ON THE PROGRAMMABILITY OF HETEROGENEOUS MASSIVELY-PARALLEL COMPUTING SYSTEMS

BY

ISAAC GELADO

Master of Science in Telecommunications Engineering
Universidad de Valladolid, 2003

Advisor: Nacho Navarro and Wen-mei W. Hwu

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Architecture

Universitat Politècnica de Catalunya
2010

Barcelona, Spain

Abstract

Heterogeneous parallel computing combines general purpose processors with accelerators to efficiently execute both sequential control-intensive and data-parallel phases of applications. This thesis aims to increase the programmability of heterogeneous parallel systems formed by CPUs and massively data-parallel accelerators. This thesis deals with the programmability problems due to separate physical memories for CPUs and accelerators, which are key to accomplish high performance. These separate memories are presented to application programmers as disjoint virtual address spaces, which harms programmability in two different ways. First, extra code is needed to transfer data between system and accelerator memories. Second, data structures are double-allocated in both memories, which results in using different memory addresses (pointers) in CPU and accelerator code to reference the same data structure. This thesis proposes mechanisms to solve these problems from three different levels: programming model, hardware modifications, and software run-time systems.

This thesis proposes a programming model that integrates accelerator and system memories into a single virtual address space, allowing the CPU code to access accelerator memories using regular load/store instructions. In this programming model, data structures shared by CPUs and accelerators are hosted in the accelerator memory and accessible from both the CPU and the accelerator code. Moreover, because a single copy of data structures exists in the application virtual address space, applications can use the same virtual memory address (pointer) to access such data structures.

The Non-Uniform Accelerator Memory Access (NUAMA) architecture is proposed as an efficient hardware implementation of this programming model. This architecture incorporates mechanisms to buffer and coalesce memory requests from the CPU to the accelerator memory to reduce the performance penalty produced by long-latency memory accesses to accelerator memory. Moreover, the memory hierarchy is modified to use a write-through cache policy for accelerator-hosted data which results in an eager update of the accelerator memory contents. This eager update ensures that most of the accelerator memory contents are updated on accelerator calls, effectively minimizing the accelerator call latency.

The Asymmetric Distributed Shared Memory Model (ADSM) is also in-

troduced in this thesis, as a run-time system that implements the proposed programming model. In ADSM, the CPU code can access all memory locations in the virtual address space, but accelerator code is restricted to access those memory addresses mapped to the accelerator physical memory. This asymmetry allows all required memory coherence actions to be executed by the CPU, which allows the usage of simple accelerators. The ADSM model keeps a copy of the accelerator-hosted data in system memory for the CPU to access it, and builds an unified virtual address space by mirroring system and accelerator virtual address space.

Finally, this thesis introduces the Heterogeneous Parallel Execution (HPE) model, which allows a seamless integration of accelerators in the existent sequential execution model offered by most modern operating systems. The HPE model introduces the execution mode abstraction to define the processor (i.e., CPU or accelerator) where the application code is being executed. The execution mode abstraction provides full backwards compatibility with existent applications and systems.

To all those giants who gently offered me their shoulders to stand on. I hope that what I reached to see was worthy their effort.

Acknowledgments

I have been told to not forget acknowledging my advisors, so I will do it in the first place. I have to acknowledge the great work my advisors, Nacho and Wen-mei, have done during all these years. These two giants are responsible for this thesis as much as the time I have devoted to its completion. I have to thank Nacho for all the time he has shared with me, not only as an advisor, but as the great person he is. I am grateful to Wen-mei, who gave me the opportunity of learning from him and guided me in the dark. Wen-mei, thank you so much for converting a visit to the middle of nowhere in such a delightful experience.

The ordering of the acknowledgments has been driving me crazy for quite a long time. Finally, I have decided to follow a quite systematic approach to not forget anybody, so I will do it in chronological ordering.

Although the memories are quite blurry, I think the first people I met were my parents. Be sure that without them, this thesis would have never been completed. My dear mother, Adelina, has been always looking after me. Although many times she did not understand what I was doing, she has always believed in me, and so I appreciate such a hard task. The first seed of this thesis was planted by my father, Juan Francisco, many years ago. He inculcated me the love for science and engineering since I was a child, he taught me to read, to calculate, and to wonder how stuff works. I sincerely believe that this thesis started when my father and I reached the agreement to buy a computer only after I learnt how it works. He is the giant I admire the most.

My brother, Jesús, was the third person I met. He has been living with me for most of my life, quite a hard task for him, believe me. I would like to thank my brother for letting me experiment with electricity on him when I was a child. I have to acknowledge his patience with me on so many occasions I, in the name of science, broke our computer at home.

I have to acknowledge Yannis Dimitriadis for his passionated lectures, where I started to love computer architecture. I also have to thank my Ms thesis advisor, Juan Ignacio Asensio, who fooled me to get into the PhD program. This thesis has been also possible because I learnt from these two giants.

Carlos Villavieja and Carlos Boneti require a special mention. Carlos Villavieja, my office mate for so many years, has been helping me during all the time I have spent in Barcelona. I will never forget the time we spent in the fridge, the days

in the lab, the conversations we had. Carlos Boneti and I were new in the city when we started our PhD, and we have shared so many weekends at UPC, so many days working together, so many pieces of our life. I do not want to thank Carlos Villavieja and Carlos Boneti for being such a great support for me, I thank them for being such great friends.

I have to acknowledge Mateo Valero for giving me the opportunity of studying at UPC and learning from him. But this thesis is just a tiny thing for being grateful to such a giant. Mateo, thank you so much for devoting your life to make UPC one of the best places in the world to do research in computer architecture. I also want to thank Alex Ramírez, a giant who has given me so many useful advises and shared so many drinks. Another giant who also has helped me during this thesis is David Kaeli, whose sincerity served me to avoid false steps.

I am deeply grateful to Yale Patt, the biggest giant, from whom I have learnt about computer architecture and teaching during his lectures, and about life while having coffee and dinner with him. Yale is also responsible for this thesis; he has provided me continuous advise and helped me to be on the track. Yale, you are done with your part of the deal, it is now my turn.

I also have to thank the lab crow. Ramón and I have shared many rejects, some gin-tonics, and once a sofa. Javi has helped with the coding, testing, and thinking of this thesis. Lluís, the tool master, is the responsible for most of the graphs. I also thank Shane Ryoo, Sain-zee Ueng, Christopher Rodrigues and Sara Baghsorkhi, who kindly have helped me when visiting Urbana. I would also thank Marie-Pierre for all the paper work she has done for me.

John Kelm has been a key player in this thesis, with whom I shared so many conversations where most of the ideas of this thesis were born. I have to acknowledge John's work polishing my text in so many papers, and teaching me educated English words.

Two giants I also have to thank are Steve Lumetta and Sanjay Patel. Steve, thank you for teaching me the importance of little details when doing research. Sanjay deserves a lot of credit for this thesis. I will never forget the meeting where Wen-mei and Sanjay triggered the idea of ADSM in my brain. John Stone also has contributed to the elaboration of this thesis with great insights about programmers' point of view, and deserves my acknowledge.

The last, but not the least, person to acknowledge is my precious Anna. I have to thank her for being with me during these last years, supporting me, taking care of me, loving me. Anna has become the main motivation of my life and the reason to finish this thesis; without her I would have never done this work.

Abbreviations

ADSM Asymmetric Distributed Shared Memory

AMC Accelerator Memory Collector

API Application Programming Interface

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DP Data-Level Parallelism

DMA Direct Memory Access

DSM Distributed Shared Memory

FPGA Field Programmable Gate Array

GMAC Global Memory for Accelerators

GPU Graphics Processing Unit

HPE Heterogeneous Parallel Execution

IPC Instructions Per Cycle

I/O Input/Output

ISA Instruction Set Architecture

MMU Memory Management Unit

NUAMA Non-Uniform Accelerator Memory Access

POSIX Portable Operating System Interface for Unix

OS Operating System

SDK Software Development Kit

TLB Translation Look-aside Buffer

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Heterogeneous Parallel Computing

Data parallel code has the property that multiple instances of the code can be executed concurrently on different data. Data parallelism exists in many applications such as physics and biology simulations, weather prediction, financial analysis, medical imaging or media processing. Most of these applications also have sequential control-intensive phases that are interleaved between data parallel phases. A computing system that efficiently executes such applications requires high multi-thread throughput for data parallel phases and short single-thread execution latency for sequential phases. However, maximizing multi-thread throughput and minimizing single-thread execution latency are two design goals that impose very different, and often conflicting, requirements on processor design. General purpose processors implement out-of-order execution, multi-instruction-issue, data caching, branch prediction and memory speculation mechanisms using high frequency clock signals to concurrently execute as many independent instructions as possible in the shortest possible time. This is in contrast to, for instance, Graphics Processing Units (GPU) that achieves high data throughput using many in-order multi-threaded cores that share their control unit and instruction cache and run at moderate frequency. Analogously, reconfigurable logic allows implementing many application-specific processing units running at low frequencies that also deliver high data throughput.

Figure 1.1 illustrates how general purpose processors and accelerators are combined to form heterogeneous parallel computing systems. In these systems, sequential control-intensive code is executed by a CPU, while data parallel application phases are executed by accelerators. There are many examples of successful heterogeneous parallel systems [PH08]. For example, the RoadRunner supercomputer couples AMD Opteron CPUs with IBM PowerXCell accelerators. If the RoadRunner supercomputer were benchmarked using only its general purpose CPUs, rather than being the top-ranked system (June 2008), it would drop to a 50th-fastest ranking [BDH$^+$08]. In addition, practically all modern desktop computers are heterogeneous parallel systems that combine CPUs and GPUs. Moreover, heterogeneous systems also deliver a higher power efficiency than traditional homogeneous systems. For instance, the NVIDIA

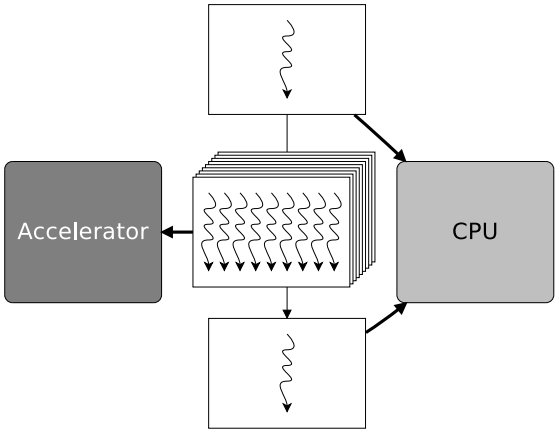


Figure 1.1: Heterogeneous parallel computing paradigm

GTX285 GPU offers 5.21 MFLOPS (single-precision) per watt when all hardware resources are utilized, whereas the Intel i7 895 CPU delivers 0.83 MFLOPS per watt. This high power efficiency in massively data parallel codes is another reason for the adoption of heterogeneous massively-parallel computing systems in both, High Performance Computing (HPC) and embedded environments.

A major architectural issue when designing a heterogeneous parallel system is the programming model for data sharing between CPUs and the accelerators. Programming models in current commercial systems typically expose separate physical memories of CPUs and accelerators, and data transfer mechanisms based on Direct Memory Access (DMA) hardware to application programmers through data copy routines. For example, in the CUDA programming model [NVI09], an application programmer can transfer data from the CPU to the accelerator device by calling a memory copy routine whose input parameters include a source pointer to the CPU memory space, a destination pointer to the accelerator memory space, and the number of bytes to be copied. OpenCL [Mun09] and the IBM Cell SDK [KDH+05] define similar interfaces. These explicit memory copy models can significantly increase the programming complexity when using accelerators in large complex applications.

1.2 Overview

The main goal of this dissertation is to optimize programmability and performance of heterogeneous parallel systems as a single metric, which could be summarized as: *to ease the coding task without decreasing the system performance*. This dissertation argues and provides experimental data to support that a programming model for heterogeneous parallel systems, where application programmers are provided with a single virtual address space that includes system and accelerator memories, accomplishes the goal of improving programmability while not affecting system performance. This dissertation further studies, ana-

lyzes and evaluates the implementation and performance issues that arise when unifying the disjoint physical address spaces of CPUs and accelerators into a single virtual address space.

Chapter 2 covers the related work of this dissertation. First, a review of existent CPU – accelerator architectures and programming models for heterogeneous parallel systems is presented. CPU – accelerators architectures are classified depending on the granularity of the computations accelerators perform. Existing mechanisms to transfer data between CPUs and accelerators are also presented. Programming models are classified depending on how accelerators are integrated in the application code. Finally, this chapter reviews the previous work in Distributed Shared Memory (DSM) systems and operating system support for distributed and heterogeneous systems that is extended by this dissertation.

Chapter 3 presents the hardware and software reference system assumed in this dissertation. The reference heterogeneous parallel system is formed by CPUs and coarse-grained massively-parallel accelerators with physically separated memories. The reference programming model expose the existence of these separate memories as disjoint virtual address spaces to programmers and accelerators are invoked as function calls. The NVIDIA CUDA programming model is presented as an example of such programming model and the specifics of CUDA used in this dissertation are discussed. Finally, the simulation and execution environments and the benchmarks used to evaluate the contributions of this thesis are detailed.

Chapter 4 presents the benefits of a model where accelerator-accessible data is only hosted by the accelerator memory, and where a global shared virtual address space that includes system and accelerator memories is used. First, three data hosting models for heterogeneous parallel systems are defined: per-call, double-buffered, and accelerator-hosted. The per-call model imposes low accelerator memory capacity requirements but it can potentially degrade the overall system performance to unacceptable levels. The double-buffered model requires a higher amount of accelerator memory than the previous model, delivers higher performance because it overlaps data communication between CPUs and accelerators with computation, but it highly penalizes programmability. Finally, the accelerator-hosted model requires all data used by accelerators to be hosted in their own memory, which imposes high memory capacity requirements on accelerators, but increases programmability because data transfers between system and accelerators memories are not required. In this dissertation, the implementability issues and optimization opportunities of this later model are discussed.

Chapter 5 presents the Non-Uniform Accelerator Memory Access (NUAMA) architecture. NUAMA is an implementation of the accelerator-hosted model defined in Chapter 4. In NUAMA, an Accelerator Memory Collector (AMC) is integrated within the system memory controller. The AMC inspects all memory

requests going into and out of the system memory controller to identify writes to data structures hosted by accelerators. The processor Translation Look-aside Buffers (TLB) entries are extended with one extra field to identify those memory pages containing data hosted by accelerators. This field is used by the AMC to identify memory requests affecting accelerator hosted data. The AMC also ensures that all pending writes to accelerator hosted data have finished before starting the execution at the accelerator. An experimental evaluation of the NUAMA architecture using a cycle-accurate simulator shows that this architecture delivers a performance similar to traditional DMA-based architectures.

Chapter 6 introduces the Asymmetric Distributed Shared Memory (ADSM) model. ADSM is a software implementation of the accelerator-hosted data transfer model. In this model, a unified virtual address space encompassing both system and accelerators memory is built. In ADSM, CPUs can access any virtual memory address within the unified virtual address space, but accelerators are constrained to access virtual memory addresses corresponding to their own physical memory. This asymmetric nature of the model allows the system to rely on CPUs to perform all memory coherence activities; accelerators, therefore, do not need to provide additional hardware support for memory coherence. The design and implementation issues of a complete ADSM system are considered in this thesis. Experimental results show that applications accomplish similar performance and higher programmability when using the ADSM model than when using commercially available run-time systems, which matches with experimental results in Chapter 5.

Chapter 7 extends the ADSM model to fully integrate heterogeneity in the application execution model. This chapter analyzes existent execution models for different accelerators, which are not fully compatible with the current sequential execution model based on execution threads. Then, the Heterogeneous Parallel Execution (HPE) model is presented. This model extends the current execution thread abstraction with a set of execution modes, which effectively represent the application capabilities when executing on different computational devices (i.e., CPUs and accelerators). The HPE model is fully compatible with the existent sequential model implemented by most operating systems. Moreover, HPE provides support for running applications for heterogeneous parallel systems in current homogeneous (i.e., without accelerators) machines. The design and implementation space for this execution model is analyzed, and the accelerator hardware support for an efficient implementation of this model is described.

Chapter 8 concludes this dissertation. The key contributions and insights presented in this thesis are reviewed. This chapter also presents potential future work and the research lines this thesis has enabled. Potential problems to be solved as part of the future work are analyzed and possible solutions are sketched.

1.3 Contributions

This dissertation makes several contributions in the programming models and computer architecture fields.

1.3.1 Accelerator-hosted Data Transfer Model

This dissertation introduces a data transfer model where accelerator-accessible data structures are hosted in the accelerator memory. Such data transfer model removes data access and marshalling overheads incurred when using traditional accelerator programming models. Existent programming models for heterogeneous parallel systems requires transferring all data used by accelerators from system memory to accelerator memory prior starting the execution at the accelerator. When data structures used by accelerators are scattered in the application virtual address space (e.g., a linked-list), the code running on the CPU first has to extract and arrange that data (i.e., data marshalling). An accelerator-hosted model does not require explicit data transfer and data marshalling, improving programmability.

1.3.2 Unified Virtual Address Space

A unified virtual address space that includes both system and accelerator memories is presented in this dissertation. Such a virtual address space improves programmability of heterogeneous parallel systems by elegantly solving the double-pointer problem present in current programming models. This problem arises from the separate physical accelerator and system memories in most heterogeneous parallel systems. Separate system and accelerator memories are presented to application programmers as disjoint system and accelerator virtual address spaces. Hence, two virtual memory addresses are required to reference a single data structure: a system virtual memory address in the CPU code, and an accelerator virtual memory address in the accelerator code. A unified virtual address space for CPUs and accelerators hides the separate physical memories to programmers and, therefore, removes the need for two different pointers to reference a single data structure.

1.3.3 Non-Uniform Accelerator Memory Access

The NUAMA architecture is introduced in this dissertation. This architecture allows applications to be programmed following the accelerator-hosted data transfer model by allowing CPUs to access accelerator data using load/store instructions. The NUAMA architecture includes hardware structures to buffer and coalesce write requests to accelerator memory data, eagerly update the contests of accelerator memories, and to cache read request for accelerator-hosted data in the CPU. Hence, NUAMA reduces the cost of reaching accelerator

memory from the CPU. Simulation results for NUAMA show negligible performance overheads and large programmability improvements with minor hardware additions.

1.3.4 Asymmetric Distributed Shared Memory Model

This thesis introduces the ADSM mode, where data used by accelerators is hosted by the memory attached to the accelerator using such data. Accelerator memories are mapped into the application virtual address space, so CPUs can access accelerator hosted data using regular load/store instructions. However, a given accelerator is constrained to access those regions of the virtual memory address space that contain data hosted by its own memory. This distribution asymmetry allows all required memory coherence actions to be performed at the CPU. The necessary API calls and memory coherence and consistency of an ADSM system are discussed in this dissertation. The design choices for a run-time system that implements the ADSM model are discussed too. A concrete operating system and accelerator independent ADSM run-time system design is detailed. The lack of hardware support in current accelerators does not allow a straightforward way of implementing ADSM run-time systems. A set of software techniques that allows building an user-level ADSM library for current accelerators on top of existing operating systems is presented. The limitations and overheads of each presented software technique are studied.

1.3.5 HPE Model

The HPE model for heterogeneous parallel systems is introduced in this dissertation. This model is fully compatible with the existent sequential model based on execution threads. In this model, the execution thread abstraction is extended with execution modes, which represent the capabilities (e.g., accessible virtual address space and ISA) of the execution thread when running on a given execution mode. In this model, accelerator calls are implemented as execution mode switches. On an execution mode switch, the application execution flow might migrate to an accelerator, if present, or fall back to a compatibility acceleration emulator. This approach allows execution applications that make use of accelerators in systems without them.

Chapter 2

Related Work

2.1 CPU – Accelerator Architectures

This section reviews existent CPU – accelerator architectures. First, a taxonomy of CPU – accelerator architectures is presented. This taxonomy classifies accelerators depending on the granularity of the dataset of the computations done by the accelerator. This taxonomy is used to help the reader to understand the scope of the present dissertation.

2.1.1 Fine-grained Accelerators

Fine-grained accelerators, also known as co-processors, are typically integrated in the CPU pipeline as functional units. Some examples of this kind of accelerators are floating-point and SIMD units, such as the SSE [Int07] and 3DNow! [AMD06] instructions found in x86 CPUs, and more general interfaces, such as the MIPS [MIP01] and ARM [Sea00] accelerator interfaces. The communication between integrated accelerator and the CPU is done by means of registers and is controlled by instructions that extend the ISA. There are several proposals for integrating more flexible and coarse-grained accelerator in a similar way. For instance, Chimaera integrates reconfigurable logic as a functional unit inside the core [HFHK04] which is able to access the general-purpose registers of the CPU and perform arbitrary arithmetic functions.

There are examples of CPU – accelerator interconnects in the field of reconfigurable accelerators. Garp [HW97] connects a reconfigurable accelerator to the CPU registers and the data cache.

2.1.2 Medium-grained Accelerators

Medium-grained accelerators are typically integrated in the same chip that CPU cores. These accelerators typically implement a small-size scratch-pad memory to store input, output, and temporary data, which is directly accessed by the accelerator logic. Medium-grained accelerators systems, such as MorphoSys [SLL+00] or MOLEN [VWG+04] implement a DMA engine to transfer data between system memory and the accelerator scratch-pad memory. An exception is OneChip, where the accelerator logic directly accesses to system

memory [JC99]. Most medium-grained accelerator platforms extend the CPU ISA with new instructions to trigger DMA transfers between system memory and the accelerator scratch-pad memory and to start the execution at the accelerator.

Commercially available platforms, such as Xilinx Virtex 5 FXT FPGAs [Xil09], allow implementing medium-grained accelerators that can be reconfigured at run-time. In Virtex 5 FPGAs accelerators are typically attached to the Processor Local Bus (PLB) and implement a DMA engine to transfer data between the accelerator scratch-pad memory and system memory. Accelerators also implement control registers that are mapped to the system physical address space. Hence, data transfers and computations are initiated writing predefined values to the accelerator control registers.

2.1.3 Coarse-grained Accelerators

Coarse-grained accelerators and CPUs typically are separated chips with separate memories. However, there are examples of coarse-grained accelerators and CPUs that are integrated in the same chip. For instance, in the Cell BE, the Power Processing Unit (general purpose processor), the Synergistic Processing Units (accelerators), the L2 cache controller, the memory interface controller, and the bus interface controller are connected through an Element Interconnect Bus in the same chip [KDH+05]. AMD Fusion chips will integrate CPU, memory controller, GPU and PCIe controller into a single chip. The Intel Graphics Media Accelerator [Int05] and the NVIDIA ION chips are examples of coarse-grained accelerators that share memory access with general purpose processors. These Integrated Graphics Processor (IGP) systems integrate a GPU inside the Graphics and Memory Controller hub, which manages the flow of information between the CPU, the system memory interface, and the I/O controller.

Examples of coarse-grained accelerators that have their own memory are NVIDIA Tesla GPUs. These accelerators are connected to the CPUs through a PCIe bus and include their own GDDR memory. Future graphics cards based on the Intel Larrabee chip will have a similar configuration [SCS+08]. The RoadRunner supercomputer is composed of nodes that include two AMD Opteron CPUs (IBM BladeCenter LS21) and four PowerXCell chips (2x IBM BladeCenter QS22). Each LS21 BladeCenter is connected to two QS22 BladeCenters through a PCIe bus [BDH+08]. The Cray XD1 connects an FPGA chip to the CPU using a direct connection through a dedicated I/O bus [FADJ+05].

2.2 CPU – Accelerator Data Transfers

The two preliminary steps for mapping applications into a data-parallel architecture are profiling to identify the computation intensive parts of the applications [EPP+01] and a detailed analysis of these parts to determine, for a given

accelerator architecture, if they are suitable to be implemented by an accelerator [GNVV04]. With these analyses in hand, the application is modified to perform data transfers and synchronization between the CPU and the accelerator. Appendix A presents a framework that automates these tasks.

CPU – accelerator systems implement data transfer as a marshalling process and a copy of the input data from system memory to the accelerator memory and vice versa using DMA [KMK01]. For instance, MorphoSys [SLL+00], the Cell processor [GHF+06], and the NVIDIA Tesla include DMA engines to copy data between the accelerator and the system memory.

Garp [HW97] and OneChip [JC99] avoid data copying by allowing the accelerator to access the CPU memory hierarchy. This approach requires implementing a memory controller for each accelerator that must implement the memory coherence protocol, perform memory translation, and ensure protection.

Guo [GNVV04] identified the data transfer process as a bottleneck in accelerator architectures. He proposes the *smart buffer*, a compiler technique that minimizes the data to be copied [GBN04]. A smart buffer compiler exploits the fact that the input data on consecutive calls to a given accelerator frequently share items with previous calls; these items do not need to be copied. There exist similar techniques for propagating values in shared memory multiprocessors, such as data forwarding [KCPT95].

2.3 Programming Models

This section presents an overview of existent programming models for CPU – accelerator systems. The scope of this section is the CPU – accelerator interface, so the accelerator programming model is omitted.

Most existent programming models present a common characteristic: accelerator and system memories are presented as two separate virtual address spaces to programmers. The programming models presented in this section significantly differ in the execution model (i.e., accelerator invocation). The different execution models involve different mechanisms to make accelerator input data accessible to the accelerators present in the system, and the accelerator output data accessible to general purpose processors. In this section we classify programming models depending on how accelerators are invoked.

2.3.1 Function Call Based Programming Models

Computations at accelerators are presented to programmers as function calls in many commercial programming models. These programming models require DMA transfers between CPUs and accelerators to move accelerator input and output data between the system and accelerator address spaces. Examples of function call based programming models are the IBM SPE Runtime Management [IBM07], NVIDIA CUDA [NVI09], ATI CTM [ATI06], and

OpenCL [Mun09].

The scope of the IBM SPE Runtime Management programming model is the Cell BE chip (e.g., a single QS22 BladeCenter), where SPUs are accelerators and the PPU the general purpose processor. Systems, such as the RoadRunner supercomputer, where the Cell BE chip is combined with general purpose processor chips (e.g., AMD Opteron chips in the RoadRunner), only use the IBM SPE Runtime Management for the code running on the Cell BE. Typically, these systems also use a function call based programming model for the interaction between the code running on the Cell BE chip and the code running on CPUs.

Function call based programming models expose DMA operations to applications programmers through memory copy routines. For instance, in the CUDA programming model an application programmer can transfer data from system memory to the accelerator memory by calling to a memory copy routine which receives three parameters: a destination pointer to the accelerator memory space, a source pointer to system memory, and the number of bytes to be copied. This memory copy interface ensures that the accelerator can only access the part of the application dataset that is explicitly requested by the memory copy parameters. In the remaining of this dissertation the terms DMA-based programming model will be use to refer to these programming models too.

Typically, DMA operations can only be initiated by the code running on the CPU. However, some DMA based programming models such as the IBM SPE Runtime Management and NVIDIA CUDA, starting at version 3.0, support DMA transfers initiated from accelerators. Furthermore, NVIDIA CUDA supports mapping system memory into the accelerator virtual address space and, thus, mapped system memory regions can be directly accessed from the accelerator. The IBM SPE Runtime Management includes the opposite feature: accelerator memory can be mapped into the application address space and, therefore, general purpose processors can access accelerator memory.

Execution at the accelerator in these programming models is completely independent of data transfers. These programming models typically define an accelerator invocation routine that takes a pointer in the accelerator virtual address space to the first instruction to be executed. Most DMA based programming models do not place any constraint on the parameters passed to accelerators. An exception is the OpenCL standard that requires all pointers used by the accelerator to be passed as parameters during the accelerator invocation [Mun09].

Function call programming models are being used in homogeneous parallel systems too. Darlingtonet. al. [DFH+93] proposed using *skeleton* functions as part of the programming model. Skeleton functions implement parallel algorithms commonly used in a given class of applications. This approach is, for instance, behind STAPL which provides parallel implementations of STL C++ functions [AJR+03].

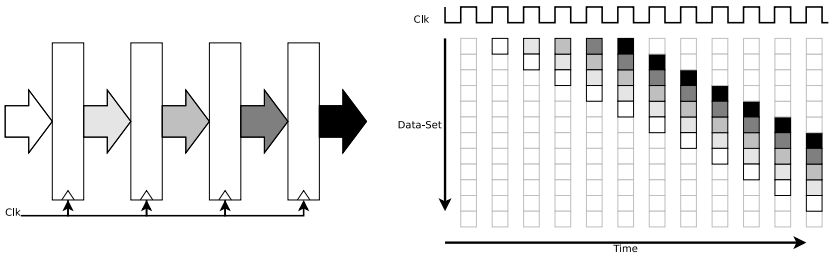


Figure 2.1: Example of application-specific pipelined logic

2.3.2 Stream Based Programming Models

Stream based programming models, such as ImpulseC [MB06] and SGI [SGI08] are mainly used for systems using FPGA based accelerators. These programming models assume that accelerators process streams of data. FPGA accelerators typically accelerate application execution by implementing application-specific logic that efficiently implements computations that are extensively used by the application. Application-specific logic is typically pipelined to exploit data parallelism, as shown in Figure 2.1. Each stage in the pipeline in Figure 2.1 operates over different parts of the application dataset. The application-specific logic is replicated few times to increase the amount of data parallelism exploited by the accelerator: each replica operates over different parts of the dataset. However, the number of replicas is typically low due to the limited reconfigurable logic area available in current FPGA chips.

Stream programming models have been also used in massive data parallel homogeneous systems, such as clusters of computers. For instance, ASSIST [Van02] decomposes programs into components that communicate through data streams. ASSIST requires the programmer to declare modules and connect them using streams. This data-dependence information is used by the ASSIST run-time to schedule the execution of modules on different processors.

Stream based programming models provides application programmers with routines to create accelerator input, output and input/output streams. There are typically two operations the code running on the CPU can perform over an accelerator stream: push data into input and input/output streams, and pull data out of output and input/output streams. These push/pull operations implement control-flow mechanisms: the application is blocked on a push operation whenever the accelerator input buffer is full and on a pull operation whenever the accelerator output buffer is empty.

Execution at the accelerator requires one input and one output stream, or one input/output stream to be passed as parameters. Accelerators are constrained to read data from and write data to streams received as parameters, so only data explicitly pushed by the application can be accessed.

2.3.3 Task Based Programming Models

Task based programming models for CPU – accelerator architectures divides applications in a set of tasks that can be executed by both general purpose processors and accelerators. Two examples of task based programming models are StarSS [BPBL06] and WorkForce. These programming models are typically build on top of DMA based programming models and aim to simplify the process of coding applications for CPU – accelerator architectures. The OpenCL specification also aims to provide a task-like approach, but in its current state the OpenCL programming model is closer to function call based programming models.

Task based programming models require programmers to define tasks composing the application. Task definition can be done at compile time (e.g., using `#pragma` constructions), such in StartSS, or at run-time, such in WorkForce. Task definition requires application programmers to identify input, output and input/output data structures for the task. This information is essential to allow efficient task scheduling (e.g., scheduling two independent tasks to run concurrently on different accelerators). Data structures used by accelerators do not require any special handling, contrary to explicit memory copies and push/pull stream operations required by DMA and stream based programming models respectively.

Tasks execution is started through task creation routines, which do not differ from traditional execution thread creation routines available in most operating systems. Additionally, task based programming models offer routines to wait for a given task to complete its execution.

2.3.4 Comparison of Programming Models

The programming models discussed previously are not exclusive, and they present different abstraction levels to programmers. Function call based programming models present the lowest-level of abstraction. Programmers explicitly launch the accelerator execution, being the programmer in charge of selecting the accelerator executing each kernel. Furthermore, separate system and accelerator virtual address spaces are presented to programmers, who must explicitly request DMA transfers before and after accelerator execution. This programming model exposes hardware details, such as the existence of separate accelerators and memory address spaces, to programmers. Such low-level of abstraction allows building other programming models on top of function call programming models. This dissertation extends the memory and execution model of function call based programming models to improve programmability and performance. Hence, higher-level programming models (i.e., task and stream based) also benefit from the results presented in this dissertation.

Stream based programming models are typically built on top of function call programming models as run-time systems. Computations at accelerators are

encapsulated as functions which are launched and terminated at accelerators by the run-time system. Stream push and pull operations are implemented through memory copy operations and synchronization primitives. For instance, stream input buffers, filled by push operations, are transferred from system to accelerator memory through memory copy routines[1].

Task based programming models are typically implemented on top of function call based programming models too. User defined tasks are mapped to function calls, being the run-time system in charge of selecting the most adequate accelerator to execute the task. The run-time system copies the task input and output data through memory copy routines provided by the low-level programming model.

2.4 Distributed Shared Memory

Many hardware and software Distributed Shared Memory (DSM) systems exist, that implement a shared address space on top of physically distributed memories. In this dissertation, Chapter 6 presents a DSM system for heterogeneous parallel systems.

Hardware DSM systems include the necessary logic to implement coherence protocols and detect accesses to shared data. Most hardware DSM systems implement write-invalidate protocols, rely on directories to locate data, and have a cache-line size sharing granularity [DSF88, WH88, LLG+90, MW90, Gus92, FBR93]. There are also hardware implementations that rely on software to implement data replication [BR90], to support the coherence protocol [ABC+95], to virtualize the coherence directory [WLT93], or to select the appropriate coherence protocol [HKO+94].

Software DSM systems might be implemented as a compiler extension or as a run-time system. Compiler based DSM systems add semantics to programming languages to declare shared data structures. At compile time, the compiler generates the necessary synchronization and coherence code for each access to any shared data structure [ACG86, BT88].

Run-time DSM implementations provide programmers with the necessary APIs to register shared data structures. A software run-time system uses the memory protection hardware to detect accesses to shared data structures and to perform the necessary coherency and synchronization operations. The run-time system might be implemented as part of the operating system [FP89, DLAR91] or as an user-level library [LH89, CBZ91, BZS93, KCDZ94]. The former allows the operating system to better manage system resources (e.g., blocking a process while data is being transfered) but requires a greater implementation effort. For instance, in Amoeba [TvRvS+90] processes can declare shared memory segments that are accessible from processes running in remote nodes, and access

[1]Different programming models define different policies to trigger data copy operations between system and accelerator memories

to shared memory segments is grated through capability delegation and copy operations. User-level DSM libraries require the operating system to bypass and forward protection faults (e.g., as POSIX signals) and to provide system calls to interact with the memory protection hardware. This approach is also taken in TreadMarks [KCDZ94], CRL [JKW95], and Shasha [SGT96].

Object-based DSM systems allow applications to invoke methods of objects that are shared among processes running on different network nodes. For instance, Linda [ACG86] allows applications to insert objects into *n-adas*. Objects within a *n-ada* are accessible by all processes in the system. A similar approach is taken in Orca [Bal90], which defines a programming language that allows the creation of shared objects and remote processes.

Heterogeneity has been also considered in software DSM systems [BF88, ZSM90]. These works mainly deal with different endianisms, data type representations and machines running different operating systems.

2.5 Operating Systems

Chapter 7 in this dissertation presents a novel execution model to integrate accelerators in the operating system (OS). Previous research on OS support for heterogeneous systems focuses on policies to manage the different properties (e.g., memory access latency) of heterogeneous processors and to manage resource sharing (e.g., shared caches in many-core processors). The MIT exokernel [EKO95] supports heterogeneity by exporting the hardware diversity to user-level applications. The Infokernel [ADADB+03] supports hardware diversity by implementing abstractions that export the internal OS kernel state to user-level. This abstractions might be used by programmers to implement application-specific policies. The Infokernel assumes that all processors in the system belong to the same process type (e.g., all processors can execute OS kernel code).

The multikernel model [BBD+09] modifies the user process abstraction to be a collection of dispatcher objects, one on each core on which it might execute. The multikernel model also assumes that all processors in the system are capable of executing OS kernel code (i.e., object dispatchers). This assumption does not hold in most current heterogeneous systems. For instance, the IBM Cell Synergistic Processing Elements, reconfigurable logic in FPGAs, and GPUs can only execute user-level code.

OS support for reconfigurable hardware has been an active research field [Bre96, WK01, WP03, KL08]. This research mostly investigates reconfigurable logic placement and virtualization and assumes an execution model where accelerators are abstracted as I/O devices.

The IBM Cell processor is currently supported by the Linux kernel [Ber05]. IBM Cell Synergistic Processing Elements are abstracted as execution threads

within the user process. This approach requires multi-threading to use accelerators, which harms programmability.

The IBM BlueGene/L OS also handles the heterogeneity of the underlying hardware. The BlueGene/L machine is formed by compute and I/O nodes, interconnected in a regular topology. The Blue Gene/L OS abstracts the machine in *psets*, which consist of one I/O node and a collection of compute nodes. A *pset* maps to several user processes, each one running on an processor of the compute nodes and one user process running in the I/O node. This latter user process is the only one allowed to perform I/O operations [GBC+05]. The IBM Blue Gene/L OS targets supercomputing environments, designed to efficiently parallel Message Passing Interface (MPI) applications.

MOSIX [BL85] is a single image operating system that allows user processes to migrate between nodes of a computer network. The Sprite network operating system also implements a user process migration mechanism [DO99]. Process migration has been also implemented in many distributed operating systems, such as Amoeba [TvRvS+90], Mach [MZDG93] and Chorus [RAA+88]. Process migration has been a quite active research area, and the reader is referred to the survey from Milojicic et. al. [MDP+00] for further information.

Chapter 3

Reference Hardware and Software Environment

3.1 Reference CPU – Accelerator Architecture

The target of this thesis are coarse-grained massively-parallel accelerators systems. For the sake of simplicity, the term accelerator is used to refer to this kind accelerators in the remaining of this dissertation.

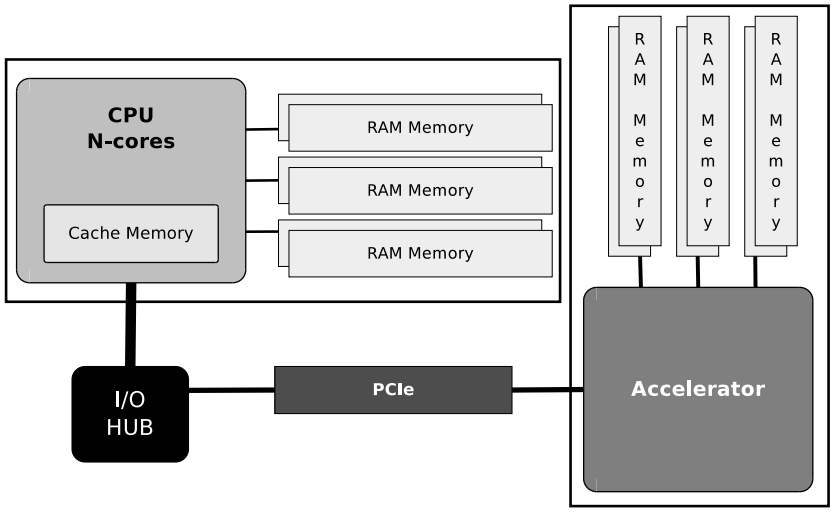


Figure 3.1: Reference Single-Accelerator Architecture

Figure 3.1 shows the base single-accelerator system assumed in this dissertation. The accelerator includes its own local memory and it is connected to the CPU cores through a PCIe bus. Additional accelerators can be connected to the PCIe bus to build a multi-accelerator system.

The base architecture in Figure 3.1 is currently implemented in most desktop systems that include a GPU card. There are examples of commercially available desktop computers including up to four GPU cards. The RoadRunner supercomputer [BDH+08] also implements the system architecture adopted in this dissertation. Figure 3.2 shows the architecture of a single RoadRunner node; The IBM LS21 BladeCenter includes two AMD Opteron chips, connected through a HyperTransport link. Two I/O Hubs, are attached each one to an AMD Opteron chip. Each I/O Hub in the LS21 BladeCenter and one IBM QS22 BladeCenter are connected through a PCIe bus. Notice that in the RoadRun-

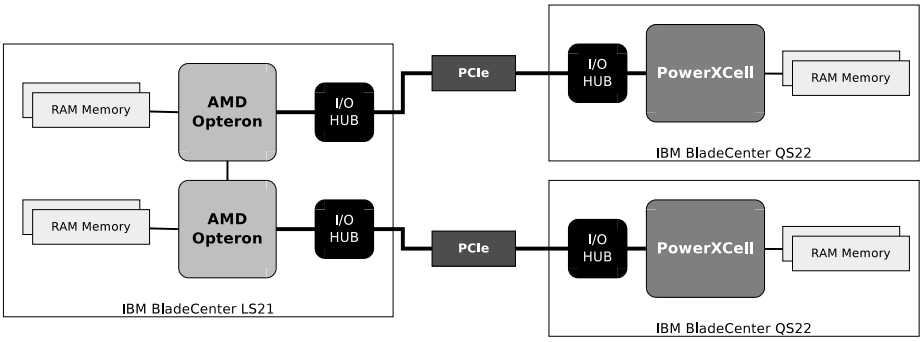


Figure 3.2: Architecture of a compute node of the RoadRunner supercomputer

ner supercomputer, QS22 BladeCenters are used as accelerators and the Power Processing Unit of the PowerXCell chips is only used to handle data transfers between QS22 and LS21 BladeCenters. This is in contrast with an approach taken, for instance, in the Sony PS/3 where the IBM Cell chip is a full heterogeneous system formed by one on-chip CPU (i.e., Power Processing Unit) and up to six medium-grained accelerators (i.e., Synergistic Processing Elements). The approach taken in the RoadRunner supercomputer is also adopted in this dissertation.

This dissertation assumes that CPUs include on-chip cache memories and Memory Management Units (MMU) that provides both memory protection and virtualization. This dissertation assumes simple accelerators without virtual memory and protection. Moreover, the reference architecture assumes that accelerators are not capable of performing I/O operations. However, accelerators might delegate the execution of these operations to CPUs using interrupts: the accelerator sends an interrupt to the CPU, which is handled by the operating system, to request I/O operations.

The base architecture assumed in this dissertation has two key properties:

- Distributed memory: CPUs and accelerators use memories that are physically separated.

- Non-coherent memory: memory accesses to memory elements that are not physically attached to the processing element (i.e., a CPU or an accelerator) do not trigger any coherence action.

Note that both distribution and non-coherency, are orthogonal properties. For instance, commodity systems based on Intel GMA, AMD Fusion, and NVIDIA ION chips connects both, GPU and general purpose processors to system memory. However, memory accesses from different processing elements do not trigger any coherence action. The orthogonality of these properties allows a separate analysis of each property.

3.1.1 Distributed Memory

Accelerators and general purpose processors impose very different requirements on memory controllers. CPUs are designed to minimize single-thread execution latency and typically implement some form of strong memory consistency (e.g., sequential consistency in MIPS CPUs). Accelerators design tries to maximize multi-thread throughput and typically implement weak forms of memory consistency (e.g., Rigel implements weak consistency [KJJ+09]). Memory controllers for general purpose processors tend to implement narrow memory buses (e.g., 192 bits for the Intel Core i7) compared to data parallel accelerators (e.g., 512 bits for the NVIDIA GTX280 GPU) to minimize the memory access time. Relaxed consistency models implemented by accelerators allow memory controllers to serve several requests in a single memory access. Strong consistency models required by CPUs do not offer the same freedom to rearrange accesses to system memory. Memory access scheduling in the memory controller has also different requirements for general purpose processors and accelerators (i.e., latency vs. throughput). CPU – accelerator systems typically attach CPUs and accelerators to separate memories to meet the different aforementioned requirements. Systems where CPUs and accelerators share the same physical memory (i.e., IGP systems) typically deliver lower performance than those systems using separate memories.

Figure 3.3 illustrates the need for a separate accelerator memory to accomplish high performance. This figure shows an estimation of the required memory bandwidth for different values of Instructions Per Cycle (IPC) in a set of the NASA Parallel Benchmarks. Interconnection fabrics used in most modern CPUs (i.e., Hyper-transport and QPI) limit the maximum IPC achievable in all benchmarks. For instance, QPI only allows a value of IPC higher than 40 in BT. However, the bandwidth delivered by the NVIDIA GTX295 GPU allows IPC values of up to 100 for all benchmarks but UA.

3.1.2 Non-coherent Memory

Application phases amenable to be implemented by accelerators typically operate over large datasets and, therefore, accelerators typically perform a high number of concurrent memory accesses while executing. Fully coherent CPU – accelerator systems might potentially produce high memory coherence traffic that would degrade the overall system performance. For instance, this situation occurs whenever data structures are sequentially accessed by general purpose processors and accelerators. Accesses from the CPU will bring data to the CPU cache memory. Consequent write accesses from the accelerator to this data will require coherence traffic to invalidate cache-lines at the CPU containing data being modified by the accelerator.

Fully coherent CPU – accelerator systems require both, CPUs and accelerators, to implement the same memory coherence protocol. Hence, it would

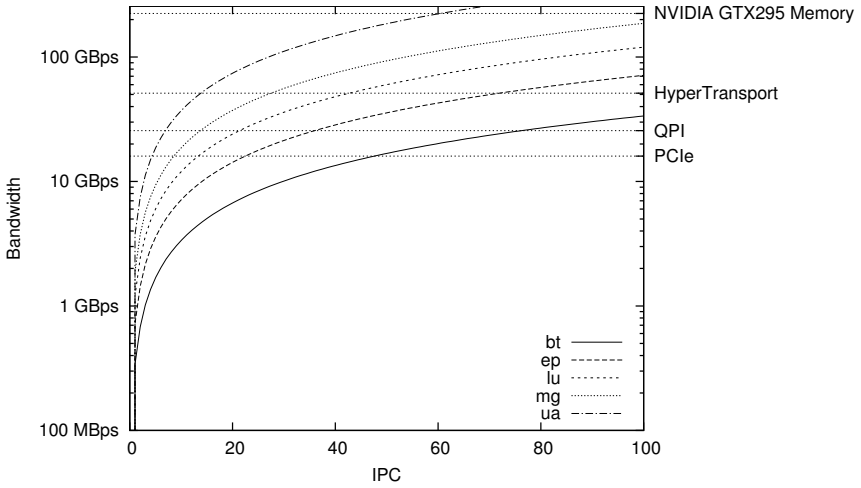


Figure 3.3: Estimated memory bandwidth for different values of instruction per cycle in some NASA parallel benchmarks

be highly difficult, if not infeasible, to use the same accelerator (e.g., a GPU) in systems based on different CPU architectures. This constraint would impose a significant economic penalty to accelerator manufacturers. Furthermore, full system coherence require accelerators to implement the coherence protocol, which would consume silicon area currently devoted to processing units and, thus, reduce the benefit of using accelerators.

Most commercially available CPU – accelerator systems do not keep coherence between system and accelerator memories. For instance, system memory is shared between the CPU and the accelerator in IGP systems based on Intel GMA, AMD Fusion, or NVIDIA ION chips. However, memory accesses from the GPU do not produce any coherence traffic to access the CPU cache hierarchy to check for an updated copy of the data being accessed.

Non-coherent accelerator memory requires the contents of CPU cache memories have to be flushed before transfers between system and accelerator memories. As a result, the application working-set has to be brought back from system memory to cache memories after data transfers.

3.2 Reference Programming Model

This dissertation assumes a programming model where accelerators are encapsulated as function calls in the application code. This functions executed by accelerators are called *kernels*. This programming model is similar to NVIDIA CUDA [NVI09], OpenCL [Mun09], future Intel Larrabee [SCS+08] graphics processors, ATI CMT [ATI06] and the RoadRunner supercomputer [BDH+08].

Listing 3.1 shows a simple example of a code that calculates the zeros of a generic function and illustrates the programming model used in this dissertation. The execution flow for the code at Listing 3.1 is shown in Figure 3.4. First,

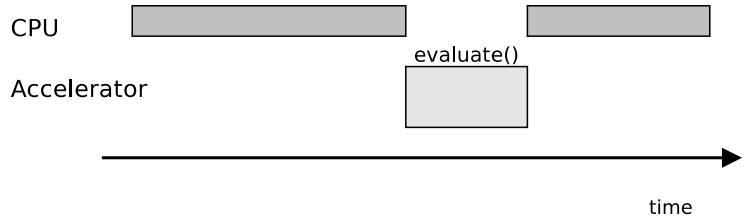


Figure 3.4: Execution flow for the code in Listing 3.1

```
1  int main(int arc, char *argv[])
2  {
3      int i;
4      float *x, *accX;
5      float *y, *accY;
6      dim3 dimGrid, dimBlock;
7
8      /* Allocate and initialize input data */
9      x = malloc(SIZE * sizeof(float));
10     for(i = 0; i < SIZE; i++) x[i] = i * DELTA;
11     accMalloc(&accX, SIZE * sizeof(float));
12     accMemcpy(accX, x, SIZE * sizeof(float));
13
14     /* Allocate output data */
15     y = malloc(SIZE * sizeof(float));
16     accMalloc(&accY, SIZE * sizeof(float));
17
18     /* Evaluate function at GPU */
19     evaluate(accY, accX);
20
21     /* Read output data and check for zeros */
22     accMemcpy(y, accY, SIZE * sizeof(float));
23     for(i = 0; i < SIZE; i++)
24         if(y[i] == 0) printf("Function␣has␣a␣zero␣at␣%f\n", i * DELTA);
25
26     /* Free memory */
27     accFree(accX);
28     accFree(accY);
29     free(x);
30     free(y);
31 }
```

Listing 3.1: Example of the programming model used in this dissertation

the CPU starts executing the code until it reaches line 19 (`evaluate(accY, accX);`), where kernel execution at the accelerator is invoked as a function call. The kernel call triggers the execution and the accelerator and waits until the kernel execution is done. When the kernel finishes executing at the accelerator, the remaining code (lines 21 to 31) is executed sequentially by the CPU.

This programming model exports to the programmer the existence of two separate virtual address spaces for system and accelerator memories, influencing the application code in two different ways. First, data structures accessed by the accelerator are allocated twice, one per address space. Second, consistency between address spaces is explicitly managed through memory copy routines. For instance, in Listing 3.1, system memory is allocated for the accelerator input data at line 9, and accelerator memory is allocated for the same data structure

```
1     evaluate<<<dimGrid, dimBlock>>>(accY, accX);
```

Listing 3.2: Example of CUDA code

at line 11. This double allocation requires two separate variables (`float *x, *accX;`) to keep the data structure memory address in system and accelerator memory respectively. Analogously, two variables (`float *y, *accY;`) are used to reference the accelerator output data, which is also double allocated (lines 15 and 16). Furthermore, allocated data structures must be released at both system and accelerator memory (lines 27 – 30). Listing 3.1 also illustrates explicit consistency management. The accelerator input data (`*x`) is initialized in the `for` loop at line 10. However, only system memory is modified by this loop, and, therefore, a explicit memory copy is required before the kernel call (line 12) to ensure that the accelerator memory contains the input data. Similarly, data produced by the accelerator is only written to accelerator memory during the kernel execution. Hence, a explicit copy from GPU to system memory is required (line 22) to ensure that the loop at lines 23 and 24 access the data produced by the accelerator.

3.3 NVIDIA® CUDA™

The software proposals done by this dissertation (Chapters 6 and 7) target NVIDIA GPUs (i.e., accelerators) and the CUDA programming model. The execution and memory model of NVIDIA CUDA is quite similar to the reference programming model previously described. However, kernel invocation in NVIDIA CUDA uses different semantics. For instance, Listing 3.2 shows an example of a kernel call in CUDA. Kernel calls in CUDA take two additional parameters (`<<<dimGrid, dimBlock>>>`) to set up the thread organization executing the kernel.

The specification of CUDA covers concepts ranging from the execution model on GPUs to the interface between CPUs and GPUs. We refer the reader to the NVIDIA CUDA documentation for a more comprehensive description of this programming model [NVI09]. Only the specific concepts that are used in the rest of the paper will be highlighted in this section.

3.3.1 Multi-GPU Support

CUDA allows applications to execute kernels in several GPUs through *CUDA contexts*. A CUDA context effectively represents a GPU device that is binded to an application execution thread. Kernel invocations and memory copy operations performed by the application thread will be done over the device associated to the CUDA context. Application threads can request the creation of new contexts, but CUDA imposes the following constrains:

Processor	Memory Subsystem
Freq: 5GHz	L1 ICache: 16K (4-way, 1 port)
Fet/Iss/Ret width: 4/4/5	L1 DCache:16K (4-way, 1 port)
LdSt/Int/FP units: 4/4/3	L1 Hit/Miss Delay: 2/13
RAS: 32	L2 Cache: 1MB (16-way, 2 ports)
BTB: 2K entries, 2-way assoc.	L2 Hit/Miss Delay: 10/105
Branch pred:	ITLB entries: 64 (4-way)
bimodal size: 16K entries	DTLB ent: 64 (4-way)
gshare-11 size: 16K entries	Main Mem: 271
I-window: 92	Acc Mem Delay: 271
ROB size: 176	Acc Link Channels: 16
Int regs: 96	Acc Link Latency: 1
FP regs: 80	
Ld/St Q entries: 56/56	
IMSHR/DMSHR: 4/16	

Table 3.1: Simulation parameters. Latencies shown in processor cycles representing minimum values.

- An application thread can only have one active CUDA context. Whenever an application thread attaches a new CUDA context, the old context becomes inactive.
- A CUDA context can only be active in one application thread. Attaching a CUDA context that is attached to other application thread will fail.

3.3.2 CUDA Streams

CUDA supports concurrent execution of kernels at the GPU and DMA transfers from and to GPU memory by means of streams. A CUDA stream is a sequence of commands that execute in order, but different CUDA streams might execute their commands out of order with respect to one another or concurrently. Thus, for kernel execution and DMA transfers to happen simultaneously, they must belong to different CUDA streams. Applications might overlap kernel execution, data transfers from system to GPU memory and from GPU to system memory using three streams. In a multi-GPU system, a CUDA stream is associated to the CUDA context where was created. Hence, all CUDA streams that belong to the same CUDA context will execute all actions (i.e., kernel calls and data transfers) in the GPU associated to such CUDA context.

3.4 Evaluation Methodology

3.4.1 Simulation Environment

Hardware modifications proposed in Chapter 5 are evaluated using execution-driven simulations. The architecture of the simulated CPU is shown in Table 3.1. The simulated DMA controller can read data from the L2 cache if present and from main memory otherwise. DMA transfers use burst reads and writes of 128 bytes in memory blocks of up to 4KB.

The evaluation is performed using a cycle-accurate simulator based on SESC [RFT+10], which is modified to incorporate DMA transfers, accelerator execution and the

System	CPU		GPU	
	Processor	Memory	Processor	Memory
GTX285	2x AMD Opteron 2222 3 GHz 2MB L2	4x 2 GB DDR2 667 MHz	2x GTX285 30 SM 1476 MHz	2x 1 GB GDDR3 1242 MHz
C870	2x Intel Xeon E5420 2.5 GHz 6MB L2	4x 2 GB DDR2 667 MHz	2x C870 16 SM 1350 MHz	2x 1.5 GB GDDR3 800 MHz

Table 3.2: Target systems used in the evaluation

hardware modifications proposed in this dissertation. The simulated CPU code is compiled using gcc version 3.4 with the `-O3` flag to MIPS binaries.

Functional and timing simulation of the code executed by the CPU is performed. The code executed by the accelerator is natively executed in a NVIDIA GPU, so only CPU execution is considered. Total application execution time is estimated using the contribution of accelerator execution time to the total application execution time. This contribution is obtained by executing the benchmark on the systems described in the following section.

3.4.2 Execution Environment

Experiments in Chapters 6 and 7 are executed in the 64-bit x86 systems outlined in table 3.2. All machines run a GNU/Linux system, with Linux kernel 2.6.32 and NVIDIA driver 195.36.15. Benchmarks are compiled using GCC 4.3.4 for CPU code and NVIDIA CUDA compiler 3.0 for GPU code.

Execution times are taken using the `gettimeofday()` system call, which offers a microsecond granularity. All experiments are executed 2048 times, and samples that are two times higher than the arithmetic average are considered outliers and discarded. All results use the arithmetic average value of the filtered results, and the standard deviation is used as an estimation of the error range.

GTX285, in Table 3.2, is a commodity GPU system typically found in high-end desktop systems; C870, in Table 3.2, is a GPU system designed for high-performance environments. The GPUs in GTX285 are from a newer generation than the GPUs included in C870. Hence, some hardware features , such as concurrent DMA transfers and GPU execution, are only available on GTX285. Moreover, the peak performance of each GPU in GTX285 is 1062 GFLOPS, while each GPU is C870 reaches up to 518 GFLOPS. The peak performance accomplished by the systems has little impact over the experimental results presented in this dissertation because only data communication mechanisms are studied. These mechanisms mainly depend on the available bandwidth between system and accelerator memory in each system. Figure 3.5 shows the effective data transfer bandwidth between CPU (host) and GPU (device) for different data transfer sizes. Both systems accomplish a maximum data transfer bandwidth for large data sizes. Moreover, in both cases, there is a significantly hop in the effective accomplished bandwidth when the data transfer size goes

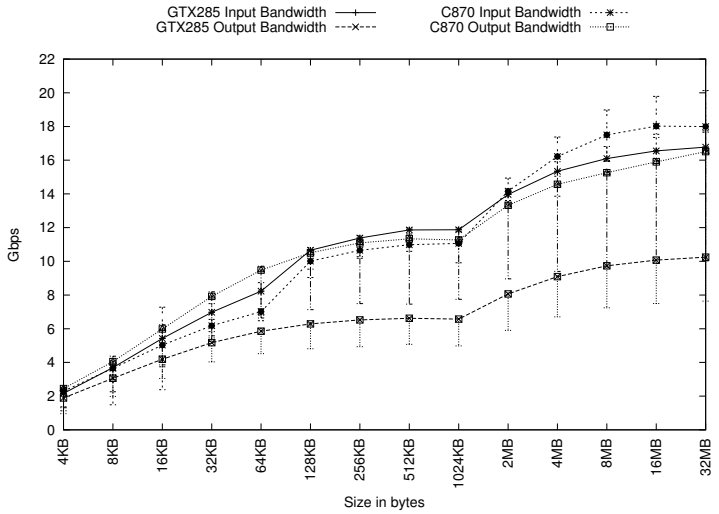


Figure 3.5: GPU bandwidth for two different GPUs (GTX285 and C870)

from 1MB to 2MB. The available bandwidth in GTX285 is smaller than in C870, so the impact of CPU – accelerator data transfers in this system is larger.

3.5 The Parboil Benchmark Suite

The Parboil benchmark suite is a set of benchmarks designed to measure the performance of heterogeneous systems formed by general purpose processors and GPUs [IMP]. The Parboil benchmark suite is used to evaluate most proposed systems in the remaining of this dissertation. This section provides a brief description of the Parboil benchmark suite, an information that is common to most of the remaining of this dissertation.

3.5.1 Benchmark Description

MRI-Q

Magnetic Resonance Imaging Q (MRI-Q) is a computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.

MRI-FHD

Magnetic Resonance Imaging FHD (MRI-FHD) is a computation of an image-specific matrix F_d^H, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.

Benchmark	Host to Device		Device to Host	
	Total (MB)	# Transfers	Total (MB)	# Transfers
MRI-Q	3.05	7	2.00	3
MRI-FHD	3.07	11	2.02	4
CP	0.61	10	1.00	1
SAD	0.05	1	8.49	1
TPACF	4.73	2	0.03	1
PNS	0.00	0	0.02	224
RPES	61.86	2	4.15	1

Table 3.3: Data transfers in the Parboil benchmark suite

CP

Coulombic Potential (CP) computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. Adapted from 'cionize' benchmark in VMD.

SAD

Sum of Absolute Differences (SAD) is the sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.

TPACF

Two Point Angular Correlation Function (TPACF) is an equation used here as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body.

PNS

Petri Net Simulation (PNS) implements a generic algorithm for Petri net simulation. Petri nets are commonly used to model distributed systems.

RPES

Rys Polynomial Equation Solver (RPES) calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.

3.5.2 Characterization

For the purposes of this dissertation, the characterization of data transfers between the CPU and the GPU is the most important factor. Table 3.3 summarizes the data transfers between CPU (host) and GPU (device) for all Parboil benchmarks obtained using application profiling.

Parboil benchmarks might be categorized into small-size transfer benchmarks (PNS) and large-size transfer benchmarks (MRI-Q, MRI-FHD, and RPES).

SAD, and TPACF fit into both categories. SAD performs small-size data transfers from CPU to GPU, but large-size transfers from GPU to CPU. Analogously, TPACF performs large-size data transfers from CPU to GPU and small-size data transfers in the other direction. Finally, CP is a medium-size transfer benchmark, where data transfers are of moderate size in both directions. All benchmarks, but PNS, perform few data transfers between CPU and GPU. PNS requires 224 data transfers between the GPU and CPU.

The ratio of transferred data size over the number of data transfers can be combined to estimate the efficiency of data transfers. The larger this ratio, the more efficient data transfers are performed. SAD, TPACF, and RPES are the benchmarks that larger data transfers perform and, therefore, best utilize the PCIe bandwidth. PNS and CP, on the other hand, perform small data transfers so data transfer bandwidth accomplished is relatively low.

Chapter 4

Programmability of Heterogeneous Parallel Systems

4.1 Introduction

This chapter discusses programmability issues of heterogeneous parallel systems from the application programmer's perspective. The two major programmability problems present in current heterogeneous parallel systems identified in this chapter are separated CPU – accelerator memories, and disjoint virtual address spaces for CPUs and accelerators. The former harms programmability by requiring application programmers to explicitly copy data structures between system and accelerator memories. CPU – accelerator communication through memory copy routines prevents by-reference parameter passing to accelerator calls, requiring all parameters to be passed by-value. Moreover, this change in the parameter passing semantics might harm the application performance due to the cost of data marshalling (i.e., collecting the data to be copied) and the memory copy. Disjoint virtual system and accelerator address spaces increase the complexity of the code because data structures are referenced by different virtual memory addresses (i.e., pointers) in CPU and accelerator code. Such a constrain might also introduce performance penalties in the accelerator call.

This chapter first presents three different models for data transfers between accelerator and system memory: per-call, double-buffered, and accelerator-hosted. The first two models are currently in use and assume separate CPU and accelerator memories. Hence, in these two models data communication between CPU and accelerator happens through system and accelerator memories copy routines. These memory copy routines present a trade-off between programmability and performance in these two models. The third model, accelerator-hosted, is a contribution of this thesis. In this model accelerator memory is accessible from the CPU code and data structures required by the accelerator are only hosted in accelerator memory. The accelerator-hosted model might increase the accelerator memory capacity requirements but greatly improves programmability of heterogeneous parallel systems. This chapter shows how the accelerator-hosted model allows programmers to write simple CPU code for heterogeneous parallel systems that resembles to the code they would produce for a homogeneous (i.e., without accelerators) system. Hardware and software implementation alternatives for this model are presented in Chapters 5 and 6

respectively.

Finally, the implications of a unified virtual address space over the accelerator code are presented. This chapter argues that a unified virtual address space that includes both, system and accelerator memories, allows accelerator code to be written in a more straightforward way, eases the task of using data structures with embedded pointers (e.g., linked-lists) in the application code, and allows for potential performance gains. This chapter also outlines potential hardware and software approaches to build this unified virtual address space from physically separated system and accelerator memories.

4.2 Accelerator Data Transfer Models

A data transfer model defines the exchange of data between accelerators and CPUs during application execution. This dissertation identifies three different data transfer models. The first model, per-call, is the simplest among the three, but achieves the lowest performance. The second model, double-buffered, achieves higher performance than the first model, but requires high level of programming effort. The third model, accelerator-hosted, achieves the highest level of programmability among the three models and allows for potential performance gains.

4.2.1 Preliminaries

Data transfer models are illustrated using a linked-list sorting benchmark code. The sorting code implements the Batcher odd-even merge sort described by Satish et. al. [SHG09]. This sorting algorithm takes a N-element unordered vector of key/value pairs as input, and produces an ordered N-element vector of key/value pairs. The sorting benchmark execution is divided in three main stages:

- Initialization. The linked-list is initialized with random values. This phase is implemented in a completely sequential way.
- Sorting. The implementation of this function in an homogeneous system (i.e., without accelerators) is shown in Listing 4.1. First, a vector of key-value pairs is extracted from the linked-list sequentially in a *marshalling* process (lines 8 – 12). This vector is passed as input to the sorting function (line 15), which produces an ordered key-value vector. The output vector is finally used to re-build the output ordered linked-list during a *unmarshalling* process (lines 18 – 23).
- Checking. The linked-list is transversed to check for correctness in a sequential way.

The initialization and checking phases are inherently sequential processes that are executed in the CPU. These two stages are common to all discussed

```
1  template<typename S>
2  S *sort(S *ptr, size_t size)
3  {
4      /* Allocate system and accelerator memory */
5      pair<float, S> *buffer = malloc(size * sizeof(pair<float, S>));
6
7      /* Marshall: Build the input vector from the linked list */
8      for(int i = 0; i < size; i++) {
9          buffer[i].value =ptr->weight;
10         buffer[i].ptr = ptr;
11         ptr = ptr->next;
12     }
13
14     /* Call sorting function in the accelerator */
15     oddEvenMergeSort<float, S, pair<float, S> >(buffer, size);
16
17     /* Unmarshall: Re-build the linked list from the output buffer */
18     S *ptr = buffer[0].ptr;
19     for(int i = 1; i < size; i++) {
20         ptr->next = buffer[i].ptr;
21         ptr = buffer[i].ptr;
22     }
23     ptr->next = NULL;
24
25     /* Release memory */
26     free(buffer);
27
28     return ptr;
29 }
```

Listing 4.1: Linked-list sorting using only the CPU

data transfer models. The sorting stage is composed by both kinds of code, sequential and parallel, and, therefore, it might be executed by both the CPU and the accelerator.

4.2.2 Per-Call Data Transfer Model

The per-call data transfer model uses the accelerator memory to hold the data required by the following call to the accelerator. This model requires enough accelerator memory to store input and output data structures for each kernel call. In the linked-list sorting example, the sorting kernel uses a single input-output vector of key-value pairs, and, therefore, only this data structure is hold in the accelerator memory. In this example, sorting a 16-million element linked-list with 4-byte key/value fields requires a total of 122 MB of accelerator memory.

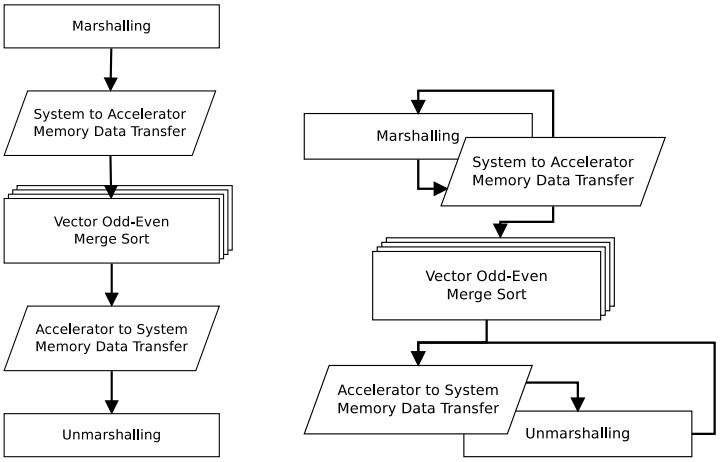
Figure 4.1(a) shows the flowchart for the sorting function in the linked-list example, whose source code is shown in Listing 4.2. In this function, the code at the CPU first allocates (lines 5 – 8) and builds (lines 11 – 15) the vector of key/value pairs traversing the linked list (marshalling). This process requires accessing each element in the list, which is a sequential process due to the existent data dependency between accesses to two consecutive elements in the list. Then, the input vector is transferred from system memory to the accelerator memory (lines 18 – 19), the sorting kernel at the accelerator is called, and the

```
1  template<typename S>
2  S *sort(S *ptr, size_t size)
3  {
4      /* Allocate system and accelerator memory */
5      pair<float, S> *buffer =
6          (pair<float, S> *)malloc(size * sizeof(pair<float, S>));
7      pair<float, S> *device = NULL;
8      accMalloc((void **)&device, size * sizeof(pair<float, S>));
9
10     /* Marshall: Build the input vector from the linked list */
11     for(int i = 0; i < size; i++) {
12         buffer[i].value =ptr->weight;
13         buffer[i].ptr = ptr;
14         ptr = ptr->next;
15     }
16
17     /* Copy the input buffer to the accelertor memory */
18     accMemcpy(device, buffer,
19         size * sizeof(pair<float, S>), memcpyHostToAcc);
20
21     /* Call sorting function in the accelerator */
22     accOddEvenMergeSort<float, S, pair<float, S> >(device, size);
23
24     /* Transfer the output buffer to system memory */
25     accMemcpy(buffer, device,
26         size * sizeof(pair<T, S>), memcpyAccToHost);
27
28     /* Unmarshall: Re-build the linked list from the output buffer */
29     S *ptr = buffer[0].ptr;
30     for(int i = 1; i < size; i++) {
31         ptr->next = buffer[i].ptr;
32         ptr = buffer[i].ptr;
33     }
34     ptr->next = NULL;
35
36     /* Release memory */
37     free(buffer);
38     accFree(device);
39
40     return ptr;
41 }
```

Listing 4.2: Linked-list sorting in the per-call data transfer model

CPU waits for the accelerator to finish executing (line 22). Once the accelerator is done, the CPU code transfers the ordered output vector from accelerator to system memory (lines 25 – 26) and uses this data to rebuild (lines 29 – 34) the ordered linked-list (unmarshalling). This latter process also presents a data dependency between the access to two consecutive elements in the list, so it is executed at the CPU. Finally, the vector of key-value pairs is released (lines 37 – 38).

Figure 4.1(a) shows two I/O operations: a data transfer from system to accelerator memory before executing the kernel (lines 18 – 19), and a data transfer from accelerator to system memory after the kernel execution (lines 25 – 26). These two data transfers, which are not present in the CPU-only code, force accelerator kernel calls to follow a by-value calling convention. Hence, changes to the parameters by the kernel code have no visibility in the caller's scope.



(a) Flowchart for the sorting function in the linked-list sorting example when the per-call data transfer model is used

(b) Flowchart for the sorting function in the linked-list sorting example when the double-buffered data transfer model is used

Figure 4.1: Flowchart for sorting function in the per-call and double-buffered data transfer models

By-value calling convention might reduce the benefit of using accelerators. For instance, in the linked-list sorting example, building the output ordered linked-list from the output ordered vector of key/value pairs is a highly parallel task. However, by-value parameter passing would require transferring the whole input list from system to accelerator memory before the accelerator call, and transferring the whole output list from accelerator to system memory on the kernel return. The cost of these two data transfers voids the benefit of using the accelerator and thus, the output linked-list is built by code executed in the CPU. Another effect of by-value parameter passing is that the vector of key-value vector is allocated (lines 5 – 8) and released (lines 37 – 38) twice, once in system memory and once in accelerator memory. This is in contrast to the CPU-only code, where only one allocation and release call is necessary. This double allocation is discussed later on in this chapter.

Data transfers in the per-call model happen just after the CPU has finished the marshalling process and just before the accelerator executes the kernel. During the data transfer, both the CPU and the accelerator, are sitting idle, waiting for the data to be transferred. This data transfer time might represent a significant amount of the total execution time. For instance, in the linked-list sorting example, the data transfer times from and to system and accelerator memory is 4.11% of the benchmark execution time.

4.2.3 Double-Buffered Data Transfer Model

The double-buffered model aims to overlap data transfer and computation in the CPU and/or the accelerator to mitigate the performance overheads of by-value parameter passing. In the linked-list sorting benchmark, the double-buffered model has the same accelerator memory capacity requirements: the input/output vector of key/value pairs. In this example, a single call to the accelerator is done and, hence, only CPU computation and data transfers are overlapped. However, to overlap data transfers and accelerator computations the double-buffer model requires twice the accelerator memory capacity as the per-call model.

Figure 4.1(b) shows the flowchart for the sorting function in the linked-list example when the double-buffered model is used, as in Listing 4.3. In this Figure, the linked-list is tailed (`bucket` elements per tile in Listing 4.3) and the marshalling (lines 14 – 28) and unmarshalling (lines 48 – 51) are implemented for each tile. Hence, an input vector tile is transferred (lines 21 – 22) while the CPU is producing the following vector tile.

The overlapping of data transfers and computation improves application performance at the cost of harming programmability. For instance, for a 16 million item linked-list, the double-buffered liked-list sorting benchmark is 1.14X faster than the per-call version of the same code. However, the code using a double-buffered model in Listing 4.3 is far more complex than the code using the per-call model in Listing 4.2. Moreover, the source code using a double-buffered model hardly resembles to a code not using accelerators. The main difference between the double-buffer and the CPU-only implementations are:

- The vector of key/value pairs is allocated (lines 5 – 8) and released (lines 60 – 61) twice in double-buffered instead of once, as occurs in the per-call model.
- The marshalling (lines 14 – 18) and unmarshalling (lines 48 – 51) loops are part of two different outer loop, where the process is done in tiles.
- Asynchronous data transfer functions (lines 21 – 22 and 42 – 43) are needed to transfer memory between system and accelerator memories.
- Synchronization calls (lines 26 and 54) are needed to ensure that asynchronous data transfers have finished.

4.2.4 Accelerator-Hosted Data Transfer Model

The accelerator-hosted model is the model adopted in this dissertation. In this model, all data required by any accelerator call is hosted in the accelerator memory. In the linked-list sorting benchmark, the accelerator-hosted model requires both, the input linked-list and the input vector of value/key values to be hosted by the accelerator memory, which means larger accelerator memory

```
template<typename S>
S *sort(S *ptr, size_t size)
{
    /* Allocate system and accelerator memory */
    pair<float, S> *buffer =
        (pair<float, S> *)malloc(size * sizeof(pair<float, S>));
    pair<float, S> *device = NULL;
    accMalloc((void **)&device, size * sizeof(pair<float, S>));

    /* Marshall and transfer data in tiles of bucket elements */
    const size_t bucket = 256 * 1024;
    for(size_t offset = 0; offset < size; offset += bucket) {
        /* Marshall: Build the input vector from the linked list */
        for(int i = 0; i < backet; i++) {
            buffer[i + offset].value =ptr->weight;
            buffer[i + offset].ptr = ptr;
            ptr = ptr->next;
        }

        /* Copy the input buffer to the accelertor memory */
        accMemcpyAsync(device + offset, buffer + offset,
            bucket * sizeof(pair<float, S>), memcpyHostToAcc);
    }

    /* Wait for the last data transfer */
    accSynchronize();

    /* Call sorting function in the accelerator */
    accOddEvenMergeSort<float, S, pair<float, S> >(device, size);

    /* Unmarshall and transfer data in tiles of bucket elements */
    S *ptr = buffer[0].ptr;

    /* Transfer the first tile to system memory */
    accMemcpyAsync(buffer, device,
        bucket * sizeof(pair<T, S>), memcpyAccToHost);

    for(size_t offset = 0; offset < size; offset += bucket) {
        /* Transfer the next tile to system memory */
        size_t next = offset + bucket;
        if(size < next) {
            accMemcpyAsync(buffer + next, device + next,
                bucket * sizeof(pair<T, S>), memcpyAccToHost);
        }

        /* Unmarshall: Re-build the linked list from the
            output buffer */
        for(int i = 1; i < bucket; i++) {
            ptr->next = buffer[i + offset].ptr;
            ptr = buffer[i + offset].ptr;
        }

        /* Wait for the next buffer tile */
        accSynchronize();
    }

    ptr->next = NULL;

    /* Release memory */
    free(buffer);
    accFree(device);

    return ptr;
}
```

Listing 4.3: Linked-list sorting in the double-buffer data transfer model

```
1  template<typename S>
2  S *sort(S *ptr, size_t size)
3  {
4      /* Allocate system and accelerator memory */
5      pair<float, S> *buffer = NULL;
6      accMalloc((void **)&buffer, size * sizeof(pair<float, S>));
7
8      /* Marshall: Build the input vector from the linked list */
9      for(int i = 0; i < size; i++) {
10         buffer[i].value =ptr->weight;
11         buffer[i].ptr = ptr;
12         ptr = ptr->next;
13     }
14
15     /* Call sorting function in the accelerator */
16     accOddEvenMergeSort<float, S, pair<float, S> >(accPtr(buffer), size);
17
18     /* Unmarshall: Re-build the linked list from the output buffer */
19     S *ptr = buffer[0].ptr;
20     for(int i = 1; i < size; i++) {
21         ptr->next = buffer[i].ptr;
22         ptr = buffer[i].ptr;
23     }
24     ptr->next = NULL;
25
26     /* Release memory */
27     accFree(buffer);
28
29     return ptr;
30 }
```

Listing 4.4: Linked-list sorting in the accelerator-hosted data transfer model

capacity requirements than in the per-call and double-buffered models. However, as discussed in Chapter 3, accelerator memory capacity is increasing, so the extra memory requirements of the accelerator-hosted model are easily met by current accelerators.

Figure 4.2 shows the flowchart for the linked-list sorting example when the accelerator-hosted model is used, as in Listing 4.4. This code resembles to the CPU-only version of the code in Listing 4.1. The accelerator-hosted model only differs from the CPU-only version on the memory allocation (line 6) and release (line 27) calls, and the sorting function call (line 16). Memory allocation and release calls in the accelerator-hosted model use special forms to indicate to the run-time system that the memory must be allocated/released in the accelerator memory. The accelerator-hosted model allows by-reference parameter passing to accelerators, but virtual addresses used by the accelerator call might differ from the virtual addresses used by the CPU. Hence, the call to the sorting function (line 16) requires translating from system to accelerator addresses (`accPtr()` when passing parameters by-reference. This constrain is analyzed in the next section.

By-reference parameter passing might also improve application performance. For instance, in the linked-list sorting benchmark in Listing 4.4, if the linked-list is allocated in accelerator memory, by-reference parameter passing enables

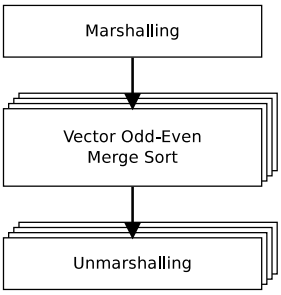


Figure 4.2: Flowchart for the sorting function in the linked-list sorting example when the accelerator-hosted data transfer model is used

implementing an accelerator function to build the output sorted linked list. Such an implementation process speeds up the unmarshalling process by 215X.

The accelerator-hosted model might greatly penalize the execution in the CPU because accesses to accelerator-hosted data have to reach the accelerator memory. This performance penalty is addressed in this dissertation using hardware (Chapter 5) and software (Chapter 6) mechanisms that allow caching accelerator-hosted data in system memory.

4.3 A Unified Shared Address Space

Chapter 3 argued that separate system and accelerator memories are key for CPU – accelerator architectures to deliver high performance. Physically separated memories typically are implemented as separate system and accelerator physical address spaces. These two separate physical address spaces are presented by most commercial programming models (see Chapter 2) as two separate virtual address spaces to programmers (e.g., host and device memories in CUDA [NVI09]). Exposing system and accelerator virtual address spaces to application programmers harms the programmability of CPU – accelerator systems because duplicated memory references (one per virtual address space) are needed. This programmability problem can be easily overcome by presenting application programmers with a unified virtual address space that includes both accelerator and system memories. This section discusses the programmability harms due to separate accelerator and system address spaces and benefits of a unified virtual address space.

4.3.1 The Double-Pointer Problem

DMA-based programming models for heterogeneous parallel systems present programmers separate system and accelerator address spaces. This situation is illustrated in Figure 4.3, where disjoint system and accelerator virtual address spaces require programmers to use different virtual memory addresses to refer to the same data objects: a system virtual memory address in the CPU code

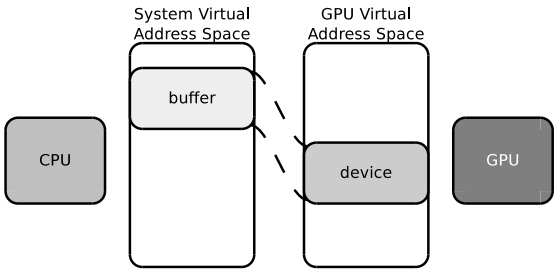


Figure 4.3: Example of disjoint processor – accelerator memory address spaces. A data object is stored in both address spaces at different virtual memory addresses.

```
4       /* Allocate system and accelerator memory */
5       pair<float, S> *buffer =
6           (pair<float, S> *)malloc(size * sizeof(pair<float, S>));
7       pair<float, S> *device = NULL;
8       accMalloc((void **)&device, size * sizeof(pair<float, S>));
```

Listing 4.5: Double-pointer requirement of disjoint address spaces

and an accelerator virtual memory address in the accelerator code. These two virtual system and accelerator memory addresses appear in the source code as two different variables (pointers in C/C++ code). The need for two pointers to reference the same data object is referred as *the double-pointer problem* in this dissertation. A similar problem also occurs, for instance, on UNIX systems when two user processes shared memory; the shared memory region is referenced by different virtual memory addresses by different processes.

Using duplicated-pointers to reference data structures is a quite straightforward approach when the per-call and the double-buffered data transfer models are used. These two models require data structures to be copied from system to accelerator memory and vice versa. Hence, using one pointer to reference the data object copy in system memory and a different pointer to access the data object copy in accelerator memory is a consistent approach. Listing 4.5 shows an extract from the sorting function in the linked-list example when the per-call data transfer model is used[1]. This code snippet illustrates the usage of double-pointer when separate system and accelerator address spaces are exported to programmers. First, the code in Listing 4.5 declares a system memory pointer (`buffer`) and allocates a chunk of system memory (line 4 – 5). Then, an accelerator memory pointer is declared (line 6) and accelerator memory is allocated (line 7). However, both pointers, `buffer` and `device`, reference the same data object: the input vector of key/value pairs.

The double-pointer problem of disjoint address spaces is mitigated by the accelerator-hosted data transfer model. When this model is used, the CPU code only requires a single pointer, which contains a system virtual memory

[1]The code snippet for the double-buffered model does not significantly differ from the code in Listing 4.5

```
16      /* Call sorting function in the accelerator */
17      accOddEvenMergeSort<float, S, pair<float, S> >(
18          accPtr(buffer), size);
```

Listing 4.6: Accelerator-mapped accelerator call example using system to accelerator memory translation

```
1  template<typename T, typename S>
2  __global__ void accUnmarshall(pair<T, S> *buffer, size_t size)
3  {
4     unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
5     if(idx < size - 1) {
6        buffer[idx].device->device_next = buffer[idx + 1].device;
7        buffer[idx].device->system_next = buffer[idx + 1].system;
8     }
9  }
```

Listing 4.7: Linked-list sorting in the per-call data transfer model

address. However, code executed in the accelerator accesses data objects using accelerator virtual memory addresses. Hence, when parameters are passed by-reference on accelerator calls, system to accelerator virtual memory address translation has to be done. This memory translation ensures that the accelerator code takes a valid accelerator virtual memory address. Listing 4.6 shows the relevant sorting function code to illustrate how system memory references are translated prior being passed by reference to accelerator calls. The call to `accOddEvenMergeSort()` in Listing 4.6 takes two parameters: `vector`, an accelerator virtual memory address where the vector of key/value pairs is located in the accelerator memory, and `size`, the number of pairs in `vector`. The `accPtr()` is used to translate from the system virtual memory address (`vector`) to the accelerator virtual memory address where the vector is hosted.

Complex data objects with embedded pointers add a new dimension to the double-pointer problem. For instance, in the linked-list sorting example, each list element includes a pointer to the next element in the list. This pointer might be used in both CPU and accelerator code and, therefore, each item actually requires two pointers: one pointer containing the system virtual memory address for the next element in the list, and one pointer with the accelerator virtual memory address to that very same element. Duplicate pointers embedded into data structures pose an additional problem: pointer consistency. Listing 4.7 illustrates the pointer consistency problem using the sorted linked-list reconstruction code executed by the accelerator. The code in Listing 4.7 uses accelerator virtual addresses to access each linked-list element (`buffer[idx]->device`), and modifies both the system (`system_next`) and accelerator (`device_next`) virtual memory addresses for the next element in the list (lines 6 and 7). Due to the complexity of dealing with separate pointers embedded into data structures, some programming models, such as OpenCL, explicitly forbid data structures to include pointers. However, such a constrain forces programmers to use relative

addressing, which adds more complexity to the accelerator code.

Besides increasing the code complexity, keeping system and accelerator pointers consistent between them might impose an important penalty on application performance too. For instance, in the CUDA code in Listing 4.7 setting the values for the system and accelerator pointers in each item requires two accesses to global memory, which are extremely costly. If the code in Listing 4.7 would require only accessing a single pointer for the next element, the execution time for the linked-list reconstruction would be reduced by a half.

4.3.2 Single Pointer Solution

In the present dissertation, the double-pointer problem is solved by removing the need for separate system and accelerator virtual memory addresses. Figure 4.4 illustrates the solution adopted in this dissertation. In Figure 4.4, system and accelerator memories are combined to form a unified virtual address space, where each data object is definitely identified by a single virtual memory address. A unified virtual address space that includes both system and accelerator memories, is the natural abstraction for the accelerator-hosted data transfer model. Unlike the per-call and double-buffered transfer models, the accelerator-hosted model assumes the existence of a single copy of data structures used by accelerators that are hosted in accelerators memory. Hence, programmers would expect using a single virtual memory address to reference data objects.

Listing 4.8 shows the linked-list sorting example function when both the accelerator-hosted and the single virtual address space is used. The main benefits provided by a single virtual address space are:

- Parameters passed by-reference to accelerator calls do not require any special handling (line 8).
- Data structures with embedded pointers get simplified because a single reference is needed (line 20).
- The accelerator code becomes simpler (lines 15 – 22) than when duplicated pointers are required.
- The accelerator code performance might be improved due to the lack of explicit code to keep pointer consistency. For instance, the number of global memory accesses in the code in Listing 4.8 is half the number required in the analogous code when duplicated pointers are required (Listing 4.7).

This dissertation presents two different approaches to build a unified virtual address space that includes both system and accelerator memories. The first approach, used in Chapter 5, maps accelerator physical addresses to virtual memory addresses using well-known virtual memory translation mechanisms, such as pagination and segmentation. When this approach is used, accelerator

```
1  template<typename S>
2  S *sort(S *ptr, size_t size)
3  {
4      /* Allocate system and accelerator memory */
5      pair<float, S> *buffer = NULL;
6      accMalloc((void **)&buffer, size * sizeof(pair<float, S>));
7
8      /* Marshall: Build the input vector from the linked list */
9      for(int i = 0; i < size; i++) {
10         buffer[i].value =ptr->weight;
11         buffer[i].ptr = ptr;
12         ptr = ptr->next;
13     }
14
15     /* Call sorting function in the accelerator */
16     accOddEvenMergeSort<float, S, pair<float, S> >(accPtr(buffer), size);
17
18     /* Unmarshall: Re-build the linked list from the output buffer */
19     S *ptr = buffer[0].ptr;
20     accUnmarshall<float, S>(buffer, size);
21     buffer[size - 1].device->next = NULL;
22
23     /* Release memory */
24     accFree(buffer);
25
26     return ptr;
27 }
28
29 template<typename T, typename S>
30 __global__ void accUnmarshall(pair<T, S> *buffer, size_t size)
31 {
32    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
33    if(idx < size - 1) {
34       buffer[idx].device->next = buffer[idx + 1].ptr;
35    }
36 }
```

Listing 4.8: Code snippet from linked-list sorting using a unified virtual address space

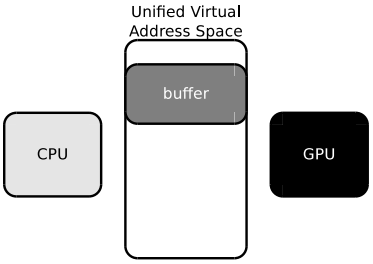


Figure 4.4: Example of disjoint processor – accelerator memory address spaces. A data object is stored in both address spaces at different virtual memory addresses.

memory allocation calls (i.e., `accMalloc()`) set up the necessary virtual memory translation data structures (e.g., page table entries) in the CPU and the accelerator to map a range of accelerator physical memory addresses into the same range of virtual memory addresses in system and accelerator memories.

If virtual memory is not supported by the accelerator or is not available to the system software (i.e., operating system), *memory space mirroring* might be used to build a unified virtual address space. Memory space mirroring ensures that accelerator-hosted data structures are referenced by the same memory address in different address spaces. In its simplest form, a range of accelerator physical memory addresses are mapped to the same address range in the system virtual memory space. Chapter 6 presents a detailed discussion about implementation alternatives for memory space mirroring. A different approach to build a unified virtual address space might be found in [SZC+09].

4.4 Summary

The present chapter has presented three different data transfer models to be used in processor – accelerator architectures:

- *Per-Call*: input data used by accelerator calls is copied to accelerators memory just before accelerator calls and output data produced by accelerators is copied to system memory just after accelerator calls.
- *Double-Buffered*: accelerator call input and output data is copied in a block-by-block basis to overlap data transfers with computations on the processor and on accelerators.
- *Accelerator-Hosted*: accelerator input and output data is hosted by accelerator memories, so no data copies are needed. Whenever the CPU requires accessing accelerator input and output data, it effectively accesses the accelerator memory.

This dissertation advocates for the accelerator-hosted model because it improves programmability. The programmability benefits offered by this model are:

- By-reference parameter passing to accelerator calls.
- Lack of explicit memory consistency and coherence code in user applications.

Per-call and double-buffered data transfer models are supported by most existent commercial programming models for heterogeneous systems. These two transfer models require programmers to explicitly copy data between system and accelerator memories and, therefore, export separate virtual address spaces for system and accelerator memories. However, the existence of separate virtual memory address spaces introduces the *double-pointer* problem: applications must keep two different references (pointers) to the same data object; one reference is used to access the data object by the code executed in the processor, while the other reference is used to access the same data object by the code running on the accelerator. Double-pointers have to be kept consistent by the application code, which might represent an important performance penalty.

The accelerator-hosted data transfer model adopted in this dissertation is better exploited when a unified virtual address space that includes system and accelerator memories is built. The programmability and performance drawbacks introduced by the double-pointer problem are cleanly solved by such a unified virtual address space. Different approaches to efficiently build a unified virtual address space are explored in the following chapters of this dissertation.

4.5 Significance

The accelerator-hosted data transfer model described in this chapter eases the task of programming heterogeneous parallel systems. This model removes the two major drawbacks present in current programming models: the lack of support for by-reference parameter passing and the explicit memory coherence and consistency management. The accelerator-hosted data is well suited for modern heterogeneous parallel systems, where accelerator memories tend to have a larger capacity than system memories.

The accelerator-hosted model benefits from a unified virtual address space that includes system and accelerator memories. This unified virtual address space solves the double-pointer problem, which has been described and analyzed in this dissertation for the first time in the literature. Solving the double-pointer problem not only eases the programming of CPU and accelerator code for heterogeneous parallel systems, but has been shown to improve the performance of some accelerator codes by reducing the total number of memory accesses.

The accelerator-hosted data transfer model and the unified virtual address space presented in this chapter are the basis for the further developments achieved during the elaboration of this thesis.

Chapter 5

Non-Uniform Accelerator Memory Access

5.1 Introduction

The hardware modifications discussed in this chapter conform a Non-Uniform Accelerator Memory Access (NUAMA) architecture model. In this model CPUs can access accelerator memories using regular load/store instructions, but accelerators are constrained to only access their own physical memory. Providing CPUs access to the accelerator memory, applications can be coded using the accelerator-hosted data transfer model presented in Chapter 4 which has been shown to improve the programmability of heterogeneous parallel systems. In NUAMA, accelerator-hosted data is cached by the CPUs so repeated access to the same data, or accesses to contiguous memory locations are not penalized. Moreover, the NUAMA architecture incorporates hardware mechanisms to buffer and coalesce data request for accelerator memory to efficiently use the available interconnection link (e.g., PCIe) bandwidth.

NUAMA allows the Operating System (OS) to build a unified virtual address space that includes system and accelerator memories. The operating system implements the address space mirroring technique discussed in Chapter 4. On an accelerator memory allocation request, the OS sets up the virtual memory translation data structures (e.g., page table entries) so the accelerator physical address and the system virtual address matches.

A subset of the Parboil benchmarks, described in Chapter 3, has been ported to use the accelerator-hosted data transfer model. This porting process highlights the programmability improvements of the accelerator-hosted data transfer model. Simulation results of the NUAMA architecture show no execution time slow-down across these benchmarks with respect to previous DMA-based CPU – accelerator architectures, which have already shown speedups from 10X to 240X with respect to a baseline processor-only configuration [RRB+08]. These simulation results show the viability of an efficient implementation of the accelerator-hosted data transfer model with small hardware additions to existent systems.

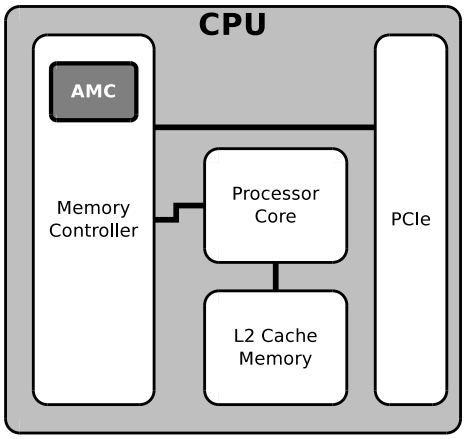


Figure 5.1: CPU architecture in NUAMA

5.2 Non-Uniform Accelerator Memory Access Architecture

The NUAMA architecture hides the overheads due to data movement between system and accelerator memory by dynamically collecting the accelerator input data as it is being produced by the CPU. An *Accelerator Memory Collector* (AMC) is integrated with the system memory controller, as illustrated in Figure 5.1. The AMC inspects every memory request going into and out of the controller to identify writes to the data that are being hosted by the accelerator memory. The AMC determines if an access is for the accelerator memory based on mappings designated by the application. Whenever a memory write involves the accelerator memory, the AMC forces a write-through action from the system memory controller to the accelerator memory. With additional buffering, the ACM coalesces writes to sequential locations for better transfer efficiency. The AMC and the write-through policy for the accelerator-hosted data structure when they reside in the L1 and L2 caches maximize the probability that, when the accelerator starts computing, all of its input data will be already present in its memory. This constraint of using a write-through L2 cache is relaxed later in this chapter.

Consider the simplified particle dynamics simulation application in Listing 5.1, which uses the sorting function discussed in Chapter 4. In this example, the marshalling and unmarshalling loops have been encapsulated as functions to simplify the application code. The application first allocates accelerator memory (lines 4 and 6) to store a linked-list of the particles (`list`) forming the system to be simulated and the vector of key/value pairs required by the sorting function. The memory accelerator allocation call (`nuamaAlloc()`) requests the OS to set up the necessary mappings. The OS code uses the address space mirroring technique described in Chapter 4 to map the allocated memory at the same virtual memory address in both the CPU and the accelerator. Then, the

```
1  int main(int argc, char *argv[]) {
2      float dt = 0;
3      /* Allocate data structures */
4      particle_t *list = nuamaAlloc(N * sizeof(item_t));
5      pair<particle_t *, float> vector =
6          nuamaAlloc(N * sizeof(pair<particle_t *, float>));
7
8      /* Init data structures */
9      initData(list, N);
10
11     /* Marshalling */
12     marshall(vector, list, N);
13
14     for(dt = 0; dt < T; dt += DELTA) {
15         /* Sort vector */
16         nuamaOddEvenSort(vector, N);
17         /* Unmarshall */
18         unmarshall(list, vector, N);
19         /* Compute trajectory */
20         computeForces(list, N);
21         /* Set new particle positions */
22         updatePositions(list, N);
23     }
24
25     printData(list, N);
26     nuamaFree(list);
27     nuamaFree(vector);
28 }
```

Listing 5.1: Main loop of sample particle dynamics simulation application

application executes a loop where the forces between particles are calculated, and the position of the particles are updated. Finally, the application prints the final position of the particles in the system.

While the application code reads the input from list of particles from the disk (line 9), the AMC sees a series of memory writes to locations mapped to the accelerator memory. The AMC buffers these writes, coalesces them into larger transfer units, and sends them to the accelerator. Analogously, the marshaling process (line 12) to build the initial vector of key/value pairs produces a series of accelerator memory reads and writes, which are also handled by the AMC. When the application calls the `nuamaOddEvenSort()` function (line 16), it first writes the input parameters (`*vector`, and `N`) into the accelerator registers. Notice that the vector of key/value pairs is passed by-reference using the `vector` pointer. When the accelerator finishes computing, the CPU gets the output value from one of the accelerator registers. A similar procedure is followed for the `unmarshall()` (line 18). A first version of the application in Listing 5.1, might implement the `computeForces()` (line 20) and `computePositions()` (line 22) functions in the CPU. In this case, memory accesses to the linked-list in these functions are seen by the AMC, which brings the data from the accelerator memory to the CPU caches.

A more elaborated version of the application in Listing 5.1 would implement both functions (`computeForces()` and `computePositions()`) in the accelerator. In this improved version, the code in Listing 5.1 remains unchanged. This

illustrates how NUAMA provides a progressive porting path for applications to use accelerators. In a first stage, computation intensive functions amenable for accelerator execution are moved to accelerators, while the rest of the application remains begin executed in the CPU. Later on, other functions might be also ported to be executed in accelerators, and so on. Once the data structures used by accelerators have been allocated in the accelerator memory (`nuamaAlloc()`), most of the application code remains unchanged, and only the functions moved to the accelerator needs to be recoded.

5.3 Accelerator Memory Collector

Accelerator data hosting is implemented using a TLB assisted mechanism: the page table entry is extended with a one-bit A field, which is set for those pages containing data hosted by the accelerator. The L1 and L2 caches also store the A field for each cache line.

The A field is used to implement a hybrid write-back/write-through L2 cache. The L2 cache controller follows a write-through/write-no-allocate policy only for those cache lines whose A field is set. A hybrid policy provides three main benefits:

1. The data will be present in the accelerator memory before launching the computation.

2. Accelerator-hosted data is still cached so repeated CPU accesses or accesses to contiguous accelerator data are not penalized.

3. Write accesses to other data structures do not incur write-though actions by the L2 cache.

The L2 cache controller sends the A field to the main memory controller for every memory write operation. Write requests whose A bit is set are handled by the AMC. The AMC implements a write buffer to store pending write operations to the accelerator memory controller. Pending requests in the write buffer are coalesced and sent to the accelerator memory through the PCIe link. A single core system requires a write buffer with as many entries as the maximum number of outstanding memory requests the L2 can support. If the ISA does not allow write reordering, coalescing write operations might lead to race conditions. For instance, a write to a lock variable might be done before actually writing the data that is locked. The accelerator memory is expected to only host "pure data" structures while synchronization variables are stored in main memory.

The AMC presents the following interface to the CPU: applications use the `sacc` (*Start Accelerator*) instruction to signal the AMC to complete the data movement necessary and begin execution of the accelerator. Figure 5.2 shows the actions performed by the AMC when it gets a `sacc` request. First, it waits for all pending write-through activities to the accelerator to finish. Then, the

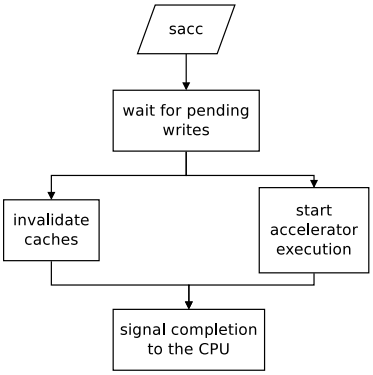
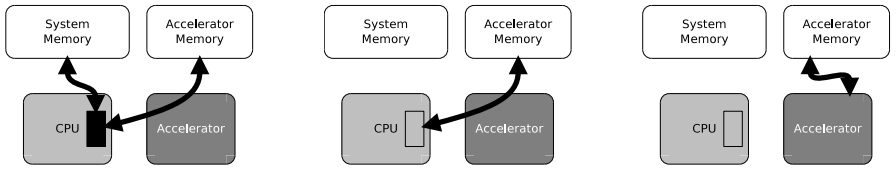


Figure 5.2: Actions performed in the AMC when a `sacc` instruction commits



(a) The application accesses data in system memory and accelerator memory through the cache

(b) Before starting the computation the AMC invalidates those cache lines containing accelerator data

(c) When the accelerator starts computing all the data is only in its own memory

Figure 5.3: Data movement in a NUAMA architecture

AMC invalidates those cache lines in the L1 and L2 caches whose A field is set and sends the `sacc` request to the accelerator. Note that these two actions can be performed in parallel if the CPU does not request any data during this process. To enforce this constraint, when the `sacc` instruction enters into the processor issue queue, the processor stops fetching new instructions. When the `sacc` instruction commits, it is sent to the AMC. At this point there are no other instructions in flight since the issue queue is empty. The AMC signals completion (all write-through activities and invalidations are done) to the CPU by asserting a line that enables fetching and issuing new instructions.

The write-through cache policy for the accelerator data allows overlapping the execution in the CPU with the data movement, as illustrated in Figure 5.3(a). When the application calls the accelerator, the AMC ensures that the single valid copy for the accelerator data is in the accelerator memory (Figure 5.3(b)). The application implements a polling loop or uses interrupts to wait for the accelerator to finish. Hence, only the accelerator can access those data (Figure 5.3(c)) while it is computing. Once the accelerator is done, any CPU access to the accelerator data will miss in the L1 and L2 caches since all the cache lines containing data hosted by the accelerator memory were invalidated. Both caches fill from the accelerator memory (Figure 5.3(a)).

5.4 Benefits and Limitations

NUAMA architectures have several implications for applications performance. The hybrid write-back/write-through L2 cache used in NUAMA is the key to overlap data transfers between the CPU and the accelerator. Furthermore, the hybrid policy increases the probability of having the data present in the accelerator memory when the CPU calls the accelerator. However, there is a potential side-effect that might reduce the benefit of the hybrid write policy of NUAMA. Every time the CPU modifies the data hosted by the accelerator memory, a write-through action is triggered at the L2 cache. Hence, repeated writes to the same memory location from the CPU before the accelerator is called lead to several write-through actions, increasing the amount of data transferred from the CPU to the accelerator memory. Thus, the average bandwidth of the L2 cache, the L2 bus, the memory controller and, the PCIe link required by NUAMA is higher than the bandwidth required by traditional DMA architectures. However, NUAMA redistributes write and reads operations to the accelerator memory across the total execution time of the application. Thus, in many cases the maximum instantaneous bandwidth required by NUAMA is much smaller than the instantaneous bandwidth required during a DMA transfer between the CPU main memory and the accelerator memory. A given hardware element only limits applications performance when it is not able to deliver the instantaneous bandwidth required by the application. Because NUAMA reduces the maximum value of the instantaneous bandwidth requirements, NUAMA is expected to perform better than current DMA-based architectures.

A benefit of NUAMA is its ability to perform DMA transfers from I/O devices to the accelerator local storage, since this is mapped in the system physical address space. In many applications, the data used by accelerators is directly read from the disk. In these applications NUAMA might decrease the amount of data transferred between the CPU main memory and the accelerator memory significantly.

5.5 Experimental Evaluation

The NUAMA architecture is evaluated using the simulation environment and benchmarks described in Chapter 3. The accelerator execution time, unless explicitly stated, is omitted from the reported results. The accelerator execution time represents most of the total benchmark execution time, and, if this time is included, the execution time differences between NUAMA and DMA-based architectures becomes inappreciable.

```
1  float results[4];
2  float *h_vars;
3  int *h_maxs;
4
5  int main(int argc, char** argv)
6  {
7      int N = atoi(argv[1]);
8      int s = atoi(argv[2]);
9      int t = atoi(argv[3]);
10     int N2 = N+N;
11     int NSQUARE2 = N * N2;
12
13     // Alloc system memory for the results
14     h_vars = (float *)malloc(t * sizeof(float));
15     h_maxs = (int *)malloc(t * sizeof(int));
16
17     // Compute the simulation on the accelerator
18     Petrinet();
19     // Compute statistics in the CPU
20     ComputeStatistics();
21
22     // Release memory
23     free(h_vars);
24     free(h_maxs);
25
26     // Print statistics
27     printf("petri␣N=%d␣s=%d␣t=%d\n", N, s, t);
28     printf("mean_vars:␣%f␣␣␣␣var_vars:␣%f\n", results[0], results[1]);
29     printf("mean_maxs:␣%f␣␣␣␣var_maxs:␣%f\n", results[2], results[3]);
30
31     return 0;
32 }
```

Listing 5.2: Main function in PNS for DMA configuration

5.5.1 Benchmark Porting

The parboil benchmark suite described in Chapter 3 is used to evaluate the NUAMA architecture. The SAD, MRI-Q and MRI-FHD are not included in the evaluation because the I/O operations they require are not supported by the simulation infrastructure.

The DMA configuration uses the NVIDIA CUDA version of each benchmark, which has already been optimized to reduce the overhead of data movement between the CPU and the accelerator. The CUDA API calls for allocating DMA memory buffers, performing data transfers and launching accelerator computation are substituted by the analogous ones provided by the simulation platform.

This porting process is illustrated in Listings 5.2, 5.3, and 5.4. Listing 5.2 shows the code of the main function in PNS for the DMA configuration, which is exactly equal in the CPU-only version of the benchmark. The main function in PNS calls two other functions: `Petrinet()` and `ComputeStatistics()`, shown in Listings 5.3 and 5.4 respectively. The `Petrinet()` function implements the per-call data transfer model discussed in Chapter 4. This function first allocates memory in the accelerator (lines 7 – 11), then the accelerator is called inside a loop for different blocks of the dataset (lines 16 – 34). Notice that after each accelerator invocation, the output data produced by the accelerator is copied

```
1  void Petrinet()
2  {
3      int unit_size = NSQUARE2 * (sizeof(int) + sizeof(char)) +
4          sizeof(float) + sizeof(int);
5      int block_num = MAX_DEVICE_MEM / unit_size;
6
7      // Allocate memory
8      int *g_places = (int *)accMalloc((unit_size - sizeof(float) -
9          sizeof(int)) * block_num);
10     float *g_vars = (float *)accMalloc(block_num * sizeof(float));
11     int *g_maxs = (int *)accMalloc(block_num * sizeof(int));
12
13     int *p_hmaxs = h_maxs;
14     int *p_hvars = h_vars;
15
16     // Launch the accelerator computation foreach block
17     for(int i = 0; i < t - block_num; i += block_num) {
18         accPetrinet(g_places, g_vars, g_maxs, N, s, 5489 * (i+1));
19
20         // Copy results to system memory
21         accMemcpy(p_hmaxs, g_maxs, block_num * sizeof(int),
22             MemcpyAccToHost);
23         accMemcpy(p_hvars, g_vars, block_num * sizeof(float),
24             MemcpyAccToHost);
25         // Move pointer to next block
26         p_hmaxs += block_num;
27         p_hvars += block_num;
28     }
29
30     // Launch the accelerator computation for the last
31     // block and copy back the results
32     accPetrinet(g_places, g_vars, g_maxs, N, s, time(NULL));
33     accMemcpy(p_hmaxs, g_maxs, (t-i) * sizeof(int), MemcpyAccToHost);
34     accMemcpy(p_hvars, g_vars, (t-i)*sizeof(float), MemcpyAccToHost);
35
36     // Free accelerator matrices
37     accFree(g_places);
38     accFree(g_vars);
39     accFree(g_maxs);
40 }
```

Listing 5.3: Function invoking the accelerator in PNS for DMA configuration

```
1  void computeStatistics()
2  {
3      float sum = 0;
4      float sum_vars = 0;
5      float sum_max = 0;
6      float sum_max_vars = 0;
7      for (int i=0; i < t; i++) {
8          sum += h_vars[i];
9          sum_vars += h_vars[i] * h_vars[i];
10         sum_max += h_maxs[i];
11         sum_max_vars += h_maxs[i] * h_maxs[i];
12     }
13     results[0] = sum/t;
14     results[1] = sum_vars/t - results[0] * results[0];
15     results[2] = sum_max/t;
16     results[3] = sum_max_vars/t - results[2] * results[2];
17 }
```

Listing 5.4: Statistics computation in PNS for DMA and NUAMA configurations

```
1  float results[4];
2  float *h_vars;
3  int *h_maxs;
4
5  int main(int argc, char** argv)
6  {
7      int N = atoi(argv[1]);
8      int s = atoi(argv[2]);
9      int t = atoi(argv[3]);
10     int N2 = N+N;
11     int NSQUARE2 = N * N2;
12
13     // Alloc system memory for the results
14     h_vars = (float *)accMalloc(t * sizeof(float));
15     h_maxs = (int *)accMalloc(t * sizeof(int));
16
17     // Compute the simulation on the accelerator
18     Petrinet();
19     // Compute statistics in the CPU
20     ComputeStatistics();
21
22     // Release memory
23     accFree(h_vars);
24     accFree(h_maxs);
25
26     // Print statistics
27     printf("petri␣N=%d␣s=%d␣t=%d\n", N, s, t);
28     printf("mean_vars:␣%f␣␣␣␣var_vars:␣%f\n", results[0], results[1]);
29     printf("mean_maxs:␣%f␣␣␣␣var_maxs:␣%f\n", results[2], results[3]);
30
31     return 0;
32 }
```

Listing 5.5: Main function in PNS for NUAMA configuration

from the accelerator memory to system memory (lines 20 – 24 and 33 – 34). Finally, once that all dataset tiles have been processed, the accelerator memory is released (lines 36 – 39). The `ComputeStatistics()` function scans through the output data produced in `Petrinet()` to compute different statistics from the Petri network simulation.

Parboil benchmarks are ported to NUAMA performing the following modifications:

- Calls to allocated data structures required by the accelerator are substituted by calls to the NUAMA API.
- Calls to accelerator memory allocation/release and data transfers between system and accelerator memories are removed.

The NUAMA porting process is illustrated in Listings 5.5, and 5.6. The NUAMA code for the main function in PNS does not significantly differ from the main function in DMA and CPU-only versions of the benchmark. The system memory allocation/release calls, `malloc()`/`free()`, are substituted by the analogous provided by the NUAMA run-time (lines 14 – 15, and 23 – 24), i.e., `accMalloc()`/`accFree()`. The `ComputeStatistics()` function in the NUAMA version is exactly the same than in the DMA and CPU-only versions of the

```
1  void Petrinet()
2  {
3      int unit_size = NSQUARE2 * (sizeof(int) + sizeof(char)) +
4          sizeof(float) + sizeof(int);
5      int block_num = MAX_DEVICE_MEM / unit_size;
6
7      // Allocate memory
8      int *g_places = (int *)accMalloc((unit_size - sizeof(float) -
9          sizeof(int)) * block_num);
10
11     int *p_hmaxs = h_maxs;
12     int *p_hvars = h_vars;
13
14     // Launch the accelerator computation foreach block
15     for(int i = 0; i < t - block_num; i += block_num) {
16         accPetrinet(g_places, p_hvars, p_hmaxs, N, s, 5489 * (i+1));
17         // Move pointer to next block
18         p_hmaxs += block_num;
19         p_hvars += block_num;
20     }
21
22     // Launch the accelerator computation for the last
23     // block and copy back the results
24     accPetrinet(g_places, p_hvars, p_hmaxs, N, s, time(NULL));
25
26     // Free accelerator matrices
27     accFree(g_places);
28 }
```

Listing 5.6: Function invoking the accelerator in PNS for NUAMA configuration

benchmark. However, the `Petrinet()` function, in Listing 5.6, differs significantly from the same function in the DMA version. A comparison of code in the `Petrinet()` function for DMA (Listing 5.3) and NUAMA (Listing 5.6) clearly shows the programmability improvements provided by NUAMA.

5.5.2 Hardware Requirements

Experimental results show that AMC sizes over eight entries do not significantly improve the performance of the NUAMA architecture in the simulated benchmarks. Such an AMC size is sufficient because benchmarks tend to write to contiguous accelerator memory locations, so several write requests are stored into a single AMC entry. For the simulated L2 cache with 128 byte lines, an AMC entry of 128 bytes is required. Such AMC entry size allows for 32 4-byte memory requests to be buffered into a single AMC entry. Hence, 8 entries provide space in the AMC buffer for up to 256 accelerator memory write requests, which is larger than the total number of outstanding memory requests supported by contemporary CPUs.

The extra hardware required by NUAMA is relatively small. The simulated 8-entry AMC requires a 1KB fully-associative non-blocking cache. Additionally, one extra bit is added to each TLB entry, and L2 cache-line. This requirement means 16 extra bytes in the simulated TLBs, and one extra kilobyte in the L2 cache. Notice that the extra storage in the L2 cache and the TLB required by

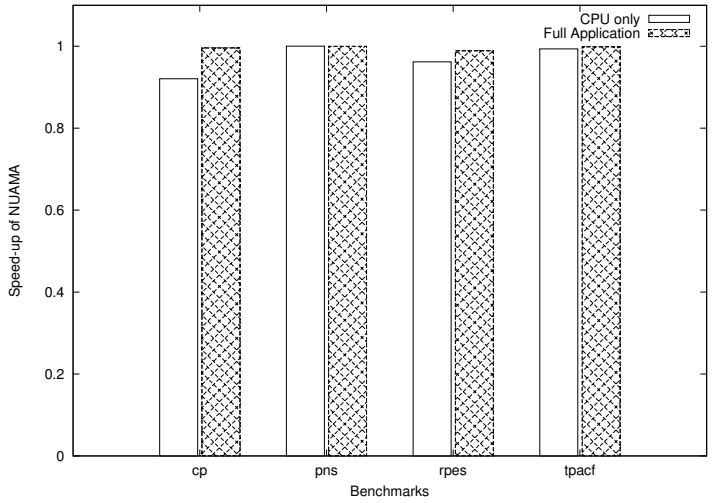


Figure 5.4: Speed-up of NUAMA with respect to DMA for simulated benchmarks

NUAMA are distributed among all entries. Hence, this modifications are not likely to increase the TLB and L2 cache access times.

5.5.3 NUAMA Performance

The performance of NUAMA with respect to DMA data transfers between CPU and accelerators is compared. Figure 5.4 shows the results for the different configurations and benchmarks. Results are shown as the speed-up in NUAMA with respect to DMA.

Execution times for NUAMA and DMA are almost the same for two of the four benchmarks we simulate, but NUAMA slightly slows-down the execution of the CPU code in CP and RPES. This small slow-down is due to the extra off-chip memory accesses performed by NUAMA, as shown in Figure 5.5. In these two benchmarks, accelerator-hosted data structures are initialized element by element. This initialization results in a series of write requests to the accelerator memory. In NUAMA, these requests are coalesced by the AMC, and sent in batches to the accelerator memory. In DMA, these write requests are cached in the L2, and accelerator data structures are initialized using one DMA transfer per data structure. The degree of coalescing is higher in DMA than in NUAMA due to the eager update of the accelerator memory done by the ACM. Figure 5.4 also shows that when the accelerator execution time is considered, both NUAMA and DMA perform equally well because the accelerator execution time is much larger than the CPU execution and data transfer times in all benchmarks.

Figure 5.5 shows that NUAMA actually produces a larger number of total off-chip memory access. The number of accesses to system memory is smaller in NUAMA than in DMA because NUAMA does not double allocate in system and accelerator memory those data structures used by the accelerator. These ex-

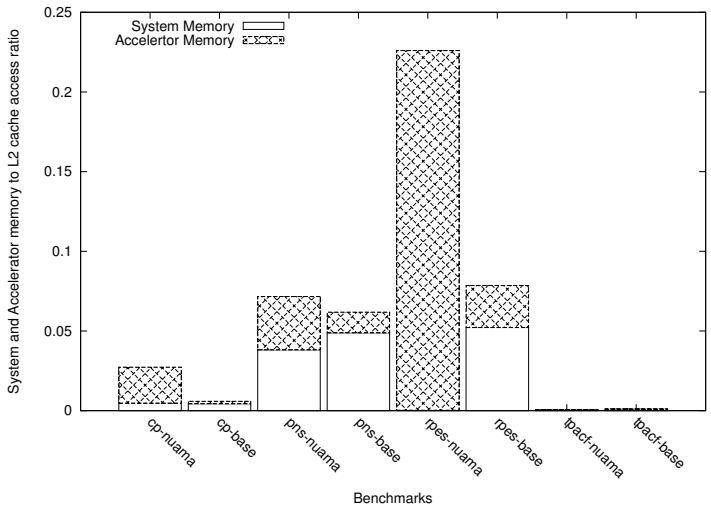


Figure 5.5: Number of accesses to the CPU main memory or the accelerator memory per access to the L2 cache. L2 write-backs and main memory reads are accesses to the CPU main memory. L2 write-through and local memory reads are accesses to the accelerator memory.

perimental results, jointly to the execution time results in Figure 5.4 illustrates that larger memory bandwidth requirements do not mean performance penalties because application performance is degraded whenever the instant memory bandwidth requirements can not be meet by the hardware.

NUAMA effects the CPU performance by increasing the number of TLB misses. The number of page table entries in NUAMA is larger because data structures hosted in accelerator memory require additional page table entries. Simulation results show that the number of TLB misses in CP, PNS and RPES is slightly larger in NUAMA than in DMA. However, NUAMA decreases the number of TLB misses by 1.1X in TPACF. In this benchmark, using separate page table entries for accelerator hosted data decreases the number of TLB conflicts due to the memory allocation algorithm. In DMA, the memory allocator maps accelerator data to virtual memory addresses that conflict with static allocated data structures. However, NUAMA maps accelerator hosted data structures into virtual addresses that do no conflict with any other allocated data structure. These experimental results show the importance of TLB-aware memory allocation to reduce the performance penalty due to TLB misses.

Figure 5.6 illustrates the effect of NUAMA over cache memories due to the write-through/write-non-allocate policy for accelerator-hosted data. The number of read and write L2 cache misses in NUAMA is higher than the DMA in CP, while it is similar in PNS, RPES and TPACF. The larger number of L2 cache misses of NUAMA with respect to DMA in CP is due to the memory access pattern in the CPU code. For instance, the CPU code in PNS, RPES and TPACF does not perform any write access to accelerator-hosted data, so the L2 cache write miss ratio in these two benchmarks is similar in both NUAMA

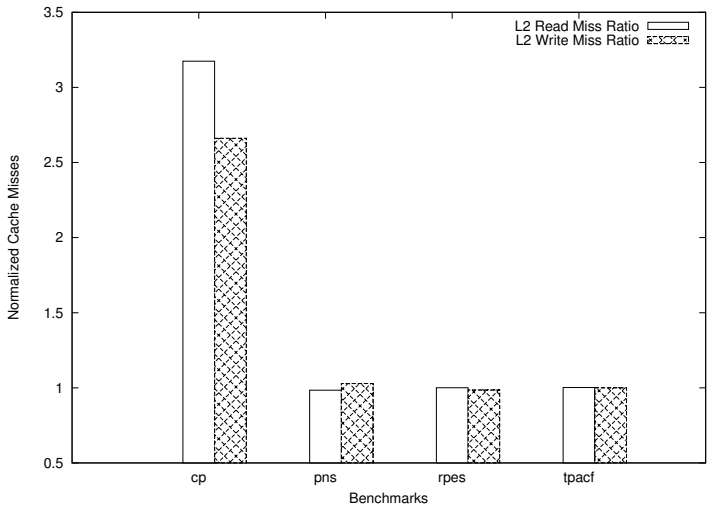


Figure 5.6: NUAMA L2 cache miss ratio normalized to DMA L2 cache miss ratio.

and DMA. The CPU code for CP performs a series of writes to initialize the accelerator input data. In this case, the write-through/write-non-allocate policy causes writes to consecutive memory positions to miss in the L2 cache. Despite the higher L2 cache miss ratio in NUAMA than in DMA for some benchmark, both architectures perform equally well. The reason for this similar performance is that the L2 cache miss latency is shorter in NUAMA than in DMA. L2 cache misses in NUAMA are distributed between system memory, for regular data, and the accelerator memory, for accelerator-hosted data. This distribution reduces the contention in the access to the system memory controller and to system memory and, which results in a shorter cache miss latency for NUAMA compared to DMA.

5.5.4 Memory Latency

Figure 5.7 shows the speed-up of NUAMA with respect to DMA for different system and accelerator memory latencies. Many existing accelerators, such as GPUs, include GDDR memories which deliver higher bandwidth compared to regular DDR memories. However, the same latency for both system and accelerator memory is assumed conservatively.

NUAMA slows-down the execution of the CPU code with respect to DMA as the memory lantency increases. This slow-down is more pronounced on CP and RPES, the benchmarks where NUAMA increases the number of off-chip memory accesses the most. In TPACF, where NUAMA and DMA require almost the same number of off-chip memory accesses, both configurations perform similarly well as the memory latency increases. NUAMA and DMA performs equally well for all memory latencies if the total application execution time is considered.

The speed-up of NUAMA with respect to DMA for PNS exhibits a quite

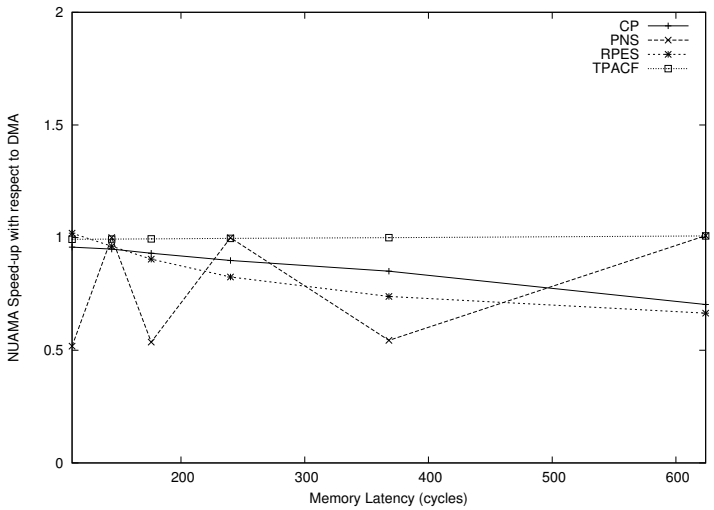


Figure 5.7: Speed-up of NUAMA with respect to DMA for different memory latencies

irregular pattern. In this benchmark, the NUAMA execution time is almost constant for all memory latencies, but the DMA execution time greatly varies for different memory latencies. This variability is due to the different simulated clock frequencies for the DMA controller and the CPU and the short execution time of PNS. DMA transfers require the application writing to the DMA status and control registers the source and destination addresses, and the transfer size. On some simulations this write happens near the end of the DMA clock cycle, and the DMA transfer starts immediately. However, in other simulations these writes happens at the start of the DMA clock cycle so the start of the DMA transfer is delayed until the next DMA clock cycle. The contribution of this delay in the start of DMA transfers is relatively large in PNS due to its short execution time and the high number of DMA transfers done by this benchmark.

5.5.5 Link Latency

Figure 5.8 shows the speed-up of NUAMA with respect to DMA for different PCIe link configurations. There is an important slow-down for CP and RPES when the bandwidth delivered by the PCIe bus is small due the high off-chip memory traffic produced by NUAMA in these benchmarks (see Figure 5.5). As the PCIe bandwidth increases, the slow-down in NUAMA for these benchmarks becomes smaller, being almost null for RPES when a 32X PCIe link is used. For PNS and TPACF, NUAMA and DMA perform similarly well for all considered PCIe configurations because both configurations produce barely the same number of accesses to the accelerator memory.

The inverse dependence between the PCIe bandwidth and the slow-down of NUAMA with respect to DMA shows that NUAMA is best suited for low-latency/high-bandwidth interconnections between CPUs and accelerators. The CPU

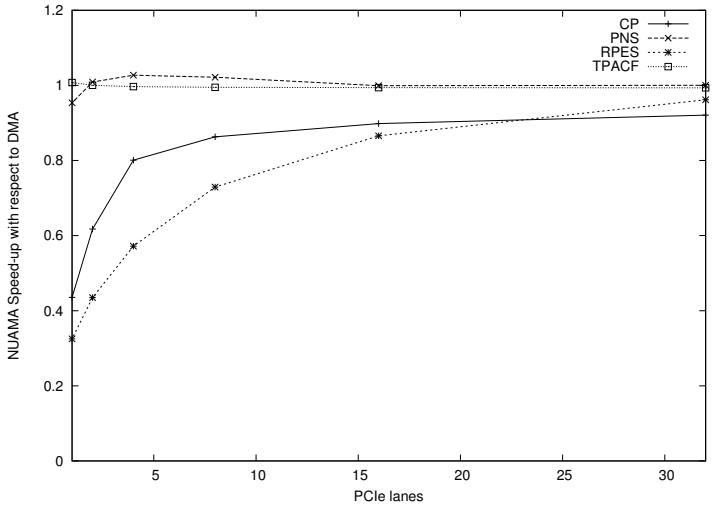


Figure 5.8: Speed-up of NUAMA with respect to DMA for different PCIe configurations

– accelerator interconnections bandwidth is growing (e.g., $250MB/s$ in PCIe 1.0 and $32GB/s$ in future PCIe 3.0). This shows that the NUAMA architecture is well suited for current and future systems.

5.6 Summary

This chapter has described NUAMA, a hardware implementation of the accelerator-hosted data model described in Chapter 4. NUAMA allows CPUs to efficiently access the accelerator memory using regular load/store instructions. The key architectural modifications required by NUAMA are:

- Page table entries are extended with an accelerator bit to identify those memory pages hosted by the accelerator memory.
- L2 cache memory implements a write-back/write-allocate for regular data, while a write-through/write-non-allocate policy is used for data hosted in the accelerator memory.
- An Accelerator Memory Collector is integrated in the memory controller to identify and coalesce memory request for the accelerator memory, and to ensure memory consistency on accelerator calls.

The write-through L2 cache policy provides an eager update of the accelerator memory contents. Due to this eager update, NUAMA overlaps computations in the CPU and data transfers to the accelerator memory. Hence, when the accelerator is called, most data is already in the accelerator memory and, therefore, the accelerator call latency is reduced. However, eager updates would under-utilize the interconnection between the CPU and the accelerator

(e.g., PCIe bus). Improving the efficiency of data transfers is the key role of the AMC, which coalesces memory request to contiguous accelerator memory locations to perform large transfer transactions.

Simulation results of the NUAMA architecture show that applications perform equally well than in traditional DMA based systems. NUAMA slightly slows-down the performance of the code executed by the CPU in certain benchmarks, but the overall effect of this slow-down over the complete application execution time is negligible. The benchmark porting experience shows that NUAMA provides a higher degree of programmability than traditional DMA-based CPU – accelerator architectures. This chapter has shown that NUAMA improves programmability without producing performance penalties with little hardware additions.

5.7 Significance

The NUAMA architecture has shown the viability of the accelerator-hosted model for CPU – accelerator systems. Benchmarks ported to NUAMA have required less number of lines of code than the original code targeted for a DMA-based data transfer model. Moreover, the source code for NUAMA simplifies the source code by removing the need of separate pointers for system and accelerator memory.

The NUAMA architecture has been only implemented within a simulator, which restricts its applicability. Moreover, the architectural modifications required by NUAMA are not likely to be adopted in the short-term in actual CPUs because the current market-share of CPU – accelerator systems is still too small. However, the design of the NUAMA architecture leads to two key observations:

- Eager update of the accelerator memory contents is essential to overlap data transfers and CPU computations, and, therefore, reduce the accelerator call latency.
- Buffering and coalescing of accelerator-hosted data is key to accomplish high data transfer bandwidths.

The evaluation of the NUAMA architecture has shown that its performance is comparable to traditional DMA-based architectures. Hence, the performance – programmability product, the metric being optimized in this dissertation, is higher in NUAMA than in existent CPU – accelerator architectures.

Chapter 6

Asymmetric Distributed Shared Memory

6.1 Introduction

Chapter 5 presented, NUAMA, a hardware implementation of the accelerator-hosted data transfer model and the unified virtual address space for CPU – accelerator systems discussed in Chapter 4. This chapter presents the Asymmetric Distributed Shared Memory (ADSM) model, the software counterpart of NUAMA, which presents several advantages. First, ADSM brings the programmability benefits of the accelerator-hosted data transfer model and the unified virtual address space to current CPU – accelerator systems. Second, a software implementation allows for different data transfer policies (e.g., eager update in NUAMA) to be available to applications. Third, a software run-time might extract high-level information from applications, which would allow for additional performance optimizations.

ADSM maintains an unified virtual memory address space for CPUs to access objects in the accelerator physical memory but not vice versa. This asymmetry allows all coherence and consistency actions to be executed on the CPU, allowing the use of simple accelerators. This chapter also presents GMAC, a user-level ADSM library, and discusses design and implementation details of such a system. Experimental results using GMAC show that an ADSM system makes heterogeneous systems easier to program with negligible performance overheads.

6.2 Asymmetric Distributed Shared Memory

Asymmetric Distributed Shared Memory (ADSM) maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. This section presents ADSM as a data-centric programming model and the benefit of an asymmetric shared address space.

6.2.1 ADSM Programming Model

In the ADSM programming model, programmers allocate or declare data objects that are processed by methods, and annotate performance critical methods (*kernels*) that are executed by accelerators. When such methods are assigned to an

```
1  __kernel__ void computeRhoPhi(int, float *, float *, float *,
2      float *, float *, float *);
3
4  __kernel__ void computeFH(int, int, float *, float *, float *,
5      kValues *, float *, float *);
6
7
8  int main (int argc, char *argv[])
9  {
10     /* Allocate data structures */
11     allocate(numX, &kx, &ky, &kz, &x, &y, &z, &phiR, &phiI, &dR, &dI);
12
13     /* Read in data */
14     readData(inFiles, numX, kx, ky, kz, x, y, z, phiR, phiI, dR, dI);
15
16     /* Create CPU data structures */
17     createDataStructs(numK, numX, realRhoPhi, imagRhoPhi, outR, outI);
18     kVals = (kValues*)calloc(numK, sizeof (kValues));
19
20     /* Pre-compute the values of rhoPhi on the GPU */
21     computeRhoPhi(numK, phiR, phiI, dR, dI, realRhoPhi, imagRhoPhi);
22     assert(gmacThreadSynchronize() == gmacSuccess);
23
24     /* Fill in kVals values */
25     fillKVals(numK, kVals, kx, ky, kz, realRhoPhi, imagRhoPhi);
26
27     /* Compute FH on the GPU (main computation) */
28     computeFH(numK, numX, x, y, z, kVals, outR, outI);
29     assert(gmacThreadSynchronize() == gmacSuccess);
30
31     writeData(params->outFile, outR, outI, numX);
32
33     /* Release Memory */
34     free(kVals);
35     release(kx, ky, kz, x, y, z, phiR, phiI, dR, dI,
36         realRhoPhi, imagRhoPhi, outR, outI);
37
38     return 0;
39 }
```

Listing 6.1: Main function in MRI-FHD in ADSM

accelerator, their corresponding data objects are migrated to on-board accelerator memory. This model is illustrated in Listing 6.1, which shows the main function of MRI-FHD from the Parboil benchmark. In this code, `computeRhoPhi()` (line 1) and `computeFH()` (line 3) are performance critical and are executed in accelerators.

ADSM removes the need to explicitly request memory on different memory spaces (line 11). Programmers assign data objects to methods that might or might not be executed by accelerators. Run-time systems can be easily built under this programming model to automatically assign methods and their data objects to accelerators, if they are present in the system. High performance systems are likely to continue having separate physical memories and access paths for CPUs and accelerators. However, CPUs and accelerators are likely to share the same physical memory and access path in low-cost systems. An application written following the ADSM programming model will target both kinds of systems efficiently. When the application is run on a high performance

system, accelerator memory is allocated and the run-time system transfers data between system memory and accelerator memory when necessary. In the low-cost case, system memory (shared by CPU and accelerator) is allocated and no transfer is done. Independence from the underlying hardware is the first benefit provided by ADSM.

Analogously to the NUAMA architecture discussed in Chapter 5, ADSM offers a convenient software migration path for existing applications. Performance critical libraries and application code are moved to accelerator engines, leaving less critical code porting for later stages [BDH+08]. For instance, in Listing 6.1, `computeFH()` (line 28) is first ported to be executed in an accelerator. However, `computeRhoPhi()` (line 21) remains being executed in the CPU in at first, and ported to the accelerator later on. Finally, `fillKVals()` (line 25) is currently executed in the CPU, but future versions of the code might also execute this function in an accelerator. CPUs and accelerators do not need to be ISA-compatible to interoperate, as long as data format and calling conventions are consistent. This programming model provides the run-time system with the information necessary to make data structures accessible to the code that remains for execution by the CPU. This is the second gain offered by the ADSM programming model.

Data objects used by kernels are often read from and written to I/O devices (e.g., a disk or network interface). ADSM enables data structures used by accelerators to be passed as parameters to the corresponding system calls that read or write data for I/O devices (e.g. `read()` or `write()`). In Listing 6.1, `readData()` (line 14) and `writeData()` (line 31) reads and writes, respectively, the data structures passed by-reference to disk. If supported by the hardware, the run-time system performs DMA transfers directly to and from accelerator memory (*peer DMA*), otherwise an intermediate buffer in system memory might be used. Applications benefit from *peer DMA* without any source code modifications, which is the third advantage of this programming model.

6.2.2 ADSM Run-time Design Rationale

As discussed in Chapter 2, distributed memories are necessary to extract all the computational power from heterogeneous systems. However, the accelerator-hosted data transfer model presented in Chapter 4 hides the complexity of this distributed memory architecture from programmers by unifying distributed memories into a single virtual address space.

A DSM run-time system reconciles physically distributed memory and logical shared memory. Traditional DSM systems are prone to thrashing, which is a performance limiting factor. Thrashing in DSM systems typically occurs when two nodes compete for write access to a single data item, causing data to be transferred back and forth at such a high rate that no work can be done. Access to synchronization variables, such as locks and semaphores, tends to

be a primary cause of thrashing in DSM. This effect is unlikely to occur in heterogeneous systems because hardware interrupts are typically used for synchronization between accelerators and CPUs and, therefore, there are no shared synchronization variables.

If synchronization variables are used (e.g., polling mode), special hardware mechanisms are typically used. These special synchronization variables are outside the scope of our ADSM design. False sharing, another source of thrashing in DSM, is not likely to occur either because sharing is done at the data object granularity.

A major issue in DSM is the memory coherence and consistency model. DSM typically implements a relaxed memory consistency model to minimize coherence traffic. Relaxed consistency models (e.g., release consistency) reduce coherence traffic at the cost of requiring programmers to explicitly use synchronization primitives (e.g. *acquire* and *release*). In the ADSM model memory consistency is only relevant from the CPU perspective because all consistency actions are driven by the CPU at method call and return boundaries. Shared data structures are released by the CPU when methods using them are invoked, and data items are acquired by the CPU when methods using them return.

DSM pays a performance penalty for detection of memory accesses to shared locations that are marked as invalid or dirty. Memory pages that contain invalid and dirty items are marked as not present in order to get a *page fault* whenever the program accesses any of these items. This penalty is especially important for faults produced by performance critical code whose execution is distributed among the nodes in the cluster. Data cannot be eagerly transferred to each node to avoid this performance penalty because there is no information about which part of the data each node will access. This limitation is not present in ADSM since the accelerator executing a method will access only shared data objects explicitly assigned to the method.

The lack of synchronization variables hosted by shared data structures, the implicit consistency primitives at call/return boundaries, and the knowledge of data structures accessed by performance critical methods are the three reasons that lead us to design an ADSM as an efficient way to support a data-centric programming model on heterogeneous parallel systems.

6.2.3 Application Programming Interface and Consistency Model

Four fundamental functions must be implemented by an ADSM system: shared-data allocation, shared-data release, method invocation, and return synchronization. Table 6.1 summarizes these necessary API calls.

Shared-data allocation and release calls are used by programmers to declare data objects that will be used by kernels. In its simplest form, the allocation call only requires the size of the data structure and the release requires the

API Call	Description
adsmAlloc(size)	Allocates *size* bytes of shared memory and returns the shared memory address where the allocated memory begins.
adsmFree(addr)	Releases a shared memory region that was previously allocated using *adsmAlloc()*.
adsmCall(kernel)	Launches the execution of method *kernel* in an accelerator.
adsmSync()	Yields the CPU to other processes until a previous accelerator calls finishes.

Table 6.1: Compulsory API calls implemented by an ADSM run-time

starting memory address for the data structure to be released. This minimal implementation assumes that any data structure allocated through calls to the ADSM API will be used by all accelerator kernels. A more elaborate scheme would require programmers to pass one or more method identifiers to effectively assign the allocated/released data object to one or more accelerator kernels.

Method invocation and return synchronization are already found in many heterogeneous programming APIs, such as CUDA. The former triggers the execution of a given kernel in an accelerator, while the latter yields the CPU until the execution on the accelerator is complete.

ADSM employs a *release consistency* model where shared data objects are released by the CPU on accelerator invocation (*adsmCall()*) and acquired by the CPU on accelerator return (*adsmSync()*). This semantic ensures that accelerators always have access to the objects hosted in their physical memory by the time a kernel operates on them. Implicit acquire/release semantics increase programmability because they require fewer source code lines to be written and it is quite natural in programming environments such as CUDA, where programmers currently implement this consistency model manually, through calls to `cudaMemcpy()`.

6.3 Design and Implementation

This section describes the design and implementation of *Global Memory for ACcelerators* (GMAC), a user-level ADSM run-time system. The design of GMAC is general enough to be applicable to a broad range of heterogeneous systems, such as NVIDIA GPUs or the IBM PowerXCell 8i. GMAC is implemented for GNU/Linux based systems that include CUDA-capable NVIDIA GPUs. The implementation techniques presented here are specific to the target platform but they can be easily ported to different operating systems (e.g. Microsoft Windows) and accelerator interfaces (e.g. OpenCL).

ADSM might be implemented using a hybrid approach, where low-level functionalities are implemented in the operating system kernel and high-level API calls are implemented in a user-level library. Implementing low-level layers within the operating system kernel code allows I/O operations involving shared data structures to be fully supported without performance penalties. All GMAC

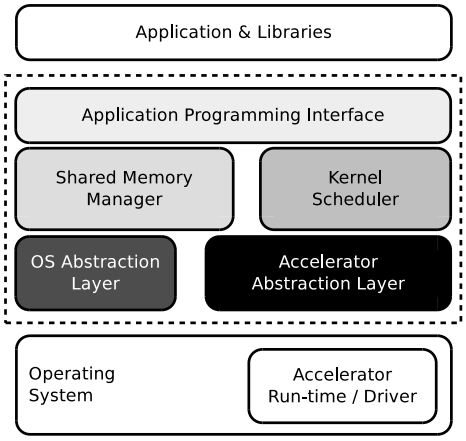


Figure 6.1: Software layers that conforms the GMAC library.

code is implemented in a user-level library because there is currently no operating system kernel-level API for interacting with the proprietary NVIDIA driver required by CUDA.

6.3.1 Overall Design

Figure 6.1 shows the overall design of the GMAC library. The lower-level layers (OS Abstraction Layer and Accelerator Abstraction Layer) are operating system and accelerator dependent, and they offer an interface to upper-level layers for allocation of system and accelerator memory, setting of memory page permission bits, transfer of data between system and accelerator memory, invocation of kernels, and waiting for completion of accelerator execution, I/O functions, and data type conversion, if necessary. The reference implementation the OS Abstraction Layer interacts with POSIX-compatible operating systems such as GNU/Linux. One Accelerator Abstraction Layer is implemented to interact with CUDA-capable accelerators: the CUDA Driver Layer. This layer interacts with the CUDA driver, which offers a low-level API and, thus, allows having full control over accelerators at the cost of more complex programming of the GMAC code base. The top-level layer implements the GMAC API to be used by applications and libraries.

The Shared Memory Manager in Figure 6.1 manages shared memory areas and creates the shared address space between host CPUs and accelerators. An independent memory management module allows testing of different management policies and coherence protocols with minor modifications to the GMAC code base. The kernel scheduler selects the most appropriate accelerator for execution of a given kernel, and implements different scheduling policies depending on the execution environment. A detailed analysis of kernel scheduling is out of the scope of the present dissertation and the reader is referred to Jimenez et al. [JVG+09] for a deeper analysis.

6.3.2 Shared Address Space

GMAC builds a shared address space between the CPUs and the accelerator. When the application requests shared memory (*adsmAlloc()*), accelerator memory is allocated on the accelerator, returning a memory address (virtual or physical, depending on the accelerator) that can be only used by the accelerator. Then, GMAC requests the operating system to allocate system memory over the same range of virtual memory addresses. In a POSIX-compliant operating system this is done through the `mmap` system call, which accepts a virtual address as a parameter and maps it to an allocated range of system memory (*anonymous memory mapping*). At this point, two identical memory address ranges have been allocated, one in the accelerator memory and the other one in system memory. Hence, a single pointer can be returned to the application to be used by both CPU code and accelerator code.

The operating system memory mapping request might fail if the requested virtual address range is already in use. In the single-GPU target system this is unlikely to happen because the address range typically returned by calls to `cudaMalloc()` is outside the ELF program sections. However, this implementation technique might fail when using other accelerators (e.g. ATI/AMD GPUs) or on multi-GPU systems, where calls to `cudaMalloc()` for different GPUs are likely to return overlapping memory address ranges. A software-based solution for this situation requires two new API calls: *adsmSafeAlloc(size)* and *adsmSafe(address)*. The former allocates a shared memory region, but returns a pointer that is only valid in the CPU code. The latter takes a CPU address and returns the associated address for the target GPU. GMAC maintains the mapping between these associated addresses so that any CPU changes to the shared address region will be reflected in the accelerator memory. Although this software technique works in all cases where shared data structures do not contain embedded pointers, it requires programmers to explicitly call *adsmPtr()* when passing references to the accelerator. Since OpenCL does not allow the use of pointers in GPU kernel code, such kernels are already written using relative memory indexing arithmetic.

GMAC also implements a multi-accelerator safe allocation code that allows using 64-bit virtual system memory addresses within the code executed by 32-bit legacy GPUs. First, memory is allocated in the accelerator (`cudaMalloc()`) and an accelerator physical memory address is returned. Then, GMAC allocates system memory in a 4 GB region in such a way that the 32 least significant bits of the system virtual memory address and the accelerator physical memory address match. For instance, assume that the allocated range in accelerator memory starts at accelerator physical address `0x00010100`. The `gmacMalloc()` code will request a system memory allocation starting at the system virtual memory address `0x100010100`. When this system virtual memory address is interpreted by the accelerator as an accelerator physical memory address, the hardware

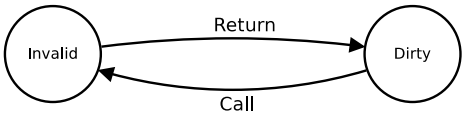
truncates the value to 32 bits (`0x00010100`), which is the accelerator physical address where the data is hosted. A second execution thread, using a different accelerator, might also request a shared data allocation, and CUDA might allocate accelerator memory starting at (or overlapping with) the same accelerator physical memory range (i.e., `0x00010100`). The code in GMAC identifies this new allocation as belonging to a different accelerator and, thus, will use a different 4 GB system virtual memory address range (e.g., `0x200010100`). This implementation approach requires a 4 GB virtual memory address range per execution thread making use of accelerators. This is an affordable cost in current systems. For instance, current 64-bit x86 processors use 47 bits to address the user-accessible virtual memory space, which means up to 32768 4 GB chunks are available. Multi-accelerator systems, such multiple GPUs card servers, will likely include few accelerators and, thus, only require a few 4 GB chunks in the virtual address space. Note that this implementation approach is only valid for 32-bit accelerators.

A good solution to the problem of conflicting address ranges between multiple accelerators is to have virtual memory mechanisms in accelerators. With virtual memory mechanisms in both CPUs and accelerators, the *adsmAlloc()* can be guaranteed to find an available virtual address in both CPU's address space and accelerator's address space. Thus, accelerators and CPUs can always use the same virtual memory address to refer to shared data structures. Additionally, accelerator virtual memory simplifies the allocation of shared data structures. In this case, the implementation of *adsmAlloc()* first allocates system and accelerator memory and fills the necessary memory translation data structures (e.g., a page table) to map the allocated physical system and accelerator memory into the same virtual memory range on the CPU and on the accelerator. Virtual memory mechanisms are implemented in latest GPUs, but not available to programmers [HS09]

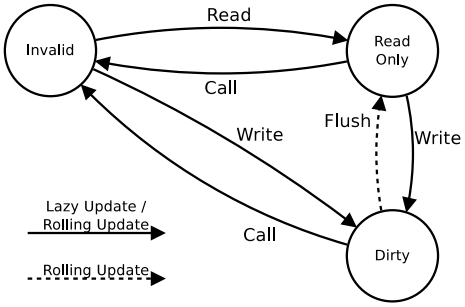
6.3.3 Memory Coherence Protocols

The layered GMAC architecture allows multiple memory coherence protocols to coexist and enables programmers to select the most appropriate protocol at application load time. The GMAC coherence protocols are defined from the CPU perspective. All book-keeping and data transfers are managed by the CPU. The accelerators do not perform any memory consistency or coherence actions. This asymmetry allows the use of simple accelerators.

Figure 6.2 shows the state transition diagrams for the coherence protocols provided by GMAC. In the considered protocols, a given shared memory range can be in one of three different states. *Invalid* means that the memory range is only in accelerator memory and must be transferred back if the CPU reads this memory range after the accelerator kernel returns. *Dirty* means that the CPU has an updated copy of the memory range and this memory range must be



(a) Batch-update



(b) Lazy-update and Rolling-update

Figure 6.2: State transition diagram for the memory coherence protocols implemented in GMAC.

transferred back to the accelerator when the accelerator kernel is called. *Read-only* means that the CPU and the accelerator have the same version of the data so the memory region does not need to be transferred before the next method invocation on the accelerator.

Batch-update is a pure write-invalidate protocol. System memory gets invalidated on kernel calls and accelerator memory gets invalidated on kernel return. On a kernel invocation (*adsmCall()*) the CPU invalidates all shared objects, whether or not they are accessed by the accelerator. On method return (*adsmSync()*), all shared objects are transferred from accelerator memory to system memory and marked as dirty, thus implicitly invalidates the accelerator memory. The invalidation prior to method calls requires transferring all objects from system memory to accelerator memory even if they have not been modified by the CPU. The memory manager keeps a list of the starting address and size of allocated shared memory objects in order to perform these transfers. This is a simple protocol that does not require detection of accesses to shared data by the code executed on the CPU. This naive protocol mimics what programmers tend to implement in the early stages of application implementation.

Lazy-update improves upon batch-update by detecting CPU modifications to objects in read-only state and any CPU read or write access to objects in invalid state. These accesses are detected using the CPU hardware memory protection mechanisms (accessible using the `mprotect()` system call) to trigger a page fault exception (delivered as a POSIX signal to user-level), which causes a page fault handler to be executed. The code inside the page fault handler implements the state transition diagram shown in Figure 6.2(b).

Shared data structures are initialized to a read-only state when they are allocated, so read accesses do not trigger a page fault. If the CPU writes to

any part of a read-only data structure, the structure is marked as dirty, and the memory protection state is updated to allow read/write access. Memory protection hardware is configured to trigger a page fault on any access (read or write) to shared data structures in invalid state. Whenever a data structure in invalid state is accessed by the CPU, the object is transferred from accelerator memory to system memory, and the data structure state is updated to read-only, on a read access, or to dirty on a write access.

On a kernel invocation all shared data structures are invalidated and those in the dirty state are transferred from system memory to accelerator memory. On kernel return no data transfer is done and all shared data objects remain in invalid state. This approach presents two benefits: (1) only data objects modified by the CPU are transferred to the accelerator on method calls, and (2) only data structures accessed by the CPU are transferred from accelerator memory to system memory after method return. This approach produces important performance gains with respect to batch-update in applications where the code executed on the accelerator is part of an iterative computation and the code executed on the CPU after the accelerator invocation only updates some of the data structures used or produced by the code executed on the accelerator.

Rolling-update is a hybrid write-update/write-invalidate protocol. Shared data structures are divided into fixed size memory blocks. The memory manager, as in *batch-update* and *lazy-update*, keeps a list of the starting addresses and sizes of allocated shared memory objects. Each element in this list is extended with a list of the starting addresses and sizes of the memory blocks composing the corresponding shared memory object. If the shared object size of any of these blocks is smaller than the default memory block size, the list will include the smaller value. The same memory protection mechanisms that lazy-update uses to detect read and write accesses to dirty and read-only data structures are used here to detect read and write accesses to dirty and read-only blocks. This protocol only allows a fixed number of blocks to be in the dirty state on the CPU, which is called *rolling size*. If the maximum number of dirty blocks is exceeded due to a write access that marks a new block as dirty, the oldest block is asynchronously transferred from system memory to accelerator memory and the block is marked as read-only (dotted line in Figure 6.2(b)). In the base implementation, an adaptive approach to set the rolling size is used: every time a new memory structure is allocated (*adsmAlloc()*), the rolling size is increased by a fixed factor (with a default value of 2 blocks). This approach exploits the fact that applications tend to use all allocated data structures at the same time. Creating a dependence between the number of allocated regions and the maximum number of blocks in the dirty state ensures that, at least, each region might have one of its blocks marked as dirty.

Rolling-update exploits the spatial and temporal locality of accesses to data structures in much the same way that hardware caches do. It is expected that codes that sequentially initialize accelerator input data structures will benefit

from rolling-update. In this case, data is eagerly transfered from system memory to accelerator memory while the CPU code continues producing the remaining accelerator input data. Hence, rolling-update will automatically overlap data transfers and computation on the CPU. Each time the CPU reads an element that is in invalid state, it fetches only the fixed size block that contains the element accessed. Therefore, rolling update also reduces the amount of data transferred from accelerators when the CPU reads the output kernel data in a scattered way.

All coherence protocols presented in this section contain a potential deficiency. If, after an accelerator kernel returns, the CPU reads a shared object that is not written by the kernel, it must still transfer the data value back from the accelerator memory. Interprocedural pointer analysis [CH00] in the compiler or programmers can annotate each kernel call with the objects that the kernel will write to, then the objects can remain in read-only or dirty state at accelerator kernel invocation. ADSM enables interprocedural pointer analysis to detect those data structures being accessed by kernels, because both CPU and accelerator use the same memory address to refer to the same memory object.

6.3.4 I/O and Bulk Memory Operations

An I/O operation might, for instance, read data from disk and write it to a shared object. First, a page fault occurs because the call to `read()` requires writing to a read-only memory block. Then, GMAC marks the memory block as read/write memory and sets the memory block to the dirty state, so the call to `read()` can proceed. However, when using rolling-update, once the first memory block is read from disk and written to memory, a new page fault exception is triggered because `read()` requires writing to the next memory block of the shared object. However, the second page fault aborts the `read()` function and after handling the second page fault, the `read()` function cannot be restarted because the first block of its data has already been read into the destination. The operating system prevents an ongoing I/O operation from being restarted once data has been read or written. GMAC uses library interposition to overload I/O calls to perform any I/O read and write operations affecting shared data objects in block sized memory chunks and, thus, avoids restarting system calls. GMAC offers the illusion of *peer DMA* to programmers, but the current implementation still requires intermediate copies in system memory.

Library interposition is used in GMAC to overload bulk memory operations (i.e. `memset()` and `memcpy()`). The overloaded implementations check if the memory affected by bulk memory operations involve shared objects and they use the accelerator-specific calls (e.g. `cudaMemset()` and `cudaMemcpy()`) for shared data structures, while forwarding calls to the standard C library routines when only system memory is involved. Overloading bulk memory operations avoids unnecessary intermediate copies to system memory and avoids triggering page

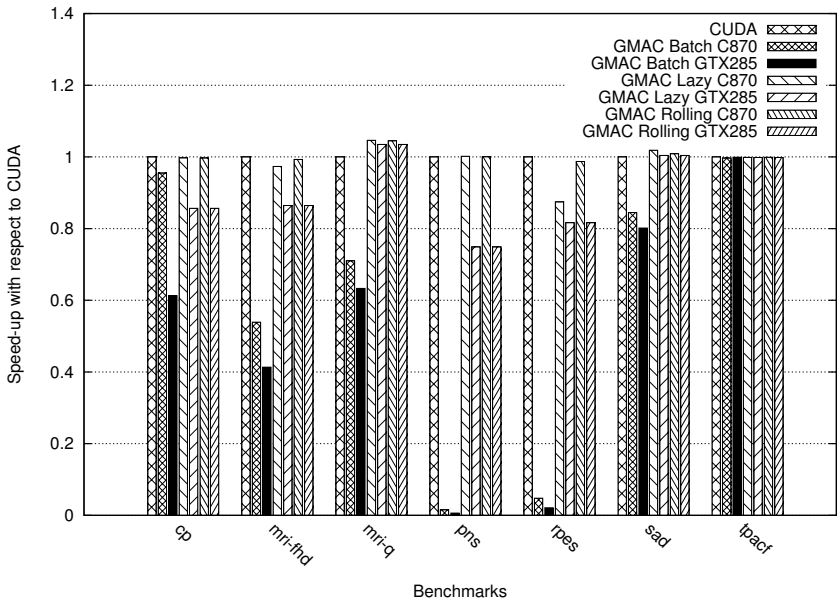


Figure 6.3: Slow down for different GMAC versions of Parboil benchmarks with respect to CUDA versions

faults on bulk memory operations.

6.4 Experimental Results

This Section presents an experimental evaluation of the GMAC library. The experiments described in this section use the benchmarks and systems described in Chapter 3 and follows the experimental methodology also discussed in that chapter.

The porting time from CUDA to GMAC for the seven benchmarks included in Parboil took less than eight hours of work. The porting process does not significantly differ from the porting of applications to NUAMA, which has been already discussed in Chapter 5.

6.4.1 Coherence Protocols

Figure 6.3 shows the slow-down for all benchmarks included in the Parboil Benchmark Suite with respect to the default CUDA implementation for GMAC using different coherence protocols. The GMAC implementation using the *batch-update* coherence protocol always performs worse than other versions, producing a slow-down of up to 65.18X in *pns* and 18.64X in *rpes*. GMAC implementations using *lazy-update* and *rolling-update* achieve performance equal to the original CUDA implementation.

Figure 6.4 shows data transferred by *lazy-update* and *batch-update* normalized to the data transferred by *batch-update*. The *batch-update* coherence pro-

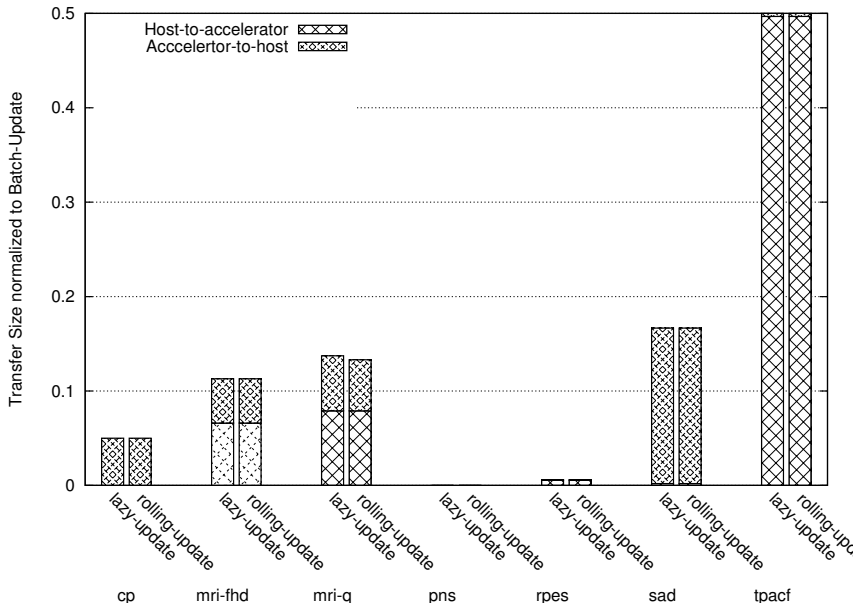


Figure 6.4: Transferred data by different protocols normalized to data transferred by *Batch-update*

tocol produces long execution times because data is transferred back and forth from system memory to accelerator memory on every accelerator invocation. This illustrates the first benefit of a data-centric programming model, where data transfers are automatically handled by the run-time system. Inexperienced programmers tend to take an over-conservative approach at first, transferring data even when it is not needed. A data-centric programming model automates data transfer management and, thus, even initial implementations do not pay the overhead of unnecessary data transfers.

The original CUDA code, *lazy-update*, and *rolling-update* achieve similar execution times. In some benchmarks, there is a small speed-up for GMAC with respect to the CUDA version. This shows the second benefit of GMAC: applications programmed using GMAC perform as well as a hand-tuned code using existing programming models, while requiring less programming effort.

Fine-grained handling of shared objects in *rolling-update* avoids some unnecessary data transfers (i.e. *mri-q* in Figure 6.4). A 3D-Stencil computation illustrates the potential performance benefits of *rolling-update*. Figure 6.5 shows the execution time of this benchmark for different input volume sizes and memory block sizes. As the volume size increases, *rolling-update* offers a greater benefit than *lazy-update*. The 3D-Stencil computation requires introducing a *source* on the target volume on each time-step, which tends to have zero values for most of the volume because it represents a small emitter localized at some point in space. In this version, the CPU executes the code that performs the *source introduction*. *Lazy-update* requires transferring the entire volume prior to introducing the source, while *rolling-update* only requires transferring the few

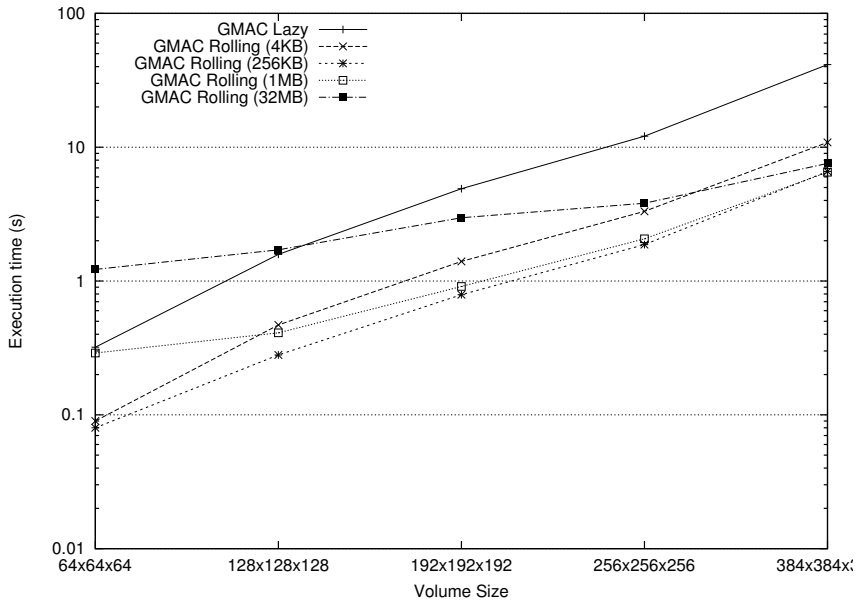


Figure 6.5: Execution time for a 3D-Stencil computation for different volume sizes

memory blocks that are actually modified by the CPU. This is the main source of performance improvement of *rolling-update* over *lazy-update*.

The combining effects of eager data transfer versus efficient bandwidth usage are illustrated in Figure 6.5 too. This Figure shows that execution times are longer for a memory block size of 32MB than for memory block sizes of 256KB and 1MB, but the difference in performance decreases as the size of the volume increases. Source introduction typically requires only accessing to one single memory block and, hence, the amount of data transferred depends on the memory block size. 3D-Stencil also requires writing to disk the output volume every certain number of iterations and, thus, the complete volume must be transferred from accelerator memory. Writing to disk benefits from large memory block because large data transfers make a more efficient usage of the interconnection network bandwidth than smaller ones. As the volume size increases, the contribution of the disk write to the total execution time becomes more important and, therefore, a large memory block size reduces the writing time.

These results shows the importance of reducing the amount of transferred data. Figures 6.3 and 6.4 show the relationship between the amount of transferred data and the execution time. The largest slow-downs in *batch-update* are produced in those benchmarks that transfer the most data, (*rpes* and *pns*). Programmer annotation and/or compiler or hardware support to avoid such transfers is a clear need.

Figure 6.6 shows the break-down of execution time for all benchmarks when using *rolling-update*. Most execution time is spent on computations on the

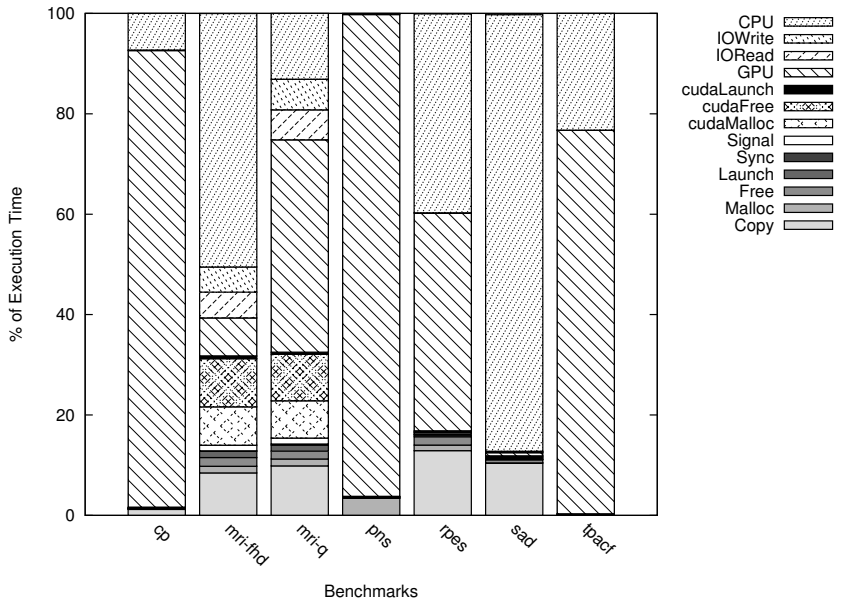


Figure 6.6: Execution time break-down for Parboil benchmarks

CPU or at the GPU. I/O operations, on those benchmarks that require reading from or writing to disk, and data transfers are the next-most time consuming operations in all benchmarks. The first remarkable fact is that the overhead due to signal handling to detect accesses to non-present and read-only memory blocks is negligible, always below 2% of the total execution time. Figure 6.6 also shows that some benchmarks (*mri-fhd* and *mri-q*) have high levels of I/O read activities and would benefit from hardware that supports peer DMA.

6.4.2 Memory Block Size

The memory block size is a key design parameter of the *rolling-update* protocol. The larger the memory block size, the less page fault exceptions are triggered in the processor. However, a small memory block size is essential to eagerly transfer data from system memory to the accelerator memory and to avoid transferring too much data from accelerator memory to system memory when reading scattered data from the accelerator memory.

The first experiment consists of running the Parboil benchmarks using different memory block sizes and fixing the maximum number of memory blocks in dirty state. Experimental results show that there is no appreciable difference in the execution time of Parboil benchmarks in this experiment, due to the small contribution by the CPU code accessing accelerator-hosted data to the total execution time.

A micro-benchmark that adds up two 8 million elements vectors is used to show how the execution time varies for different memory block size values. Figure 6.7 shows the execution time for different block sizes of this synthetic

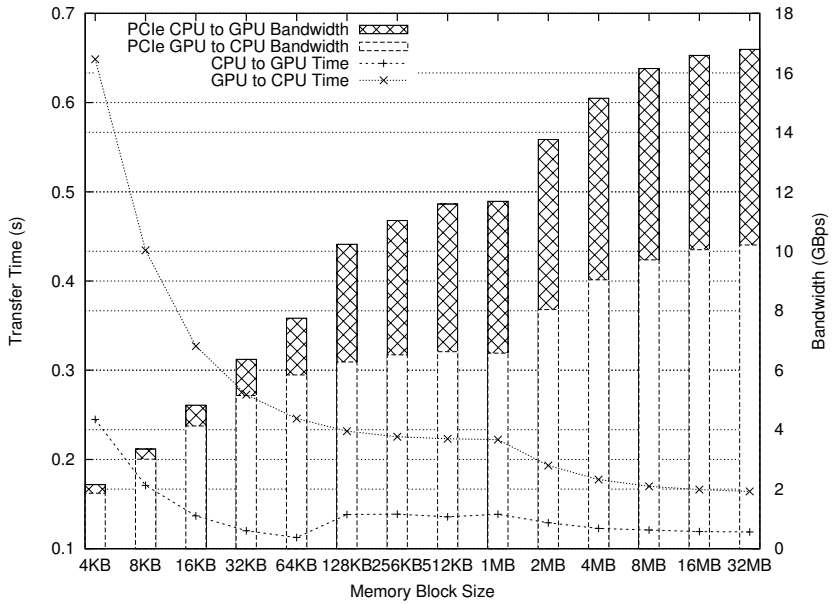


Figure 6.7: Execution times (lines) and maximum data transfer bandwidth (boxes) for vector addition for different vector and block sizes

benchmark using different block sizes. The average transfer bandwidth for each block size is also plotted. The data transfer bandwidth increases with the block size, reaching its maximum value for block sizes of 32MB. Data transfer times for vector addition decrease as the transfer bandwidth increases. The execution time reduction when moving from memory block sizes from 4KB to 8KB and from 8KB to 16KB is greater than the increase in the data transfer bandwidth. Small memory block sizes produce many page faults to be triggered by the CPU code producing and consuming the data objects. On a page fault, the GMAC code searches for the faulting block in order to modify its permission bits and state. GMAC keeps memory blocks in a balanced binary tree, which requires $O(log_2(n))$ operations to locate a given block. For a fixed data object size, a small memory block size requires more elements to be in the balanced binary tree and, thus, the overhead due to the search time becomes the dominant overhead. A large memory block size allows an optimal utilization of the bandwidth provided by the interconnection network and reduces the overhead due to page faults.

There is an anomaly in Figure 6.7 for a block size of 64KB. The CPU-to-accelerator transfer time for a 64KB memory block is smaller than for larger block sizes. The reason for this anomaly is the eager data transfer from the CPU. A small block size triggers a higher number of block evictions from the CPU to the accelerator which overlaps with other computations in the CPU. In this benchmark, when the block size goes from 64KB to 128KB, the time required to transfer a block from the CPU to the accelerator becomes longer than the time required by the CPU to initialize the next memory block and, therefore,

evictions must wait for the previous transfer to finish before continuing. Hence, a small enough memory block size is essential to overlap data transfers with computations at the CPU.

6.4.3 Rolling Size

This experiment uses a variable *rolling size* (maximum number of memory blocks on dirty state). The rolling size affects application performance in two different ways. First, the smaller the rolling size, the more eagerly data is transferred to accelerator memory and, thus, the more overlap between data transfers and computation. Second, memory blocks are marked as read-only once they are evicted from the memory block cache. Hence, any subsequent write to any memory location within a evicted block triggers a page fault and the eviction of a block from the memory block cache. This experiment shows that for all Parboil benchmarks, except for *tpacf*, the rolling size does not affect application performance in an appreciable way due to the way they are coded.

Execution time results for the *tpacf* benchmark illustrates quite a pathological case that might be produced by small rolling sizes. Figure 6.8 shows the execution time of *tpacf* for different block sizes using rolling sizes of 1, 2, and 4. For rolling size values of 1 and 2, and small memory block values, data is being transferred from system memory to accelerator memory continuously. The *tpacf* code initializes shared data structures in several passes. Hence, memory blocks of shared objects are written only once by the CPU before their state is set to read-only and they are transferred to accelerator memory. As the memory block size increases, the cost of data transfers becomes higher and, thus, the execution time increases. When the memory block size reaches a critical value (2MB for *tpacf-2* and 4MB for *tpacf-1*), memory blocks start being overwritten by subsequent passes before they are evicted, which translates to a shorter execution time. Once the complete input data set fits in the rolling size, the execution time decreases abruptly because no unnecessary updates are done. For a rolling size value of 4, the execution time of *tpacf* is almost constant for all block sizes. In this case, data is still being transferred, but the larger number of memory blocks that might be in dirty state allows memory blocks to be written by all passes before being evicted.

This results reveals an important insight that might be especially important for a hardware ADSM implementation. In such an implementation, given a fixed amount of storage, there is a trade-off between the memory block size and the number of blocks that can be stored. Experimental results shows that it is more beneficial to allow a higher number of blocks in dirty state than providing a large block granularity. Other considerations, such different costs for DMA transfers and page faults must be taken into account when designing a hardware ADSM system.

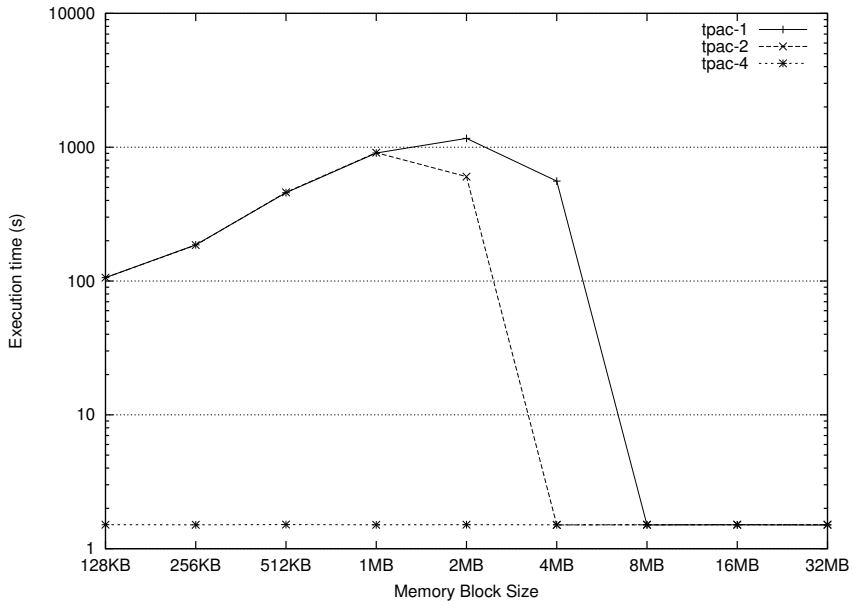


Figure 6.8: Execution time for *tpacf* using different memory block and rolling sizes

6.5 Summary

This chapter has introduced the ADSM programing model for heterogeneous parallel systems. ADSM implements the accelerator-hosted data transfer model that presents programmers with a shared address space between general purpose CPUs and accelerators. CPUs can access data hosted by accelerators, but not vice versa. This asymmetry allows memory coherence and consistency actions to be executed solely by the CPU. ADSM exploits this asymmetry (*page faults* triggered only by CPU code), the characteristics of heterogeneous systems (lack of synchronization using program-level shared variables), and the properties of synchronous function call programming models (*release consistency* and data object access information) to avoid performance bottle-necks traditionally found in symmetric DSM systems.

GMAC, a software user-level implementation of ADSM, and the software techniques required to build a shared address space and to deal with I/O and bulk memory in an efficient way using mechanisms offered by most operating systems has been discussed. Three different coherence protocols to be used in an ADSM system has been introduced: batch-update, lazy-update and rolling-update, each one being a refinement of the previous one. Experimental results show that rolling-update and lazy-update greatly reduces the amount of data transferred between system and accelerator memory and performs as well as existent programming models.

Based on the experience with ADSM and GMAC, it might be argued that future application development for heterogeneous parallel computing systems

should use ADSM to reduce the development effort, improve portability, and achieve high performance. Moreover, there is a clear need for memory virtualization to be implemented by accelerators in order to ADSM systems to be robust for heterogeneous systems containing several accelerators. Furthermore, hardware supported *peer DMA* can increase the performance of certain applications.

6.6 Significance

The ADSM model brings the accelerator-hosted model and the unified virtual address space to existent CPU – accelerator systems. Moreover, ADSM provides architecture independence, legacy support, and efficient I/O support.

The ADSM model illustrates how distribution asymmetric in terms of asymmetric visibility of the virtual address space and asymmetry in the memory coherence management is able to achive performance comparable to their counterparts using programmer-managed data transfers. Moreover, with the adequate hardware support, ADMS is likely to achieve higher performance than current programming models for processor – accelerator systems.

GMAC introduces two novel approaches to build an unified virtual address space on top of existent operating systems. GMAC also shows how library interposition might be used to optimize the performance of I/O and bulk memory operations. This very same technique can be use to allow ADSM systems to also achieve high performance on MPI applications.

Different memory coherence protocols for ADSM has been implemented in GMAC. The experimental results show the importance of ADSM coherence protocols that minimizes the amount of data transferred and are able to eagerly update the contents of the accelerator memory.

GMAC is current publicity available and has been successfully used in production applications.

Chapter 7

Heterogeneous Parallel Execution Model

7.1 Introduction

This chapter describes the Heterogeneous Parallel Execution (HPE) model for applications programmed following the accelerator-hosted data transfer model presented in Chapter 4. The HPE model extends the synchronous function call programming model discussed in Chapter 3 and defines how accelerators are integrated into user applications

This chapter first argues that existent OS abstractions are not sufficient for heterogeneous parallel systems and have to be modified, but these modifications should be backwards compatible with existent applications and systems. The OS extensions adopted by existent function call based programming models for CPU – accelerator systems are analyzed and shown to increase the complexity of programming heterogeneous parallel systems.

The HPE model extends the existent execution thread abstraction with execution modes. An execution mode defines the hardware resources (e.g., CPU or accelerator) accessible by an execution thread. In the HPE model all execution threads belonging to the same user process share a single CPU execution mode and each execution thread owns as many additional execution modes as accelerator types are supported by the system. The shared CPU execution mode provides compatibility with the existent execution model and the accelerator execution modes accelerator virtualization. In HPE, execution threads can only be in one execution mode at a time and accelerator calls are implemented as execution mode switches, so accelerator calls are synchronous. The HPE model based on execution modes provides full backwards compatibility with existent applications and systems.

This chapter also presents two different implementations of the HPE model in the GMAC library introduced in Chapter 6. There is a trade-off between performance and memory isolation between these implementations. The accelerator hardware support necessary to allow a efficient HPE implementation that provides memory isolation is described. Experimental results show that the HPE model, while improving programmability of heterogeneous parallel systems, produces little overheads.

7.2 HPE Model

This section presents an execution model for ADSM systems. First, the necessity for operating system (OS) abstractions to be modified to include different classes of processors (i.e., CPUs and accelerators) is identified. Furthermore, such modifications must be backwards compatible with existing abstractions in order to succeed. Modifications to the OS abstractions in current execution models for heterogeneous systems are then analyzed. Based on this analysis, the fundamental properties of an execution model for heterogeneous parallel computing systems are described.

7.2.1 Rationale and Guiding Principles

Most OSs export system resources to application programmers via user processes. User processes are composed by three main OS abstractions: a virtual address space, file descriptors and execution threads. The virtual address space abstracts the memory available to the programmer, file descriptors abstract I/O devices, and execution threads abstract CPUs. These three abstractions define an execution model where programmers create execution threads to spawn new execution flows. In this model, execution threads can access all file descriptors and virtual memory addresses of the process they belong to. This property is based on the assumption that all processors in the system have the same set of capabilities (e.g., the ability of accessing I/O devices). However, such an assumption does not remain valid on a heterogeneous system (e.g., GPUs cannot access I/O devices). Hence, existing OS abstractions have to be modified to accommodate the reality of heterogeneous systems.

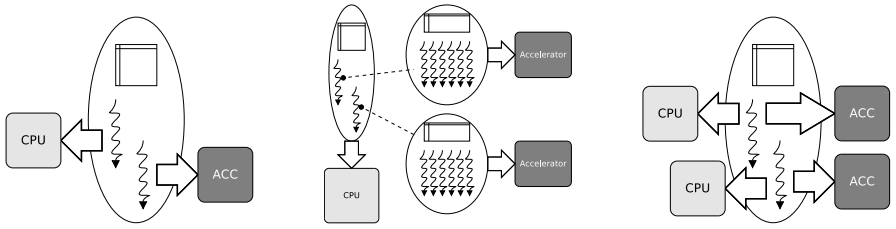
Existing OS abstractions and execution models for homogeneous systems are well understood by most programmers and are used in most applications. An execution model that is incompatible with existing applications would require a huge porting effort that most users are not likely to afford. Moreover, most programmers are likely to be reticent to adopt an execution model that is incompatible with the model that they have been successfully using for years. Furthermore, a backwards-compatible execution model cleanly supports existing (homogeneous) systems, by disabling or emulating the OS accelerator support. Hence, a single OS code-base might be able to support existing (homogeneous) and heterogeneous systems. Therefore, backward compatibility is a key requirement for the HPE execution model.

7.2.2 Existing Heterogeneous Execution Models

A first approach to integrate accelerators into the OS is to define a new abstraction for these devices. Such an abstraction would represent an execution flow running on the accelerator. However, application execution flows are already abstracted via execution threads, therefore such an abstraction would produce

```
1  // create a compute context with GPU device
2  ctx = clCreateContextFromType(CL_DEVICE_TYPE_GPU);
3
4  // create a work-queue
5  q = clCreateWorkQueue(ctx, NULL, NULL, 0);
6
7  // create the compute program
8  prg = clCreateProgramFromSource(ctx, 1, &src, NULL);
9
10 // build the compute program executable
11 clBuildProgramExecutable(prg, false, NULL, NULL);
12
13 // create the compute kernel
14 krn = clCreateKernel(prg,   k e r n e l  );
15
16 // set the args values
17 clSetKernelArg(krn, 0, (void *)&foo, 4, NULL);
18 clSetKernelArg(kr, 1, (void *)&bar, 4, NULL);
19
20  // execute kernel
21 clExecuteKernel(q, krn, NULL, range, NULL, 0, NULL);
```

Listing 7.1: Example OpenCL code that initializes an accelerator, generates code suitable to be executed on the accelerator and executes the code on the accelerator



(a) Execution Model in the IBM Cell SDK

(b) Execution Model in NVIDIA CUDA run-time API

(c) HPE Model

Figure 7.1: IBM Cell SDK, NVIDIA CUDA run-time API, and ADSM execution models for heterogeneous systems. In the figure ovals represent processes, tables virtual address spaces, and arrows execution threads. File descriptors are omitted to simplify the figure. All examples assume two execution threads per process.

an incongruent execution model. Such incongruence is likely to require OS code duplication (e.g., duplicated scheduling code for CPUs and accelerators), increasing chances of introducing bugs in the OS code and the complexity of the OS code and its maintainability. A different approach consists of reusing existing OS abstractions and incorporating accelerators into the execution model. This is the approach adopted by existing commercial programming models for heterogeneous systems using different base abstractions.

OpenCL [Mun09] and the CUDA driver API [NVI09] integrate accelerators into the execution model using file descriptor semantics. In this model, accelerators are abstracted as *contexts* and different context operations (e.g., allocating accelerator memory or launching a computation) available to programmers. These semantics increase the complexity of using accelerators. For instance, Listing 7.1 shows the code required to run a program that only performs a sin-

gle accelerator call in OpenCL. The required code in this example, is far more complex than, for instance, the single line of code required to perform a function call or to create a new execution thread in traditional systems. However, these semantics are quite convenient when managing accelerators inside the OS.

The IBM Cell SDK [IBM07] abstracts accelerators as execution threads, as illustrated in Figure 7.1(a). In this model, applications create new accelerator execution threads to execute code at accelerators. This model effectively requires programmers to transform sequential applications to multi-threaded applications when using accelerators. As a consequence, inter-thread synchronization code is required to use accelerators, increasing the complexity of programming heterogeneous systems. Moreover, the implementation of this model requires a *shadow* CPU thread per accelerator thread because accelerators included in the IBM Cell chip are not capable of executing OS kernel code. This implementation effectively couples accelerator execution to CPU execution and, for instance, when an accelerator thread is created, it has to wait for their correspondent *shadow* thread to be selected for execution by the OS to start running in the accelerator. This coupling might also have additional undesired side-effects in the overall system performance [MGN08].

The NVIDIA CUDA run-time API, in contrast to the CUDA driver API, uses processes to abstract accelerators, as illustrated in Figure 7.1(b). In this model each execution thread is associated to an accelerator process (i.e., CUDA context) for its whole lifetime. Accelerator processes are composed of an accelerator virtual address space and accelerator execution threads. In this model, computations in the accelerator are Remote Procedure Calls [BN84] (RPC) and, therefore, accelerator code is isolated from the CPU code and parameters are passed by-value. This approach is hard to adopt in the ADSM model, which implements single virtual address space for the code executed by both CPUs and accelerators, and, thus, allows by-reference parameter passing.

7.2.3 Execution Modes

The key insight to design the proposed HPE model is that programmers typically integrate accelerators into applications as execution flow migrations: the execution flow leaves the CPU and starts executing in the accelerator. From this perspective, the code executed by the CPU and by the accelerator belongs to the same execution flow, that is, to the same execution thread.

However, High Performance Computing (HPC) programmers tend to use *asynchronous* accelerator calls; on an asynchronous accelerator call, the application execution flow is split into a CPU and an accelerator execution flows. Asynchronous accelerator calls are used to execute accelerator and CPU code concurrently. This thesis argues that asynchronous accelerator calls should be allowed by the HPE model, but not supported natively. Despite the performance gains, asynchronous accelerator calls pose several problems if supported

directly by the OS. First, the execution thread's state (i.e., running, waiting, blocked) becomes undetermined. For instance, consider an application that asynchronously calls an accelerator and continues executing code in the CPU. Then, the OS scheduler preempts the execution thread because its time quantum has expired. At this point, the execution thread is in a *quantum state*: the execution thread is both waiting for the CPU and running in the accelerator. A potential solution to this problem is to avoid preempting threads that are running in the accelerator. However, such a solution would waste CPU cycles in the usual case of the code at the CPU is simply waiting for the accelerator to finish. Second, applications programmed assuming asynchronous accelerator calls might potentially fail on systems where such calls are not supported by the underlying hardware. Hence, this dissertation advocates for an execution model where execution threads are extended to include both CPUs and accelerators. In this model, illustrated in Figure 7.1(c), an execution thread might be executing on the CPU or on an accelerator, but not on both of them at the same time. This approach integrates accelerators as such, devices intended to speed up the execution of some computations. In this model, asynchronous accelerator calls, that is parallel CPU and accelerator execution, are supported by spawning new application execution threads. Moreover, existing run-time systems might be used on top of the HPE model to provide programmers with asynchronous accelerator calls.

In the HPE model, execution threads have different *execution modes*. Each thread owns as many different execution modes as processor classes (e.g., CPU or GPU) are present in the system. An execution mode defines the thread's execution environment:

- The processor properties, such as Instruction Set Architecture (ISA) (e.g., x86 or cubin) of the instructions executed by the thread on the execution mode. Processor properties also include the thread's ability to execute OS code, preemptibility, or accessibility of I/O devices in each execution mode.

- Virtual address space visibility. Each execution mode defines the subset of the user process virtual address space accessible by the execution thread. For instance, the CPU execution mode (i.e., the execution thread is running in the CPU) allows the thread to access any virtual memory address. However, the accelerator execution mode (i.e., the execution thread is running on the accelerator) only allows access to virtual memory addresses mapped to the accelerator physical memory.

The ADSM model is mapped to execution modes as follows: execution threads belonging to the same user process share a single CPU mode, but each execution thread has its own accelerator mode. In the HPE model, an accelerator call is translated into a switch from the CPU execution mode to an accelerator execution mode, analogously to a system call that is a privilege-level

switch from user to kernel-level. Returning from the accelerator switches the thread's execution mode back to the CPU. For the sake of simplicity, in the remainder of this chapter assumes two execution modes (CPU and accelerator) per thread. However, the concepts here developed are easily generalizable to an arbitrary number of thread execution modes.

7.2.4 Execution Mode Operations

Execution modes might be understood as a set of thread capabilities; an execution mode is a token that gives the thread permission to execute instructions from a specific ISA, and access to a subset of the virtual address space and file descriptors [Lev84]. This view is taken in the HPE model and, hence, two fundamental execution mode operations are offered to programmers: delegation and copy.

Delegation transfers the execution mode from the thread invoking the operation to a target thread, effectively revoking the permission of the caller thread to switch to such execution mode. For instance, context delegation might be used in streaming applications, where each execution thread switches to an accelerator mode to perform some computation over a tile of the application dataset, and, when this computation is done, the execution thread delegates its accelerator execution context to the following thread in the pipeline.

Execution mode copy duplicates the execution mode of the invoking thread and transfers one copy to a target thread. This operation effectively allows several execution threads to share one accelerator execution mode. One sample application that uses execution mode copy is, for instance, hybrid filtering where several filters are applied to an input set and the output data from all filters are combined into a single output. However, sharing one accelerator execution mode among several threads might serialize the execution of these threads in accelerator mode. The accelerator execution mode copy operation provides a means to share data between several threads when running in accelerator mode. The data sharing granularity accomplished by execution mode copy is the whole virtual address sub-space accessible from the accelerator execution mode.

7.2.5 Benefits and Limitations

The HPE model simplifies the task of programming heterogeneous multi-accelerator systems. This model is fully compatible with existing programming models and simplifies the task of porting applications to use accelerators. Performance-critical functions can be substituted by accelerator calls in our model because it offers the same calling semantics. This simple porting path is not possible in the other execution models previously discussed. For instance, the IBM Cell SDK requires encapsulating performance critical functions into separate execution threads. This might require major changes in the application code if non-thread-safe libraries or code is used. The CUDA driver API and

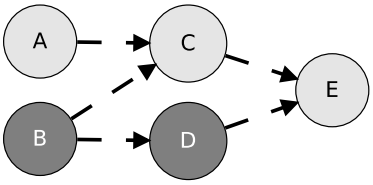


Figure 7.2: Sample data-flow that illustrates the importance of fine-grained synchronization between parallel control-flows in CPUs and accelerators.

OpenCL may require a major re-write of application code to incorporate context creation and management. The NVIDIA CUDA run-time API does not support by-reference parameter passing, so complex wrapping functions are required, as discussed in Chapter 4.

A potential limitation of the ADSM model is the need for specific calls to allocate data objects used in the accelerator code, but this limitation is also present in all existing programming models for heterogeneous systems. However, shared data object allocations are done with a single call, while the other models require two separate calls to allocate system and accelerator memory. This is key to accomplish backward compatibility; ADSM shared data allocations in accelerator-less systems only allocate system memory. However, accelerator memory allocation calls in other models simply fail, aborting the application execution. HPE cleanly complements ADSM to allow applications using accelerators to run on accelerator-less systems. In these systems, accelerator execution mode triggers the emulation of the accelerator code using the CPU [DKK09]. This accelerator emulation mode does not differ from floating-point emulation already implemented in by most OS's [HH97].

Asynchronous accelerator execution is not directly supported in the HPE model. On the contrary, all accelerator calls in OpenCL and CUDA execution models are asynchronous. Moreover, these models compel programmers to extensively use asynchronous accelerator calls and to defer accelerator synchronization (i.e., waiting for the accelerator to finish) as much as possible [NVI09]. Hence, the lack of asynchronous accelerator calls might be viewed as a major limitation of our execution model. However, HPE intentionally prevents asynchronous accelerator calls because they do not follow the sequential execution model and provides a limited form of parallel execution on CPUs and accelerators.

For instance, consider the data-flow in Figure 7.2, where dark-grey circles represent accelerator computations and light-grey circles CPU computations. This data-flow is a simplified version of a pattern found, for instance, in an Finite Difference Time Difference simulation of electromagnetic wave propagation in unbounded volumes, where snapshots of electromagnetic fields are taken every few iterations. In the data-flow in Figure 7.2, computations A and B, and C and D can proceed in parallel. However, the code in the CPU has to ensure that B has finished before starting C, but the GPU code can start D just after

Operation	Description
accAlloc	Allocates memory at the accelerator
accRelease	Releases accelerator memory
accLaunch	Launches a computation in the accelerator

Table 7.1: Basic accelerator object interface using in GMAC

finishing B. Asynchronous accelerator calls do not provide an easy mapping of this data-flow. The programmer asynchronously calls to B and compute A concurrently. Then, a synchronization call between CPU and accelerator is required to ensure that B has finished, before launching D asynchronously and start computing C. This CPU – accelerator synchronization, required due to the data dependency between B and C, degrades the application performance. For instance, the execution of D is delayed because the CPU has not finished computing A. HPE provides an elegant way to map the data-flow in Figure 7.2. The application uses one execution thread to compute A and C in the CPU, and another execution thread to compute B and D in the accelerator. A semaphore, for instance, can be used to avoid start computation C before B has finished: the first execution thread waits on the semaphore after finishing A, and the second thread increments the semaphore after finishing computation B. As illustrated in this example, the fine-grained inter-thread synchronization provided by HPE allows efficient CPU – accelerator parallel execution.

7.3 GMAC Design and Implementation

This section discusses the design of the HPE model. Different implementation options for this model in GMAC. The implementation techniques discussed in this section are applicable into the OS kernel-level. Furthermore, a OS kernel-level implementation would be preferred, so OS thread scheduling and memory management policies could be applied globally to both CPUs and accelerators.

7.3.1 Accelerator Management

Accelerators present in the system are discovered during the GMAC bootstrap process. Physical accelerators (e.g., ACC in Figure 7.3) are encapsulated inside accelerator objects, whose implementation depends on the accelerator architecture (e.g., CUDA or OpenCL). All accelerator objects implement the common interface shown in Table 7.1, so most GMAC code is accelerator-independent. Accelerator objects in GMAC are stored in a map of accelerator lists indexed by an accelerator-architecture key.

Accelerator objects are accessible to execution threads via accelerator contexts (or contexts, in short). The internal structure is illustrated in Figure 7.3. A context contains a reference to an accelerator object and one reference to a memory map object. The memory map object keeps the information about

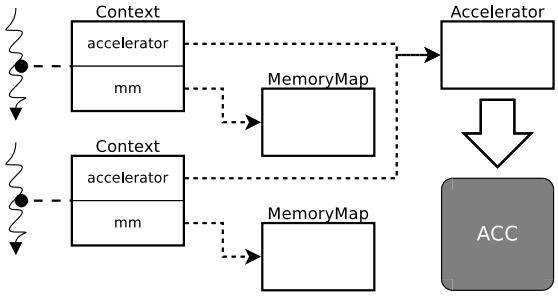
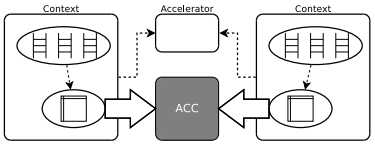
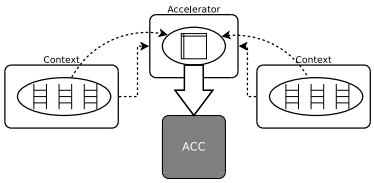


Figure 7.3: Internal accelerator management GMAC structure for two execution threads on a single-accelerator system. White boxes represent GMAC abstractions and ACC a physical accelerator



(a) Execution model implementation using one CUDA context per context



(b) Execution model implementation using one CUDA context per accelerator

Figure 7.4: Execution model model implementation alternatives for CUDA GPUs. Queues in ovals represent CUDA streams and tables accelerator virtual address spaces.

CPU – accelerator shared data. A reference to an accelerator object is required at context creation-time to allow the eager update of the accelerator memory, as presented in Chapter 6. At thread creation time, a context reference for each accelerator type present in the system is stored in the thread-specific data [iee09]. In an OS kernel-level implementation, this accelerator reference would be stored in the thread's task structure. Contexts are initialized lazily (i.e., when first required by the thread) to avoid context initialization times on threads not using accelerators.

Figure 7.4 illustrates two possible accelerator and context implementations for NVIDIA CUDA GPUs. In the implementation in Figure 7.4(a), each accelerator keeps a different CUDA device identifier, and each context a different CUDA context and a set of CUDA streams (represented as queues in elipses in Figure 7.4). In the implementation in Figure 7.4(b), accelerators keep a CUDA device identifier and one CUDA context, while contexts only use a set of CUDA streams. In both implementations, the set of CUDA streams per context allows automatic overlapping of execution and data transfers.

The implementation in Figure 7.4(a) is a natural approach, because the use of separate CUDA contexts provides accelerator memory isolation across contexts, as defined by the ADSM model. However, this implementation might produce important overheads due to the CUDA context creation and switch

costs. Moreover, using separate CUDA contexts also prevents exploitation of concurrent GPU kernel execution in new NVIDIA Fermi GPUs [HS09]. The implementation in Figure 7.4(b) uses one CUDA context per accelerator to eliminate the potential for the aforementioned performance overheads. However, such an implementation does not provide memory isolation across contexts when scheduled to run in the same physical GPU. GMAC allows selection of either implementation at compile-time, so users can choose the trade-offs between performance and protection.

Additional GPU hardware support can potentially remove the trade-off between performance and protection in the HPE implementation. The first possibility is hardware modifications to improve CUDA context creation and switching times. These two latencies are greatly reduced in new NVIDIA Fermi GPUs [HS09], but CUDA context creation and switch still are quite costly and are likely to remain relatively long due to the internal GPU architecture [LNOM08]. A second possibility is the addition of hardware structures to provide memory protection at the CUDA stream level. There is no fundamental design obstacle that prevents such hardware from being included in the GPU. For instance, a potential implementation adds a stream identifier register per processor (or Streaming Multiprocessor in CUDA terminology). GPU page table entries are also extended with a permission bitmap, where each bit indicates the access rights of the memory page for each stream. The stream identifier register is used to check the permission bitmap on memory accesses to ensure protection. This implementation provides full backward-compatibility by simply setting all permission bits in the permission bitmap.

7.3.2 Delegation, Copy and Migration

Execution mode delegation and copy operations are implemented in GMAC in two new API calls. These new calls delegate or copy the current context to the thread specified as a parameter. A per-thread context receive buffer is implemented to store contexts sent to each thread. On a context copy call, GMAC increments the usages count of the caller thread context and pushes this context into the target thread's receive buffer. Context delegation first detaches the context from the caller thread and, then, pushes this context into the destination thread receive buffer. Execution threads dequeue contexts from their receive buffers by invoking a context call receive operation.

The previous implementation of accelerator copy and delegation effectively changes the physical accelerator used in the accelerator execution mode because contexts are assigned to a physical accelerator at creation time. Hence, these operations might become an important source of accelerator load imbalance; for instance, consider an application running on a dual-GPU system where all threads assigned to the second accelerator copy their contexts to threads assigned to the first accelerator. Another source of load imbalance is an unequal

Benchmark	Consecutive Calls	GPU Time (μsec)	Sync Calls Overhead	Dataset (MB)
CP	10	13633	0.05%	1.00
MRI-FHD	2.5	4734	0.13%	5.05
MRI-Q	1.5	6075	0.08%	5.02
PNS	1	9912	0.04%	686.65
TPACF	1	940862	0.00%	4.77
SAD	3	424	1.52%	8.53
RPES	71	1752	0.40%	79.40

Table 7.2: Accelerator usage data from the Parboil benchmark suite

distribution of accelerator workload among application threads. Load balancing can be accomplished by migrating contexts from overloaded accelerators to any that are underutilized. GMAC implements a simple load balancing algorithm using the usages count for each physical accelerator to estimate the accelerator load. Whenever the usage count of one accelerator is much higher than the average accelerator usage count, a context assigned to this accelerator is considered for migration to the least utilized accelerator. The migration is only performed if the data movement cost is below a given threshold. More elaborate migration algorithms might be implemented, but a detailed analysis of context migration algorithms is left as future work.

7.4 Experimental Evaluation

7.4.1 Asynchronous Accelerator Calls

The major difference between the HPE model and existent OpenCL and CUDA execution models is the asynchronicity of accelerator calls; OpenCL and CUDA implement asynchronous accelerator calls, while HPE uses synchronous accelerator calls. In this experiment, we measure the overhead produced due to synchronization after accelerator calls in GMAC. The extra performance gains due to parallel CPU and GPU execution of asynchronous calls is not considered because concurrent CPU and GPU execution in HPE is supported using multiple execution threads.

Two synthetic benchmarks are used: *Null.Async* and *Null.Sync*. *Null.Async* performs several *null* asynchronous accelerator calls in a row, and then waits for the accelerator to finish computing. *Null.Sync* also performs several *null* accelerator calls, but waits for the accelerator to finish computing just after each accelerator call. The *null* accelerator call returns immediately after the invocation, so its execution time can be approximated to be zero. Hence, the execution time difference between these two benchmarks measure the cost of synchronization calls in HPE. Additionally, we have also run several tests that execute *non-null* accelerator calls with different GPU execution time values to check that the costs remain constant regardless the complexity of the code executed by the GPU.

Figure 7.5 shows the average difference in accelerator call cost between

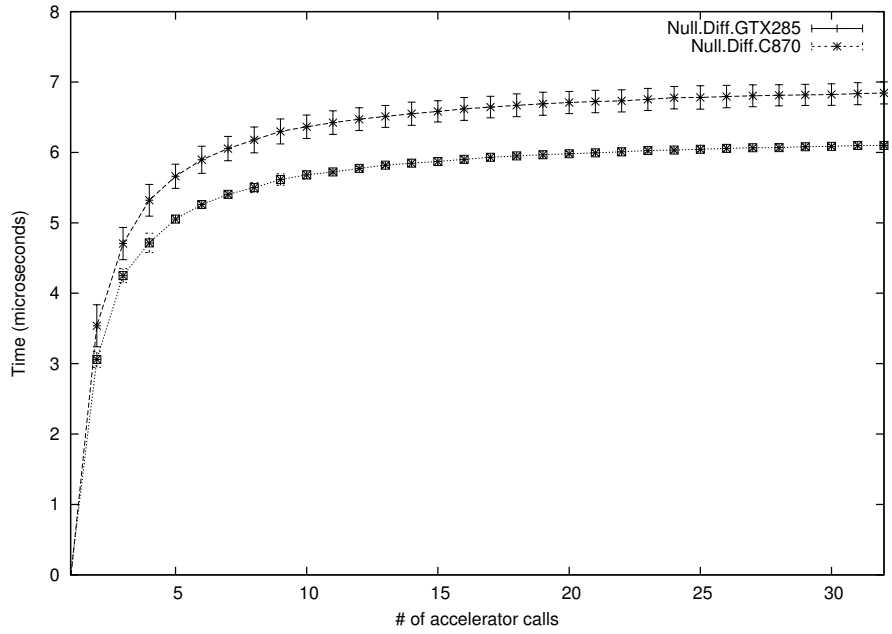


Figure 7.5: Execution time difference (in μsec) between consecutive asynchronous and synchronous accelerator calls.

Null.Sync and *Null.Async* for different NVIDIA GPUs. Asynchronous accelerator calls are 6 μseconds faster per call than synchronous calls for C870 GPUs, and 7 μseconds for GTX285 GPUs when the number of consecutive accelerator calls is large. However, for a small number of consecutive accelerator calls, the time difference decreases, becoming 4 μseconds in both systems for three consecutive accelerator calls. These extra costs might be contextualized using profiling data from the Parboil benchmark suit [IMP], shown in Table 7.2. The accelerator execution time overhead due to synchronous accelerator calls is 0.08% for *MRI-Q*, and 0.05% for *CP* due to the long average latency execution of each accelerator call. There is a 0.40% overhead in *RPES*, which performs a high number of short-latency consecutive accelerator calls. The largest overhead is produced in *SAD* because the very short average accelerator call execution time. These experimental results show that synchronous accelerator calls introduce a negligible overhead in the total accelerator execution time.

7.4.2 Context Creation and Switching

Two different execution model implementations for NVIDIA GPUs were discussed in the previous section. These implementation differ in the usage of CUDA contexts; the first implementation uses one CUDA context per accelerator context, while the second implementation uses one CUDA context per accelerator. The performance difference between these two implementations is measured using synthetic benchmarks. The *Create* benchmark measures the

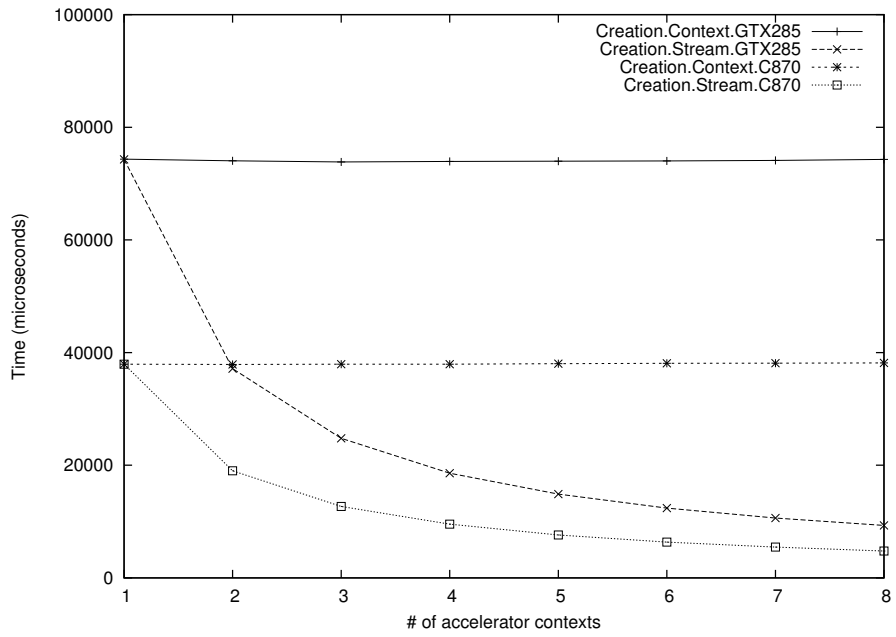


Figure 7.6: Average Per-context creation time (in microseconds) of an accelerator context

performance difference on accelerator context creation time. This benchmark consists of a loop where several GMAC contexts are created and initialized. The accelerator context switch time is measured using the *Switch* benchmark. This benchmark performs several consecutive *null* accelerator calls from different execution threads, and the accelerator call execution time is reported.

Figure 7.6 shows the creation time of an accelerator context as the number of accelerator contexts per user process increases. This figure shows results for the two implementation approaches: using one CUDA context per accelerator context (*Creation.Context*) and using one CUDA context per accelerator (*Creation.Stream*). Both implementations produce the same average overhead when a single accelerator context per physical accelerator is used. However, this overhead decreases exponentially as the number of accelerator contexts increase in *Creation.Stream*, while it remains constant in *Creation.Context*. Accelerator context creation time is almost constant in *Creation.Context* because a new CUDA context is allocated and initialized at the GPU on each call. *Creation.Stream*, on the contrary, presents decreasing average accelerator context creation times as the number of accelerator contexts grows. In this case, one CUDA context is created, and this cost is only paid on the first accelerator context creation, which requires initializing the accelerator.

Accelerator context switch cost is barely noticeable when the number of contexts is one because no switch is done. However, for two or more accelerator contexts, the switch in *Switch.Context* is much larger (200 μsecs in C870 and 460 μsecs in GTX285) than in *Switch.Stream* (10 μsec in C870 and GTX285).

Accelerator context switches in *Switch.Context* requires setting-up the GPU control information, because a new CUDA context becomes active. This is in contrast with *Switch.Stream* where the same CUDA context is used by all accelerator contexts and the GPU configuration is not modified.

These experimental results show that applications that create more threads than the number of accelerators to increase accelerator utilization can greatly benefit from the *Stream* implementation.

7.4.3 Context Copy and Delegation

Context delegation and copy operations provide a means for inter-thread communication and accelerator sharing. A *ping-pong* benchmark is designed to measure the round-trip cost of context delegation. This benchmark consists of two execution threads interchanging a single context; initially, the main thread delegates its context to the secondary thread and waits for the secondary thread to delegate a context to it. Analogously, the secondary thread first waits for the main thread to delegate its context and, as soon as it gets the context, the secondary thread delegates it to the main thread. This pattern repeats in an infinite loop. The time spent by both operations, context delegation and receive, on each loop iteration is measured. The round trip cost, in average, is 28 μseconds on the GTX285 system and 15 μseconds on the C870 system. A *null ping-pong* benchmark, where two threads wait/post on two shared semaphores is executed to measure the cost of inter-thread synchronization when no GMAC code is involved. The round-trip cost in this case is 18 μseconds on the GTX285 system and 11 μseconds on the C870 system. As expected, these results show that context delegation mainly depends on the performance of CPU mutual exclusion locks since no GPU operation is involved in context delegation.

Context copy allows several execution threads to share the same *visibility* of the virtual address space when running in accelerator mode. This functionality has little applicability currently due to the lack of concurrent stream execution in the current GPU generation. However, upcoming GPUs (and other kinds of accelerators) will allow several streams to run concurrently on the same physical accelerator [HS09]. The context copy time is measured using a synthetic benchmark formed by two execution threads. The main thread executes a loop where its context is copied to the secondary thread on each loop iteration. The secondary thread executes a loop, where a context is received on each iteration. The measured context copy times are 4 μseconds on the GTX285 system and 3 μseconds on the C870 system.

7.4.4 Context Migration

The cost of context migration between accelerators is a key factor in developing load-balancing policies in multi-threaded applications and multi-programmed systems. This section measures the cost of context migration and sets the base

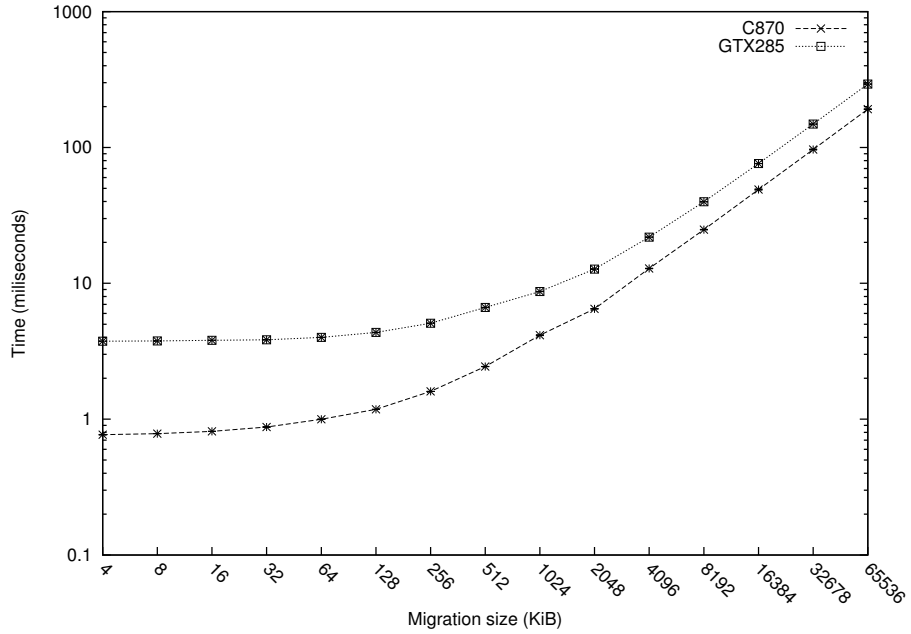


Figure 7.7: Context Migration

for future accelerator scheduling research. A synthetic benchmark is used to evaluate the cost of context migration. This benchmark forces an accelerator context migration changing the thread's accelerator affinity through a GMAC call. Before setting the accelerator affinity, the benchmark allocates shared data structures of a configurable size.

Figure 7.7 shows the context migration time for different application dataset sizes. Context migration is an expensive operation, taking about 4 milliseconds in GTX285 for very small dataset values and up to 1 second for datasets of 1GB. The context migration time is mainly formed by two components: context creation and data transfers. The context creation time, which does not depend on the dataset size, is about 4 milliseconds in GTX285 (see Figure 7.6) if no previous context exists in the target accelerator. However, the context creation time becomes negligible when compared with the data transfer time if the target accelerator already has active contexts. These experimental results show that the target accelerator state (i.e., initialized vs. idle) must be considered for context migration policies.

Context migration time becomes dominated by data transfer cost as the context dataset grows. Figure 7.7 shows that the context migration cost follows a linear dependency with the data transfer size for dataset sizes larger then 2MB. Such a simple dependency might be easily integrated in the context migration policy to estimate the cost of migrating different contexts. This migration time might be used together history-based algorithms [JVG+09] to trigger context migrations.

Profiling information from the Parboil benchmark suite in Table 7.2 shows

that context migration might be only triggered on situations of high load unbalance. For instance, the cost of migrating PNS is two orders of magnitude higher than the accelerator execution time. In Parboil benchmarks, context migration would represent little overhead for the case of TPACF. This benchmark has a dataset of 4.77MB (i.e., a migration time of about 20ms in GTX285) and an average accelerator execution time of 940ms per accelerator invocation. Unfortunately history-based algorithms are of little help in this case because TPACF performs a single accelerator call. These experimental results highlight the importance of algorithms to select the accelerator assignment on context initialization time.

7.5 Summary

This chapter has introduced the HPE model for heterogeneous parallel systems. HPE integrates accelerators into the execution thread abstraction, provided by most operating systems. In HPE, execution threads are extended with execution modes, which defines the hardware resources accessible by the execution thread. All execution threads belonging to the same user process share a common CPU execution mode, which is active when the application is running at the CPU. Moreover, execution threads also own one execution mode per kind of accelerator present in the system. In HPE, applications call accelerators by switching its active execution mode from the CPU to the accelerator being invoked. The HPE model is fully compatible with existent applications, keeping the sequential programming model that programmers are used to. Furthermore, the HPE model provides backwards compatibility by allowing the emulation of accelerators in software whenever they are not present, such in legacy systems.

Execution modes provide application programmers with a programming model with synchronous accelerator calls. In this programming model, the application execution flow can be only running in one processor (i.e., CPU or accelerator) at the time. HPE allows parallel CPU – accelerator execution by spawning new user threads, in the same way that parallel execution on multi-core CPUs is accomplished in current operating systems. Using separate execution threads to allow concurrent CPU – accelerator execution enables fine-grained synchronization between the code executed in the CPU and the code executed in the accelerator, which is not possible in existent programming models where accelerators are called asynchronously.

Two implementation approaches for the HPE model in GMAC has been presented. There is trade-off between system performance and memory isolation in these implementations due to the current lack of memory protection in accelerators. This chapter has outlined the modifications on the accelerator hardware and the accelerator virtual memory structures required to support memory protection while being compatible with the existent hardware. Experimental results have shown that the HPE model produces little overhead.

7.6 Significance

The HPE model integrates accelerators in the execution model implemented by most contemporary operating systems, providing full backwards compatibility with existent applications and systems. The HPE mode, by providing application programmers with an execution model they are familiar with, improves the programmability of heterogeneous parallel systems and eases the adoption of accelerator in general purpose applications.

The HPE model introduces the concept of execution mode to define the different processors (i.e., CPUs and accelerators) where execution threads might be executed. Execution modes allow the implementation of an execution model where accelerators are called by requesting an execution mode switch to the OS. Execution mode switches are analogous to privilege-level switches, currently used to perform system calls. Execution modes also allow a simple mechanism for the operating system to allow executing applications on systems without accelerator: on an execution mode switch, the OS emulates in software the accelerator being invoked.

The HPE model implementation in GMAC has illustrated the need for memory protection mechanisms to be implemented by accelerators. The necessary hardware support outlined in this chapter would allow an efficient implementation of the HPE model while being fully compatible with current applications.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Heterogeneous systems formed by general purpose processors and accelerators allow efficient execution of a wide range of applications formed by sequential control-intensive and data-parallel phases. These kind of systems are becoming widely adopted in the high-performance computing field in supercomputing centers, industry and academia.

Millions of heterogeneous desktop systems that include CPUs and GPUs are sold every year. However, general purpose applications, such as word processors, spreadsheets and multimedia applications, are not exploiting the additional computational power present in current desktop systems. The major drawbacks for the adoption of the heterogeneous computing paradigm in general purpose applications are the complex programming model and lack of backwards compatibility.

This dissertation has used a comprehensive approach to improve programmability and backwards compatibility of heterogeneous systems. An analysis of the existent data transfer models for heterogeneous systems have reveled the lack of by-reference parameter passing to accelerators and the usage of separate virtual address spaces for system and accelerator memories. Current data models require parameters to be passed by-value to accelerators and, therefore, extra code to perform memory copies between system and accelerator memories is needed. These memory copies harm the application performance, which might be hidden using double-buffering techniques that further increases the task of programming heterogeneous systems. Separate system and accelerator virtual address spaces produce the double-pointer problem: a single data structure is referenced by different pointer in the CPU and accelerator code. Keeping these two pointers consistent also increases the complexity of programming heterogeneous systems.

This dissertation has presented the accelerator-hosted data transfer model, where data structures used by accelerators are only hosted by the accelerator memory. Such a model allows parameters to be passed by-reference to accelerators and, therefore, removes the need of explicit memory copy operations. This model has been also extended to build an unified virtual address space, shared

by CPUs and accelerators. This unified virtual address space solves the double-pointer problem by allowing data structures to be referenced by the same virtual memory address in both, the CPU and accelerator code. The accelerator-hosted data transfer model and the unified virtual address space for CPUs and accelerators are the two first contributions of the this dissertation.

The Non-Uniform Accelerator Memory Access (NUAMA) architecture has been also introduced in this dissertation. The NUAMA architecture allows CPUs to access the accelerator memory using regular load/store instructions. Data structures hosted by the accelerator memory are identified using one extra bit in the page table entries. An Accelerator Memory Collector (AMC) is integrated within the system memory controller to identify memory accesses to accelerator-hosted data. The AMC buffers and coalesces memory requests for the accelerator to improve the throughput of data transfers to the accelerator memory. The NUAMA architecture also uses a write-through/write-non-allocate policy in the L2 cache for accelerator-hosted data, while keeping the write-back/write-allocate policy for the data hosted in system memory. This hybrid L2 cache policy is the key to eagerly update the contents of the accelerator memory, so on an accelerator call few data needs to be transferred. Moreover, by allowing accelerator-hosted data to be cached in the CPU, repeated accesses and accesses to contiguous memory locations are not penalized. The NUAMA architecture is the third main contribution of this dissertation.

This dissertation has introduced the Asymmetric Distributed Shared Memory (ADSM) system. ADSM allows the implementation of the accelerator-hosted model and the unified virtual address space on top of existent systems. The key insight of ADSM is the double asymmetry of distribution:

- Accelerators can only access data hosted on their own memory, while CPUs can access any location of the virtual address space.
- All coherence actions are performed by CPUs.

This asymmetry allows the usage of simple accelerators without any additional hardware support. The design and implementation of Global Memory for Accelerators (GMAC), an ADSM system as a user-level library, has been also discussed in this dissertation. The description of ADSM and the GMAC runtime, which is publicly available, are the forth and fifth contributions of this dissertation, respectively.

Finally, Heterogeneous Parallel Execution (HPE) model for ADSM systems has been presented in this dissertation. The HPE model accomplishes full backwards compatibility with existent sequential execution models based on execution threads, which provides the following benefits:

- Existent applications can run unmodified on top of the ADSM systems.
- Applications requiring accelerators can execute on top of legacy systems, being accelerators emulated in software.

The design and implementation in GMAC of such execution model has been presented, and is the sixth contribution of this dissertation.

8.2 Future Work

The work presented in this dissertation has opened new research lines for CPU – accelerator systems in the fields of memory management, accelerator scheduling, and programming models. This new research lines are translated in potential future work, which is outlined in this section.

8.2.1 Accelerator Memory System

On an accelerator call, NUAMA and ADSM requires invalidating accelerator hosted data in the cache hierarchy and system memory respectively. This invalidation ensures that after an accelerator call, the CPU access the updated copy of the data hosted in the accelerator memory. This approach has a potential performance overhead because accelerator-hosted data structures are invalidated even if the accelerator does not actually modified them (i.e., read-only input parameters). Hence, after an accelerator call, CPU accesses to these read-only data structures have to reach the accelerator memory.

To avoid the invalidation of accelerator-hosted on an accelerator calls, accelerators should provide a means for the CPU to identify those data that has been modified during the accelerator execution. The necessary hardware support and control structures to be implemented by accelerators to avoid accelerator-hosted data invalidation is a potential future work.

8.2.2 Accelerator Memory Manager

The experimental work in NUAMA and ADSM showed that large data transfer sizes are the key to accomplish an efficient utilization of the PCIe bandwidth. Currently, most applications for CPU – accelerator systems come from the high-performance computing and use large datasets. This data layout allows for medium and large size transfers between system and accelerator memories. However, many existent applications use dynamic data structures, such as linked-lists or binary trees, which are formed by large number of small memory chunks. The dynamic nature of these data structures results in a memory layout, where data from different structures is interleaved in memory. Such a memory layout prevents large data transfers between system and accelerator memory to happen.

An accelerator memory allocation that dynamically packs data structures into contiguous accelerator memory regions would allow to large data transfers between CPUs and accelerators. Such an allocator has to be able to dynamically identify data structures for each accelerator memory allocation, so a object cache per data structure can be used.

8.2.3 Accelerator Virtual Memory

The accelerator-hosted data transfer model increases the accelerator memory capacity requirements. Currently, accelerator memory capacity is increasing (i.e., 6GB on NVIDIA Fermi). However, some applications might use data structures larger than the available physical accelerator memory. There are software and hardware mechanisms to support such applications. For instance, host-mapped memory in NVIDIA GPUs might be used to increase the amount of accelerator memory at the cost of lower accelerator performance. The research of efficient accelerator virtual memory is a potential future work of this thesis.

8.2.4 Accelerator Scheduling and Operating System Integration

The ADSM execution model integrates accelerators into the execution thread abstraction. This integration allows operating system schedulers to consider the accelerator execution time into scheduling policies to improve scheduling fairness. Current operating system schedulers favor applications that are mostly executed by accelerators due to their short CPU execution time.

A potential future work is to integrate the ADSM execution model inside the operating system kernel code. This integration will allow to further investigate the necessary operating system support for heterogeneous systems. This potential future work presents the following challenges:

- Definition of an accelerator interface. Due to the variety of existent accelerators (e.g., NVIDIA, AMD and Intel GPUs), a generic interface for all accelerators is needed to keep a hardware-independent operating system kernel code.

- Scheduling metrics. Current operating system scheduler typically use the execution time as main metric to take scheduling decisions because all processes can be executed by all processors in the system. In the ADSM execution model, this assumption does not remain valid; some processes might be only executed by an accelerator depending on the execution mode. Accomplishing load-balancing between CPUs and accelerators might require using different metrics.

- Accelerator emulation. Applications using accelerator can be executed on systems without accelerators using a fall-back emulation mode. The trade-offs between operating system kernel-level and user-level accelerator need to be analyzed to develop reliable and efficient accelerator emulation.

8.2.5 Multi-Accelerator Programming

The ADSM execution model supports multi-accelerator systems by spawning new execution threads. This approach allows building higher-level abstractions

in the run-time to hide thread creation to application programmers. The analysis of the problems posed by accelerator memory distribution and the research of mechanisms to hide the complexity of using several accelerators is also potential future work.

Appendix A

Application Partitioning for Heterogeneous Systems

A.1 Introduction

This appendix presents a methodology for guided application partitioning for CPU – accelerator systems. This methodology is demonstrated using a prototype of the Non-Uniform Accelerator Memory Access (NUAMA) architecture. The key feature of NUAMA is that accelerator-accessible data structures are hosted by the accelerator memory, but are still accessible by the CPU. Together, the techniques described in this appendix enable a software developer to take a piece of software and map it to a heterogeneous multi-core system. Moreover, this methodology lays the groundwork necessary to enable a methodical and automated approach to application partitioning for NUAMA and ADSM CPU – accelerator systems. In the remaining of this chapter, a base NUAMA systems is assumed, but the concepts here developed are equally applicable to ADSM systems.

The first contribution of this work is a methodology which leverages both developer knowledge of an application and dynamic application profiling techniques to partition data applications for NUAMA systems. The second contribution is an emulation platform for rapid prototyping of NUAMA applications. The emulation platform provides enhanced visibility, control, and speed to the software developer, with the added capability of emulating NUAMA systems.

A.1.1 Related Work

A variety of approaches have been proposed to extract greater performance from the increasing number of transistors available to microprocessor designers. One example is homogeneous multi-core designs integrating multiple, identical cores on a single die [KAO05, Sea05]. These designs are adept at exploiting thread-level parallelism, but are not the most power-efficient computational substrate for data-parallel codes. Several efforts have explored heterogeneous systems as a means to gain better performance with greater power and area efficiency. The Cell processor integrates groups of smaller, in-order vector units with a high-performance super-scalar core [GHF+06]. To avoid the need for a heterogeneous programming model, other work has investigated the use cores of varying capabilities from the same Instruction Set Architecture (ISA) on a die [KTR+04].

Furthermore, heterogeneous multi-core processors are shown to provide favorable power/performance benefits to a wide array of applications [KFJ+03], but may still be limiting for irregular, data-parallel codes.

An alternative approach is to use accelerators that are specifically designed to exploit data parallelism with a high level of power and area efficiency. Specifications are already underway to incorporate fine-grained examples of such accelerators into commodity microprocessors [Ram06]. The goal of such systems is to complement high-performance processor cores by incorporating domain-specific functionality that is implemented as fast, closely-coupled logic (e.g., [SBB+05]). For a study of data parallelism and its effect on microarchitecture, see [SKMB03]. The end result is a heterogeneous mix of general-purpose and accelerator cores that can exploit the various forms of parallelism present in applications [Aea06].

The design of interconnect between the CPU and the accelerators is critical to the performance achievable by the system. Commercial examples of accelerator interconnects include system buses (e.g., HyperTransport and PCIe) and instructions provided by the ISA (e.g., MIPS Coprocessor Interface). System bus interfaces provide high-bandwidth, high-latency connections between system memory, CPU, and accelerators without imposing upon the ISA. On the other hand, commercially-available ISA extensions provide a low-latency access mechanism for accelerators, but with register granularity that requires entangling accelerator interfaces with the processor pipeline. Both models provide *pass-by-value* semantics whereby the data is explicitly delivered to the accelerator and the CPU does not keep a reference to the data. The NUAMA architecture avoids modifying the ISA of the general-purpose CPU and utilizes a *pass-by-reference* model where persistent data is shared between the CPU and accelerator.

A commercial example of a data-parallel accelerator and development environment is the NVIDIA G80 and its corresponding CUDA [NVI09] software development environment. CUDA is an environment for developing software that will run on the accelerator; However, it currently does not provide a means to determine an appropriate partitioning of applications across the CPU and accelerator as the proposed methodology achieves. The key difference between the CUDA model and NUAMA is that the CUDA provides only a pass-by-value model for the CPU to access its local memory, while NUAMA provides both pass-by-value and, as extensively used in this study, pass-by-reference semantics.

In order to effectively use the data hosting feature, the application and its data must be properly partitioned between the CPU and accelerators. Application partitioning has been widely studied in the field of design automation. The FLAT tool set [SNV+03] uses source code profiling and simulation to identify the compute-intensive loops of applications. In the field of application-specific instruction set processors (ASIP), the μP tool set [KAFK+05] uses fine-grained assembly-level profiling and simulation to identify instruction extensions

for general purpose cores. The development environment for Stretch [Gon06] is an example of using simulation and native execution to rapidly prototype and debug reconfigurable designs. While the Stretch tool flow allows for direct performance measurements, it requires possibly time-consuming synthesis and place-and-route steps to be performed before evaluating a design. Their model does not use a prototyping mechanism to reduce the time spent debugging large-scale accelerator designs as the proposed methodology aims to do via emulation.

A.1.2 Motivation

For heterogeneous NUAMA systems to become prevalent, there is a need of methodologies such as the one presented in this appendix that allow software designers, with as little added effort as possible, to take common applications and partition them across CPUs and accelerators. Such tools should not deviate from accepted development and debugging practices while also allowing the developed applications to remain portable. Partitioning techniques that require programmer intervention are explored, but time consuming tasks are deferred as much as possible to converge on a correct design more rapidly.

The NUAMA architecture and this methodology are designed to help software developers achieve three important objectives when utilizing data-parallel accelerators. First, data sharing should consume the least possible amount of interconnect bandwidth and incur shortest possible latencies. Second, the smallest possible number of changes should be made to the programming environment and processor architecture so that software can be easily ported between systems with and without accelerators. Third, the software developed using the tools should be easy to debug. The latter two objectives are achieved partly by providing an emulation platform that eases debugging the communication between the part of the application running on the general purpose core and that executed by the accelerator. It should be noted that the proposed methodology is not meant to evaluate the performance of a partitioned design, but instead has the orthogonal goal of developing a correct design.

The partitioning flow starts with a software application that runs on the general purpose processor. The application developer then uses this methodology to identify the parts of the application that are amenable to accelerator execution and select the data structures that are accessed frequently by those software components. The developer can then use our emulation platform to test, debug, and verify both software and hardware-instrumented versions of the application there by doing code generation. The emulation platform allows for the flow to be completed, after the hosted data mappings and synchronization mechanisms are in place and debugged.

Case studies are presented to demonstrate the use of this methodology and NUAMA to enable the efficient use of data-parallel accelerators. Three

SPEC2006 integer benchmarks are mapped to the emulation platform using analysis and profiling infrastructure: one to guide the development of this methodology and two others to verify its capabilities. This appendix illustrates that a software designer can take a software application and quickly prototype designs that incorporate possibly non-existent accelerators with high visibility for debugging purposes.

A.1.3 Contributions

The contributions of this appendix are as follows:

- A profiling methodology that assists software developers in mapping software applications and data objects into NUAMA architectures.
- A rapid prototyping environment that allows for the debug and test of NUAMA designs.
- An emulation technique that uses softcore processors to avoid the time consuming process of hand implementation, or high-level synthesis, of accelerators during partitioning.
- A design flow that starts with a purely software application and results in that application debugged, tested, and partitioned to run on a NUAMA architecture.

A.2 Design Flow

Here a design flow, which aids developers in partitioning applications on to NUAMA architectures is presented. Then a prototype emulation platform that supports rapid prototyping and debugging of alternative partitioning strategies is described. The prototyping platform supports a novel emulation technique that allows substantial testing and validation of partitioning prior to the availability of accelerator hardware implementations. The prototyping platform is used to emulate the NUMA architecture.

A.2.1 Analysis and profiling tools

The methodology presented here enables a software developer to identify the subroutines and data structures amenable to accelerator implementation and data structure hosting, respectively. This methodology approach arose from experiences partitioning a SPEC application, 462.libquantum. To demonstrate the developed methodology, this methodology is applied to two other benchmarks.

A high-level diagram of the design process appears in Figure A.1. The application (upper left corner) is first fed into a suite of profiling and analysis tools. The developer uses the analysis as a guide to selecting both functions

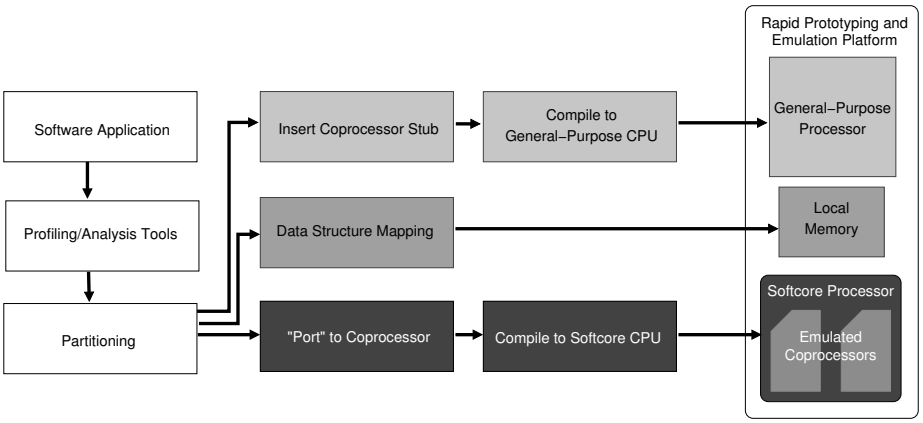


Figure A.1: Design Flow Overview

and data objects, which are then executed and hosted by the accelerator, respectively. The developer's choices are then used to partition the program, producing three results: stub code for controlling the interaction between the general-purpose processor and the accelerator; a data structure mapping that controls placement of objects into the accelerator memory as well as the associated allocation management; and an implementation of the accelerator functions suitable for emulation by a softcore processor. The tool flow (the block marked "Profiling/Analysis Tools" in Figure A.1) works as follows:

First, subroutine candidates are found by profiling the application and determining the data parallelism present. Data to be hosted by the accelerator memory is discovered by correlating the load/store intensity of dynamically allocated data structures with the execution periods of high computation time subroutines. The resolution of the tools can be modified to enable quicker analysis while searching for interesting regions of execution and slower, in-depth analysis of those regions found with coarser resolution. The visualizations produced provide a guide that enables developers to determine what subroutines are candidates for acceleration and what objects those subroutines will access for a given input set.

At a more detailed level, this methodology consists of the following steps:

1. **Preprocessing** to add annotations into the code at compile time to simplify and accelerate profile analysis stage.

2. **Profiling** isolates compute-intensive areas of the application (*candidate subroutines*). The `gprof` profiling tool is used for the experiments.

3. **Data-Level Parallelism (DLP) Discovery** evaluates the amenability to accelerator implementation of the candidate subroutine based on the amount of DLP present.

4. **Access Intensity** determines which program objects are heavily accessed by the candidate subroutines.

5. **Liveness Analysis** measures the persistence of the data structures and to see whether it is possible for multiple objects to time-multiplex the accelerator memory without large data transfers.

6. **Data Synchronization Granularity** maps the access pattern of the hosted data structures to one of the access models in Figure A.2.

Performance profiling of the application is a well-understood filter for selecting candidate subroutines for accelerator implementation. Due to Amdahl's Law, speedup will be limited by the time spent executing any sequential code. As such, those subroutines that dominate execution time are chosen with profiling. A threshold of 10% execution time is set before considering a subroutine for accelerator implementation. The developer may choose to evaluate all subroutines; however, setting such a limit reduces the number of subroutines that must be tracked, reducing the time required for analysis.

The amount of data-level parallelism available in candidate subroutines is how the appropriateness of implementing a candidate subroutine as a accelerator is evaluated. DLP is evaluated by counting the total number of arithmetic instructions and dividing it by the height of the dependence tree for the loop body code. Greater degrees of DLP can be exposed by exploiting cross-iteration parallelism for all of benchmarks.

The ability of NUAMA to provide demonstrable speedups depends heavily on discovering strong correlations between data structures and code regions that can be accelerated by accelerators. Dynamic data profilers to show how much and how often data is accessed from inside candidate subroutines have been developed. High *access intensity* is defined as a large number of loads and stores to a particular object in a time interval. High access intensity for a data object during a candidate routine marks it as an attractive choice for hosting in the accelerator memory. Correlating the dynamic data profiler results with the subroutine execution periods allows the software developer to make informed decisions about what data structures should be hosted while using different candidate subroutines.

Accurate selection of data structures for inclusion in the accelerator memory is necessary for successful accelerator execution. However, the ability to access hosted data objects with low latency from both the CPU and accelerator is exploited to allow for overly eager mappings to accelerator memory. The only guarantee that must be made is that all of the data that the *accelerator* accesses be present in its memory prior to its execution. Data locality is enforced explicitly at the point where the processor hands off control to the accelerator. Otherwise, since the accelerator memory is accessible and cacheable by the CPU, correct execution will be maintained without performance degradation, even if objects are unnecessarily hosted by the accelerator memory.

For NUAMA to remain scalable and general-purpose, the architecture requires the capability of mapping large accelerator working sets into smaller

accelerator memories (virtualization). Determining how much data is accessed from candidate subroutines allows the developer to gage the level of accelerator memory virtualization that must be done. For applications where the amount of data accessed is small enough, dynamic data profiling will show that the candidate objects will fit into the given accelerator memory and no virtualization is needed. In cases where the objects are larger than available memory, software management similar to the virtualization of system memory can be employed.

As applications progress through different phases of execution, different accelerator or data structure hosting arrangements may be appropriate. The proposed methodology provides a data object liveness analysis technique to aid developers in finding opportune periods to make such runtime changes. An object is said to be *dead* during the interval between a load and the first store that starts a period in which all values in that object are overwritten without an intervening load. Any dead objects can simply be discarded as soon as the last load prior to the dead interval, since it will be re-created prior to any future loads accessing its contents. Liveness is also a measure of an object's persistence in memory, with long lifetimes indicating possibly good candidates for data hosting compared to short-lived values.

Liveness analysis is useful for the situation in which a accelerator must iterate through data structures larger than the available accelerator memory, or when a single object is equivalent to a sequence of distinct objects. Furthermore, if the accelerators are reconfigurable (e.g., FPGA-like) or are programmable (e.g., SPE in the Cell processor), the opportunity exists for various accelerator functions to be utilized throughout the execution of the program. In either case, knowing that objects are dead during periods of execution allows for applications to simply discard the values in the accelerator memory and start using the accelerator memory to host another object. For objects that will exist later in the application, but are simply dead for a period of time, the application must change the address mapping for the object. The key feature of liveness profiling is its ability to uncover periods where objects need not be preserved, thus guiding the developer to parts of the code where accelerator reconfiguration and data structure remapping can take place efficiently.

The last aspect of analysis is the granularity of data flow in a candidate subroutine. Figure A.2 shows four possible modes. As part of this flow, a technique to analyze the data flow granularity and access behavior of our benchmarks has been developed. Figure A.2(a) shows the situation where all data control is transferred at the same point. Such a model requires less architecture support, but unnecessarily limits concurrency. When the computation of the subsequent block depends on a late write by the preceding block is called stack access pattern. A stack access pattern is limiting for accelerator concurrency and if such a pattern is found, accelerators may only be able to execute sequentially. Full streaming, as depicted in Figure A.2(b), is the situation where each element is transferred independently and in an order that is consistent across control

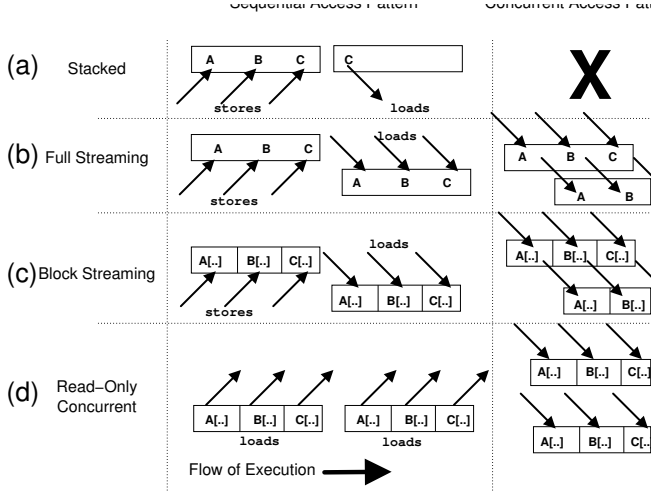


Figure A.2: Data Movement in Concurrent Data Access Models

paths. In such a model, it may be possible to execute multiple accelerators concurrently and synchronize with a mechanism similar to the presence bits used in dataflow machines [Vee86]. Figure A.2(c) shows block streaming where the access pattern between blocks is exploited, but within blocks applications may access data in an arbitrary order. Some applications have sequential code regions that only access objects in a read-only fashion, allowing for the concurrent execution shown in Figure A.2(d). Access patterns of applications are also studied to give direction to the developer about what computations may be able to stream data or be co-scheduled in order to increase parallelism.

The tools used as part of this methodology are developed using perl and PIN [LCM+05]. PIN tools perform the dynamic analysis of applications by instrumenting the binaries, providing high-speed access to dynamic program behavior. The resolution of the data collection is variable whenever possible to allow for faster feedback loops for developers. Using these tools, tens of millions of instructions per minute can be profiled at a fine resolution and, at coarser resolution, hundreds of millions of instructions per minute.

A.2.2 Emulation platform

A prototype emulation platform for NUAMA-based systems has been developed to allow *software versions of accelerators* to be rapidly prototyped and the corresponding software to be tested and debugged without requiring developers to adopt entirely new development methods. Greater visibility is provided by the emulation platform, resulting in reduced debugging times compared to other accelerators that rely on "black box" approaches, which do not expose their interfaces directly to the developer. By exposing the accelerator memory to the user application, the usage of standard compiler (e.g., `gcc`) and debugging tools (e.g., `gdb`) is enabled, reducing the complexity experienced by developers

wishing to migrate from conventional platforms.

The emulation platform allows a developer to run applications in any of four modes: *software-only profile*, *software memory debug*, *accelerator debug*, and *accelerator profiling*. Using these four modes of operation, the developer can take the results of the partition methodology and implement an emulated partitioning of the application. The emulated accelerator platform can then be used to evaluate trade offs in the partitioned application's design space and to debug applications with concurrent execution on general-purpose cores and accelerators.

The prototyping platform consists of an FPGA-based development board with embedded hard and soft processor cores that runs a fully functioning operating system and development environment. The Xilinx Virtex-II Pro FPGA [Xil05], which integrates a PowerPC (PPC) processor core on-die, is used in this work. The software for the platform consists of Linux 2.6.18 with a full suite of GNU development tools, libraries, and utilities. The software portion of applications to be prototyped are compiled for the PowerPC processor and linked against standard PPC Linux C libraries. To emulate accelerators, the Xilinx MicroBlaze softcore processor running a modified form of the original source code of the region to be accelerated is used. The restrictions on that code are the same as those required of the accelerators themselves. Furthermore, multiple accelerators can be emulated by synthesizing multiple softcore processors, or, if they do not have overlapping lifetimes, on the same softcore processor. The hosting of application data structures is done by embedded SRAM, accessible by both the MicroBlaze and PPC processors, that is mapped into the application's address space. The embedded PPC processor, softcore MicroBlaze processors, embedded memories, and system software represent a platform that allows developers to prototype applications targeting future NUAMA systems.

There are two modes on the emulation platform that do not use accelerators: the software-only mode and the software memory debug mode. The software-only mode is the original software application running on the emulation platform. Applications can be developed using conventional compilation and debugging tools under the software-only mode to ensure a stable base prior to partitioning into accelerator and general-purpose software modules. The software memory debug mode executes all of the code on the general-purpose processor, however, the data structures to be hosted by the accelerator memory are allocated to the embedded memories of the emulation platform. Placing the selected objects into the accelerator memory allows for initial debugging of the software/accelerator interface and an initial analysis of the caching and bus contention behavior of the final design.

The accelerator debug and profiling modes are used post-partitioning to debug and evaluate the partitioned designs, respectively. In the accelerator debug mode, emulated accelerators are used that are functionally identical to the final accelerator design, however, they run as a software module on a softcore

processor. The accelerator debug mode allows the developer to remove bugs in the synchronization and data partitioning between the CPU and accelerator. For applications that have many objects needing to be hosted and for virtualization of the accelerator memory, ensuring that the accelerator local memory contains the correct set of values is critical for correct execution. Having the ability to verify the partitioning scheme using emulation as opposed to simulation can reduce debugging time by two orders of magnitude, as the results show (see Table A.2). Furthermore, the emulation platform provides a high degree of visibility and control for the developer to alter and examine the state of the executing application using symbolic debuggers and external interfaces via JTAG.

The accelerator profile mode is the final step in the design flow, removing the emulated accelerator from the hardware interface and inserting the actual accelerator. Doing so enables the developer to evaluate the performance of the accelerated and *debugged* design. Using these two modes, the developer can remove bugs and evaluate performance in that order allowing for rapid debugging of the software/hardware interface followed by performance evaluation.

A.3 Case Studies

The application which drove the methodology development, along with the tools themselves, is presented followed by two case studies of application partitioning. Examples are drawn from the SPECint2006 suite in three application domains characterized by computation amenable to accelerator acceleration: mathematical libraries and simulation (462.libquantum), scientific computing (456.hmmer), and multimedia (464.h264ref). Finally, the utility of a rapid prototyping platform that allows software developers to target future hybrid NUAMA architectures with their applications is demonstrated.

A.3.1 Design driver: 462.libquantum

The driving application used to develop the partitioning infrastructure is the 462.libquantum integer benchmark. The benchmark was chosen for its small code size and easily discoverable regions with data-level parallelism (DLP), making it an easy testbed for the analysis techniques. The benchmark simulates a quantum computer running Shor's algorithm for integer factorization relying heavily upon the libquantum library. On the emulation platform, two library subroutines are migrated and their associated data into emulated accelerators. The benchmark spends over 3/4 of the original uniprocessor runtime in these two functions. Profile tools have enabled to determine the data structures appropriate for accelerator memory hosting. The application is instrumented to work for the different platform computation modes. A discussion about how debugging proceeds in the model, resulting in a semantically correct version of

Benchmark	Total Instrs.	ALU Instrs.	Loop-body DLP	Cross-iteration Mechanism
libquantum	8	3	3	Loop Unrolling
hmmer	84	22	40	Loop Skewing
h264ref	223	62	90	Streaming

Table A.1: Data-level parallelism present in loop bodies and mechanisms for exploiting the cross-iteration parallelism

the partitioned application.

After applying the proposed design flow, the developer has a set of subroutines amenable to accelerator implementation, the data objects to be hosted by those accelerators, knowledge of the persistence and periods of liveness of those data objects. The current implementation of the tools only provides information, leaving all code modification and final inspection to the developer. Greater automation of partitioning is desirable, but due to its inherent complications, leaving automation to future work. The process followed to develop the partition methodology is present using 462.libquantum as an illustrative example.

Preprocessing: A set of scripts that add annotations to the source to simplify and expedite the profiling and analysis process. For example, this step adds tags indicating to the tools the names of dynamically allocated objects. Static analysis and correctness checks in this stage could provide feedback and semantic information for the subsequent steps, but are not investigated here.

Profiling: Application profiling is used to determine candidate subroutines based on their contribution to the overall computation time. For 462.libquantum it is found that 52% and 26% of the computation time is spent in the subroutines `quantum_toffoli()` and `quantum_sigma_x()`, respectively, making them good targets for acceleration.

DLP Discovery: The chosen SPEC benchmarks are tuned for sequential execution and thus require some code transformations to expose cross-iteration loop parallelism. Table A.1 shows the DLP the method found within loop bodies. There is not an explicit technique for choosing a method for exposing more parallelism, but after inspection of the DLP regions found during DLP discovery, simple transformations could expose more DLP and are shown in the table. The DLP present in loops of candidate subroutines is calculated by comparing the store set of each iteration with the load set of subsequent iterations. If there is a cross-iteration loop read-after-write dependence, the loop is marked as not data-parallel and therefore not amenable to accelerator implementation.

Access Intensity: the proposed flow provides the ability to visualize the correlation between different data objects and candidate subroutines. Figure A.3 shows three candidate subroutines for the libquantum benchmark and two candidate data objects, `reg->node` and `reg`. The width of the data struc-

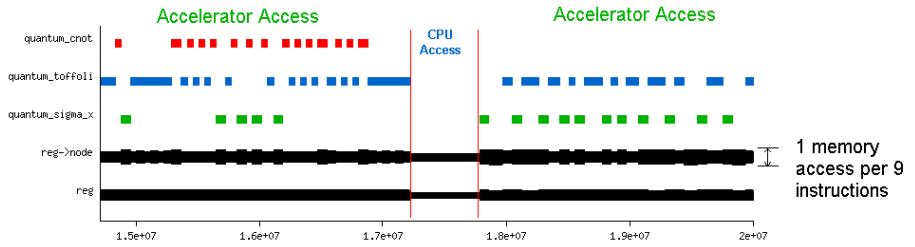


Figure A.3: Memory access intensity for 462.libquantum. The top three lines indicate which function is executing at each point in time.

ture line is proportional to the number of memory accesses performed in a time interval, or the access intensity. During the interval depicted, as is true for much of the application, the access intensity of these objects is far greater than of any other objects, and in fact the three functions shown do not access any other data objects. The peak access intensity is one in nine instructions being either a load or a store to the given object. Furthermore, even in the regions between the execution of candidate subroutines, where the CPU would be accessing the data, moderate access intensity for candidate data objects is found. NUAMA exploits this access pattern by providing unequal sharing of the structures hosted in the accelerator memory, optimizing for the more frequent accelerator data accesses, but allowing cacheable CPU access while the accelerator is not executing.

Liveness Analysis: The Access Intensity stage shows there to be a high correlation between accesses to certain data objects and the candidate subroutines. Being able to measure the lifetime of these objects provides information necessary for the developer to decide whether a pass-by-reference or a pass-by-value model is appropriate. Our liveness technique, a scriptable combination of profiling and instrumentation, demonstrates that for 462.libquantum, the candidate objects are persistent, i.e., they stay live for the duration of the application's execution, and are therefore amenable to hosting throughout the application's execution. Furthermore, since the accelerator memory is limited in size and hosted data is not backed by system memory, liveness analysis can be used to identify data that need not be copied back to memory upon remapping of the accelerator memory.

Data Synchronization Granularity: the required granularity of data synchronization for 462.libquantum has been investigated. Having coarse-grained data synchronization allows for a reduced complexity architecture and programming model where the transfer of data object control between processing elements occurs in whole-object transactions as shown in Figure A.2(a). However simple, such a coarse-grained model may result in limited concurrency since one operation must completely relinquish control of an object before the next operation can access it. A more fine-grained approach would result in added concurrency when the results of one operation are forwarded to the next (Fig-

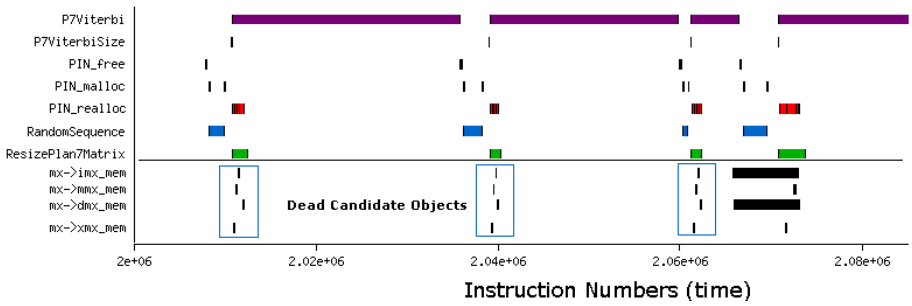


Figure A.4: Liveness results for 456.hmmer with horizontal bars for data indicating dead regions. (Note: Due to the resolution of the image, function invocations appear to overlap.)

ure A.2(b)), essentially pipelining multiple operations inside the accelerator. However, the added synchronization may result in unjustifiable overhead compared to a coarse-grained mechanism.

Visual inspection of 462.libquantum found that accelerator operations could be overlapped. The analysis using the proposed methodology has shown this to be the often the case with the data produced by `quantum_sigma_x()` available to `quantum_toffoli()` when the calls are adjacent, allowing for a pipelined implementation. However, due to limitations of the prototype emulation platform, concurrent emulated accelerators can not be demonstrated.

Using the approach developed in the design driver, case studies of benchmarks evaluated using the proposed infrastructure are presented.

A.3.2 Case study: 456.hmmer

The 456.hmmer benchmark is a DNA sequencing application that uses Hidden Markov Models to perform DNA sequence alignments. At the thread level, the benchmark can be partitioned such that independent threads each process independent blocks of data from an assigned work pool. Within each thread, the benchmark's core contains exploitable DLP. The proposed methodology and tools are applied to uncover the DLP, map it into NUAMA, and prototype it in the emulation platform.

Profiling indicates that 456.hmmer spends between 71% and 97% of its execution time in the `P7Viterbi()` subroutine. The analysis indicates that `P7Viterbi()`'s inner loop accesses four data objects heavily, all of which are arrays: `mx->imx_mem`, `mx->mmx_mem`, `mx->dmx_mem`, and `mx->xmx_mem`. Liveness analysis is provided in Figure A.4, demonstrating that these data objects all become dead early in each invocation of `P7Viterbi()`, and some of the objects are occasionally dead between invocations as well. This information led us to further inspect the code and to notice that data-structure resizing occurs in `ResizePlan7Matrix()`, called from `P7Viterbi()` before executing the compute-intense code portions. We discovered that a call to the C library function

`realloc()` is the source of loads from the objects that extend the liveness intervals beyond `P7Viterbi()`. Thus, between invocations to `P7Viterbi()`, the objects contain no useful state, indicating that there is no need to transfer the arrays into or out of the accelerator between invocations. Additionally, the deadness of the arrays indicates that accelerator context switching overhead is minimal between invocations.

456.hmmer has been implemented on the prototype emulation platform, hosting the four candidate data objects in the accelerator memory and the inner loops of the `P7Viterbi()` emulated on the softcore processor. The proposed infrastructure has allowed to arrive at this partitioning and the emulation platform has allowed to rapidly prototype the design without having a physical accelerator implemented.

A.3.3 Case study: 464.h264ref

The 464.h264ref benchmark is a multimedia application that encodes raw video into a compressed form using the H.264 specification. The benchmark is a streaming application which applies various transformations to blocks of data as they are read from the input video and are incorporated into the encoded output. One stage in the encoding process using the proposed methodology is isolated and implement it as an emulated accelerator.

Profiling indicates that two subroutines, `SetupFastFullPelSearch()` and one of its children `SATD()`, account for roughly 45% of 464.h264ref's execution time. `SATD()` computes the sum of absolute differences (SAD) over an array passed to it by `SetupFastFullPelSearch()`. The input array represents the current block (`SAD_block`), composed by pulling pointers from an array representing the current video frame.

The DLP tools were able to discover that `SATD()` invocations are independent and that the SAD computation has a high level of instruction-level parallelism. Thus `SATD()` is amenable to accelerator acceleration. Our liveness analysis showed that `SAD_block` is live upon `SATD()` invocation, but dead when `SATD()` returns, indicating that the function returns its computational result only via its return value. Therefore `SATD()` does not modify its input, and the accelerator need not transfer `SAD_block` back to the CPU after execution. Thus, the SAD kernel is mapped into an emulated accelerator, hosting the current `SAD_block` in the accelerator memory.

The `SATD()` subroutine has a simple if-then-else structure that would require added resources or added complexity if implemented as a accelerator. Profiling indicated that for the input sets, the branch is highly biased and it is possible to implement a data-parallel emulated accelerator excluding the uncommon control path. Accounting for all possible execution paths involves a trade-off between a more complicated accelerator and the overhead of the exception handling mode.

A.3.4 Emulation platform evaluation

The accelerator emulation platform has been developed to enable software developers to rapidly prototype applications partitioned using the analysis infrastructure. As an example of its use, two kernels of code from the 462.libquantum benchmark have been isolated and executed on a softcore processor. The emulated accelerator runs independently of the CPU, with synchronization performed via memory-mapped registers and explicit cache flushes. Two of the data structures isolated using the proposed methodology were mapped into the accelerator of the emulated accelerator, with that memory being accessible to both the CPU and the accelerator. The only changes that were made to the application were to allocate the memory used by the mapped objects to the accelerator memory and to perform synchronization at the call site of the now accelerated subroutine. Following this same flow, 456.hmmer and 464.h264ref have been additionally implemented on the emulation platform based on the partitioning found using the proposed methodology.

A strong motivation for using softcore accelerator emulation is that the platform executes the partitioned application much closer to its original speed than does a software-only system simulator. Table A.2 demonstrates the time required to run the SPEC test input sets for the benchmarks. The first row of Table A.2, labeled 'Native Execution', is the time required to run on a 3.2 GHz Intel Pentium 4, representing a contemporary system. Note that this result does not include the instrumentation present and represents best case performance for the machine. The next row compares the execution time of the platform running the code solely in software on the embedded hard processor of the platform. The next two rows provide the results of using emulated accelerators with the benchmark mostly running on the embedded hard processor and the accelerator running as software on a softcore processor. For each of these, emulated accelerators with their hosted data structures cached and not cached are shown. The last row of the table shows the time to simulate the benchmarks using a cycle-accurate simulator.

The instrumentation occurs on the native platform, providing the partitioning information quickly. The emulated accelerator prototype platform provides a means of developing, debugging, and evaluating a partitioned design. Non-cached local store access simplifies debugging, since it allows the developer to stop execution and view a coherent global state, including local memories, system memory, the register state of the general-purpose processor, utilizing the pre-existing software tools for the FPGA platform. Having this degree of visibility while suffering less than a two order of magnitude slowdown in performance allows for faster, easier development of partitioned applications. To more realistically evaluate a partitioned design, caching can be enabled. Once a partitioned design is debugged, performance evaluation of a *correct* design can be carried out on a simulator.

	462.libquantum		456.hmmer		464.h264ref	
Native Execution	1x	(0.30s)	1x	(0.10s)	1x	(1m13s)
Emulation Platform w/o Accel	40x	(12.1s)	43x	(4.33s)	26x	(32m11s)
Emulation w/Accel+NoCache	60x	(18.1s)	71x	(7.10)	30x	(36m23s)
Emulation w/Accel+Cache	56x	(16.7s)	73x	(7.33)	30x	(36m26s)
Simulation w/o Accel	2437x	(12m11s)	1180x	(1m58s)	3151x	(3833m50s)

Table A.2: Slowdown for alternate execution modes with example applications.

While one to two orders of magnitude slower than native execution, the emulation platform allows for incremental mapping of applications into a prototype of NUAMA, without suffering the three to four orders of magnitude slowdown experienced when using cycle-accurate simulation. The speed of the emulation platform allows developers to partition their applications using the proposed methodology and to debug them effectively without suffering the high turnaround time associated with simulation or full-blown accelerator generation. When the flexibility of a simulation platform is needed, the developer can move his now-debugged design onto the simulator for evaluation. Future research could leverage performance characteristics tracked using the cycle-accurate simulator to better model NUAMA on the emulation platform, providing the developer with both the speed of emulation and the accuracy and flexibility of the simulator.

A.4 Conclusions

The main contribution of this work is a methodology for mapping applications into heterogeneous CPU – accelerator architectures such as NUAMA. Examples of general-purpose applications that were mapped into a NUAMA emulation platform have been provided. The emulation platform allows for software designers to partition their software applications across accelerators and general-purpose processor domains. Using the presented design flow and tools, developers can rapidly prototype software that targets heterogeneous systems incorporating general-purpose processors and application-specific accelerators.

References

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '95*, pages 2–13, New York, NY, USA, 1995. ACM.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Transactions on Computers*, 19(8):26–34, Aug. 1986.

[ADADB+03] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. *SIGOPS Oper. Syst. Rev.*, 37(5):90–105, 2003.

[Aea06] Krste Asanovic and et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[AJR+03] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208, 2003.

[AMD06] AMD Staff. *AMD64 Architecture Programmer's Manual*. AMD Corporation, September 2006.

[ATI06] ATI Staff. *ATI CTM Guide*, 2006.

[Bal90] Henri E. Bal. Orca: a language for distributed programming. *SIGPLAN Not.*, 25(5):17–24, 1990.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09*, pages 29–44, New York, NY, USA, 2009. ACM.

[BDH+08] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Supercomputing '08*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[Ber05] A. Bergmann. The Cell Processor Programming Model. *Linux-Tag*, June 2005.

[BF88] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, Aug 1988.

[BL85] Amnon Barak and Ami Litman. Mos: a multicomputer distributed operating system. *Softw. Pract. Exper.*, 15(8):725–737, 1985.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Supercomputing '06*, page 86, New York, NY, USA, 2006. ACM.

[BR90] Roberto Bisiani and Mosur Ravishankar. Plus: a distributed shared-memory system. *SIGARCH Comput. Archit. News*, 18(3a):115–124, 1990.

[Bre96] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336, 1996.

[BT88] H.E. Bal and A.S. Tanenbaum. Distributed programming with shared data. In *ICCL '88*, pages 82–91, Oct 1988.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Compcon Spring '93*, pages 528–537, Feb 1993.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *SOSP '91*, pages 152–164, New York, NY, USA, 1991. ACM.

[CH00] Ben-Chung Cheng and Wenmei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00*, pages 57–69, New York, NY, USA, 2000. ACM.

[DFH+93] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[DKK09] Gregory Diamos, Andrew Kerr, and Muil Kesavan. Translating GPU binaries to tiered SIMD architectures with ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technilogy, 2009.

[DLAR91] P. Dasgupta, Jr. LeBlanc, R.J., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Transactions on Computers*, 24(11):34–44, Nov 1991.

[DO99] Fred Douglis and John Ousterhout. Transparent process migration: design alternatives and the sprite implementation. pages 56–86, 1999.

[DSF88] G. Delp, A. Sethi, and D. Farber. An analysis of memnet—an experiment in high-speed shared-memory local networking. In *SIGCOMM '88*, pages 165–174, New York, NY, USA, 1988. ACM.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95*, pages 251–266, New York, NY, USA, 1995. ACM.

[EPP+01] Rolf Enzler, Marco Platzer, Chistian Plessl, Lothar Thiele, and Gerhard Troester. Reconfigurable processors for handhelds and wearables: Application analysis. In *Reconfigurable Technology*, pages 135–146, Denver, CO, USA, August 2001.

[FADJ+05] M.R. Fahey, SR Alam, T.H. Dunigan Jr, J.S. Vetter, and P.H. Worley. Early Evaluation of the Cray XD1. *Cray User Group Conference*, 2005.

[FBR93] S. Frank, III Burkhardt, H., and J. Rothnie. The KSR 1: bridging the gap between shared memory and MPPs. In *Compcon Spring '93*, pages 285–294, Feb 1993.

[FP89] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. In *SOSP '89*, pages 211–223, New York, NY, USA, 1989. ACM.

[GBC+05] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM J. Res. Dev.*, 49(2):195–212, 2005.

[GBN04] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *ACM SIGPLAN Notices*, 39(7):249–256, 2004.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *FPGA*, pages 162–170, New York, NY, USA, 2004. ACM Press.

[Gon06] Ricardo E. Gonzalez. A software-configurable processor architecture. *IEEE Micro*, 26(5):42–51, 2006.

[Gus92] Davib B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, 1992.

[HFHK04] Scott Hauck Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera reconfigurable functional unit. *IEEE Transactions on VLSI*, 12(2):206–217, Feb. 2004.

[HH97] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: combining emulation and binary translation. *Digital Tech. J.*, 9(1):3–12, 1997.

[HKO+94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswider Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *ASPLOS '94*, pages 274–285, New York, NY, USA, 1994. ACM.

[HS09] Wenmei W. Hwu and John Stone. A programmers view of the new GPU computing capabilities in the Fermi architecture and cuda 3.0. White paper, University of Illinois, 2009.

[HW97] John R. Hauser and John Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM '97*, pages 12–21, Apr 1997.

[IBM07] IBM Staff. *SPE Runtime Management Library*, 2007.

[iee09] Information technology - portable operating system interface (posix) operating system interface (posix). *ISO/IEC/IEEE 9945 (First edition 2009-09-15)*, pages c1 –3830, 15 2009.

[IMP] IMPACT Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[Int05] Intel Staff. *Intel 945G Express Chipset Product Brief*, 2005.

[Int07] Intel Staff. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. Intel, May 2007.

[JC99] Jeffrey A. Jacob and Paul Chow. Memory interfacing and instruction specification for reconfigurable processors. In *FPGA*, pages 145–154, New York, NY, USA, 1999.

[JKW95] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: high-performance all-software distributed shared memory. In *SOSP '95*, pages 213–226, New York, NY, USA, 1995. ACM.

[JVG+09] Víctor Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[KAFK+05] K. Karuri, MA Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained application source code profiling for ASIP design. In *DAC 24*, pages 329–334, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[KCPT95] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *ICS*, pages 255–264, New York, NY, USA, 1995. ACM Press.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the CELL multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[KFJ+03] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. *Microarchitecture, IEEE/ACM International Symposium on*, 0:81, 2003.

[KJJ+09] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09*, pages 140–151, New York, NY, USA, 2009. ACM.

[KL08] John H. Kelm and Steven S. Lumetta. Hybridos: runtime support for reconfigurable accelerators. In *FPGA '08*, pages 212–221, New York, NY, USA, 2008. ACM.

[KMK01] D. Kim, R. Managuli, and Y. Kim. Data cache and direct memory access in programming mediaprocessors. *IEEE Micro*, 21(4):33–42, 2001.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Normal P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Hanapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann, Newton, MA, USA, 1984.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *ISCA '90*, pages 148–159, New York, NY, USA, 1990. ACM.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March-April 2008.

[MB06] Peter Messmer and Ralph Bodemer. Accelerating scientific applications using FPGAs. *XCell*, 2006.

[MDP+00] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[MGN08] Julio Merino, Isaac Gelado, and Nacho Navarro. Evaluation of the Cell BE SPU scheduling for multi-programmed systems. In *WIOSCA'08*, July 2008.

[MIP01] MIPS Staff. *MIPS32 Architecture for Programmers*. MIPS Technologies, March 2001.

[Mun09] Aaftab Munshi. *The OpenCL Specification*, 2009.

[MW90] C. Maples and L. Wittie. Merlin: A superglue for multicomputer systems. In *Compcon Spring '90*, volume 90, pages 73–81, 1990.

[MZDG93] Dejan S. Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese. Task migration on the top of the mach microkernel. In *USENIX MACH III Symposium*, pages 273–290, Berkeley, CA, USA, 1993. USENIX Association.

[NVI09] NVIDIA Staff. *NVIDIA CUDA Programming Guide 2.3*, 2009.

[PH08] Sanjay Patel and WenMei W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4–12, July-Aug. 2008.

[RAA+88] M. Rozier, V. Abrossimov, F. Arm, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, et al. Overview of the Chorus distributed operating system. *Computing Systems*, 1988.

[Ram06] R.M. Ramanathan. Extending the world's most popular processor architecture. Whitepaper, Intel, September 2006.

[RFT+10] Jose Renau, Basilio Fragela, James Tuck, Wei Liu, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator. http://sesc.sourceforge.net, May 2010.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wenmei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08*, pages 73–82, New York, NY, USA, 2008. ACM.

[SBB+05] Valentina Salapura, Randy Bickford, Matthias Blumrich, Arthur A. Bright, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Michael Gschwind, Manish Gupta, Shawn Hall, Rudd A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Martin Ohmacht, Rick A. Rand, Todd Takken, and Pavlos Vranas. Power and performance optimization at the system level. In *CF '05*, pages 125–132, New York, NY, USA, 2005. ACM.

[SCS+08] Larray Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[Sea00] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Pusblishing Co., Inc., Boston, MA, USA, 2000.

[Sea05] B. Sinharoy and et al. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[SGI08] SGI Staff. *Reconfigurable Application-Specific Computing User's Guide*, 2008.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS-VII*, pages 174–185, New York, NY, USA, 1996. ACM.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. pages 1–10, 2009.

[SKMB03] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. Universal mechanisms for data-parallel architectures. In *MICRO 36*, page 303, Washington, DC, USA, 2003. IEEE Computer Society.

[SLL+00] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[SNV+03] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03*, pages 189–198, New York, NY, USA, 2003. ACM.

[SZC+09] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI '09*, pages 431–440, New York, NY, USA, 2009. ACM.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, 1990.

[Van02] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

[VWG+04] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[WH88] D.H.D. Warren and S. Haridi. Data Diffusion Machine-a scalable shared virtual memory multiprocessor. In *Fifth Generation Computer Systems 1988*, page 943. Springer-Verlag, 1988.

[WK01] Grant Wigley and David Kearney. The first real operating system for reconfigurable computers. *Aust. Comput. Sci. Commun.*, 23(4):130–137, 2001.

[WLT93] Jr. Wilson, A.W., Jr. LaRowe, R.P., and M.J. Teller. Hardware assist for distributed shared memory. In *DCS '03*, pages 246–255, May 1993.

[WP03] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *ERSA '03*, pages 284–287. CSREA Press, 2003.

[Xil05] Xilinx. *Virtex-II Pro and Virtex-II Pro X Plaform FPGAs: Complete Data Sheet*, October 2005.

[Xil09] Xilinx Staff. *Virtex-5 Family Overview*, Feb 2009.

[ZSM90] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *DCS '90*, pages 30–37, May 1990.