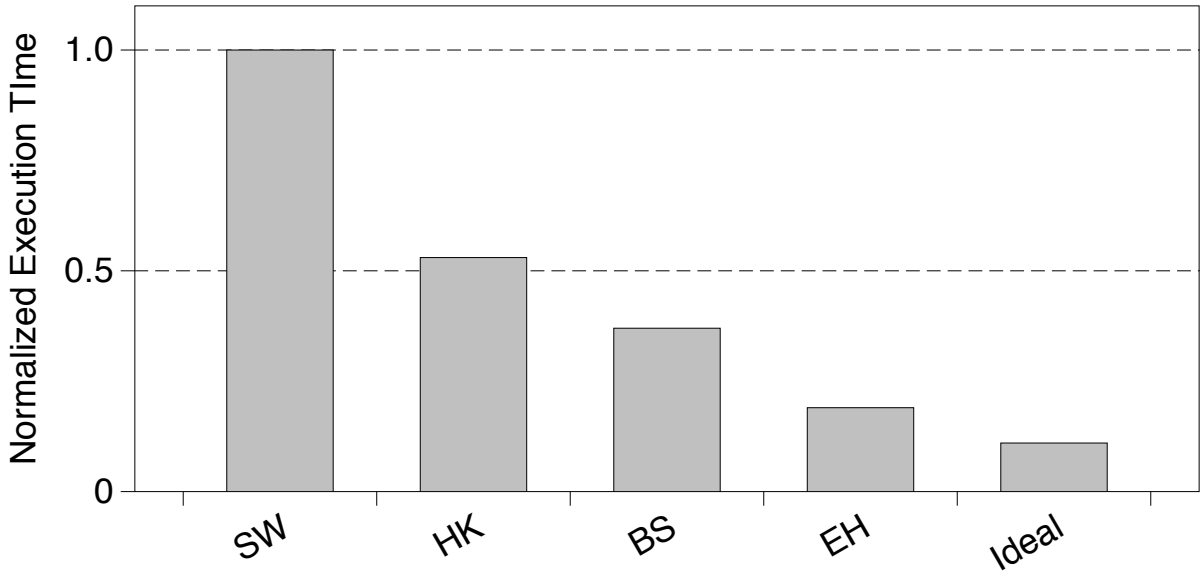


Figure 4.5: Comparison of results following various scheduling algorithms.

4.2.4 Results

As showing in Figure 4.5, enhanced scheduling performs better than basic scheduling. Differences between *SW* and *BS* mainly result from the advantages for hardware kernels to address large input data sizes. From *EH* to *Ideal*, the PR overheads are ignored when different applications are switched. When handling small input data size, software implementation has slightly advantages over hardware one with PR enabled. Therefore, *BS* performances better than *HK*.

Chapter 5

Conclusion

Hardware accelerators are capable of achieving higher performance than general-purpose processors. However, designing dedicated hardware accelerators usually lacks the productivity and the flexibility compared with programming on general-purpose processors. Multiprocessor system-on-chip (MPSoC) incorporating software cores are designed to express parallelism lying within applications to achieve higher performance. In this work we present a prototype of a unified OpenCL-flavor parallel programming model to combine both software and hardware kernels into our hybrid multiprocessor system-on-chip with multiple memory hierarchies. In addition, we propose the corresponding automatic design flow by generating software and hardware kernels. With the HOpenCL hardware and software libraries, as well as the compatible hardware interfaces, users do not need to re-write separate hardware kernels when applying hardware accelerators into the system.

Further, we extend our hybrid co-design computing framework to support dynamic partial reconfiguration and correponding scheduling methods for different hardware kernels. With the help of partial reconfiguration on FPGAs, dynamic profiling and scheduling algorithms are proposed for allocating computation resources. Experiments are carefully conducted on the Xilinx Kintex-7 platform.

We use matrix multiplication as our benchmark to examine the potential of our hybrid system in the first and the second approaches in terms of performance, scalability, and productivity. Two 512×512 matrixes as input are given to the kernel. From host side, various local sizes with the associated group sizes are tested. For each option of local size, we generate both hardware and software kernels with DMA either enabled or disabled. The results show that using hardware kernels reaches more than 10 times speedup compared with the software kernels. DMA can help improve the performance by $\sim 25\%$. Our prototype platform also demonstrates a good performance

scalability when the number of group computation units (GCUs) increases from 1 to 6 until it becomes a memory bound problem. Compared with the hard ARM core on the Zynq 7045 device, we find that the performance of one ARM core is equivalent to 2 or 3 GCUs with software kernel implementations. On the other hand, a single GCU with hardware kernel implementation is 5 times faster than the ARM core.

Using the hardware architecture in the third approach, we fully implemented three micro benchmarks using both software kernels and hardware ones. With different input sizes and applications, an execution queue on hybrid platform is generated. Totally 180 tasks in the queue are executed. The results shows that with dynamic scheduling, the performance is 5.2 times better than the one using purely software implementations.

5.1 Future Work

In this section, the future work will be discussed in two aspects: performance and extensibility. We would like to maximize the performance in terms of hardware and software kernels. Also, we would like to improve the framework as it can be easier to use.

5.1.1 Performance

1. Optimizing hardware kernel design. Although hardware kernels are designed by HLS tools. However, the performance of hardware kernels can be further improved by optimizing HLS source codes.

2. Improving performance of partial reconfiguration. The performance of PR is mainly limited by the performance of AXI bus, as well as the vendor-designed ICAP modules. Xilinx provides the wrapper of ICAP which can be connected to the AXI bus. However, this wrapper can only transfer bitstream files in AXI-lite mode. We would like to enable burst mode, and pipelined transferring to replace the original ICAP wrapper.

3. Mixed kernel running. Current HOpenCL system can only run the same kernel programs

at one time no matter software or hardware kernels. Scheduling mechanism by enabling execution of mixed kernel programs should be focused on.

4. Better scalability interconnection networks. The scalability of current system is limited by the number of GCUs and the frequency of system bus. However, as the number of GCUs increase, the scalability will be worse even increasing the frequency of system bus. In the future, we would like to utilize network-on-chip (NoC) with better scalability into our framework.

5.1.2 Extensibility

1. Supporting more OpenCL APIs. At this moment, we only support a limited set of original OpenCL APIs on our HOpenCL platform. We plan to support more OpenCL APIs in the future development.

2. OS migration. HOpenCL runs as a standalone framework. This single framework is limited by the compiler, as well as the library supports. We intend to migrate the current framework into an OS supported system. In this way, the new system can support much larger PR bitsream files.

3. Accepting more general kernel programs. Software kernel programs are converted into hardware kernel programs (namely, HSL codes) through kernel2HLS Translator. However, current translator cannot accept all kinds of kernel programs. Besides, the synthesized HLS codes cannot be fully guaranteed to run as expected.

References

[1] Altera Corporation. *OpenCL for Altera FPGAs – http://www.altera.com/products/software/opencl/opencl-index.html*. Last accessed December 4, 2014.

[2] Alexandros Bartzas and George Economakos. A methodology for efficient use of OpenCL, ESL and FPGAs in multi-core architectures. In *Proceedings of the 18th International Conference on Parallel Processing Workshops*, Euro-Par'12, pages 507–517, 2013.

[3] S. Alexander Chin and Paul Chow. Opencl memory infrastructure for FPGAs (abstract only). In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 269–270, 2012.

[4] Dan Connors, Eric Grover, and Blake Caldwell. Exploring alternative flexible OpenCL (FlexCL) core designs in FPGA-based MPSoC systems. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 3:1–3:8, 2013.

[5] Hongyuan Ding and Miaoqing Huang. A unified opencl-flavor programming model with scalable hybrid hardware platform on fpgas. In *Proc. 2014 International Conference on Reconfigurable Computing and FPGAs (ReConFig'14)*, December 2014.

[6] Hongyuan Ding and Miaoqing Huang. An automatic design flow for hybrid parallel computing on mpsocs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '15, 2015.

[7] D. Göhringer, M. Hübner, V. Schatz, and J. Becker. Runtime adaptive multi-processor system-on-chip: RAMPSoC. In *Proc. IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–7, April 2008.

[8] P.O. Jääskeläinen, C.S. de la Lama, P. Huerta, and J.H. Takala. OpenCL-based design methodology for application-specific processors. In *Proc. 2010 International Conference on Embedded Computer Systems (SAMOS)*, pages 223–230, July 2010.

[9] Khronos OpenCL Working Group. *OpenCL 2.0 Specification*. Khronos Group, November 2013.

[10] 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 *Proc. 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig'10)*, pages 7–12, December 2014.

[11] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. OpenRCL: Low-power high-performance computing with reconfigurable devices. In *Proc. 20th International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 458–463, August 2010.

[12] Dominik Meyer and Bernd Klauer. Multicore reconfiguration platform an alternative to rampsoc. *SIGARCH Comput. Archit. News*, 39(4):102–103, September 2011.

[13] Muhsen Owaida, Nikolaos Bellas, Christos D. Antonopoulos, Konstantis Daloukas, and Charalambos Antoniadis. Massively parallel programming models used as hardware description languages: The opencl case. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 326–333, Piscataway, NJ, USA, 2011. IEEE Press.

[14] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. Synthesis of platform architectures from OpenCL programs. In *Proc. 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*, pages 186–193, May 2011.

[15] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[16] Hiroyuki Tomiyama. Challenges of programming embedded many-core SoCs with OpenCL. In *Proc. 11th International Forum on Embedded MPSoC and Multicore (MPSoC'11)*, July 2011.

[17] Xilinx, Inc. *Vivado High-Level Synthesis (2014.2)*, June 2014.