universität wien

MASTERARBEIT

Titel der Masterarbeit

"High-level Parallel Programming Support for Heterogeneous Systems"

Verfasser

Bo Wu, Bakk.techn.

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 066 940
Studienrichtung lt. Studienblatt:	Scientific Computing
Betreuer:	Ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2013 Bo Wu

Abstract

This master thesis focuses on several high-level parallel programming models for heterogeneous systems that have been becoming increasingly popular in the field of high-performance computing. Heterogeneous systems are an inexpensive and effective way for further performance improvements. A powerful combination of graphics processing units (GPUs) and central processing units (CPUs) is one of the most common types of heterogeneous systems. Currently, three of the world's ten fastest supercomputers are based on heterogeneous designs. The preponderance of using these systems is not only a marked increase in performance, but also more energy-efficient and cost-effective.

For software developers, programming on such platforms is a new challenge because of the more complex architecture. Despite two recently developed programming models, CUDA and OpenCL, programming of heterogeneous architectures is still much more difficult than programming of homogeneous architectures. Both of them are very low-level approaches. Programmers need to have in-depth knowledge of the hardware characteristics. Other challenges are the localization of parallelism, as well as the complex memory model and execution model. A further problem is that existing programs developed for homogeneous multicore processors cannot be easily ported to heterogeneous systems, and therefore, a rewriting of such programs is required. Hence, high-level programming approaches are of great importance to mitigate the problems of low-level programming.

The purpose of the thesis is to outline the current development of high-level programming models for heterogeneous systems. The thesis explores different representative high-level programming models. By analyzing the characteristics of such approaches for applications in computational science, programmability, performance and portability are evaluated, compared and discussed. Different scientific codes have been implemented with PGI Accelerator, HMPP Workbench and OpenMPC, and experiments have been performed as well.

Zusammenfassung

Diese Masterarbeit behandelt unterschiedliche high-level parallele Programmiermodelle für heterogene Systeme, die mittlerweile sehr verbreitet im Bereich High Performance Computing sind. Heterogene Systeme sind eine sehr kostengünstige und effektive Art weitere Performanzsteigerungen zu erzielen. Der häufigste Ansatz ist, leistungsfähige Grafikprozessoren (GPUs) als Acceleratoren zu den Hauptprozessoren (CPUs) hinzuzufügen. Derzeit sind drei der topzehn Supercomputer der Welt heterogene Systeme. Wichtig für die Verwendung von solchen Systemen ist nicht nur die Performanz allein, sondern auch der niedrigere Energieverbrauch und der günstigere Preis.

Für Softwareentwickler stellen solche Plattformen eine neue Herausforderung dar, da es sich um eine komplexere Architektur handelt. Die Programmierung auf einer heterogenen Architektur ist viel schwieriger als auf einer homogenen Architektur, trotz der beiden jüngst entwickelten Programmierungsmodelle CUDA und OpenCL. Beide sind sehr low-level Ansätze. Die Programmierer müssen die genaue Architektur auf Hardware-Ebene kennen. Andere Herausforderungen sind auch die Lokalisierung der Parallelität, sowie das komplexe Speichermodell und Ausführungsmodell. Ein weiteres Problem ist, dass bereits existierende Programme für homogene Multicore-Prozessoren nicht einfach auf diese heterogenen Systeme portiert werden können, und daher ein Umschreiben solcher Programme notwendig ist. Daher sind high-level Programmieransätze von großer Wichtigkeit, um die Probleme der low-level Programmierung zu entschärfen.

Das Ziel dieser Masterarbeit ist es, einen Überblick über die aktuelle Entwicklung von high-level Programmiermodelle für heterogene Systeme zu geben. Diese Arbeit untersucht unterschiedliche repräsentative high-level Programmiermodelle. Durch die Analyse der Eigenschaften von solchen Ansätzen für Anwendungen aus Computational Science, werden die jeweilige Programmierbarkeit, Performanz und Portabilität evaluiert, verglichen und diskutiert. Dazu werden unterschiedliche Scientific Codes mit PGI Accelerator, HMPP Workbench und OpenMPC realisiert, und Experimente werden auch durchgeführt.

Contents

CHAPTER 1

Introduction

Through the popularization of personal computers, game consoles and mobile devices, heterogeneous systems are growing rapidly. The combination of CPU and GPU can be found in almost every modern computer system. This common concept also earns hardware and software developers' interest because of their highly parallel structure and inherent superiority in massively data parallel computation capabilities. After the first heterogeneous supercomputer Roadrunner [Barker et al., 2008], which contains two different types of processors: IBM PowerXCell 8i processors and AMD Opteron cores. Systems using accelerators or co-processors have been remarkably increasing over the past two years.

According to the latest edition (November 2012) of the Top 500 [Top 500, 2013] world's most powerful supercomputers, 12.4% of them are using accelerators or co-processors in their systems. A year ago, only 7.8% systems used this combination of different processors. The actual world's fastest supercomputer Titan, which has achieved a performance of 17.59 Petaflop/s, has a total of 560640 processors, including 261632 NVIDIA Tesla K20x GPU accelerator cores [Top 500, 2013]. Another interesting system in the top 10 is the Stampede supercomputer. It uses the new Intel Xeon Phi (Knights Corner) processors as co-processors, and this system is the first supercomputer based on the Intel Xeon Phi processors. Currently, as can be seen from Table 1.1, there are three accelerator-based systems on the Top 10 list. Heterogeneous systems are becoming a relatively new rising category in the field of HPC.

Rank	Supercomputer	Accelerator/Co-processor	Peak performance	Power
1	Titan	NVIDIA Tesla K20x	17.590 Petaflop/s	8209 KW
7	Stampede	Intel Xeon Phi	2.660 Petaflop/s	-
8	Tianhe-1A	NVIDIA Tesla C2050	2.566 Petaflop/s	4040 KW

Table 1.1: Accelerator-based systems in Top 10 supercomputers, November 2012

According to [Kogge et al., 2008], the first Exascale (10^{18} floating-point operations per second) supercomputer is expected between 2018 and 2020. This supercomputer would be a

GPU-based supercomputer or GPU-like system [Vuduc and Czechowski, 2011], to be more precise, an accelerator-based supercomputer. The key factor of using accelerators is, current GPUs deliver more peak performance and bandwidth in comparison with current CPUs. Another important factor for assembling GPUs is that they are relatively low-cost and energy-efficient. This performance benefit will be a great help to many areas of scientific computing, for instance, climate, energy, materials, etc.

The main problem associated with the use of heterogeneous systems is the difficulty of their programming support. It is too hard to program on heterogeneous architecture. Many existing common parallel programming models, for example, OpenMP or Cilk, which are designed for traditional homogeneous multicore processors, do not support heterogeneous hardware systems. Typically, developers are required to learn a new programming language or a new library to write applications for heterogeneous systems. Meanwhile, unfortunately, these applications are mostly hardware-independent, thus, developers must also have an extra knowledge about this new hardware architecture.

Although the basic programming support, such as CUDA or OpenCL, provides improved programmability, writing CUDA or OpenCL code is still complicated, time-consuming and error-prone. For achieving good performance, programmers need to carefully control all of the low-level details of data transfers, and thread management and synchronization. In addition, optimizing these applications requires a great knowledge of the low-level programming application-programming interfaces (APIs) and the details of hardware architecture. The high-level programming support would help developers to reduce the burden on them. Its high-level abstraction allows developers to take full advantage of the heterogeneous systems. In the field of high-performance computing (HPC), developers have also noticed the benefits of high-level programming support. Recently, a few high-level programming models, including PGI Accelerator and HMPP Workbench, have been already installed in the Titan supercomputer [OLCF, 2013]. These two approaches are also the main research objects in this thesis.

The intention of this thesis is to outline the current development of high-level programming support for heterogeneous systems. A comparative study between several high-level approaches has been made in this work. Three representative programming models, PGI Accelerator, HMPP Workbench and OpenMPC source-to-source compiler have been evaluated by two compute-intensive application kernels. The results of the experiments show that there is a performance gap between these high-level implicit approaches and low-level explicit CUDA, but in fact, in some cases, they can achieve similar or comparable performance to CUDA. Unlike the low-level basic programming support, these three high-level approaches are much more user-friendly, and they can provide high productivity. Through step-wise experiments, this study also shows the strategy to improve the performance results with these software tools.

Thesis organization

The thesis is divided into seven parts. Chapter 2 presents detailed hardware architecture of these systems and the relevant background information of the basic programming support. Chapter 3 introduces several different representative high-level programming approaches for heterogeneous systems: PGI Accelerator, HMPP Workbench, OpenACC programming standard and

OpenMPC source-to-source compiler. Chapter 4 describes two experimental application kernels and the methodology of the experiments. Chapter 5 describes the implementation of two experiments, which are realized by using three high-level programming approaches. In addition, the performance results are also discussed in this Chapter. Chapter 6 introduces six additional high-level programming models that relate to Chapter 3. Chapter 7 concludes the thesis by summarizing the experimental results.

CHAPTER 2

Heterogeneous systems and basic programming support

This chapter introduces different types of heterogeneous systems, and it focuses on the most common GPU-based heterogeneous systems. Four case studies in two categories are proposed for the better understanding of the hardware architecture. In the end, the current basic parallel programming models, such as CUDA and OpenCL, are discussed as well.

2.1 Heterogeneous systems

Heterogeneous systems, which combine different types of processors, are popular systems today. Generally speaking, beside a homogeneous general-purpose processor, there is at least a special-purpose processor in a heterogeneous system. GPU, Digital Signal Processor (DSP), Field Programmable Gate Array (FPGA), Intel Xeon Phi processor and Cell processor are common examples that are often used today as hardware accelerators or co-processors for improving performance. However, heterogeneous systems are bringing new challenges to software developers because of the unusual, sometimes even domain-specific architecture. This thesis targets mainly GPU-based heterogeneous systems.

Nowadays, a combination of CPU and GPU can be found in every computer system. These powerful CPU-GPU hybrid systems obtain an optimum balance between the computational performance and generating graphics output. In fact, GPU is relatively young in comparison to CPU. In early computer systems, CPU performs not only all arithmetic and logic operations, but also graphic display operations. Up until the early 1990s, ”there was no such thing as a GPU” [Nickolls and Kirk, 2008]. In 1990s, a Video Graphics Array (VGA) controller was widely used to perform graphics. It had two basic parts: a display generator and a memory

controller with some attached Dynamic Random Access Memory (DRAM). It was connected to the PCI bus, and it drove the video display from memory buffer. In 1999, the term GPU was first coined by NVIDIA with the release of "the world's first GPU" the GeForce 256 [NVIDIA, 2013]. In 2006, NVIDIA introduced the GeForce 8800, which was designed on the unified shader architecture with up to 128 processing elements. The GeForce 8800 is also the first GPU, which is compatible with Compute unified Device Architecture (CUDA), the industry's first C-based development environment [Glaskowsky, 2009]. Since graphics devices grew up to be true processors, modern GPUs have been evolved from fixed-function graphics units to all-purpose, powerful computing engines with special-purpose functionality.

Fundamentally, CPU and GPU have significantly different internal architectural designs with different purposes. CPU is designed as a general-purpose processor, which can perform all the operations and tasks. Usually, CPU can obtain better performance on sequential code by increasing transistor counts to caches and control logic [Jang et al., 2011]. On the contrary, GPU is originally designed for rendering or generating three-dimensional graphics. Generally, an early GPU was not designed for general-purpose computations, and it could not perfectly perform all the operations and tasks. However, modern GPUs have the inherent parallel nature to be a high-performance parallel processor. Figure 2.1 shows a schematic comparison between CPU and GPU. GPU contains far more Arithmetic Logic Units (ALUs) than CPU, and it has also more control logic with some cache memory. The design strategy of GPU tends to use many parallel processors to create enough multiple threads rather than use multi-level caches to hide long memory access latency. Although state-of-the-art GPU could provide significant performance benefits over state-of-the-art CPU, the best performance for many applications comes from using both CPU and GPU [Nickolls and Kirk, 2008].

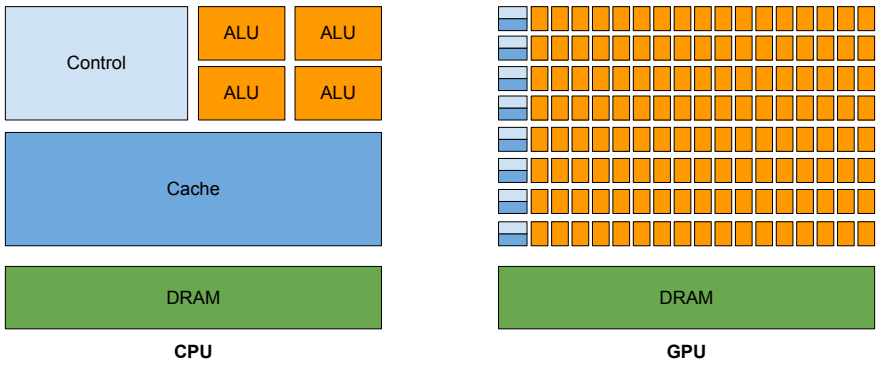


Figure 2.1: Illustration of the differences between CPU and GPU design.

In a modern personal computer, there are at least one CPU and one GPU, and both of them are connected to the same motherboard. This combination offers the basic computing power of a system. Figure 2.2 illustrates a perspective on the basic structure of an Intel processor-based computer system. The CPU and the GPU are connected directly to the north bridge, which enables communication between the CPU, the GPU and the memory. Instead of using the Front Side Bus (FSB) between the CPU and the north bridge, there is a PCI-Express connection

between the GPU and the north bridge. In an AMD processor-based system, the interconnection is very similar to an Intel processor-based system, the discrete GPU is also attached via the PCI-Express link to the chipset, but the north bridge and the CPU are integrated on the same die.

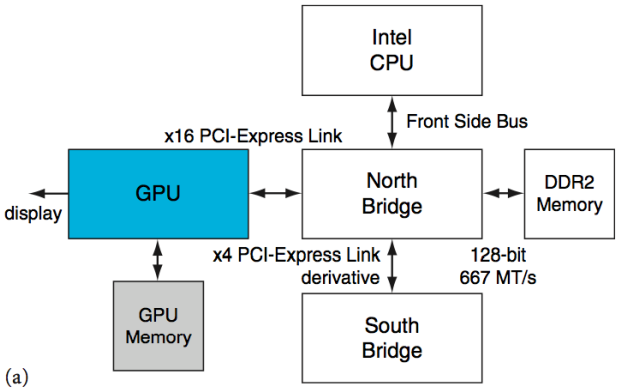


Figure 2.2: Illustration of modern PC with Intel CPU (Source: [Patterson and Hennessy, 2008])

In general, CPU and GPU have their own memory subsystems: the main memory and the GPU memory respectively. Because they have their own separate memory spaces, applications implemented on multicore processors cannot directly access the GPU memory space. In fact, this distinct interconnection causes a performance bottleneck of a CPU-GPU heterogeneous system, and the PCI-Express connection is one major performance issue [Schaa and Kaeli, 2009], [Daga et al., 2011]. For example, a 16-lane PCI-Express 2.0 connection offers up to 8 GB/sec transfer rate in each direction [Patterson and Hennessy, 2008], this is much less than some GPUs have, for instance, GeForce 8800 GTS had already a memory bandwidth 86.4 GB/sec [NVIDIA, 2012]. The memory bandwidth of GPUs is generally higher than the memory bandwidth of CPUs. Many current GPUs have been designed to deliver higher memory bandwidth. They can deliver approximately 160-200 GB/s as compared to traditional host memory systems that can provide 8-20 GB/s [Farber, 2011].

According to the interconnection technology between CPU and its accelerator unit, as well as their respective memory subsystems, there are generally two kinds of heterogeneous systems: loosely-coupled and tightly-coupled heterogeneous systems. Some researchers have also classified heterogeneous systems by their execution models into similar two categories: a device driver-based loosely-coupled execution model and an ISA-based tightly-coupled approach [Wang et al., 2007], [Linderman, 2009]. To avoid the conflict, in this thesis, the interconnection behavior is used to distinguish between loosely-coupled and tightly-coupled systems.

2.2 Loosely-coupled heterogeneous systems

In a loosely-coupled heterogeneous system, CPU and its hardware accelerator are usually not integrated together on the same silicon die, and moreover, the communication between them is

not direct. They have their own separate memory spaces. As a result, data transfers between different memory spaces are required. Some examples of loosely-coupled accelerators are discrete GPUs, Intel Many Integrated Core (MIC) processors [INTEL, 2013], FPGAs, CELL processors, etc.

In comparison to tightly-coupled heterogeneous systems, loosely-coupled heterogeneous systems are more computation-efficient by reason of the more powerful discrete accelerators. However, the limitations of the memory latency and the bandwidth have always been the primary matters that affect the final performance. The main disadvantage of loosely-coupled heterogeneous systems is the split between different memory spaces, because this data transfer cost will have a significant impact on the performance. However, for accelerator architects, the loosely-coupled concept can provide more flexibility for them [John L. Hennessy, 2011], [Linderman, 2009].

In this thesis, the experimental platform is also a loosely-coupled heterogeneous system equipped with a discrete GPU, the NVIDIA Tesla C2050.

In the next two subsections, there are two typical examples of loosely-coupled accelerators.

2.2.1 NVIDIA Fermi architecture

"Fermi" is the code name of NVIDIA's current generation CUDA architecture, and Fermi-based GPUs are very popular hardware accelerators in HPC. Figure 2.3 illustrates a block diagram of NVIDIA Fermi GPU architecture with a streaming multiprocessor (SM). A Fermi GPU contains a large number (up to 512) of streaming processors (SPs) in many streaming multiprocessors. For example, the NVIDIA Tesla C2050 has a total of 448 SP cores in 14 SMs. Each SM contains two groups of 16 SP cores, a total of 32 SP cores per SM. The SP core is the basic computational unit. Each SP has an integer Arithmetic Logic Unit (ALU) with a Floating-Point Unit (FPU), moreover, it can execute one single precision fused multiply-add instruction per clock [NVIDIA, 2009]. One multiply-add instruction can be regarded as two floating-point instructions. For instance, the single precision floating-point capability of the 1.15 GHz Tesla C2050 with 448 SP cores is:

$$448\ cores \times 2\ \frac{Flop}{instruction}/core \times 1\ \frac{instruction}{clock} \times 1.15 \times 10^9\ \frac{clockcycles}{second}$$

$$= 1030.4 \times 10^9\ \ (Fflop/second) = 1.0304\ \ (Tflop/second)$$

Fermi GPU supports the new IEEE 754-2008 double precision standard [NVIDIA, 2009]. The peak double precision floating point performance is about 50% of the peak single precision floating point performance [NVIDIA, 2009].

The Fermi architecture allows launching many thousands of parallel threads simultaneously. To manage a large number of simultaneously active threads, two-level thread schedulers are used at different levels. At the chip level, the GigaThread is used to distribute thread blocks to SMs or to quickly switch context between GPU computing applications and normal graphics

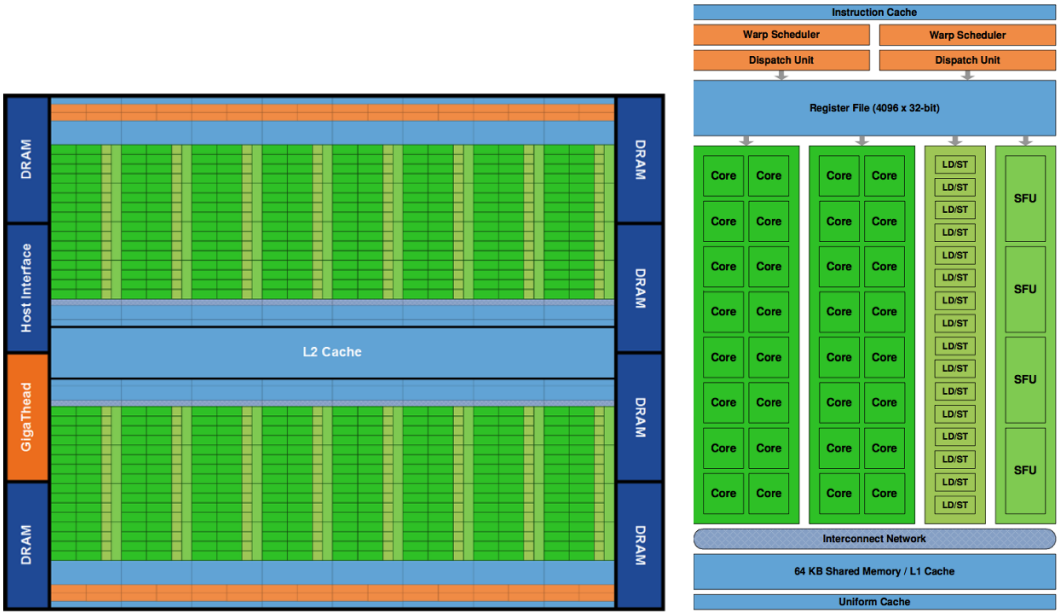


Figure 2.3: An overview of NVIDIA Fermi architecture and a streaming multiprocessor (SM). (Source: [NVIDIA, 2009])

applications. At the SM level, each SM has two warp schedulers, each of which schedules and distributes a set of 32 threads on each SM.

The Fermi architecture offers 32 Kilobyte (KB) of register files in each SM, and moreover, each SM has a configurable local memory for the L1 cache and shared memory. There are two options for different needs: either 16 KB of L1 cache with 48 KB of shared memory or 48 KB of L1 cache with 16 KB of shared memory [NVIDIA, 2009]. For more predictable data re-use among threads, using more shared memory is more efficient because of the reduction of memory traffic. In contrast, a large L1 cache is needed for irregular applications, which basically have unpredictable memory access patterns. All the SMs share a common unified L2 cache (up to 768 KB), which connects the host interface with PCI-Express and six DRAM interfaces, each 64 bits wide, up to 6 GB of GDDR5 DRAM can be attached.

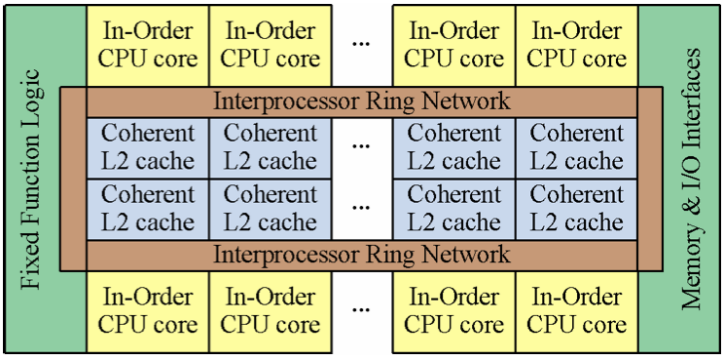
In addition, the Fermi architecture is the first GPU architecture with error correcting code (ECC) memory supports [Glaskowsky, 2009] to avoid memory errors and to ensure data integrity. The register files, shared memories, L1 caches, L2 caches, as well as DRAM are all ECC protected. Data sensitive computations such as medical imaging and financial options will benefit from the ECC's protection [NVIDIA, 2009].

2.2.2 Intel Many Integrated Core (MIC) architecture

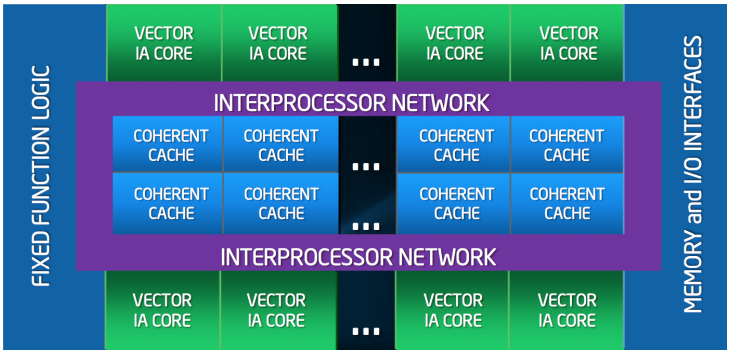
Intel Many Integrated Core (MIC) architecture is derived from three Intel's projects: the Larrabee visual computing project, the 80-core Tera-scale research chip program and the single-chip cloud computer initiative [INTEL, 2013]. The main targets for this architecture are highly parallel

applications in HPC, workstation and data centre. The Intel MIC architecture, including a prototype board with 32 cores, the code name “Knights Ferry”, was unveiled as an Intel co-processor architecture at the international Supercomputing Conference (ISC) in 2010 [Sakaugen, 2010]. At the International Supercomputing Conference (ISC) 2012, the MIC architecture based products were named a new product family name, the “Xeon Phi”. In addition, the first commercial Xeon Phi co-processor, “Knights Corner”, is built on 22-nanometer 3-D Tri-Gate transistor technology with more than 50 Intel Architecture (IA) cores on a single board [Sakaugen, 2011].

The MIC architecture has a different design concept compared to other hardware accelerators. In fact, this architecture has inherited a large proportion of the design concept from Intel’s abandoned Larrabee project [Seiler et al., 2008]. Figure 2.4 indicates that the Larrabee and MIC are very similar and almost identical. Many general-purpose in-order cores (a modified Intel Pentium P54C), each supports 4 hardware threads and wider vector units (VPU), are connected together through a high-bandwidth inter-processor ring network with fixed function logic, memory and I/O interfaces [Seiler et al., 2008]. Furthermore, both of them have the same basic cache structure and the coherent cache hierarchy, each in-order x86 core has two cache levels: L1 cache with 32 KB, and a shared L2 cache with 256 KB.



(a) Larrabee many-core architecture block diagram. (Source: [Seiler et al., 2008])



(b) MIC (Many Intergrated Core) co-processor architecture blolck diagrm. (Source: [Sakaugen, 2010])

Figure 2.4: Intel’s Larrabee and MIC (Many Integrated Core) architecture design

Although the MIC architecture and Larrabee have lots in common, they are still very different in many aspects. Basically, they are designed for different purposes and markets. The original Larrabee was designed not only for general purpose computing, but also as a discrete high-end graphics card, whereas the new MIC architecture concentrates only on the high performance computing.

The Intel Knights Ferry software development platform, a platform combined one Knights Ferry co-processor (1.2 GHz, 2GB GDDR5) with two Intel Xeon processor X5680 (3.3 GHz, 6 cores, 12 MB L3 cache), has been already tested by some selected Intel's MIC partners [Sakaugen, 2011], including the Research Group "Scientific Computing" of the University of Vienna. There are two attractive features of the MIC architecture based products: high performance and software programmability advantage. According to the Intel's demonstrations at ISC in 2011, the Knights Ferry software development platform achieved over one Teraflops (10^{12} calculations/second) for the SGEMM routine. In the Intel MIC architecture, because of its standard x86 architecture benefits, many familiar parallel programming models and math libraries, such as OpenMP, MPI, Intel Array Building Blocks, Intel Cilk or Intel's Math Kernel Library (MKL) can be used.

2.3 Tightly-coupled heterogeneous systems

Tightly-coupled heterogeneous systems can sometimes be described as heterogeneous chip multiprocessors [Kumar et al., 2005]. In these systems, the tightly-coupled accelerator and CPU are directly connected, and they are normally integrated on the same chip. A tightly-coupled heterogeneous system is mostly a unified memory architecture (UMA) system, in which a common system memory is put to use. Consequently, the CPU and its accelerator can communicate with each other directly, and the communication will be faster than a loosely-coupled heterogeneous system. Processors based on Intel Sandy Bridge architecture [Yuffe et al., 2011] or AMD Fusion architecture [Daga et al., 2011] are typical tightly-coupled heterogeneous systems.

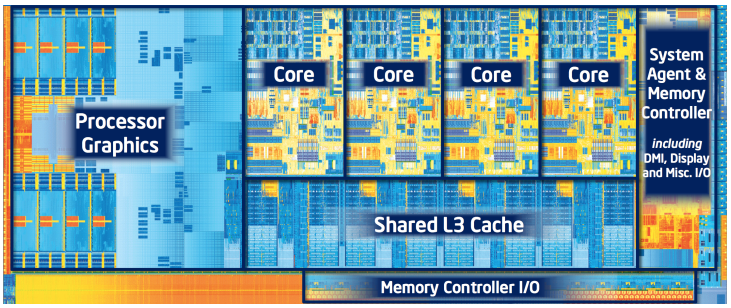
Generally, tightly-coupled heterogeneous systems are more energy-efficient and less expensive than loosely-coupled systems. Another benefit of tightly-coupled heterogeneous systems is the die area. Tightly-coupled systems are considerably smaller than loosely-coupled systems. Because of their self-limiting area, power-consuming and heat dissipation, tightly-coupled heterogeneous systems would not provide the same level of performance as loosely-coupled heterogeneous systems.

Following are two typical examples of tightly-coupled heterogeneous systems.

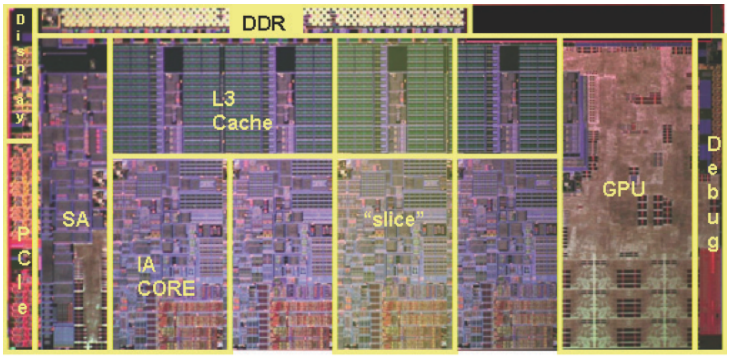
2.3.1 Intel Ivy Bridge

Ivy Bridge is the Intel's code name for the Intel's current CPU processor architecture. In an Ivy Bridge processor, several IA (Intel Architecture) cores, an integrated GPU, a memory controller, a PCI-Express controller and a display controller are fully integrated in the same die [Damaraju et al., 2012]. It is the "Tick" in Intel's "Tick-Tock" development model, and it is manufactured on 22-nanometer process technology. Its 32-nanometer predecessor, also known as "Sandy Bridge",

is the latest "tock" in its design cycle [Yuffe et al., 2011]. In its largest incarnation, Ivy Bridge is about 25% smaller than die size of Sandy Bridge, but has 20% more transistors. The Ivy Bridge architecture is essentially the same as Sandy Bridge. Figure 2.5 illustrates the layout of Intel Ivy Bridge and Sandy Bridge processors. One remarkable change is the graphics portion of the die area on Ivy Bridge. The GPU-CPU ratio has been increased visibly. In fact, the number of execution units in the Ivy Bridge's integrated GPU is increased from 6 or 12 up to 8 or 16.



(a) 22nm Ivy Bridge, 1.4 billion transistors in 160 mm^2. (Source: [Heaney, 2012])



(b) 32nm Sandy Bridge, 1.16 billion transistors in 212 mm^2. (Source: [Yuffe et al., 2011])

Figure 2.5: Ivy Bridge processor vs. Sandy Bridge processor.

Figure 2.6 illustrates the block diagram of an Ivy Bridge quad-core, including the main components and the interconnection. In this new architecture, the north bridge is completely merged with the CPU into the same processor die. It is already a part of the Ivy Bridge CPU, and only the south bridge is still a chipset outside the CPU. One particular feature is that all IA cores, the graphics processor, the level 3 cache memory and the system agent (SA) are all connected together via a high-bandwidth 256-bit/cycle ring-based interconnection. This inter-processor ring network was introduced in 2008 on Intel's Larrabee many-core architecture to link many in-order x86 CPUs [Seiler et al., 2008]. According to a report from Intel at IDF2010 (Intel Developer Forum), at a clock frequency of 3 GHz, the ring interconnection can offer a peak transfer rate of 96 GB/s per each ring stop, with four cores, the ring interconnection can deliver a total bandwidth of 384 GB/s. Another important feature is that there is a shared last-level cache

(LLC), level 3 cache for GPU and CPU. These two features, the ring interconnection and LLC, can provide many characteristic features, including general-purpose computation, graphics, etc. [Rotem et al., 2012]. In addition, they also provide an opportunity to obtain performance benefits from this tightly-coupled heterogeneous system.

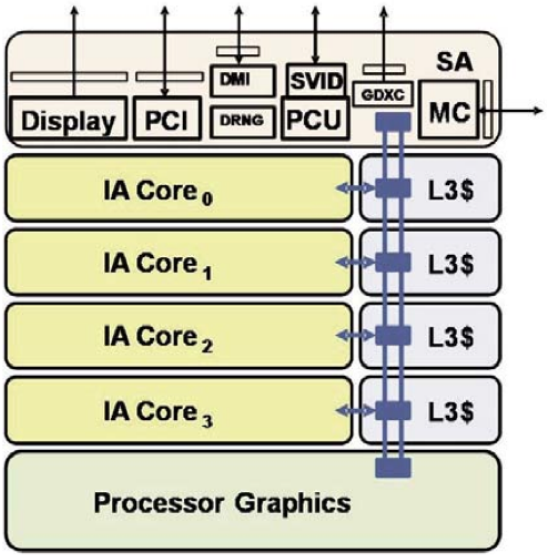


Figure 2.6: Ivy Bridge porcessor block diagram. (Source: [Damaraju et al., 2012])

2.3.2 AMD Accelerated Processing Unit

The AMD's Accelerated Processing Unit (APU) [AMD, 2013] is another representative example of tightly-coupled heterogeneous systems. It is very similar to Intel Ivy Bridge architecture. Both of them combine the general-purpose CPU with the parallel processing engine GPU together into the same package for high performance and energy efficiency.

Figure 2.7 shows the layout of the AMD's mainstream "Llano" variant. An integrated graphics core (Graphics SIMD Array), and multimedia resources, including a unified video decoder and a display controller, are integrated with four x86 CPU cores on a 227 mm^2 die area. As shown in this figure, a large part of die area is the GPU, which has resided approximately half of die area. This ratio is much bigger than in Sandy Bridge or Ivy Bridge. It is similar to the Intel's state-of-the-art CPUs, the north bridge and the memory controller are integrated to the same die as well.

Figure 2.8 shows the high level diagram of an AMD APU quad-core processor, including the major functional parts as well as the interconnection. The integrated north bridge is an important management and communication centre. CPU cores, GPU, I/O interface, graphics memory controller and two 64-bit channels of dynamic random access memory (DRAM), all these main components are connected to the north bridge chipset.

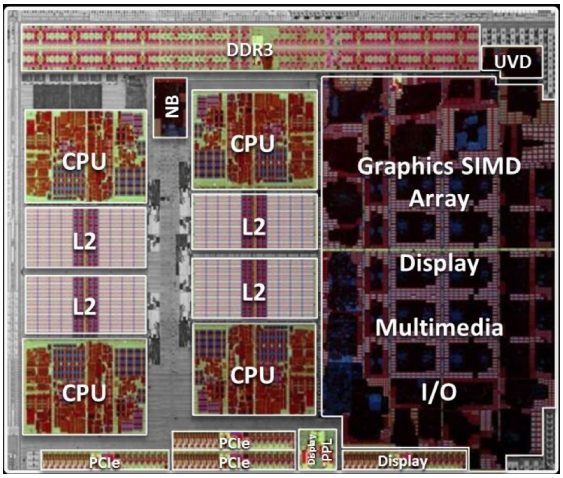


Figure 2.7: Photograph of a 32nm AMD Fusion "Llano" APU die. (Source: [Rogers, 2011]).

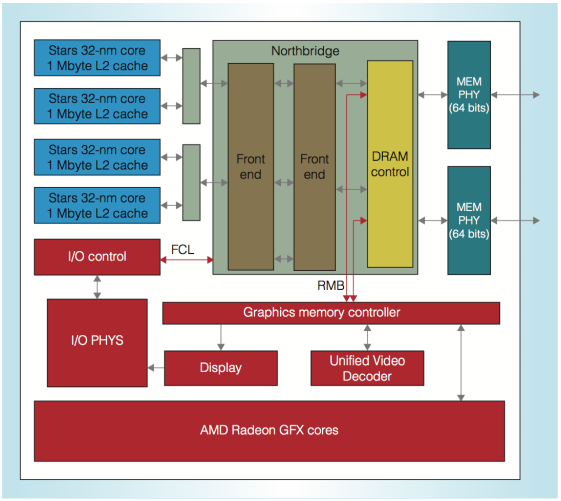


Figure 2.8: "Llano" APU block diagram. (FCL: Fusion Control Link; RMB: Radeon Memory Bus. MEM PHY: memory physical layers.) (Source: [Branover et al., 2012])

Instead of using PCI-Express connection, two new buses are used in AMD's APU: Fusion Compute Link (FCL) between the north bridge and the CPU, and Radeon Memory Bus (RMB) between the north bridge and the GPU. These two buses satisfy different needs of this heterogeneous architecture. The FCL can provide cache coherency for fine-grained data sharing between the CPU cores and GPU [Spafford et al., 2012], and to hide the long memory access latency between GPU and memory, the RMB, a non-coherent 256-bit data path, is directly connected to local memory. The RMB enables up to 29 GB/sec of memory bandwidth [Branover et al., 2012].

2.4 Basic parallel programming support for heterogeneous systems

The biggest challenge using heterogeneous systems is the complexity of their programming models. They are totally different from many existing parallel programming models for traditional homogeneous systems. Figure 2.9 summarizes programming models for GPU-based heterogeneous systems in general.

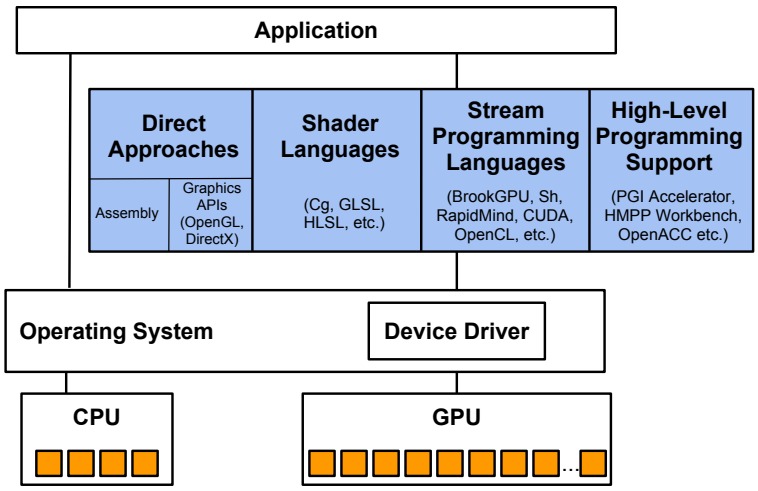


Figure 2.9: Heterogeneous CPU/GPU programming models.

The earliest stage of the GPGPU (General-Purpose programming on GPU) is the direct approaches. They are quite difficult to use, and are similar to the assembly language in the early days of computer programming. Only a few of "adventurous" programmers master these assembly instructions or domain-specific graphics APIs, such as OpenGL or DirectX models. Thompson et al. (2002) have developed a framework with vector instruction set for general-purpose computing using graphics architecture [Thompson et al., 2002]. This low-level assembly instruction programming model could be able to provide a high degree of efficiency. However, its poor usability and high hardware-dependency are two main weaknesses.

The next are the shader programming models, which are normally based on the syntax of the C programming language. Shader is a computer program that is usually used to operate on graphics data like a vertex or a pixel fragment. Some representatives of high-level shading languages are C for graphics (Cg) [Mark et al., 2003], High-Level shading language (HLSL) [Oneppo, 2007] or OpenGL Shading Language (GLSL) [Marroquim and Maximo, 2009]. The limitation and complexity of assembly instructions are relatively reduced through the high-level programming abstractions and a rich set of library functions. On the other hand, the inherency of the fixed-function graphics pipeline is still a major disadvantage. Programmers are obliged to express all the general-purpose computation within a graphics pipeline. In addition, the functionality of a program is also restricted within narrow limits because of the limitation of graphics

APIs [Kirk and Hwu, 2010]. In short, shader programming models are not easy to approach.

The stream programming models are the next phase. The main concept of these models is that the GPU is abstracted as a streaming *co-processor*, and they are more high-level than the shader programming models. The early representatives are ANSI C-based BrookGPU [Buck et al., 2004] or SH [McCool et al., 2004], a C++ library API. In a stream programming model, data that can be operated in parallel is often represented as the stream, and a function that operates many streams is represented as the kernel. Stream and kernel are two essential concepts in a stream programming model, and they ensure an abstraction approach without using graphic terms such as vertices, textures, fragments, etc. The usability and efficiency are much better than the shader programming models. On this stage, CUDA and OpenCL (Open Computing Language) are two currently dominant basic parallel programming models for heterogeneous systems.

The last stage is the high-level programming support, which is also the main research object in this thesis. With high-level programming approaches, programmers do not have to manually manage memory and parallel executions. More information can be found in Chapter 3 and Chapter 6.

2.4.1 Computer Unified Device Architecture (CUDA)

CUDA is not only NVIDIA's parallel computing architecture, but also a heterogeneous serial-parallel programming model for massive data parallelism. It is widely used in many scientific applications and interdisciplinary studies. In this thesis, CUDA is also used to evaluate several high-level heterogeneous programming models.

In the view of the CUDA programming model, there are two main parts of the computing system: *host* processor and *device*. The CPU is considered to be a *host* processor, and the GPU is considered to be a *device* or co-processor. A complete CUDA development system contains at least one host processor plus one or more devices.

As Figure 2.10 illustrates, a CUDA program involves several steps, and each of which is done either by the host or by the device. Serial code and logical operations are normally performed by the host processor. The data parallel portions, which are often referred to as kernels, are executed on the device. A CUDA kernel is a fundamental part of a CUDA program. Generally, CUDA kernel is a C function with some restrictions. In addition, at a time, only one kernel function can execute on the device GPU. A kernel function can create thousands of threads on the device to accomplish data parallelism. Normally, the host CPU and the device GPU have their own separated memory spaces, the main memory and GPU memory. Hence, before launching a kernel function on the device, enough data space on the device memory space must be allocated. After that, the required data must be transferred into the device memory from the host memory. A simple basic example of CUDA processing flow consists of the following steps:

1. Allocate memory in the device;

2. Transfer data from the host processor to the device;

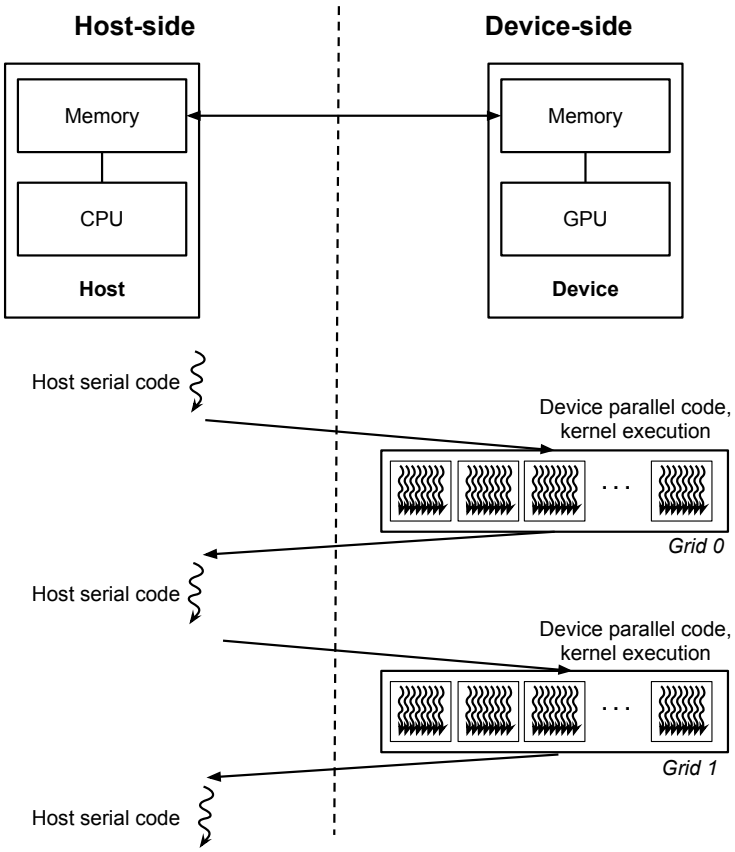


Figure 2.10: Execution model of a CUDA program

3. Launch the kernel in the device in parallel;

4. Transfer data from the device to the host processor;

5. Free the allocated memory.

When a kernel function is launched, a very large number of threads are generated by this kernel function on the device GPU. The creation process takes only few cycles. To manage this massive amount of running threads, the CUDA programming model uses a two-level hierarchy design, and all the launched threads are organized by a grid of blocks. As Figure 2.11 illustrates, at the top of the multi-level hierarchy is a grid, which is composed of one or a group of thread blocks. The threads in different blocks cannot communicate directly. Moreover, a thread block consists of a set of concurrent threads, which can communicate with each other or synchronize by using a barrier function inside the same block. Usually, programmers need to define the number of thread blocks as well as the number of threads per block in advance.

Each thread within a thread block can identify itself by its own ID using 1-D (dimensional), 2-D, or 3-D thread index. Likewise, each block within a grid has also its unique ID, identified

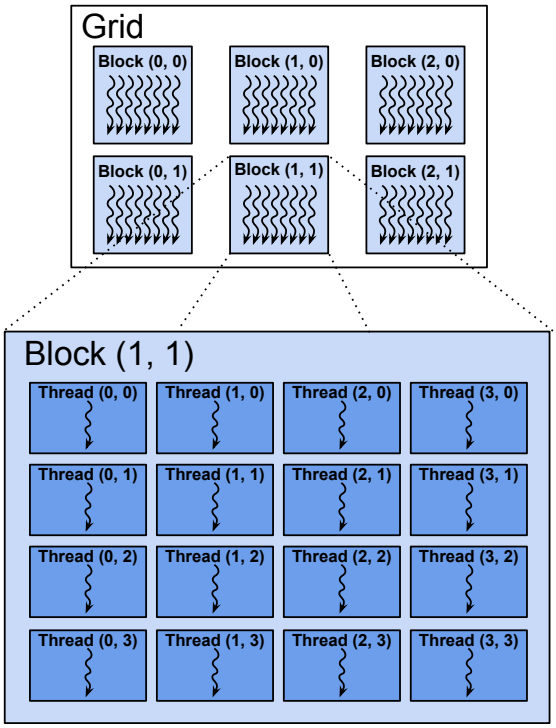


Figure 2.11: CUDA threads hierarchy

using 1-D, 2-D, or 3-D block index. Thus, the global index of a thread can be specified by the combination of these two IDs. In the CUDA programming model, the thread ID and block ID are used to make control decisions or to determine the region of data that a thread has to work on. For more convenient programming, a series of built-in variables can be used as well (with types) :

- unit3[1] *threadIdx*; // The *thread index* within a thread block.
- unit3 *blockIdx*; // The *block index* inside a grid.
- dim3[2] *blockDim*; // The dimensions of a block.
- dim3 *gridDim*; // The dimensions of a grid.

For instance, Figure 2.11 shows that all threads are organized by a 2-D (3×2) grid of thread blocks, and each thread block is organized into the 2-D (4×4) array of threads. The global thread ID can be easily calculated with its unique block and thread coordinates:

[1] A few built-in vector data types are provided by CUDA paradigm, and they can be considered as previously defined structures, for example, unit3 can be considered as a C structure with three unsigned integer fields.

[2] The type dim3 is based on uint3 to specify dimensions.

- 1-D thread block, $threadID = threadIdx.x$.
- 2-D thread block with size (D_x, D_y), $threadID = threadIdx.x + threadIdx.y \times D_x$.
- 3-D thread block with size (D_x, D_y, D_z), $threadID = threadIdx.x + threadIdx.y \times D_x + threadIdx.y \times D_x \times D_y$.

Each thread block is mapped to a single SM on the device GPU. Because of the restriction of the hardware resource, the maximum number of threads within a block is limited. This number depends on the compute capacity of the device, for example, like Fermi architecture, up to 1024 threads can execute concurrently within a block. However, it is possible to create a lot of thread blocks to execute the same kernel. The total number of threads is the product of the number of the threads per block and the number of the thread blocks.

In CUDA programming model, another important abstraction hierarchy is the multi-level memory hierarchy that is illustrated by Figure 2.12.

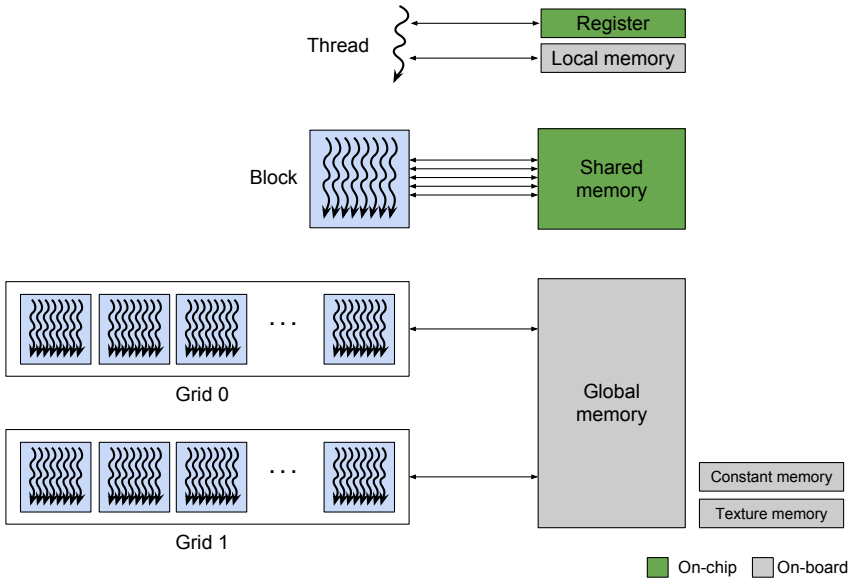


Figure 2.12: CUDA memory hierarchy

The memory hierarchy is also organized into different levels corresponding to the thread organization. Each thread can access its own private register file and local memory. All threads residing in the same block have exclusive access to on-chip shared memory. Every generated thread in a grid can access the same on-board global memory, constant memory as well as texture memory. Accessing the on-chip register or shared memory is much faster than the on-board local memory and global memory, when there are no bank conflicts. The shared memory is accessible by all the threads within a thread block, and it enables inter-thread communication and cooperation. Moreover, the size of shared memory as well as the number of registers are limited.

On the other hand, accessing the on-board local memory or global memory is much slower than register, and random access can be very expensive. Because of higher access latency for global memory and local memory, on-chip register and shared memory should be used as much as possible.

2.4.2 Open Computing Language (OpenCL)

While CUDA is a proprietary framework from NVIDIA, and restricted to NVDIA's GPUs only, Open Computing Language (OpenCL) is designed for different platforms, including GPUs, multicore CPUs, Cell processors as well as DSPs. OpenCL is an open standard, which was initiated by Apple, and is currently maintained by the Khronos Group[3].

Basically, CUDA and OpenCL have very similar concepts, which are actually borrowed from the early stream programming models, especially from BrookGPU, for example, the concept of hiding details of underlying hardware from the programmer, using the GPU as a co-processor, or the concept of "kernels".

Generally, CUDA and OpenCL have many features in common but also with a few exceptions. Both of them are low-level APIs for heterogeneous computing. As can be seen from Table 2.1, various terms are used to describe the same features and concepts of CUDA and OpenCL. CUDA applications can be rewritten without any difficulty in OpenCL.

CUDA	**OpenCL**
Streaming Multiprocessor (SM)	Compute Unit (CU)
Streaming Processor (SP)	Processing Element (PE)
Kernel	Kernel
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory and register	Private memory

Table 2.1: Comparison of CUDA and OpenCL

It is generally believed that CUDA can offer better performance than OpenCL on NVIDIA's hardware. Fang et al. have performed a performance comparison of CUDA and OpenCL, for selected applications, CUDA performed up to 30% better than OpenCL. However, under "a fair comparison", OpenCL could have nearly the same performance as CUDA on NVIDIA hardware [Fang et al., 2011].

In general, OpenCL is very similar to CUDA. Both of them address the foundational issues for programming on heterogeneous systems. The main advantage of OpenCL is its portability across different platforms, which is exactly the reason that the OpenCL programming model is more complex than CUDA.

[3] http://www.khronos.org/opencl

CHAPTER 3

High-level programming support for heterogeneous systems

For programmers, the most difficult challenge of heterogeneous systems is the complexity of their programming support. Without software support, the benefit of the high degree of performance of heterogeneous systems cannot be obtained. The intention of this chapter is to explore four high-level programming models for heterogeneous systems. Additional high-level programming solutions can be found in Chapter 6.

3.1 PGI Accelerator programming model

The PGI [1] Accelerator programming model is a high-level programming model for heterogeneous systems, and this approach is very similar in design and scope to the OpenMP programming model [Wolfe, 2010]. It is a commercial product, and was first introduced by Portland Group Inc. in 2009. At the moment, the PGI Accelerator compiler supports only CUDA-enabled NVIDIA GPUs. In the PGI Accelerator programming model, GPUs are expressed as hardware accelerators. C/C++ and Fortran programs can be easily rewritten using a few PGI compiler directives without restructuring.

3.1.1 Overview

The main purpose of the PGI Accelerator programming model is to use the compiler to generate executables that work on heterogeneous CPU-GPU architecture, and ideally, they can achieve

[1] http://www.pgroup.com

the same performance like an experienced programmer can get by writing the same CUDA program. This approach greatly reduces the burden on the programmers. In comparison to CUDA programming model, the PGI Accelerator programming model is an implicit approach for offloading codes to an accelerator. Through adding OpenMP-like compiler directives, the PGI compiler manages not only all the data transfers between the host processor and the device, but also the mapping of computationally intensive portions of a program to the device GPU. One additional benefit of this model is that the same PGI program can be executed on the host CPU without any changing, because the normal C or Fortran compiler will ignore the PGI compiler directives. This is also an advantage of many directive-based programming approaches.

In the PGI Accelerator programming model, the source code is divided into structured code blocks, which are also called as code *regions*. The concept of *region* is the basic element in this programming model. There are mainly three groups of compiler directives to define code regions: compute region directives, data region directives and loop directives.

Usually, parallelizable loops belong to a PGI compute region. They can be mapped as kernel functions onto the accelerator, and loop iterations can be mapped onto the block and thread indices. The compute region is the only required region [Wolfe, 2010], and it is a single entry, single exit region. The compute region cannot contain some statements, such as `goto`, `return`, or `exit`, which cause an early termination before the iteration count reaches a certain value. A PGI program can have many compute regions, but each compute region cannot contain other compute regions or data regions [The Portland Group Inc., 2010].

The data region is not always necessary, but still important. It allows data movements only within the region, including data allocation, deallocation on the device and copying data between the host processor memory and the device memory. This explicit region offers developers the chance to manage data movements freely. The data region is commonly used to enclose compute regions, and it can contain other data regions as well.

The PGI Accelerator compiler directives are very similar to OpenMP directives. In C language, the compiler directives are defined by using the `#pragma` constructs, and in Fortran language, the directives are defined by using stylized comments to instruct the compiler.

Figure 3.1 and Figure 3.2 show two PGI directive-based examples writing in C and Fortran. In the PGI Accelerator programming model, the first step is always to identify the parallelism in a sequential program, which must be done by the programmer at the moment. The accelerator compiler will map the loop parallelism onto the device, and the compiler will automatically manage all the data movements between the host processor and the device.

```
1 #pragma acc region
2 {
3     for (int i=0; i< n; i++)
4     {
5         C[i] = A[i] + B[i];
6     }
7 }
```

Figure 3.1: A vector addition with the PGI Accelerator compiler directive, in C.

```
1 !$acc region
2    do i=1, n
3       C(i)= A(i)+ B(i);
4    enddo
5 !$acc end region
```

Figure 3.2: A vector addition with the PGI Accelerator compiler directive, in Fortran.

With following commands, PGI compiler directive-based programs can be compiled:

- For C programs:

```
pgcc -o program_name program_name.c -ta=nvidia -Minfo
```

- For Fortran programs:

```
pgfortran -o program_name program_name.f90 -ta=nvidia -Minfo
```

In these commands, the flag `-ta=nvidia` is the target flag, and it tells the compiler that the targeted accelerator is a NVIDIA graphic card. If there is no accelerator available, the target can be set as `-ta=host`. With the flag `-Minfo`, the PGI compiler can generate useful compiler feedback, which can help programmers to find the performance bottleneck.

3.1.2 PGI Accelerator directives

The PGI Accelerator consists of three basic directives: the compute region directive, the data region directive and the loop mapping directives. The basic syntax of the PGI Accelerator complier directive is following:

- In C language:

```
#pragma acc directive-name[clause[,clause]...]new-line
Structured block or for loop
```

- In Fortran language:

```
!$acc directive-name[clause[,clause]...]
structured block or do loop
!$acc end directive-name
```

The directive-name can be `region`, `data region`, or `for` in C or `do` in Fortran, which correspond respectively to the PGI compute region directive, the data region directive, and the loop mapping directive.

- Compute region directive

  The compute region directive with the directive name `region` is the most important instruction, and a PGI program must contain at least one compute region directive. It insures that the compiler will understand that the surrounded code region must be compiled for an accelerator device.

  Normally, a PGI compute region contains at least one loop body, because loops are the main targets that should be mapped into kernel functions for the accelerator device. The PGI Planner, one of the components of the compiler, does the actual mapping process [Wolfe, 2010].

- Data region directive

  The data region directive with the directive name `data region` is an optional directive, which determines data allocation as well as deallocation, and the overall data movements within the user-defined data region. A PGI data region can contain other data regions and compute regions. If a PGI program has only data region directive, the PGI compiler will not generate any kernel functions.

- Loop mapping directive

  The PGI compiler can map the parallelizable loops within a compute region into the device parallelism. However, additional loop mapping directives can be added to explicitly manage the mapping process.

  There are some helpful clauses available on the loop mapping directive, such as the `independent` clause, which is used to tell the compiler there is no data dependency, or the `cache` clause to instruct the compiler to move some data into cache memory, the highest level of memory hierarchy.

Table 3.1 shows all the PGI clause options that can be used with the compiler directive to assist the compiler. More information about the compiler directives and their clauses can be found in [The Portland Group Inc., 2010].

3.2 HMPP Workbench

HMPP Workbench is a Heterogeneous Multi-core Parallel Programming (HMPP) environment, which offers a high-level abstract of stream programming [CAPS enterprise, 2012]. It is a directive-based approach and very similar to OpenMP and PGI Accelerator programming model. One important feature of this programming model is the ability of supporting different accelerator targets, including NVIDIA GPUs, AMD GPUs, as well as Intel MIC accelerator. The HMPP

Compute region directive clauses	Data region directive clauses	Loop mapping directive clauses
copy (list)	copy (list)	host [(width)]
copyin (list)	copyin (list)	parallel [(width)]
copyout (list)	copyout(list)	seq [(width)]
local (list)	local (list)	vector [(width)]
deviceptr (list)	deviceptr (list)	unroll (factor)
update device (list)	update device(list)	independent
update host (list)	update host (list)	kernel
if (condition)	mirror (list)	private (list)
async [(handle)]	mirror (list)	cache(list)

Table 3.1: PGI Accelerator directive clauses.

Workbench is a commercial product from the company CAPS [2]. It consists of a set of HMPP directives, a runtime, and C and FORTRAN compiler drivers with CUDA and OpenCL code generators.

3.2.1 Overview

HMPP Workbench offers a high-level directive-based programming interface to hide hardware details from programmers, and knowledge of accelerator architecture is not required. This method closely resembles PGI Accelerator approach as well as OpenMP. Programmers have to identify a function that must be run on the hardware accelerator by using a set of HMPP compiler directives.

The principal concept in HMPP Workbench is the so-called *codelet*. A codelet is a pure function that consists of the computationally intensive part of an application, and it can be remotely executed on an accelerator. The HMPP codelet directive specifies a HMPP codelet function with well-defined inputs. The HMPP Workbench provides a codelet remote procedure call (RPC) model for function executions on an accelerator as well as managing computing resources. As Figure 3.3 indicates, a complete RPC process consists of these following 5 steps:

Figure 3.3: HMPP remote procedure calls (RPC) processing flow

1. Distribute a HWA (Hardware Accelerator) and allocate memory space on the HWA;

[2] www.caps-entreprise.com

2. Transfer data from the host processor's main memory to the HWA memory;

3. Launch the codelet function (the kernel function) on the HWA;

4. Transfer data back to the host memory;

5. Release the HWA and free the allocated memory.

This HMPP RPC processing flow and the CUDA processing flow are on the whole the same. However, with HMPP Workbench, programmers do not need to implement each step manually.

This programming environment supports both synchronous and asynchronous codelet RPCs. The synchronous codelet RPC is defined by default, but an asynchronous codelet RPC can be applied for more efficient data transfer and function execution. In addition, this asynchronous operation is hardware independent. A codelet function can be represented as a kernel function in CUDA or OpenCL. It has at least one corresponding *callsite* that is the location for launching the codelet function call. The "region" concept, which is the basic concept in PGI Accelerator, is introduced by HMPP Workbench as well. However, in fact, the HMPP "region" is a combination of the codelet and callsite.

Figure 3.4 shows a simple C example of the vector addition with HMPP annotation. The HMPP codelet and callsite directives must be identified by a unique label for the whole application life. For instance, as shown in line 1 and 13, the label of the codelet is `funLabel` in this example. Line 1 also shows a series of useful HMPP directive clauses. The clause `args[*].transfer=atcall` indicates an implicit data transfer between the host memory and hardware accelerator memory at the place of the callsite, the notation "*" denotes all the related parameters in the codelet function `vecAdd` need be automatically transferred. In this example, the data transfer is performed as follows: callsite (funLabel): 10000 -> n, a -> A, b -> B, c -> C, c <- C. The clause `target=CUDA` indicates the hardware accelerator is a CUDA-enabled GPU. The current HMPP compiler supports different targets for codelet generation, including CUDA and OpenCL.

```
1  #pragma hmpp funLabel codelet, args[*].transfer=atcall, target=CUDA
2  void vecAdd(int n, int A[n], int B[n], int C[n]){
3     int i;
4     for (int i=0; i< n; i++)
5     {
6        C[i] = A [i] + B[i];
7     }
8  }
9
10 int main(void){
11    int  a[10000], b[10000], c[10000];
12    // ...
13 #pragma hmpp funLabel callsite
14    vecAdd(10000, a, b, c)
15    // ...
16 }
```

Figure 3.4: Vector addition with HMPP codelet/callsite, in C.

With the following commands, programs with HMPP compiler directives can be compiled:

```
hmpp [HMPP_OPTIONS] HOST_COMPILER [HOST_COMPILER_OPTIONS] files
```

For example:

```
hmpp gcc program_name.c -o program_name
```

The third-party compiler `HOST_COMPILER` is one of regular general-purpose compilers, for example, GNU's gcc and gfortran, or Intel's C and Fortran compilers. The HMPP back-end code generator translates HMPP codelet parts of this application into target languages, such as CUDA or OpenCL, and the rest of application is compiled by this third-party host compiler. The HMPP runtime is responsible for managing all of the important services, including initialization of the HWA, handling communication, memory management and execution of codelets parts.

3.2.2 HMPP Workbench directives

The HMPP Workbench programming model provides a set of elaborate directives and clauses for increasing operational flexibility. The general syntax of HMPP Workbench directives is following:

- In C language:

```
#pragma hmpp  codelet_lable  directive [, clause]*  [&]
```

- In Fortran language:

```
!$hmpp codelet_lable  directive [, clause]*  [&]
```

The notation of an asterisk means there are 0 or more of the preceding elements, and the notation of "&" indicates to continue this directive on the next line [CAPS enterprise, 2012].

Table 3.2 shows the HMPP directives in different categories. The HMPP directives can be simply divided in two parts: declaration and operational directives. According to their different functions, they can be also divided into control flow and data management instructions.

In comparison with the PGI Accelerator directives, as can be seen from their directive clauses, the HMPP Workbench directives are more explicit. More specific information about the HMPP compiler directives and clauses can be found in [CAPS enterprise, 2012].

3.3 OpenACC application program interface

The OpenACC [3] is a new standard for the high-level programming model for heterogeneous host + accelerator architecture, and was first introduced in 2011 at SC2011. This approach is largely

[3] http://www.openacc-standard.org

	Control flow instructions	Data management
Declarations	codelet group function	resident map mapbyname
Operational directives	callsite synchronize region parallel	allocate free acquire release advancedload delegatedstore disregard

Table 3.2: HMPP directives [CAPS enterprise, 2012].

based on the PGI Accelerator programming model (see in 3.1), and promoted by a group of four vendors: PGI, Cray, NVIDIA, and CAPS.

This is also a directive-based parallel programming method, very similar to OpenMP, and it inherits many features and aspects from the PGI Accelerator programming model. According to Douglas Miles, the director of the Portland Group, the OpenACC standard can be expressed as "a subset of the PGI Accelerator programming model". This cross-platform API can be described as a common set of compiler directives that are supported by compilers from PGI, Cray and CAPS.

The OpenACC API and the PGI Accelerator programming model have almost the same basic design concept and share a lot in common. Both of them use previously defined compiler directives to explicitly identify parallelizable loops and regions for instructing the compiler to generate parallel accelerator kernel functions. In comparison with the PGI Accelerator programming model, instead of using the term *region*, the term *construct* is used in OpenACC to specify a code region. This terminology is closer to the OpenMP standard.

The general syntax of OpenACC directives is shown as the following:

- In C language:

```
#pragma acc directive [clause[,clause]...]new-line
structured block or for loop
```

- In Fortran language:

```
!$acc directive [clause[,clause]...]
structured block or do loop
!$acc end directive-name
```

In comparison with the PGI Accelerator programming model, although the syntax of the OpenACC remains unchanged, the directive names are new defined and a few new constructs are supplemented. Moreover, there are much more directive clauses can be chosen. More information about the OpenACC directives and clauses can be found in [OpenACC, 2011].

To convert a PGI program into the OpenACC specification is not very difficult, Table 3.3 shows four necessary replacements of directives.

PGI Accelerator model	**OpenACC API**
#pragma acc region [clause]	#pragma acc kernels [clause]
#pragma acc data region [clause]	#pragma data [clause]
#pragma acc for [clause]	#pragma acc loop [clause]
#pragma acc region for [clause]	#pragma acc kernels loop [clause]

Table 3.3: Comparison of PGI Accelerator programming model and OpenACC API.

Figure 3.5 shows an OpenACC directive-based program, which is converted from the program in Figure 3.1 into the OpenACC specification. To compile this program, the compiler flag `-acc` will enable the PGI compiler to understand the OpenACC specification.

```
1 #pragma acc kernels
2 {
3     for (int i=0; i< n; i++)
4     {
5         C[i] = A[i] + B[i];
6     }
7 }
```

Figure 3.5: A vector addition with OpenACC specification, in C.

There is a noteworthy feature of OpenACC API: its execution model has three levels of parallelism, and three new terms are created to express them: *gang*, *worker* and *vector*. If the hardware accelerator is a NVIDIA's GPU, they respectively correspond to the block, warp, and threads of a warp.

3.4 OpenMPC

OpenMPC (OpenMP extended for CUDA) is an automatic OpenMP to CUDA translation compiler framework [Lee et al., 2009]. It is built on the CETUS project[4], a source-to-source compiler infrastructure for C programs written in Java. It was first introduced by Purdue University in 2009. This high-level parallel programming approach not only increases the reusability and

[4]Http://cetus.ecn.purdue.edu

portability of existing OpenMP programs for heterogeneous CPU+GPU systems, but also decreases the programming complexity.

3.4.1 OpenMP on heterogeneous system

OpenMP (Open Multiprocessing or Open specifications for Multi-Processing) is a high-level representative sample for multicore parallel programming. Although OpenMP is not designed for heterogeneous systems, its paradigm and ideas can also be taken into this new area. In this subsection, the possibility of extending OpenMP to a GPU-based heterogeneous system will be discussed.

OpenMP is a high-level application programming interface, which is specified for shared memory multiprocessing parallel programming. It is a wide-used industry standard for programming on current multicore processors. OpenMP can be described as a language extension. Its primary parallel constructs are based on existing sequential languages, such as C, C++ or Fortran. OpenMP abstracts the details of hardware, and provides a simple method to obtain parallelism. It specifies a clean interface for writing threading parallel programs, and the threading and the concept of concurrency are abstracted together. The generating parallel code from a traditional sequential code requires only some minor changes, and the original basic structure of the sequential code remains the same. Only a few preprocessor directives need to be added around the potential parallel parts.

Though OpenMP applications cannot be directly executed on a heterogeneous system, there is a series of advantages to using OpenMP as a programming pattern for heterogeneous systems:

First, the relationship between the host CPU and the device GPU can be represented as the OpenMP's fork and join constructs [Lee et al., 2009]. OpenMP is based on the classical fork-join pattern. As Figure 3.6 indicates, an OpenMP program always starts with a single master thread, which continues in a sequential fashion until it encounters a parallel region. After reaching a "fork" statement, the single master thread can spawn a team of concurrent threads as needed. In this parallel area, threads can communicate via sharing variables. When all the threads in this team are finished with their works and after reaching a "join" statement, all of them will be terminated. Only the master thread executes sequentially. The single master thread can be easily expressed as a serial code executing on the host CPU, and the team of parallel work threads can be expressed as massive running threads on the device GPU. Additionally, the OpenMP shared variables can be stored in the global memory in the device to enable the communication between all the running threads.

Second, OpenMP uses the loop parallelism pattern and the Single Program Multiple Data (SPMD) pattern. Generally, many problems using the SPMD pattern are also loop-based [Mattson et al., 2004]. These two patterns are perhaps the most suitable parallel programming patterns for data parallel computation on GPUs. In addition, using the thread ID can offer some degree of parallel logic between the loop iterations.

Next, OpenMP is designed around two essential concepts: sequential equivalence and incremental parallelism [Mattson et al., 2004]. The accuracy in computations is the precondition of parallel computing regardless of homogeneous or heterogeneous systems. The incremental

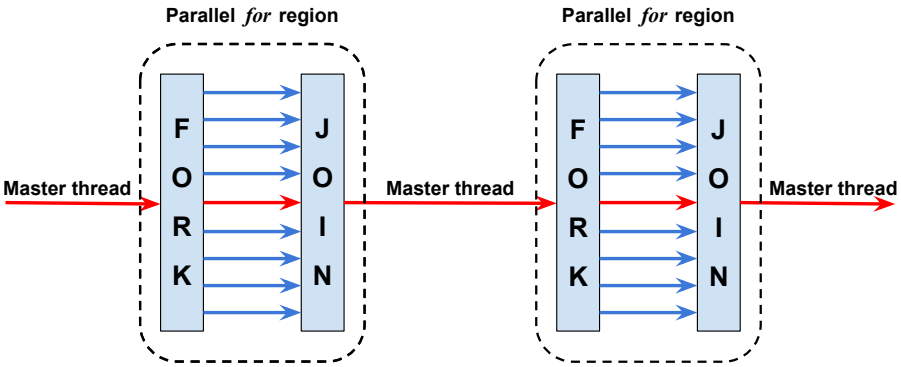


Figure 3.6: OpenMP fork-join model.

parallelism, a step by step from sequential programs to parallel programs programming style, can also be used on CPU-GPU architecture to obtain a better chance of success.

Finally, OpenMP is an explicitly programming language in which all of the parallel executions are under programmer's control. This explicit parallelism can also be used on the device GPU to offer very efficient codes.

OpenMP as a programming paradigm for heterogeneous systems has a lot of benefits, but it is still impossible to directly use OpenMP to solve the heterogeneous programming problem. On the one hand, the major reason is that OpenMP does not include a notion of non-uniform memory access times [Mattson et al., 2004], it is not well-suited to heterogeneous systems in which the host processor and the device co-processor have different memory spaces. On the other hand, heterogeneous systems have generally a nested multi-level parallelism. At the top level, parallelism can be found between the host CPU and the device co-processor. At the same time, deeper internal multi-level parallelism can also be found in the device. Unfortunately, OpenMP is not suitable for this particular nested multi-level parallelism.

However, the OpenMPC programming model attempts to take full advantage of the OpenMP programming model, and the OpenMPC compiler can interpret a standard OpenMP program into a CUDA program, which can be run on a GPU-based heterogeneous system.

3.4.2 Overview

To put it simply, OpenMPC is a translator, which can translate an OpenMP program into a CUDA program. It is based on the standard OpenMP API with a set of directives and environment variables, and it is a high-level programming interface for the abstraction of the CUDA programming model [Lee et al., 2009]. OpenMPC consists of a compilation system and a tuning system. First, the OpenMPC compiler analyses the OpenMP semantics, and the native OpenMP directives can help the compiler to find the possible candidates that can be interpreted into CUDA kernel functions. Then, the exclusive directives and environment variables from OpenMPC API can help the compiler for the final output code generation and optimization. The

OpenMPC compiler obtains a user-assisted tuning system as well. The OpenMPC directives and environment variables are used as the basis parameters of this tuning system for compile-time optimization [Lee and Eigenmann, 2012].

The following pragma can be used to assign a code candidate region, which will be translated into a CUDA kernel function:

```
#pragma cuda gpurun [clause [,] clause ]...]
```

The clauses can be inserted for managing thread batching, data mapping, and other optimizations. For example, `maxnumofblocks(Number)`, `threadblocksize(Number)`, `sharedRO(list)`, `constant(list)`, etc. A complete list can be found in [Lee and Eigenmann, 2010].

The OpenMPC API offers many environment variables for optimizations, including CUDA thread batching, data mapping, and code generation configurations for the output CUDA program. These environment variables can be inserted as compiler flags to assist OpenMPC compiler with the code generation.

Figure 3.7 shows an overall basic compilation flow of OpenMPC [Lee and Eigenmann, 2010]:

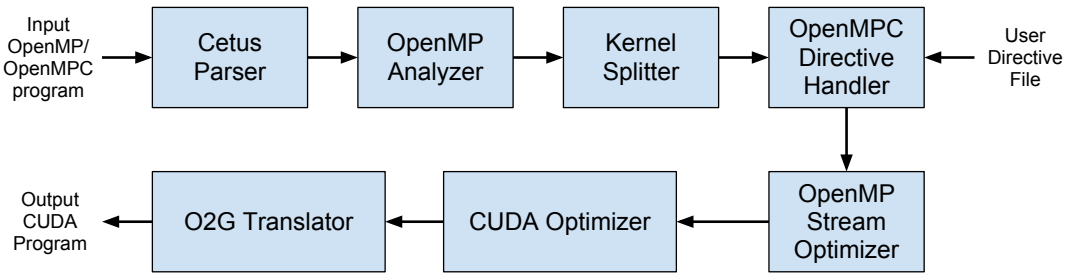


Figure 3.7: OpenMPC: overall basic compilation Flow

1. First of all, the Cetus Parser interprets an input OpenMP program, and it produces a Cetus internal representation.

2. Second, the OpenMP analyzer identifies all original OpenMP compiler directives, and then analyses the OpenMP semantics for synchronization.

3. Third, the Kernel Splitter splits the input program into kernel regions at every synchronization point.

4. Next, an `ainfo` directive is inserted at each kernel region is by the OpenMPC Directive Handler to give an ID, and a user directive file can also be analyzed.

5. Then, the OpenMP Stream Optimizer is an OpenMP-to-OpenMP translator, and the output program is an optimized OpenMP program.

6. After that, the CUDA Optimizer optimizes the OpenMP program with CUDA specifications.

7. Finally, the O_2G (OpenMP-to-GPGPU) translator generates an output CUDA program.

3.4.3 OpenMP-to-CUDA translation

In fact, only the last phase of the Figure 3.7, the O_2G (OpenMP-to-GPGPU) translator performs the real translation of an OpenMP program to a CUDA program. This translation is based on the following two steps, as shown in Figure 3.8 [Lee and Eigenmann, 2010]:

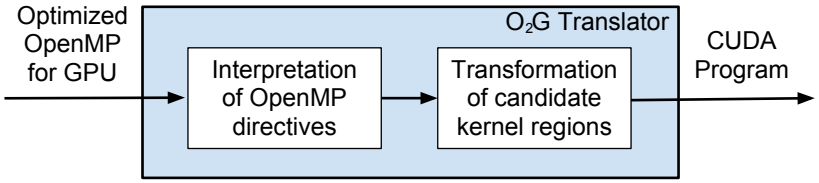


Figure 3.8: Two-step baseline translation of OpenMP into CUDA

- Interpretation of OpenMP directives under CUDA specifications.

  The first step is the foundation of the baseline translation, and the OpenMPC compiler interprets all the OpenMP code constructs under CUDA specifications. More importantly, some code regions such as OpenMP parallel constructs will be determined as the potential candidate kernel regions, which can be converted to CUDA kernel functions. Lee et al. (2009) have suggested that the OpenMP directives can be organized into four groups (Table 3.4): parallel constructs, work-sharing constructs, synchronization constructs, and directives specifying data properties [Lee et al., 2009], each of which can be converted into CUDA specification.

OpenMP directives	**CUDA specifications**
PARALLEL region construct `omp parallel`	Kernel functions
Work-sharing constructs `omp for`, `omp sections`	Split the work between the threads on GPU
Synchronization constructs `omp barrier`, `omp flush`, etc.	Split points, into two kernel functions (a global synchronization is necessary)
Directives specifying data properties `omp shared`, `omp private`,etc.	Data mapping into the address space of GPU

Table 3.4: Conversion from OpenMP constructs into CUDA specifications

- Transformation of candidate kernel regions

  After all potential candidate code regions are identified in the first step, the compiler begins to transfer these regions into CUDA kernels. Every OpenMP `omp parallel` construct is a main target that will be transferred into kernel functions. This process consists of two major steps as well:

  - Work partitioning

    Every `omp for` loop iteration and every section of `omp section` will be distributed to a CUDA thread on a device GPU. For mapping the threads on the device, the compiler determines the maximum number of threads that can execute together a kernel function. This amount of concurrent threads is organized as the number of thread blocks and the number of threads per thread block to match the work partitioning. Adding compiler directives or insetting flags into the compilation command can configure these two key parameters.

  - Data mapping

    Data mapping is another important step to convert an existing OpenMP program to a CUDA program. OpenMP directives, which specify data properties (for example, `omp shared`, `omp private`, or `omp threadprivate`), offer a great opportunity for data mapping, because they already contain some degree of data locality information between threads. For instance, OpenMP `private` data are private to each thread, they can be mapped to local memory or register in the CUDA memory model, or OpenMP `shared` data are shared among all threads, they can be mapped to global memory [Lee and Eigenmann, 2010].

3.4.4 Compiler optimizations

OpenMPC has developed a two-phase compiler optimization system [Lee et al., 2009] as shown in Figure 3.9.

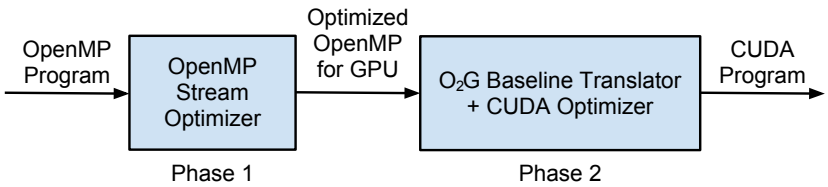


Figure 3.9: OpenMPC: two-phase OpenMP-to-CUDA translation

- OpenMP stream optimizer:

  This optimizer is an OpenMP-to-OpenMP source-to-source translator, which uses two compile-time optimization methods, parallel loop-swap transformation and parallel loop-collapsing transformation [Lee et al., 2009], respectively to exploit parallelism at the

loop level for regular applications, and to remove irregular control flow and irregular data access. With these two techniques, traditional OpenMP applications will be translated into optimized OpenMP programs to prepare for the next step conversion.

- O_2G (OpenMP-to-GPGPU) baseline translator and CUDA optimizer:

  The O_2G baseline translator translates an optimized OpenMP program from the OpenMP stream optimizer into a CUDA program. At the same time, O_2G CUDA optimizer uses different optimization methods to take full advantage of the CUDA specific features, such as data caching optimization for frequently-used data, matrix transpose for mapping thread private data, or reducing unnecessary data transfer between the host CPU and the device GPU [Lee et al., 2009]. These optimization techniques are based on a data-flow analysis, which is done by the compiler.

CHAPTER 4

Preparing experiments

This chapter introduces two experimental applications and the methodology of experiments in detail. Both applications are common matrix operations: one is solving of 2D Laplace's equation with Jacobi iterative method and the other is matrix multiplication. They are implemented and optimized by three high-level directive-based programming approaches described in Chapter 3: PGI Accelerator, HMPP Workbench, and OpenMPC source-to-source conversion. The current PGI Accelerator release supports both PGI and OpenACC specifications, because these two approaches are very similar to each other, and they could have similar performance considering using the same PGI compiler. Thus, OpenACC is not used to perform our experimental applications.

The methodology of experiments and the naïve sequential version of each experimental application are discussed in this chapter. The sequential version is used as the basis for further optimization using high-level programming approaches on a GPU-based hybrid experimental platform. For comparison, an OpenMP version running on the host CPU and a CUDA version running on the whole hybrid system are also implemented as well.

All the details of the experimental platform and the relevant compliers are also explained in this Chapter.

4.1 Methodology of experiments

During the evaluation process, each experiment composes of several steps for generating a well-organized study.

- The first step is the evaluation setup. It starts from a sequential version of an application kernel, and then, the sequential program is parallelized using both OpenMP and CUDA programming models for comparison and analysis. The OpenMP program is also used as the input source by evaluating OpenMPC programming approach.

- The next step is a baseline implementation of corresponding high-level programming approach specifications for heterogeneous systems. No optimization options are applied in this second step.
- Based on the second step, the next few steps focus on optimization processes for performance improvement.

The execution time, including data transfer overhead, is measured between the start and the completion of the kernel execution of each application. The C library routine `gettimeofday()` is called for determining the execution time. The performance of each high-level programming model for heterogeneous systems is compared with a corresponding hand-written CUDA version on the same problem size. In the first experiment, the total number of Jacobi iterations is set as a fixed number of 500. All the matrices used in the experiments are considered as square matrices with dimension `N` $\times$ `N`.

4.2 Explanation of experiments

4.2.1 Experiment 1: Solving of 2D Laplace's equation with Jacobi iterative method

The first experimental application is solving a second-order partial differential equation (PDE) - the Laplace's equation. It is an important example in many fields of scientific applications, for instance, astronomy, fluid potentials and electromagnetism. A simple example of Laplace's equation problem is the steady state heat conduction in a rectangular plate with the initial temperature on the critical boundary. Its mathematic expression can be written as:

$$\nabla^2 \varphi = 0, \tag{4.1}$$

where φ is a scalar function and ∇^2 is the Laplace operator.

In two dimensions, (4.1) can also be expressed as:

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = 0. \tag{4.2}$$

To solve a Laplace's equation, the basic technique is the relaxation method. Jacobi iteration, Gauss-Seidel and Successive Over Relaxation are basic iterative methods for Laplace's equations. In the first experiment, the Jacobi Iteration method is implemented, and the basic concept of this widely used method is the iterative convergence to correct values.

As can be seen from Figure 4.1, a new value for each point is computed from its four nearest neighbouring points during each iteration cycle. This is a typical stencil operation, and the Jacobi method can be expressed as following:

$$u_{new}(i,j) = \frac{u(i-1,j) + u(i+1,j) + u(i,j+1) + u(i,j-1)}{4}. \tag{4.3}$$

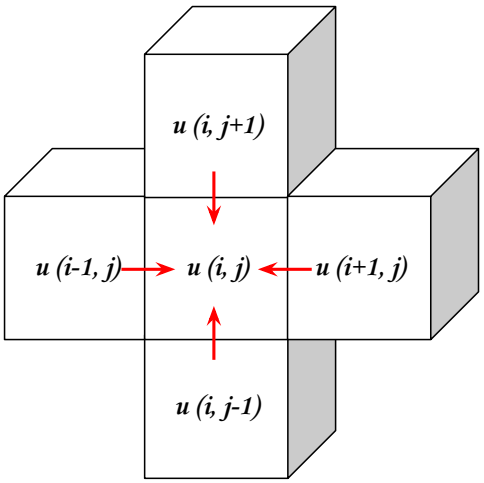


Figure 4.1: 2 dimensional Jacobi iteration method

This equation shows the calculation at each point is independent, thus, each point can be updated at the same time with the average of data from its neighbours. In this experiment, the input 2D array is created with initial values and fixed boundary conditions. During the iterations, all values are updated by using old values. For this reason, an extra 2D array is required for keeping all new values.

An example implementation of Jacobi method is shown in Figure 4.2. For this serial version algorithm, the computational complexity is $\mathcal{O}(N^2 \times iter)$, where N is the input size, and $iter$ is the number of iterations. The matrix `a` is updated during each iteration. This Jacobi pseudocode contains 3 nested loops, and it is obvious that there is a data access conflict between each iteration of the outermost `iter` loop (the line 4), thus, it cannot be parallelized because of the loop-carried dependency. However, the inner `i` and `j` loops are parallelizable. It can be noticed that there is a copy overhead in this simple implementation, after computing new values (line 7 to line 11), and there is an element-wise copy (line 14 to line 18) which causes a performance bottleneck. This `jacobi` function is a suitable candidate for acceleration on a GPU-assisted heterogeneous system, and it is the main target during the evaluating process.

The OpenMP programming model is relatively easy to use, and programmers only need to add OpenMP compiler directives to the sequential program. Figure 4.3 shows the OpenMP implementation of Jacobi method. Both inner nested i, j loops are parallelizable, thus, two compiler directives `#pragma omp parallel for shared(a, a_temp) private(i,j)` are added over the i loops to indicate the following blocks must be executed in parallel. In addition, the variable `a` and `a_temp` are declared as shared variables, and the variables `i` and `j` are declared as private variables among the threads.

The initial hand-written CUDA version is implemented with multiple thread blocks, and the input matrix is partitioned into smaller blocks. In practice, only one block of threads is often not enough for large input data in the CUDA programming model. Thus, developers are required to partition a computation into many blocks of threads. In the CUDA programming model, all

```
1  void jacobi(float a[N][N], float a_temp[N][N]){
2     int i, j, iter;
3
4     for(iter = 0; iter < MAX_ITER; iter++){
5
6        // compute new values
7        for (i = 1; i < N-1; i++) {
8           for (j = 1; j < N-1; j++){
9              a_temp[i][j] = 0.25 * (a[i-1][j] + a[i+1][j] + a[i][j+1]
                  + a[i][j-1]);
10          }
11       }
12
13       // swap a and a_temp, copy overhead
14       for (i = 1; i < N-1; i++) {
15          for (j = 1; j < N-1; j++){
16             a[i][j] = a_temp[i][j];
17          }
18       }
19    }
20
21 }
```

Figure 4.2: Pseudocode for solving 2D Laplace's equation using Jacobi method (sequential version)

```
for(iter = 0; iter < MAX_ITER; iter++){

   // compute new values
#pragma omp parallel for shared(a, a_temp) private(i,j)
   for (i = 1; i < N-1; i++) {
      for (j = 1; j < N-1; j++){
         a_temp[i][j] = 0.25 * (a[i-1][j] + a[i+1][j] + a[i][j+1] + a
             [i][j-1]);
      }
   }

   // swap a and a_temp, copy overhead
#pragma omp parallel for shared(a, a_temp) private(i,j)
   for (i = 1; i < N-1; i++) {
      for (j = 1; j < N-1; j++){
         a[i][j] = a_temp[i][j];
      }
   }

}
```

Figure 4.3: OpenMP implementation of Jacobi method

threads are organized by a grid of many thread blocks. Each thread deals with only a small portion of an input data. To make it more intuitive, as a simple example, Figure 4.4 shows that a whole matrix is partitioned into 4 submatrices. Each CUDA thread block handles only one submatrix, and each thread within a thread block calculates only one element of the submatrix. The coordinates of a thread can be used to identify the position of an element, for example, the element (12, 13) will be calculated by the thread id (4, 5) in the block (1, 1).

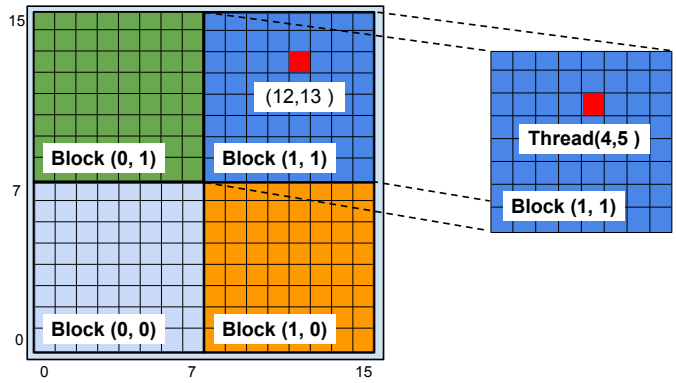


Figure 4.4: Example of a 4 block-decomposed submatrix block.

In fact, the number of thread blocks as well as threads within a thread block are dependent on the input size. However, choosing a global optimal thread block size is critical, the limits of shared memory and registers are mixed together with the coalescing global memory access. In order to simplify the problem, matrices are considered as square matrices that have dimensions divisible by 16, and fixed thread block size is used in the CUDA implementation. The input matrix is divided into `BLOCK_SIZE` × `BLOCK_SIZE` submatrices, where the `BLOCK_SIZE` is set to 16. Thus, each thread block has a total of 256 threads, and each submatrix is computed by one thread block of threads. Each element of the original input matrix can be identified using `blockIdx` and `threadIdx`. In addition, this CUDA implementation does not use shared memory, and all threads directly read values from the device global memory.

Figure 4.5 shows the CUDA code structure with data transfer for solving 2D Laplace equation by using Jacobi algorithm. Compared to the OpenMP version, this CUDA implementation is much more complicated. In the `main` function, the `cudaMalloc()` function is used to allocate memory on the device memory, and the `cudaMemcpy()` is used for explicit data transfers between the host CPU and the device GPU. The matrix `a` is transferred from the host memory to the device memory before the Jacobi iteration begins. Two kernel functions, `parallel_jacobi_kernel` for calculating new values and `jacobi_update_kernel` for swapping values, are launched in each iteration. During the iterations, there are no data movements between the host and the device. After the completion of the last iteration, the result matrix `gpu__a` is copied back from the device memory to the host memory.

```
#define N 2048  //dimension of input matrix NxN
#define BLOCK_SIZE 16  //thread block size
#define MAX_ITER 500 //number of iterations

__global__ void parallel_jacobi_kernel(float* a, float* b){  //compute new values
   int bx = blockIdx.x;  //block index
   int by = blockIdx.y;

   int tx = threadIdx.x;  //thread index
   int ty = threadIdx.y;

   int row = by*BLOCK_SIZE + ty;  //row index of input matrix
   int col = bx*BLOCK_SIZE + tx;  //column index of input matrix

   if(row == 0 || row == N-1 || col == 0 || col == N-1){//dirichlet boundary condition
      b[row*N + col] = a[row*N + col];
   }else{
      b[row*N + col] = 0.25f*(a[row*N + col -1] + a[row*N + col + 1] + a[(row+1)*N + col] + a
           [(row-1)*N + col]) ;   //compute new values
   }
}

__global__ void jacobi_update_kernel(float* a, float* b){  //swap values
   int bx = blockIdx.x;  //block index
   int by = blockIdx.y;

   int tx = threadIdx.x;  //thread index
   int ty = threadIdx.y;

   int row = by*BLOCK_SIZE + ty;  //row index of input matrix
   int col = bx*BLOCK_SIZE + tx;  //column index of input matrix

   a[row*N + col] = b[row*N + col];  //swap values
}

void main(){
   ...

   //number of blocks of each dimension of grid, for example, nBlocks = 2048/16=128
   int nBlocks = N / BLOCK_SIZE;

   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE); //2D (16x16) block
   dim3 dimGrid(nBlocks, nBlocks);  //2D (nBlocks x nBlocks) grid

   cudaMalloc((void **) &gpu__a, size);  //allocate gpu__a in the device global memory
   cudaMalloc((void **) &gpu__tmp, size); //allocate gpu__tmp in the device global memory

   //copy the a matrix from the host memory to the gpu__a matrix in the device memory
   cudaMemcpy(gpu__a, a, size, cudaMemcpyHostToDevice);

   for(iter =0; iter < MAX_ITER; iter++){
      parallel_jacobi_kernel<<<dimGrid, dimBlock>>>(gpu__a, gpu__tmp);  //launch kernel
      jacobi_update_kernel<<<dimGrid, dimBlock>>>(gpu__a, gpu__tmp);  //launch kernel
   }

   //copy the gpu__a matrix from the device memory to the a matrix in the host memory
   cudaMemcpy(a, gpu__a, size, cudaMemcpyDeviceToHost);
   ...
}
```

Figure 4.5: Code section from Jacobi CUDA implementation

4.2.2 Experiment 2: Matrix multiplication

The second experimental application is matrix multiplication, which is one of most widely used application in many scientific applications or benchmarks. This application kernel is a fundamental operation in high-performance computing and scientific computing. It is also a good example to show the advantage of parallel programming.

In this experiment, matrices are considered as large dense N-by-N square matrices. As can be seen from Figure 4.6, for a product of two matrices $c = a \times b$, each cell of the result matrix $c_{i,j}$ is computed as a dot product of matching rows from matrix a and matching columns from matrix b.

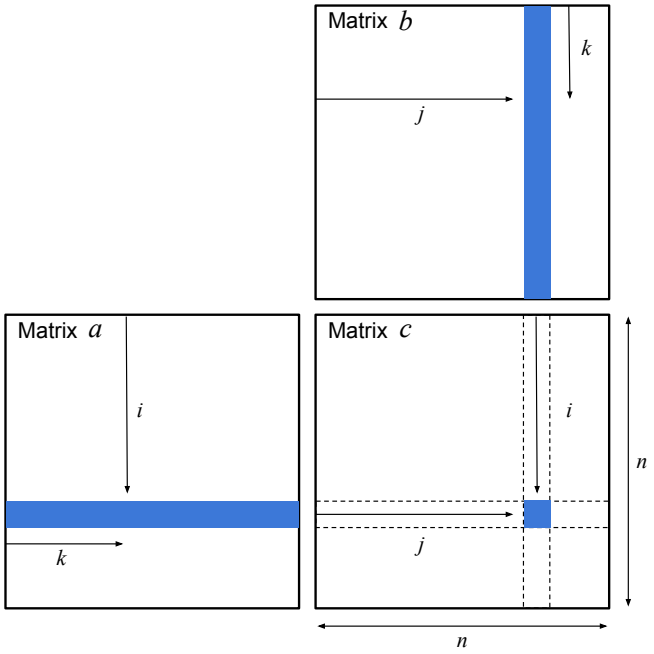


Figure 4.6: Matrix multiplication $c = a \times b$.

Its mathematical expression is oft written as:

$$\begin{aligned} c_{i,j} &= a_{i,1} \times b_{1,j} + a_{i,2} \times b_{2,j} + \cdots + a_{i,n} \times b_{n,j} \\ &= \sum_{k=1}^{n} a_{i,k} \times b_{k,j}, \end{aligned} \tag{4.4}$$

where, $1 \leq i \leq n$ and $1 \leq j \leq n$.

Figure 4.7 shows an example implementation of the matrix multiplication kernel that consists of three nested loops. This nested loop construct has a computational complexity $\mathcal{O}(N^3)$. The sequential version of the matrix multiplication suffers not only from high computational

```
1  void matrix_multiplication(float a[N][N],float b[N][N],float c[N][N]){
2
3     int i, j, k;
4
5     for (i = 0; i < N; i++) {
6        for (j = 0; j < N; j++) {
7           for (k = 0; k < N; k++){
8              c[i][j] += a[i][k]*b[k][j];
9           }
10       }
11    }
12 }
```

Figure 4.7: Pseudocode for Matrix multiplication(sequential version).

cost, but also from the poor memory locality. This simple algorithm can be optimized by parallel execution to achieve high performance. As shown in Figure 4.7, the `i` and `j` loops can be executed in parallel. This application kernel is a good candidate in the second experiment as well. Based on the sequential version, the OpenMP implementation of matrix multiplication is shown in Figure 4.8. This OpenMP version is used as the input source for OpenMPC source-to-source compiler during the second experiments. The CUDA implementation is based on the tiled matrix multiplication example implemented by Kirk and Hwu, further details can be found in [Kirk and Hwu, 2010]. This CUDA version is an optimized implementation, which takes the advantage of the shared memories.

```
1  void matrix_multiplication(float a[N][N],float b[N][N],float c[N][N]){
2
3     int i, j, k;
4
5  #pragma omp parallel for shared(a,b,c) private(i,j,k)
6     for (i = 0; i < N; i++) {
7        for (j = 0; j < N; j++) {
8           for (k = 0; k < N; k++){
9              c[i][j] += a[i][k]*b[k][j];
10          }
11       }
12    }
13 }
```

Figure 4.8: OpenMP implementation of Matrix multiplication

4.3 Testing environments

The experimental platform is a CPU-GPU heterogeneous system (Red Hat Linux (4.1.2-50)). All the tests are launched on the same workstation, which contains two Intel Xeon quad-core X5550 with 2.66 GHz and a NVIDIA Tesla C2050 card (Fermi). The detailed hardware specifications of the experimental platform are listed in Table 4.1. The experimental workstation is one node of the CORA GPU cluster [1].

	CPU **Intel Xeon X5550**	**GPU** **NVIDIA Tesla C2050**
Cores	4	448
Code name	Nehalem	Fermi
Clock rate (GHz)	2.66	1.15
Number of threads	8	14 (SM) $\times$ 1536 = 21504
Cache	8 MB	64 KB L1 (32 cores) 768 KB L2 cache
Single precision peak performance (GFLOPS)	85.1	1030
Double precision peak performance (GFLOPS)	42.6	515
Memory bandwidth (GB/sec.)	32	144
Power consumption (W)	95	238

Table 4.1: Hardware specifications during the experiments

The software tools and compiler toolkits are listed in Table 4.2. The experiments in the multicore environment are performed by using Intel icc 13.0.0, and multithreading is realized with OpenMP. In the heterogeneous environment, PGI Accelerator compiler 12.8, HMPP workbench 3.0.5 and OpenMPC compiler 0.3.1 are used to evaluate.

Programming model	**Compiler**
PGI Accelerator	PGI Release 2012 version 12.8
HMPP Workbench	HMPP version 3.0.5
OpenMPC	OpenMPC Release 0.3.1
CUDA	nvcc version 4.2
C & OpenMP	icc version 13.0.0

Table 4.2: Software tools

[1] http://cora.gridlab.univie.ac.at

CHAPTER 5

Measurements and discussion of experiments

In this Chapter, different representative directive-based high-level parallel programming approaches for heterogeneous systems - PGI Accelerator, HMPP Workbench, and OpenMPC source-to-source compiler - are evaluated through a stepwise implementation of two computationally intensive applications described in Chapter 4. Moreover, the test results are discussed in this Chapter as well.

5.1 Evaluation of the experiment 1

In the first experiment, the problem of 2D Laplace's equation is solved with the Jacobi iterative method. The sequential version of this method (Figure 4.2) used in this evaluation is optimized with different high-level parallel programming approaches in the experimental target machine architecture.

Evaluation setup: OpenMP and CUDA implementation

In the evaluation setup, an OpenMP version and a CUDA version of this application are implemented for comparison in advance. Table 5.1 indicates a performance comparison of using OpenMP and CUDA in this setup. It is apparent that the hand-written CUDA version has a better performance than the multithreading OpenMP version with 16 threads. These initial data are used to compare against other high-level approaches, and the table is updated during the experiment.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92

Table 5.1: Experiment 1: Performance results of the OpenMP and CUDA implementation at evaluation setup, matrix size: 2048, iteration number: 500

PGI variant 1: Baseline implementation

The first variant is a baseline implementation without any optimizations, and only the basic PGI compiler directives are applied. At the beginning of the PGI implementation, programmers have to determine where to add PGI compiler directives to the sequential code (Figure 4.2), and then, they manually insert the PGI compiler directives into the sequential program. In this case, the nested loops are the main targets. Mostly, the nested loop structure is the potential parallelizable code region, which can be offloaded to the device GPU.

It can be seen from Figure 5.1 that only two pragmas (marked with a grey shaded area), `#pragma acc region`, are inserted into the sequential code to specify two PGI compute regions. PGI compute region is the only required compiler directive [Wolfe, 2010] in the PGI Accelerator programming model. The PGI compiler will map the loop body within a compute region as a kernel function. In this example, the compiler will generate two kernel functions for these two compute regions.

```
for(iter = 0; iter < 500; iter++){

#pragma acc region
    for (i = 1; i < N-1; i++) {
        for (j = 1; j < N-1; j++){
            a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][j+1] +
                a[i][j-1]);
        }
    }

#pragma acc region
    for (i = 1; i < N-1; i++) {
        for (j = 1; j < N-1; j++){
            a[i][j] = a_temp[i][j];
        }
    }

}
```

Figure 5.1: Experiment 1: PGI Accelerator implementation - variant 1

The PGI compiler provides a series of compiler options. The option `-Minfo` is a useful op-

tion, which enables a detailed compiler-to-user feedback at compile time. The feedback enables programmers to determine whether the loops are parallelizable or not, and whether a compute region is translated into a kernel function or not. In addition, it also indicates the possible performance issue for further optimizations.

Figure 5.2 shows the compiler-to-user feedback in the PGI variant 1. The first two messages `Generating copyin` and `Generating copyout` indicate that the matrix `a` needs to be copied from the host memory to the device memory before launching the kernel function, and the matrix `a_temp` needs to be copied back after the kernel execution. The message `Loop is parallelizable` indicates that each iteration of a loop can be executed in parallel. According to this feedback, the compiler determines that all the nested loops in compute regions are parallelizable. The message `Accelerator kernel generated` is an important message in this feedback, which indicates that a user-defined compute region is successfully transferred into a kernel function for the device GPU. In this example, this message appears twice, so two kernel functions are generated from the compiler. The message `#pragma acc loop gang` and the message `#pragma acc loop gang, vector(128)` indicate the loop schedule, which specifies the mapping of loop-level parallelism onto the device GPU. In addition, there is a vector width argument, which is set to 128 by the compiler. The vector width argument corresponds to the thread block size here. The `gang` corresponds to the block in CUDA model, and `vector` corresponds to CUDA thread.

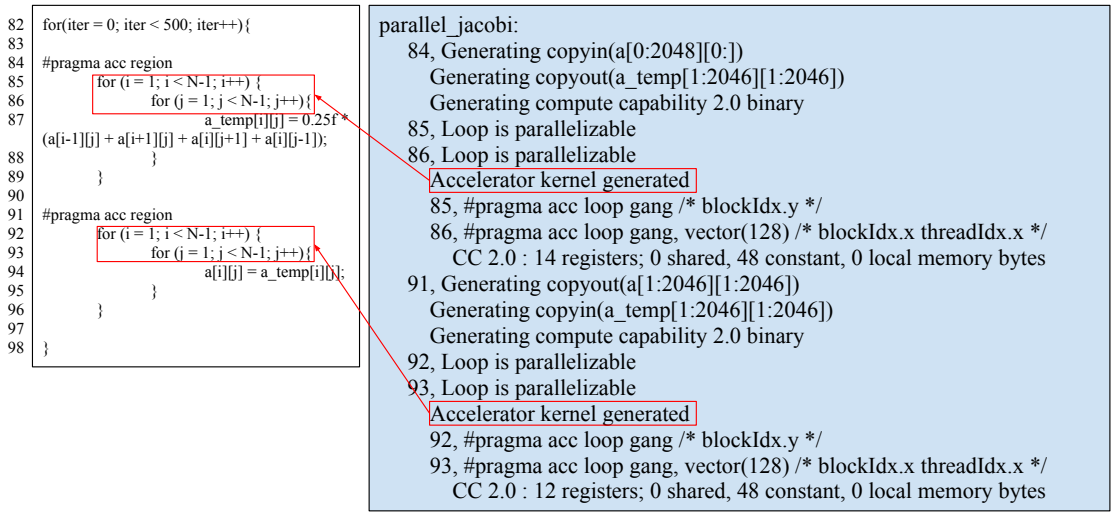


Figure 5.2: Experiment 1: PGI compiler-to-user feedback - variant 1

In the first step, the PGI compiler determines and controls all data allocations and movements. This can significantly save the development time, but this does not always lead to superior performance. Sometimes, this automatic operation can cause unnecessary performance degradation because of useless data transfers. Figure 5.3 shows the data transfers in this baseline implementation. In each iteration of the `iter` for loop, the matrix `a` is first copied from the host to device, after launching the first generated kernel function, the matrix `a_temp` is copied

back from the device to the host. Then, the `a_temp` is copied again from the host to the device before launching the second kernel function, and the `a` is transferred back again. These above four steps repeat in each iteration of the `iter` loop, and the total number of data transfers is dependent on the number of iterations in the first variant. For 500 iterations, there is a total of 2000 times of data transfers between the host and the device. This data transfer overhead is an obvious performance bottleneck.

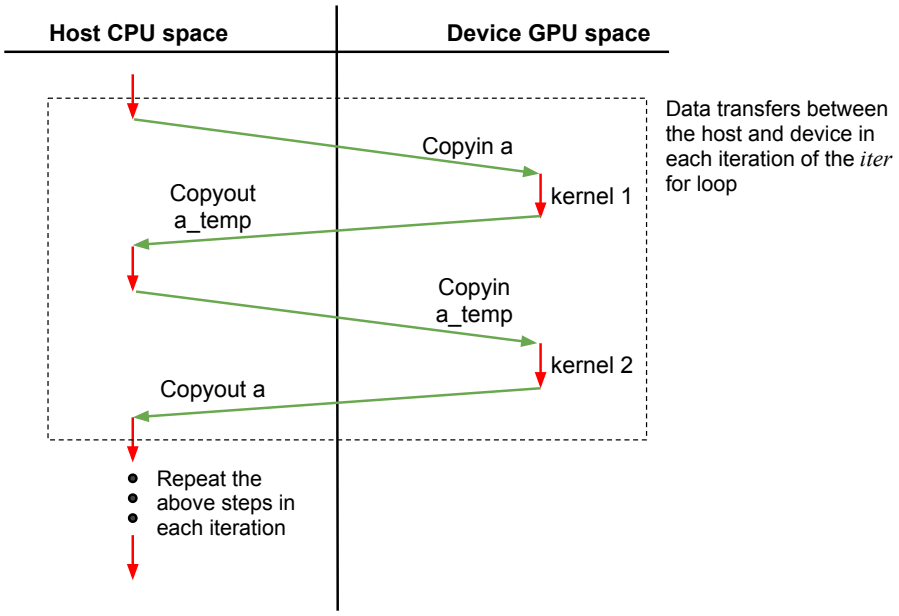


Figure 5.3: Experiment 1: 2000 times of Data transfers between the host CPU memory and the device GPU memory - PGI variant 1

Table 5.2 shows the performance results of the PGI Accelerator baseline implementation in the first variant. The PGI Accelerator implementation works extremely slow, and this parallel execution has almost lost its meaning. As pointed out, this version is very inefficient. The reason for this dramatic performance drop is the large data transfer overhead, because unnecessary data movements occur during each iteration.

Version	**Execution time (second)**	**Speedup**
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22

Table 5.2: Experiment 1: Performance results after PGI implementation in variant 1, matrix size: 2048, iteration number: 500

After the execution of the program, programmers can have further profile information. Figure 5.4 shows the PGI profile timing information. The compiler finds two compute regions, which are generated into two kernel functions, and each compute region is entered 500 times. The first compute region at the line 84 takes a total of 7.649 seconds, only 353639 μs (microsecond) are spent on executing the kernel function, and 6.976612 seconds are spent on moving data. The second compute region at the line 91 takes a total of 9.816 seconds, only 285314 μs are spent on executing its kernel, but about 9.352 seconds are spent on moving data. According to this provided information, a total of 16 seconds are spent on moving data between the host and the device. This is about 96% of the total execution time, and only 4% is spent on kernel execution. This profile timing information shows an unbalanced workload and the major performance problem in the first implementation, the large overhead of data transfers.

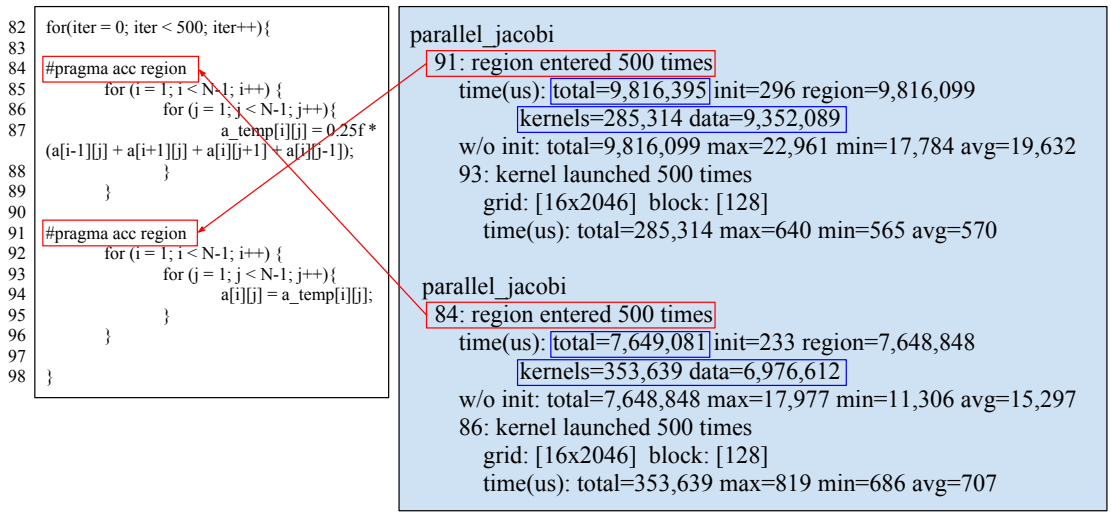


Figure 5.4: Experiment 1: PGI profile timing information - variant 1

PGI variant 2: Reducing data transfer costs

The second variant of PGI Accelerator implementation is an optimized version, which addresses the performance bottleneck of the previous step. In the second variant, data allocations and movements are no longer determined by the compiler. The PGI data region is applied to reduce unnecessary data movement.

In this variant, two strategies are used to improve the whole performance: the first is reducing the frequency of unnecessary data transfer, and the second is avoiding useless data copying. In this example, only the matrix `a` is required to be transferred between the host and the device. It needs to be copied to the device memory before reaching the `iter` for loop, and after the computation on the device, it needs to be copied back. The matrix `a_temp` is used to temporarily store the new values temporarily, thus, for the matrix `a_temp`, only the data allocation in the device is required.

Figure 5.5 shows the resulting code with a data region directive `#pragma acc data region` (marked with a grey shaded area) to obtain efficient data movements. This optional compiler directive defines a PGI data region that contains two PGI compute regions. The data clause `copy` instructs the compiler to allocate the matrix `a` in the device memory, and then, the matrix `a` is copied from the host to the device memory, when it reaches the data region. When it leaves this data region, the matrix `a` needs to be copied back from the device to the host. The data clause `local` instructs the compiler to allocate the matrix `a_temp` on the device memory only.

```
#pragma acc data region copy(a[N][N]) local(a_temp[N][N])
{
    for(iter = 0; iter < 500; iter++){

#pragma acc region
        for (i = 1; i < N-1; i++) {
            for (j = 1; j < N-1; j++){
                a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][
                    j+1] + a[i][j-1]);
            }
        }

#pragma acc region
        for (i = 1; i < N-1; i++) {
            for (j = 1; j < N-1; j++){
                a[i][j] = a_temp[i][j];
            }
        }
    }
}
```

Figure 5.5: Experiment 1: PGI Accelerator implementation - variant 2

Figure 5.6 shows the compiler-to-user feedback in the second PGI implementation. Compared with the feedback obtained from the first variant (Figure 5.2), there are two new messages, which are marked with a blue frame. They show the data transferring in this variant. The rest information of this feedback is identical to the first variant. The message `Accelerator kernel generated`, which appears twice, indicates that two kernel functions are generated.

In the second variant, the PGI data region directive is applied to determine and control all data allocations and transfers. Figure 5.7 shows the data transfers in the second variant implementation. As shown in this figure, the dada transfers happens only twice now, before the first iteration and after the last iteration. In the first variant, there are 2000 times of data movements between the host and device. In this new variant, the total number of data transfers does not depend on the number of iterations of the `iter` loop, and moreover, needed data are kept in device memory after the computation on the device begins.

Table 5.3 shows a performance improvement of PGI Accelerator implementation in the second variant. The execution time PGI variant 2 is reduced from 17.467 seconds to only 0.684

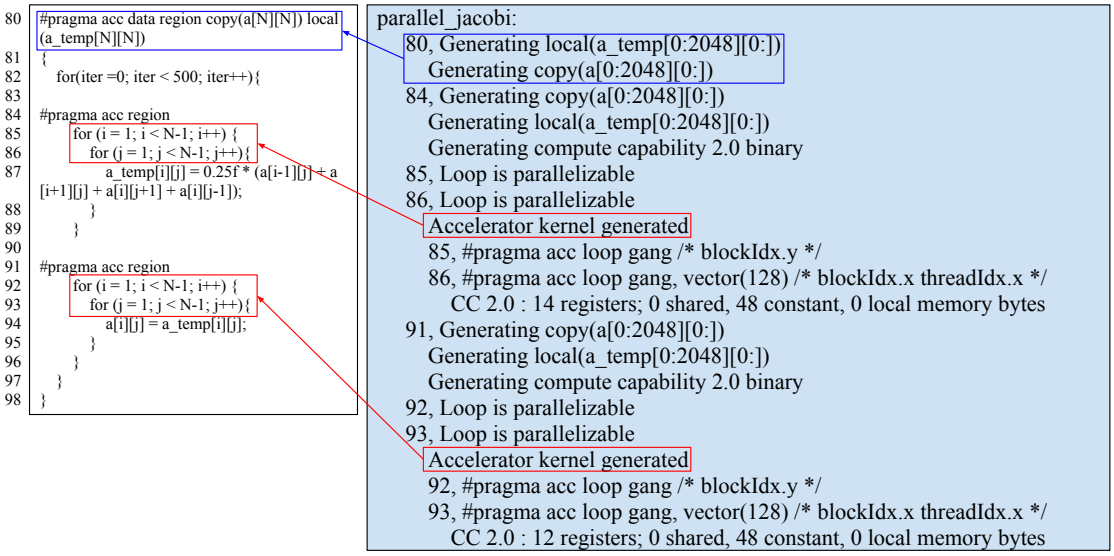


Figure 5.6: Experiment 1: PGI compiler-to-user feedback - variant 2

seconds, which means there is a large increase in performance. The speedup over the sequential version is increased from 0.22x to 5.55x. Meanwhile, the PGI variant 2 achieves a nearly similar performance to the hand-written CUDA version. The main reason for this performance improvement is reducing data movement costs by creating a beneficial data region.

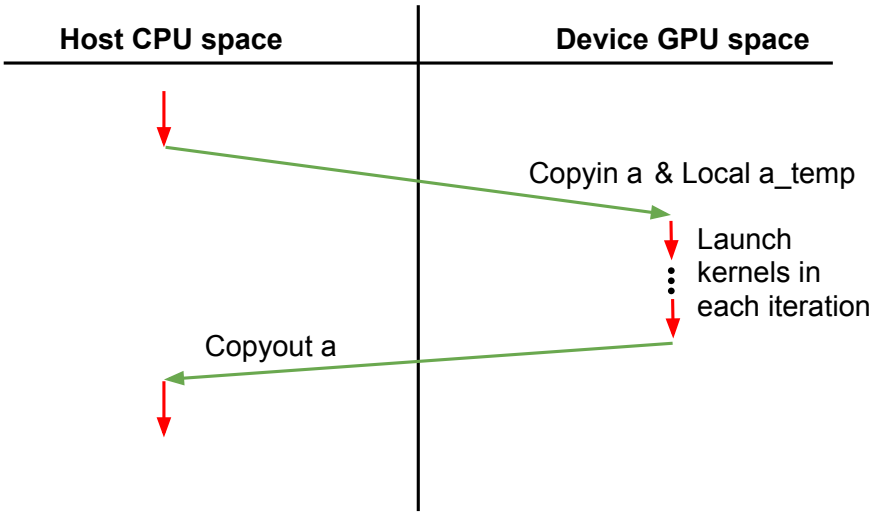


Figure 5.7: Experiment 1: Data transfers between the host CPU memory and the device GPU memory - PGI variant 2

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55

Table 5.3: Experiment 1: Performance results after PGI implementation in variant 2, matrix size: 2048, iteration number: 500

Figure 5.8 illustrates the profile timing information in the second variant. The user-defined data region at line 80 is entered only once, and the time spent in data transfer significantly drops from 16 seconds to only 13838 μs. This is a large difference between first and second variants. This figure also shows that both these two compute regions (line 84 and line 91) are entered 500 times, but there are no data movements within these compute kernels now. The kernel execution time remains almost unchanged.

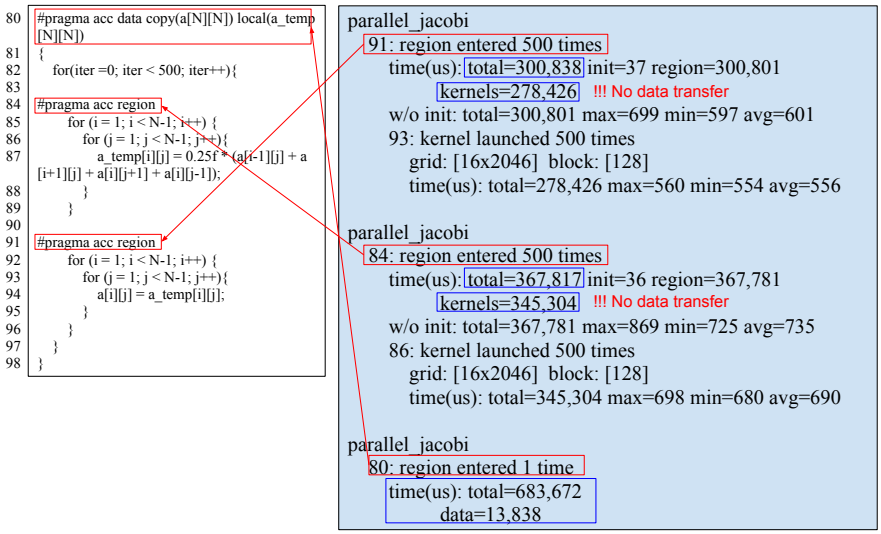


Figure 5.8: Experiment 1: PGI profile timing information - variant 2

PGI variant 3: Tuning the loop schedules

On the basis of the previous step, the implementation of the PGI third variant is a further optimized version, which tunes the loop scheduling in order to enhance its performance. By default, the PGI Planner will translate loop indices into CUDA grid and thread indices. In fact, the loop mapping process is complicated, and choosing a suitable grid and block size is also not easy. The grid and block size are two critical optimization parameters. They are related not only to

the size and structure of input data, but also to targeted hardware architecture. The PGI compiler can choose a grid and thread block size automatically, but trying different sizes is still worth to observe their impact on the application performance.

In the third implementation, different loop schedules are tested manually by using the PGI loop mapping directive. The vector width argument is the main target in the tuning process. The vector width argument is an important influence factor of the performance tuning, but unfortunately, there is no definitive answer to choosing an appropriate vector width. The best practical proposal begins with the PGI compiler-to-user feedback and profile timing information, then fine-tune the automatically determined vector width argument to observe its impact.

Determining a global optimal vector width in PGI is complicated, because many performance factors can be influenced by it, for example, the usage of on-chip shared memory and register, off-chip global memory access, and different types of compiler optimization. To choose a suitable vector width, there are two following basic principles:

- First and foremost, make sure that vector width argument must assist the loop mapping on a device with an appropriate thread block size, which should always be a multiple of warp size (32 parallel threads). In the CUDA model, threads in a thread block are divided into warps. For instance, if a block size is set to 40, two warps (64 threads) must be scheduled, so 24 threads will be wasted.

- Second, vector width argument should lead a thread block size, which can generally achieve high occupancy to keep the device busy and to hide latency. Occupancy can be basically considered as the utilization of the device GPU. However, this can be tricky sometimes, because higher occupancy rate does not always guarantee high application performance. Better performance can also be achieved by applications at lower occupancy that attain instruction-level parallelism [Farber, 2011]. The CUDA Occupancy Calculator[1] can help developers to investigate occupancy and determine vector width or CUDA thread block size.

Figure 5.9 shows the resulting code in this implementation and two new additional lines (marked with a grey shaded area) are added to the code. The PGI loop mapping directive `#pragma acc for` is used here to tune the code and it gives programmers more freedom to manage the loop mapping process on the device GPU, for instance, to describe the type of parallelism or to determine the number of threads per block. More information about this mapping directive and its clauses can be found in [The Portland Group Inc., 2010]. The loop scheduling clause `vector` with a width argument indicates that the immediately following `j` loop will be run in a vector model (synchronous thread-level parallelism between threads in a block) on the device GPU. During the tuning process, the vector width is presently set to 64, 256, 512 and 1024 (1024 is the maximum number of threads per block for our experimental platform). Without the loop vector directive, the vector width will be determined by the compiler. Just like first and second variants, the vector width or thread block size is set to 128 by the PGI compiler.

Table 5.4 shows that five different vector width arguments can lead to different occupancy and execution time. The vector width of 256 and 512 can provide an occupancy of 100%,

[1] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

```
#pragma acc data copy(a[N][N]) local(a_temp[N][N]){
    for(iter =0; iter < MAX_ITER; iter++){

#pragma acc region
        for (i = 1; i < N-1; i++) {
#pragma acc for vector(256)
            for (j = 1; j < N-1; j++){
                a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][
                    j+1] + a[i][j-1]);
            }
        }

#pragma acc region
        for (i = 1; i < N-1; i++) {
#pragma acc for vector(256)
            for (j = 1; j < N-1; j++){
                a[i][j] = a_temp[i][j];
            }
        }
    }
}
```

Figure 5.9: Experiment 1: PGI Accelerator implementation - variant 3

and both of them achieve better performance than others. In variant 1 and 2, the vector width argument is set to 128 by the compiler, and it can achieve an occupancy of 67%. According to the PGI compiler-to-user feedback (Figure 5.6), the CUDA threads are organized as a 2-D grid of 1-D thread blocks. These loop schedules can be expressed as execution configuration parameters in CUDA model, in other words, they can be indicated as <<<dimGrid(N/Vec, N), dimBlock(Vec)>>>, where N is the dimension of the input square matrix, and Vec is the vector width. These experimental results indicate that there is a direct correlation between vector width argument and occupancy. The vector width argument, which achieves higher occupancy, is generally more efficient.

Vector width argument	Execution time (second)	Occupancy
64	0.970	33%
128 (Variant 2)	0.684	67%
256	**0.630**	100%
512	0.639	100%
1024	0.862	67%

Table 5.4: Execution time and occupancy of jacobi method with different PGI vector width arguments

Table 5.5 shows a further performance improvement in variant 3 by using PGI loop mapping directive. The execution time of PGI variant 3 is reduced to 0.630 seconds, and the speedup

increases up to 6.03x.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03

Table 5.5: Experiment 1: Performance results after PGI implementation in variant 3, matrix size: 2048, iteration number: 500

HMPP variant 1: Baseline implementation

The first variant of HMPP Workbench implementation is a fundamental version without optimizations. Only the essential required HMPP compiler directives are inserted into the sequential code (Figure 4.2). Instead of targeting parallelizable loop constructs, computationally intensive functions are the main objective in the HMPP programming model.

Figure 5.10 illustrates the code that consists of a pair of HMPP `codelet` / `callsite` directives. It can be seen from this figure, the function `parallel_jacobi()` is marked with `codelet` directive with the label `jacobi`. In the HMPP programming model, a function, which will be offloaded to an accelerator, is described as a codelet. The clause `target=CUDA` tells the compiler to generate CUDA code for the targeted experimental platform. The last clause `arg[*].transfer=atcall` tells the compiler to transfer all function arguments before and after launching this `codelet`. This is a simple policy, because all arguments from the codelet must be transferred without exceptions. For the invocation of this codelet, in the main function, the codelet `parallel_jacobi()` caller is marked with `hmpp jacobi callsite` directive with the same label `jacobi`.

The HMPP compiler can provide compiler-to-user feedback as well, but this information is rough compared to the PGI compiler's feedback. Figure 5.11 indicates the diagnostic analysis report from the HMPP compiler. Three main points emerge from the feedback. First, the codelet is successfully translated into CUDA kernel functions, which is written in the compiler-generated CUDA file `jacobi_cuda.hmc.cu`. Second, the `iter` loop cannot be "gridified", which means this loop cannot be parallelized for the device GPU, because of inter-iterations dependencies. Lastly, the inner `i` and `j` loops are parallelized for the device GPU, and two kernel functions can be launched with a 2-dimensional grid of thread blocks.

Table 5.6 shows the performance comparison of the HMPP Workbench baseline implementation with other implementations. As shown in the table, the execution time of the HMPP first variant implementation is better than the PGI baseline variant, but not as good as the naïve CUDA and the optimized PGI third variant version. However, it easily achieves a speedup up to 4.77x over the sequential version.

```
#pragma hmpp jacobi codelet, target=CUDA, args[*].transfer=atcall
void parallel_jacobi(float a[N][N],float a_temp[N][N]){

   for(iter = 0; iter < MAX_ITER; iter++){

      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][j+1] + a[i][j-1]);
         }
      }

      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a[i][j] = a_temp[i][j];
         }
      }

   }
}

int main(int argc, char **argv){
   ...

#pragma hmpp jacobi callsite
   parallel_jacobi(a_cpu, a_temp_cpu);

   ...
}
```

Figure 5.10: Experiment 1: HMPP Workbench implementation - variant 1

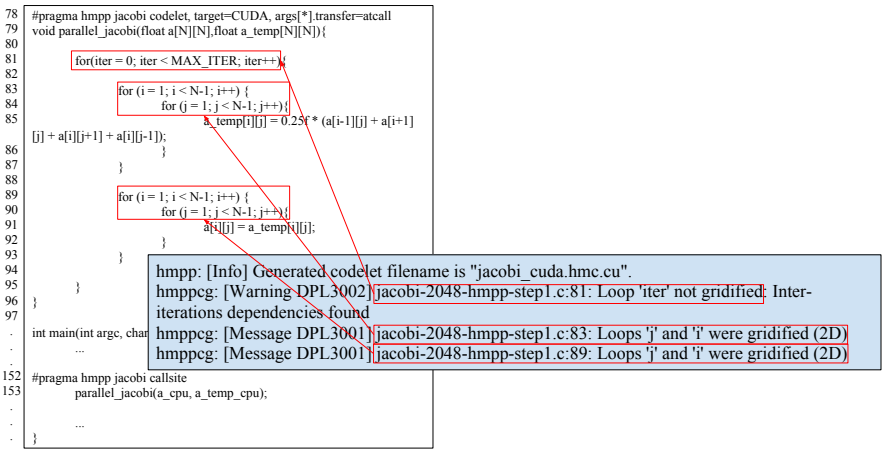


Figure 5.11: Experiment 1: HMPP compiler-to-user feedback - variant 1

After exporting the HMPP environment variable `HMPPRT_LOG_LEVEL=INFO`, a runtime log can be generated from the execution of the code. Figure 5.12 shows the HMPP runtime log in the baseline variant. It demonstrates the complete codelet remote procedure call processing flow for the first HMPP code version. More importantly, this information can help programmers to find the opportunities for further potential optimization. For example, there is an obvious problem: the matrix `a_temp` is uselessly transferred between the CPU memory and GPU mem-

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77

Table 5.6: Experiment 1: Performance results after HMPP implementation in variant 1, matrix size: 2048, iteration number: 500

ory. This will cause a performance drop due to the unneeded data transfer.

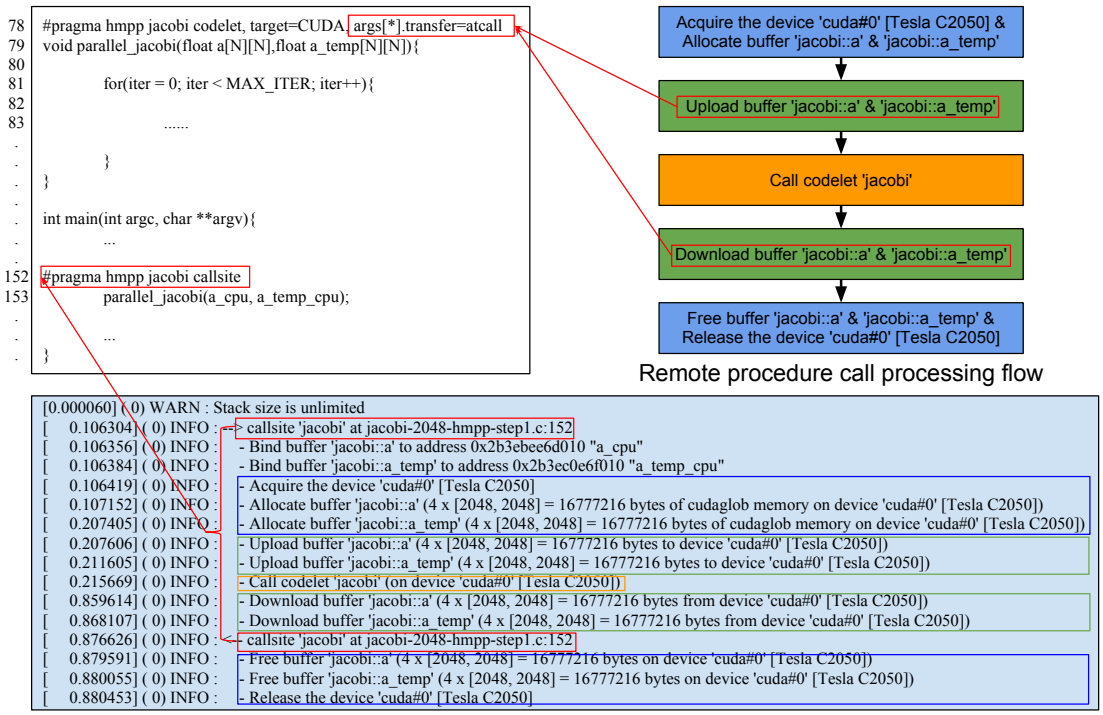


Figure 5.12: Experiment 1: HMPP runtime log with relevant information - variant 1

HMPP variant 2: Reducing data transfer costs

The HMPP variant 2 is an optimized implementation that allows useless data transfers between the host and the device memory to be avoided. Instead of transferring all arguments of the

codelet by the HMPP runtime protocol in the first implementation, in the second variant, the manual transfer policy is used to optimize data movements.

HMPP Workbench provides a set of data transfer policies. The basic transfer policy `transfer=atcall`, which is applied in the previous variant, is the easiest strategy to use and uncomplicated to perform. All arguments of a codelet must be transferred automatically. Generally, this policy is often used to test a codelet. To achieve better performance, this basic transfer policy should be changed to the manual transfer approach `transfer=manual` for reducing data movement costs.

The updated code for this HMPP second code version is shown in Figure 5.13. The first improvement is a replacement for the data transfer policy. The last argument of the HMPP codelet directive is adjusted with `args[*].transfer=manual`, which enables programmers to explicitly control the data transfers. The second change is adding the `advancedload` directive with the same label name before callsite directive in the main function. This manual transfer directive deals with data transfers from the CPU memory to the hardware accelerator memory. The third change is adding corresponding directive `delegatedstore` after the callsite directive for data transfers in opposite direction, from the accelerator memory to the CPU memory. The additional arguments, `args[a], args[a].hostdata="a_cpu"`, are added to the `advancedload` directives to tell the compiler that only the matrix `a_cpu` needs to be transferred to the device before launching the remote execution of the codelet `jacobi` (codelet label).

```
#pragma hmpp jacobi codelet, target=CUDA, args[*].transfer=manual
void parallel_jacobi(float a[N][N], float a_temp[N][N]){

   for(iter = 0; iter < MAX_ITER; iter++){

      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][j+1] + a[i][j-1]);
         }
      }

      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a[i][j] = a_temp[i][j];
         }
      }

   }
}

int main(int argc, char **argv){
   ...

#pragma hmpp jacobi advancedload, args[a], args[a].hostdata="a_cpu"
#pragma hmpp jacobi callsite
   parallel_jacobi(a_cpu, a_temp_cpu);
#pragma hmpp jacobi delegatedstore, args[a]
   ...
}
```

Figure 5.13: Experiment 1: HMPP Workbench implementation - variant 2

The HMPP second variant has an identical diagnostic analysis report with the first variant (Figure 5.11). However, as can be seen from Figure 5.14, the HMPP compiler generated runtime log for this second implementation is completely reorganized. The remote procedure call processing flow is mainly divided into three different phases: `advancedload`, `callsite` and `delegatedsotre`. Furthermore, the data transfer costs is reduced, because only the matrix `a` is transferred between the host and device memory.

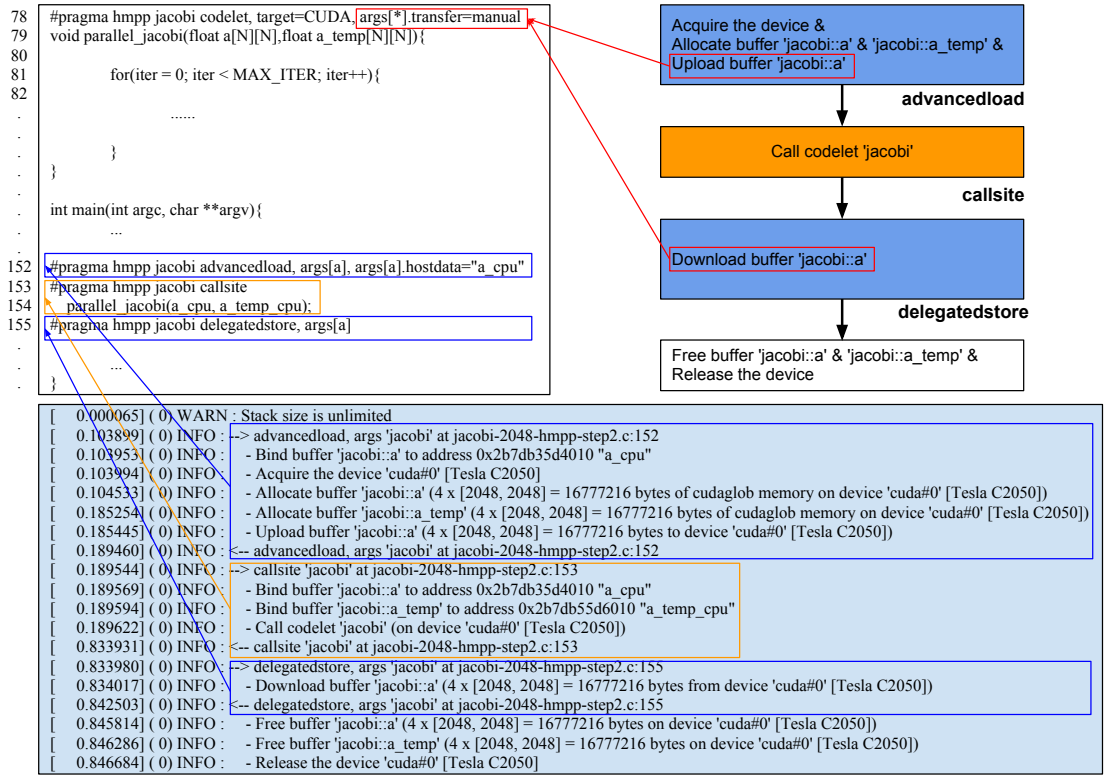


Figure 5.14: Experiment 1: HMPP runtime log with relevant information - variant 2

The execution time and speedup of the HMMP second variant implementation are depicted in Table 5.7. As might be expected, compared with the previous variant, the execution time of this version drops to 0.783 seconds. Although this is only a slight performance change, it is worth to use the HMPP manual transfer policy to control and to optimize data transfers. Sometimes, not all parameters of a `codelet` must to be transferred.

In the HMPP variant 2, the limited speedup is due to the size of the input problem. In this experiment variant, the input matrix size is set to 2048 $\times$ 2048. In addition, only the transferring of the matrix `a_temp` is avoided. According to the runtime log in the previous variant (see Figure 5.12), the time spent on transferring of the matrix `a_temp` from the CPU memory to the GPU memory is only about 4064 milliseconds (0.215669 - 0.211605), after the completion of the HMPP codelet function, the time spent on transferring of this matrix back to the CPU memory is about 8519 milliseconds (0.876626 - 0.868107). Hence, there is only a small increase in

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85

Table 5.7: Experiment 1: Performance results after HMPP implementation in variant 2, matrix size: 2048, iteration number: 500

performance over the previous variant.

HMPP variant 3: Tuning the loop schedules

Based on the previous variant, the HMPP third variant focuses on the loop mapping schedule and other further optimization to achieve better performance. Normally, mapping parallel loops onto the hardware accelerator can be operated automatically. However, the HMPP Workbench provides opportunities for managing the mapping process. This high-level programming approach offers a set of HMPP Codelet Generator (HMPPCG) directives, which are generally designed to control the loop "gridification" process and to improve code performance by the code (CUDA or OpenCL code) generation.

As mentioned in PGI variant 3, the grid size and the block size are two important parameters for performance improvement. Unlike the PGI compiler will determine a suitable size and dimensions of the thread block depending on the size and structure of input data and hardware resources, the HMPP compiler tends to map nested loops on fixed 2-dimensional thread blocks with a default value of `32x4`. This can be expressed in the CUDA model as following execution configuration parameters: <<<dimGrid(N/32, N/4), dimBlock(32, 4)>>>, where N is the dimension of the input square matrix. In the HMPP programming model, if there is no HMPPCG directive to specify a new thread block size, the thread block size is always set to this default value. Generally, this block size of 128 (`32x4`) is sufficient, but this size of thread blocks does not fit all applications and problem sizes.

Figure 5.15 shows the code in third variant implementation. In this variant, a new HMPPCG `gridify` directive is applied to manage the loop mapping process. In `parallel_jacobi()` function, two new pragmas `#pragma hmppcg gridify blocksize 256x2` are added before two nested i, j loops. The clause `blocksize 256x2` specifies a 2-dimensional thread block configuration (`dimBlock(256,2)`) and the total number of threads per block is increases from the 128 (variant 1 and 2) to 512. The new value of the `blocksize` enables each CUDA thread block to run 512 threads concurrently. During the implementation, different values or combinations of the `blocksize` are performed. According to the HMPP Codelet Generator Directives Reference Manual (HMPP Workbench 3.0), some typical values are: 16x16,

32x8, 64x2, 32x4. However, the search space of this parameter is too big. There are a lot of combinations, and hardware architecture also has an influence on this value. After many multiple experiments with different values of `blocksize`, the value of `256x2` can achieve better performance. Accordingly, the occupancy rate is also increased from 67% (variant 1 and 2) to 100%. In addition, there is a second modification of this program. An extra HMPP directive `allocate` is used to pre-allocate two arguments `a` and `a_temp` from the codelet function on the hardware accelerator memory before the data is copied. In the third implementation, the same manual transfer policy `transfer=manual` is also used to optimize data transfers.

```
#pragma hmpp jacobi codelet, target=CUDA, args[*].transfer=manual
void parallel_jacobi(float a[N][N],float a_temp[N][N]){

   for(iter = 0; iter < MAX_ITER; iter++){
#pragma hmppcg gridify blocksize 256x2
      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a_temp[i][j] = 0.25f * (a[i-1][j] + a[i+1][j] + a[i][j+1] + a[i][j-1]);
         }
      }

#pragma hmppcg gridify blocksize 256x2
      for (i = 1; i < N-1; i++) {
         for (j = 1; j < N-1; j++){
            a[i][j] = a_temp[i][j];
         }
      }

   }
}

int main(int argc, char **argv){
   ...

#pragma hmpp jacobi allocate, args[a;a_temp].size=mm_size, mm_size
#pragma hmpp jacobi advancedload, args[a], args[a].hostdata="a_cpu"
#pragma hmpp jacobi callsite
    parallel_jacobi(a_cpu, a_temp_cpu);
#pragma hmpp jacobi delegatedstore, args[a]

   ...
}
```

Figure 5.15: Experiment 1: HMPP Workbench implementation - variant 3

For this third implementation, the HMPP compiler still generates the same diagnostic report (Figure 5.11) as first and second variants do. There is no information about the new value of the block size and other changes. The runtime log for this implementation is illustrated in Figure 5.16. It can be mainly divided into four parts: `allocate`, `advancedload`, `callsite`, and `delegatedstore`. The new part of the runtime log (marked with a yellow box) is a part of `advancedload` in the second variant.

The results of the HMPP third variant can be found in Table 5.8. The execution time of HMPP variant 3 drops to only 0.607 seconds, and it achieves up to 6.26x speedup compared to the serial implementation. This performance benefit is obtained by changing the value of `blocksize` and by using the `allocate` directive. Generally, values of `blocksize` with

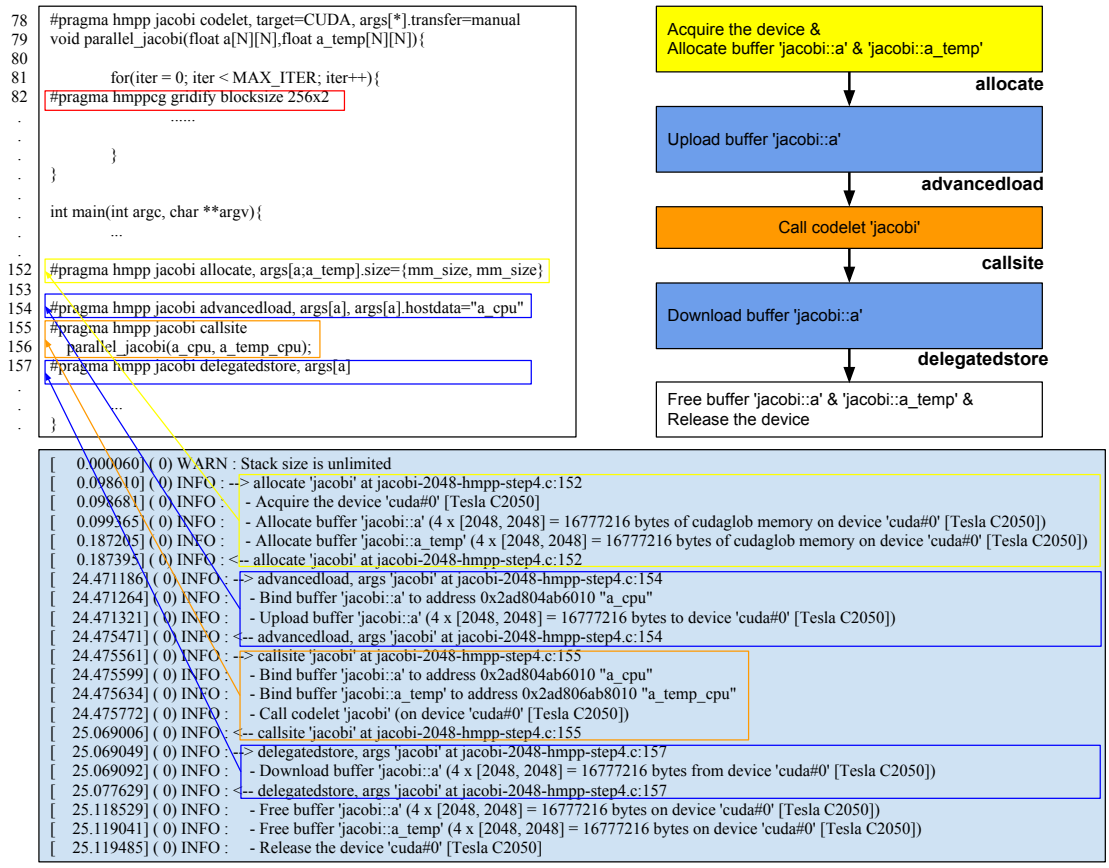


Figure 5.16: Experiment 1: HMPP runtime log with relevant information - variant 3

higher occupancy have better performance than values with lower occupancy. However, finding a global optimal value of the `blocksize` is often difficult in practice, not only the big size of search space, but also the hardware-dependent. These experimental results indicate that this HMPP Workbench can also achieve a similar performance compared to the hand-written CUDA code.

OpenMPC variant 1: Baseline translation

The first variant of OpenMPC implementation is also a basic translation without any optimizations. The original OpenMP program (Figure 4.3) is used as the input source code for the OpenMPC source-to-source compiler to generate the targeted CUDA code. For this reason, the input source code can remain unchanged or only with small modifications.

Similar to the PGI Accelerator and HMPP Workbench, the OpenMPC compiler can provide informational messages during the transformation. Figure 5.17 illustrates the compiler feedback information from the baseline implementation. The OpenMPC compiler feedback explains

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85
HMPP variant 3	0.607	6.26

Table 5.8: Experiment 1: Performance results after HMPP implementation in variant 3, matrix size: 2048, iteration number: 500

how the input OpenMP code is analyzed and translated into the CUDA program. For programmers, the most meaningful information is the `INFO` line, which describes how many candidate kernel regions are found in the input OpenMP program. For our OpenMP code, the compiler has recognized two original OpenMP parallel constructs, which are determined as the potential candidate regions. Furthermore, the compiler has translated them successfully into two CUDA kernel functions.

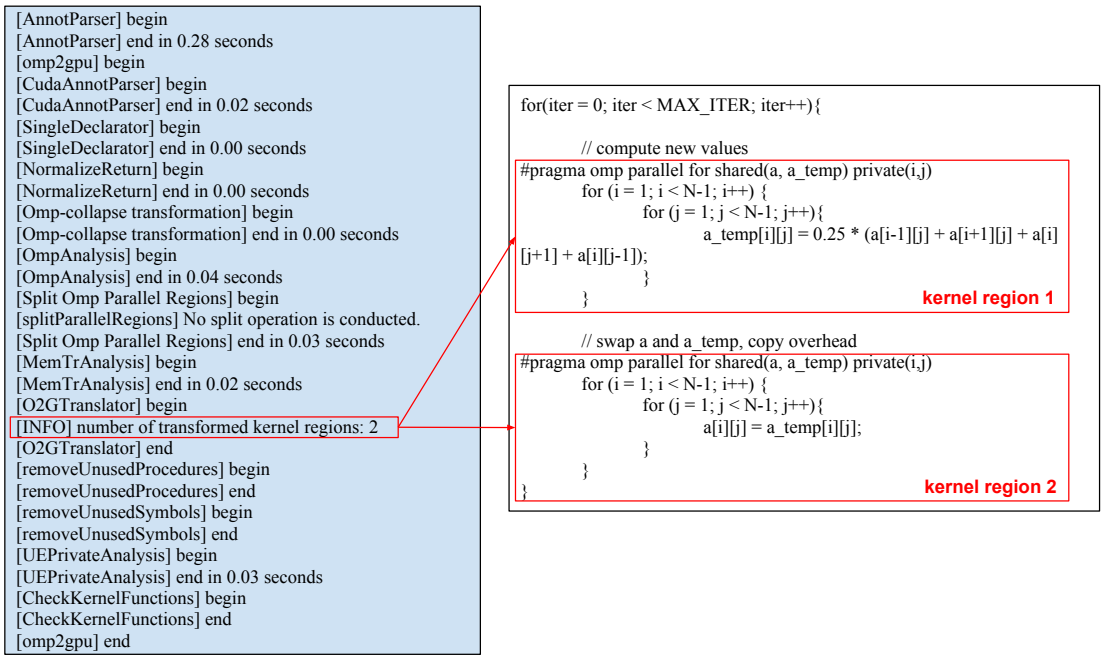


Figure 5.17: Experiment 1: OpenMPC compiler-to-user feedback - variant 1

Figure 5.18 shows two CUDA kernel functions, which are created by OpenMPC compiler

from the naive OpenMP program (Figure 4.3). As can be seen, the initial two `omp parallel for` constructs are interpreted into two CUDA kernel functions, and they inherit the same function names from the OpenMP code with different suffix `_kernel(number)`. Every initial `omp parallel for` loop iteration is distributed to one CUDA thread. In addition, based on the OpenMP data property constructs, such as `shared(a, a_temp)` or `private(i, j)`, the OpenMPC compiler maps the shared data `a` and `a_temp` to the global memory, and the private data `i` and `j` are mapped to the local memory. As can be seen from the figure, these two kernel functions do not take full advantage of the large number of CUDA threads, and only a very limited number of threads (< 2048) are used in both kernel functions. In fact, many CUDA-enabled devices are able to launch many thousands of threads at once. This number of threads is just a very small proportion. In addition, as mentioned above, the shared data `a` and `a_temp` are stored in the device off-chip global memory. To hide this global memory access latency, we need to have enough active threads.

```
1  __global__ void parallel_jacobi_kernel0(float a[2048][2048], float a_temp[2048][2048])
2  {
3  int i;
4  int j;
5  int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
6  int _gtid = (threadIdx.x+(_bid*blockDim.x));
7  i=(_gtid+1);
8  if (i<(2048-1))
9  {
10 for (j=1; j<(2048-1); j ++ )
11 {
12 a_temp[i][j]=(0.25F*(((a[(i-1)][j]+a[(i+1)][j])+a[i][(j+1)])+a[i][(j-1)]));
13 }
14 }
15 }
16
17 __global__ void parallel_jacobi_kernel1(float a[2048][2048], float a_temp[2048][2048])
18 {
19 int i;
20 int j;
21 int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
22 int _gtid = (threadIdx.x+(_bid*blockDim.x));
23 i=(_gtid+1);
24 if (i<(2048-1))
25 {
26 for (j=1; j<(2048-1); j ++ )
27 {
28 a[i][j]=a_temp[i][j];
29 }
30 }
31 }
```

Figure 5.18: Experiment 1: Generated output CUDA kernel code (Figure 4.3 as input), from OpenMPC compiler - variant 1

By default, the CUTIL utility library `cutil.h` is used during generating CUDA code. Thus, the CUDA SDK is required to compile the generated CUDA code. In this created output CUDA program, the runtime data transfer method `cudaMemcpy()` is used to implement data transfers, and the `CUDA_SAFE_CALL` macro is used for error checking.

In contrast to the PGI and HMPP approaches, the OpenMPC compiler generates only a CUDA program not an executable file. Thus, the initial `nvcc` compiler must compile the output

program again. Table 5.9 shows the performance measurement of the OpenMPC baseline translation. From the data in this Table, it is apparent that the baseline translation does not have good performance, and the execution time is almost the same as the sequential version. One of the main causes of the poor performance is the use of a small number of threads in kernel functions and high global memory latency.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85
HMPP variant 3	0.607	6.26
OpenMPC variant 1	6.371	0.59

Table 5.9: Experiment 1: Performance results after OpenMPC implementation in variant 1, matrix size: 2048, iteration number: 500

OpenMPC variant 2: Translation with all safe optimization

The current version of OpenMPC supplies a set of environment variables, which can control program-level behaviours of various optimizations, such as thread batching or other translation configurations [Lee and Eigenmann, 2012]. Some environment variables, which are always safe and beneficial, can be applied generally in most cases, and some environment variables may be beneficial, but aggressive and unsafe. In the second variant, all safe OpenMPC optimization options are used to control the code generation process. Following compiler flags are applied:

- `useParallelLoopSwap`

  This is an optional optimization option, which is developed especially for regular applications to enhance inter-thread locality, and it is introduced by the OpenMPC approach as a parallel loop-swap transformation to achieve coalesced memory access.

- `useMallocPitch`

  With this option, the CUDA function `cudaMallocPitch()` will be used to allocate two-dimensional arrays. This function is recommended by the CUDA programming guide for allocations of two-dimensional arrays to confirm that the allocation is appropriately padded to meet the alignment requirements that ensure efficient coalesced global memory access. This option could be used in this application, because all shared variables are two-dimensional arrays.

- `useGlobalGMalloc`

  This option is invariably beneficial, and can be always used. The main purpose of this option is to reduce data transfers between CPU and GPU memories.

Table 5.10 shows that this optimized variant can provide a performance win by using these three environment variables. The speedup increases from 0.59x to 2.87x over the sequential version. With these three safe OpenMPC optimization options, the new generated CUDA program contains also two kernel functions, as the first variant does, but these kernels remain unchanged. The main reason for this performance improvement is the use of the above mentioned three safe options, which will mainly optimize data transfers and global memory access.

Version	**Execution time (second)**	**Speedup**
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85
HMPP variant 3	0.607	6.26
OpenMPC variant 1	6.371	0.59
OpenMPC variant 2	1.317	2.87

Table 5.10: Experiment 1: Performance results after OpenMPC implementation in variant 2, matrix size: 2048, iteration number: 500

OpenMPC variant 3: Translation with user-assisted tuning

OpenMPC contains not only a compilation system but also a tuning system. The automatic compilation system has been already tested by first and second variants. In this variant, the tuning system is the first test object. This tuning system is based on its abundant environment variables and directives, which can directly manage the code translation and optimization process. These variables and directives can serve as tuning parameters to build search space. The most recent version of the OpenMPC approach supports only a simple tuning mechanism, which tries out different combinations of the tuning parameters suggested by the compiler.

Although the OpenMPC tuning system includes a search space pruner to reduce the number of tuning parameters and optimization space [Lee and Eigenmann, 2012], the total number of generated tuning configuration files is still high. In this experimental variant, for the same input OpenMP program, the OpenMPC tuning configuration generator creates a cumulative total of 96 tuning configuration files, and this has been done only at the program level. When the tuning occurs at the kernel level, this number should be higher than 96. With each tuning configuration file, the OpenMPC compiler can translate the input OpenMP program into a corresponding

output CUDA program. Because the current tuning engine does not directly produce executables form generated CUDA programs, programmers have to compile them using the nvcc compiler manually.

In this variant, not all tuning configuration files are used to create output CUDA programs. To reduce the cost of development and avoid duplication of work, 12 configuration files from 96 files are selected to assist the OpenMPC compiler to generate 12 CUDA code. Table 5.11 shows different configuration files lead to completely different results. The experimental results show that not all the tuning configurations have good performance, and some options are worse than the baseline translation. With this user-assisted tuning, the "best" experimental result produced by the tuning system can be chosen to represent the final optimal result. According the results from the Table 5.11, the number 40 of tuning configuration files offers a better performance than the others.

Configuration File Nr.	0	1	10	20	30	40
Execution time (sec.)	6.262	6.338	6.337	1.365	6.260	1.312
Configuration File Nr.	50	60	70	80	90	95
Execution time (sec.)	6.950	1.370	1.360	6.943	1.319	6.407

Table 5.11: Execution time of jacobi method using OpenMPC tuning configuration files

Table 5.12 shows that the OpenMPC user-assisted tuning increases the performance, but this performance improvement is very small compared to the OpenMPC second variant.

Version	**Execution time (second)**	**Speedup**
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85
HMPP variant 3	0.607	6.26
OpenMPC variant 1	6.371	0.59
OpenMPC variant 2	1.317	2.87
OpenMPC variant 3	1.312	2.89

Table 5.12: Experiment 1: Performance results after OpenMPC implementation in variant 3, matrix size: 2048, iteration number: 500

OpenMPC variant 4: Translation with a modified input program and all safe optimization

In three previous variants, the OpenMPC compiler has successfully transformed the OpenMP program into a CUDA program. Although its optimization techniques can improve the performance, there is a noticeable performance gap between the hand-written CUDA version and previous variants created by the OpenMPC compiler. As mentioned in the OpenMPC variant 1, this performance difference is due to the lack of active threads. In the previous variants, only a few threads are used, which are not enough to hide the long global memory access latency. In the OpenMPC model, each iteration of `omp for` loops is assigned to a thread [Lee and Eigenmann, 2010], thus, for multiple nested loops, only the outermost loop will be parallelized. In this last variant, the original OpenMP program is modified manually with the native OpenMP clause `collapse` to satisfy the compiler's needs. This clause is used to collapse perfectly nested loops into large iteration space.

Figure 5.19 shows two kernel functions, which are generated by the OpenMPC compiler from the manually modified input OpenMP program. As can be seen in line 9 and line 26, the total number of thread is increased up to 4186116, and now they do not contain any loop body. In the first three variants, a `for` loop can always be found in each generated kernel function, so this new variant must be more efficient than the previous three variants.

In this variant, all OpenMPC safe optimizations are used, and the thread block size is set to 256. Table 5.13 shows a noticeable performance improvement from this variant. It achieves a speedup up to 6.38x over the sequential version.

Version	**Execution time (second)**	**Speedup**
Sequential reference (icc -O3)	3.800	-
OpenMP (16 threads)	2.689	1.41
CUDA	0.641	5.92
PGI variant 1	17.467	0.22
PGI variant 2	0.684	5.55
PGI variant 3	0.630	6.03
HMPP variant 1	0.797	4.77
HMPP variant 2	0.783	4.85
HMPP variant 3	0.607	6.26
OpenMPC variant 1	6.371	0.59
OpenMPC variant 2	1.317	2.87
OpenMPC variant 3	1.312	2.89
OpenMPC variant 4	0.595	6.38

Table 5.13: Experiment 1: Performance results after OpenMPC implementation in variant 4, matrix size: 2048, iteration number: 500

```
1  __global__ void parallel_jacobi_kernel0(float * a, size_t pitch__b__main, float * a_temp, size_t
       pitch__b_temp__main)
2  {
3  int _ti_100_0;
4  int i;
5  int j;
6  int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
7  int _gtid = (threadIdx.x+(_bid*blockDim.x));
8  _ti_100_0=_gtid;
9  if ((_ti_100_0<4186116))
10 {
11 j=((_ti_100_0%2046)+1);
12 i=((_ti_100_0/2046)+1);
13 ( * (((float * )((char * )a_temp)+(i*pitch__b_temp__main)))+j))=(0.25F*((( * (((float * )(((
       char * )a)+((i-1)*pitch__b__main)))+j))+( * (((float * )(((char * )a)+((i+1)*pitch__b__main
       )))+j)))+( * (((float * )(((char * )a)+(i*pitch__b__main)))+(j+1))))+( * (((float * )(((
       char * )a)+(i*pitch__b__main)))+(j-1)))));
14 }
15 }
16
17
18 __global__ void parallel_jacobi_kernel1(float * a, size_t pitch__b__main, float * a_temp, size_t
       pitch__b_temp__main)
19 {
20 int _ti_100_0;
21 int i;
22 int j;
23 int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
24 int _gtid = (threadIdx.x+(_bid*blockDim.x));
25 _ti_100_0=_gtid;
26 if ((_ti_100_0<4186116))
27 {
28 j=((_ti_100_0%2046)+1);
29 i=((_ti_100_0/2046)+1);
30 ( * (((float * )(((char * )a)+(i*pitch__b__main)))+j))=( * (((float * )(((char * )a_temp)+(i*
       pitch__b_temp__main)))+j));
31 }
32 }
```

Figure 5.19: Experiment 1: Generated output CUDA kernel code(with modified input), from OpenMPC compiler - variant 4

5.2 Evaluation of the experiment 2

Matrix multiplication kernel is a popular application kernel to study parallel programming frameworks. In the second experiment, this matrix multiplication application kernel (Figure 4.7) is considered as a basis to accelerate, similar to the first experiment, these three nested loops are the main target for high-level programming supports.

Evaluation setup

In this step, the sequential code (Figure 4.7) version is compiled by the Intel `icc` compiler with an optimization level of `-O3`. The matrix multiplication application is also implemented by using OpenMP and CUDA. The main purpose of this evaluation setup is to allow the following comparisons between the basic and high-level supports for heterogeneous systems.

As can be seen from Table 5.14, the hand-optimized CUDA version achieves a speedup of 27.50x compared to the sequential version. This performance benefit is obtained by using shared

memory and its optimal algorithm with blocking.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50

Table 5.14: Experiment 2: Performance results of the OpenMP and CUDA implementation in evaluation setup, matrix size: 2048

PGI variant 1: Baseline implementation

The baseline implementation using PGI Accelerator compiler directives is to determine the potential parallelizable code region, then to add the most important PGI compiler directive `#pragma acc region` into the source code.

In the matrix multiplication function, finding the suitable code region is relatively easy. As can be seen from Figure 5.20, only one pragma line (marked with a grey shaded area) is inserted into the original source code, and the three nested for loops are the primary candidate. The PGI compiler directive `#pragma acc region` is applied to specify this compute region, and to tell the compiler to map the following loops as kernel functions onto the accelerator.

```
#pragma acc region
    for (i = 0; i< N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k< N; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
```

Figure 5.20: Experiment 2: PGI Accelerator implementation - variant 1

Figure 5.21 indicates the PGI compiler feedback for the program. It contains important information for further optimization. First, the complier determines that all three two-dimensional matrices are used inside the compute region, and they need to be copied from the CPU memory to the accelerator GPU memory. Two `Generating copyin` messages indicate that the matrix `a` and `b` only need to be copied from the host memory to the device memory. The message `Generating copy(c[0:2048][0:])` indicates that the matrix `c` needs to be transferred in both directions before and after kernel function execution. Figure 5.22 illustrates how the data are transferred in the first variant. Second, the `i` and j loops are diagnosed by the compiler as parallelizable, whereas the `k` loop could only be executed in sequential because the loop-carried dependencies. Third, the message `Accelerator kernel generated` indicates that the compiler has transferred the compute region to a device kernel function.

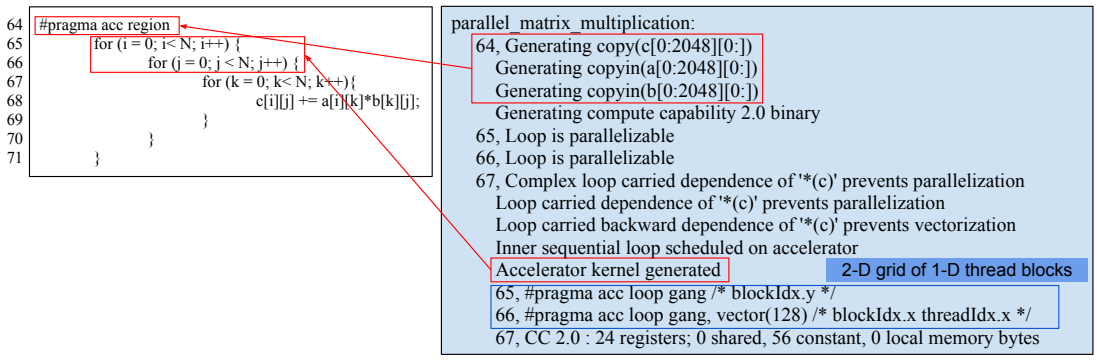


Figure 5.21: Experiment 2: PGI compiler-to-user feedback - variant 1

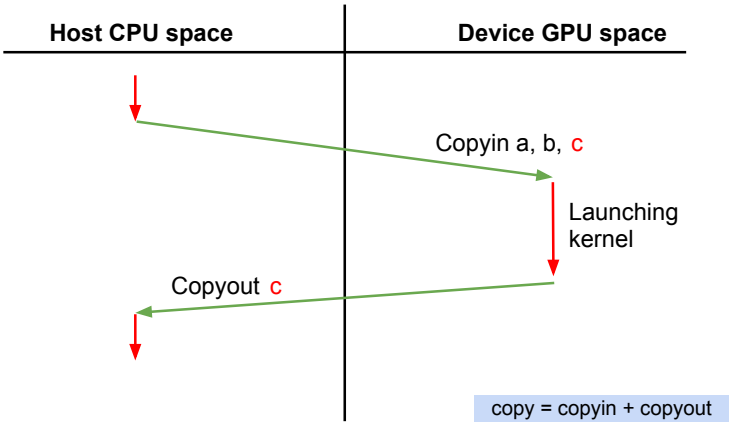


Figure 5.22: Experiment 2: Data transfers between the host CPU memory and the device GPU memory - PGI variant 1

The performance results are shown in Table 5.15. With only one additional PGI directive line increases the performance up to 12.57x over the sequential version, and it achieves already 45.72% of the performance of the hand-optimized CUDA program.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57

Table 5.15: Experiment 2: Performance results after PGI implementation in variant 1, matrix size: 2048

PGI variant 2: Reducing data movement costs

The second variant of PGI Accelerator implementation deals with reducing data movement costs. According to this compiler feedback (Figure 5.22), the matrix `c` is transferred from the CPU memory to the accelerator memory, but in fact, it needs only to be assigned values in the accelerator memory and then to be copied back to the CPU memory.

To explicitly manage the data movement processes, the PGI data clauses `copyin` and `copyout` are applied into the compute region directive. In addition, to assist the compiler to understand this purpose, the source code is slightly modified. The initial values from the parameter matrix `c` are assigned within the compute region.

Figure 5.23 shows the modified code in second variant PGI program implementation, and new lines are marked with a grey shaded area. First of all, a combination of the compute region directive and loop mapping directive, `#pragma acc region for`, is used to surround the nested loops. This combination is a shortcut for specifying a loop directive nested immediately inside a compute region [The Portland Group Inc., 2010]. The data clauses `copyin` and `copyout` specify that the matrix `a`, and `b` only need to be copied from the CPU to the accelerator memory before the kernel function execution, and the matrix `c` only needs to be copied back to the CPU memory at the end of the compute region to avoid the useless transfer in the first variant made by the compiler. As shown in Figure 5.24, the new compiler feedback indicates that the values of matrix `c` are only copied once now. This means that the total time spent on data transfers can be reduced.

Then, two new loop mapping scheduling clauses, `independent` and `seq` are applied to assist the compiler in determining data independence. It is apparent that the iterations of `i` and `j` are data independent, so these iterations can be executed in parallel. On the contrary, the `k` loop must be executed sequentially on the hardware accelerator because of its loop carried dependence.

```
#pragma acc region for independent copyin(a[N][N],b[N][N]) copyout(c[N][N])
   for (i = 0; i< N; i++) {
#pragma acc for independent
      for (j = 0; j < N; j++) {
         c[i][j] = 0.0f;
#pragma acc for seq
         for (k = 0; k< N; k++){
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }
```

Figure 5.23: Experiment 2: PGI Accelerator implementation - variant 2

Table 5.16 shows the performance results after the PGI implementation in second variant, in which the data movement costs are reduced. As can be seen from the data from the table, that the speedup increases from 12.57x to 12.83x. Furthermore, according to the PGI profile timing

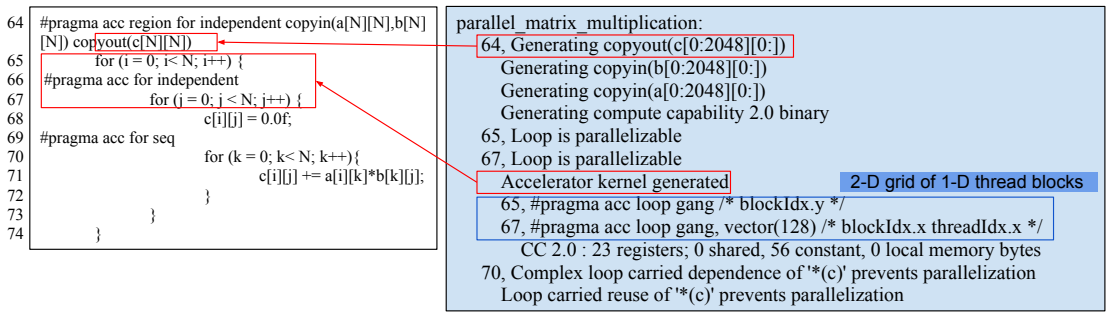


Figure 5.24: Experiment 2: PGI compiler-to-user feedback - variant 2

information, the time spent on data movement is reduced from 22299 microseconds to 15547 microseconds. This version achieves 46.65% of the performance of the CUDA program.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83

Table 5.16: Experiment 2: Performance results after PGI implementation in variant 2, matrix size: 2048

PGI variant 3: Loop Schedule Tuning

Based on the second variant, to obtain better performance, the loop schedule tuning is the main method for performance optimization. By default, the generated kernel function will be automatically scheduled on the hardware accelerator, but this process is not always optimal. According to compiler feedback in first and second variants (Figure 5.21 and Figure 5.24), the nested `i` and `j` for loops are mapped on a 2-dimensional grid of 1-dimensional thread blocks with a thread block size of 128 (vector width).

In this new variant, different combinations of vector widths are tested manually. Instead of mapping these nested loops onto 1-dimensional thread blocks, we tried to map them onto 2-dimensional thread blocks. Accordingly, two vector widths are required to specify the dimensions of the 2-dimensional thread block. That means the search space is bigger than the PGI variant 3 in the experiment 1, which has only 1-dimensional thread blocks. After many multiple experiments with different values of vector width, the combination of `2` and `128` can deliver a better performance.

As presented in Figure 5.25, two PGI loop scheduling clause `for parallel, vector` with their width arguments are inserted into the program. The `i` loop is with a vector width of

2, and the `j` loop is with a vector width of 128, they correspond to the dimensions of the thread block. As a consequence, these nested `i` and `j` loops will be mapped onto 128×2 2-dimensional thread blocks. This corresponds to the <<<dimGrid(N/128, N/2), dimBlock(128, 2)>>> in CUDA model, where N is the dimension of the input square matrix. In addition, the thread block size is increased from 128 to 256. Meanwhile, the occupancy is also increased from 67% (first and second variant) up to 100%.

```
#pragma acc region for parallel, vector(2) copyin(a[N][N],b[N[N]) copyout(c[N][N])
   for (i = 0; i< N; i++) {
#pragma acc for parallel, vector(128)
      for (j = 0; j < N; j++) {
         c[i][j] = 0.0f;
#pragma acc for seq
         for (k = 0; k< N; k++){
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }
```

Figure 5.25: Experiment 2: PGI Accelerator implementation - variant 3

Table 5.17 shows the results of the performance measurements. With the applicable loop tuning in this implementation variant, the execution time is reduced predictably from 0.269 seconds to 0.216 seconds, and the speedup increases up to 15.96x, which achieves 58% of the performance of the hand-written CUDA program. The experimental results show that vector widths, which can achieve good occupancy rates, can also provide good performance.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83
PGI variant 3	0.216	15.96

Table 5.17: Experiment 2: Performance results after PGI implementation in variant 3, matrix size: 2048

HMPP variant 1: Baseline implementation

The baseline implementation of HMPP Workbench for the matrix multiplication kernel is only adding the basic required compiler directives into the source code.

Figure 5.26 shows the resulting code with the basic required HMPP directives. The pair of directives `codelet`/`callsite` with the label `mm` is inserted into the original sequential C

program. The clause `target=CUDA` specifies the targeted language for the HMPP codelet generation, thus, all the `parallel_matrix_multiplication` function arguments are copied from the CPU memory to the accelerator GPU memory. After the kernel execution, these arguments are transferred back from the GPU memory to the CPU memory.

```
#pragma hmpp mm codelet, target=CUDA, args[*].transfer=atcall
void parallel_matrix_multiplication(TYPE a[N][N],TYPE b[N][N],TYPE c[N][N]){
   int i, j, k;

   for (i = 0; i< N; i++) {
      for (j = 0; j < N; j++) {
         for (k = 0; k< N; k++){
            c[i][j] += a[i][k]*b[k][j];
         }
      }
      }
}

int main(int argc, char **argv){
   ...

#pragma hmpp mm callsite
   parallel_matrix_multiplication(a, b, c);

   ...
}
```

Figure 5.26: Experiment 2: HMPP Workbench implementation - variant 1

Table 5.18 shows the performance results after the first variant of HMPP implementation without any optimizations. As shown in the table, the execution time of the HMPP baseline implementation is slower than the PGI Accelerator version and the CUDA version. It achieves only 20.72% of the performance of the CUDA program.

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83
PGI variant 3	0.216	15.96
HMPP variant 1	0.605	5.70

Table 5.18: Experiment 2: Performance results after HMPP implementation in variant 1, matrix size: 2048

HMPP variant 2: Reducing data transfer costs

In this variant, to reduce the data movement costs, the data transfer policy is adjusted as explicit manual transfer. With the basic transfer policy `transfer = atcall` used in the previous

variant, before launching the kernel function, the matrix `c` is copied from the host memory to the device memory. But in fact, only the matrix `a` and `b` need to be copied from the CPU to the accelerator memory before the HMPP codelet execution. Thus, the explicit manual transfer policy `transfer = manual` can improve the application performance.

Figure 5.27 indicates the HMPP program in second variant. The HMPP `advancedload` and `delegatedstore` directives are applied to avoid unnecessary data transfer. Similar to the PGI implementation in variant 2, the original C program is slightly modified too, and the initial values from the matrix `c` are assigned within the HMPP codelet.

```
#pragma hmpp mm codelet, target=CUDA, args[*].transfer=manual
void parallel_matrix_multiplication(TYPE a[N][N],TYPE b[N][N],TYPE c[N][N]){
   int i, j, k;

   for (i = 0; i< N; i++) {
      for (j = 0; j < N; j++) {
         for (k = 0; k< N; k++){
            c[i][j] += a[i][k]*b[k][j];
         }
      }
      }
}

int main(int argc, char **argv){
   ...

#pragma hmpp mm advancedload, args[a], args[a].hostdata="a"
#pragma hmpp mm advancedload, args[b], args[b].hostdata="b"
#pragma hmpp mm callsite
   parallel_matrix_multiplication(a, b, c);
#pragma hmpp mm delegatedstore, args[c]
   ...
}
```

Figure 5.27: Experiment 2: HMPP Workbench implementation - variant 2 - matrix multiplication

Figure 5.28 shows a comparison of the HMPP compiler runtime log information between HMPP first and second variants. It is very clear that the number of data transfers from host to device is reduced from 3 to 2.

As can be seen in Table 5.19, after reducing data movement costs the execution time is reduced from 0.605 seconds to 0.574 seconds, and the speedup increases up to 6.00x over the sequential C program. However, this performance is not as good as the implementations with PGI Accelerator approach, and it reaches 21.82% of the performance of the CUDA program.

HMPP variant 3: Tuning loop schedules

The third variant of the implementation using HMPP Workbench concentrates on its loop tuning feature. On the basis of the previous variant, to obtain better performances, the HMPPCG directives are applied to improve program performance by controlling the loop mapping process.

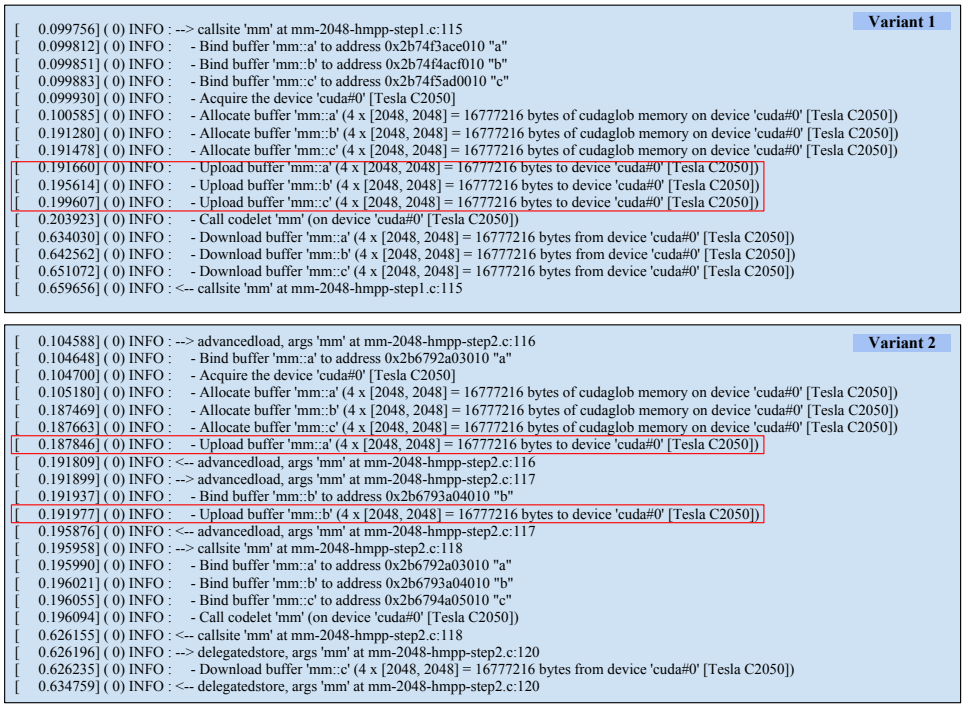


Figure 5.28: Experiment 2: HMPP runtime log with relevant information - variant 1 vs. variant 2

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83
PGI variant 3	0.216	15.96
HMPP variant 1	0.605	5.70
HMPP variant 2	0.574	6.00

Table 5.19: Experiment 2: Performance results after HMPP implementation in variant 2, matrix size: 2048

Figure 5.29 shows the resulting program for the implementation using the HMPPCG `gridify blocksize` directive. By default, loops will be always mapped onto 32×4 two-dimensional blocks. In this program, the 256×2 is the new block size. With this new size, the occupancy will be increased from 67% to 100%. Generally, this `blocksize` and the `vector` in the PGI model have a lot in common. Both of them are used to specify the thread block size and dimensions. Similar to the PGI Accelerator approach, there is no definitive solution for choosing an

optimal block size, and this depends on the targeted hardware accelerator architecture and the nested loops. However, our experimental results show that `blocksize` with higher occupancy rates will lead to better performances. During the implementation, different values of block sizes are tested. For our experimental platform, the `blocksize` 256×2 is a good combination after several values were tested. In addition, just like in the first experiment in the main function, the HMPP directive `allocate` is applied to pre-allocate the three matrices on the hardware memory.

```
#pragma hmpp mm codelet, target=CUDA, args[*].transfer=manual
void parallel_matrix_multiplication(TYPE a[N][N],TYPE b[N][N],TYPE c[
    N][N]){
    int i, j, k;

#pragma hmppcg gridify blocksize 256x2
    for (i = 0; i< N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k< N; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
        }
}

int main(int argc, char **argv){
    ...

#pragma hmpp mm allocate, args[a;b;c].size=mm_size, mm_size
#pragma hmpp mm advancedload, args[a], args[a].hostdata="a"
#pragma hmpp mm advancedload, args[b], args[b].hostdata="b"
#pragma hmpp mm callsite
    parallel_matrix_multiplication(a, b, c);
pragma hmpp mm delegatedstore, args[c]}...
```

Figure 5.29: Experiment 2: HMPP Workbench implementation - variant 3 - matrix multiplication

Table 5.20 indicates the performance measurement in this implementation variant. This third variant has the best performance in comparison with other HMPP implementations. It achieves a speedup of 9.39x over the sequential C program, and reaches about 34.14% of the performance of the CUDA program.

Implementation with OpenMPC

Similar to the first experiment, the evaluation of using the OpenMPC source-to-source compiler is performed by four variants too:

- Variant 1: Baseline translation

Version	Execution time (second)	Speedup
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83
PGI variant 3	0.216	15.96
HMPP variant 1	0.605	5.70
HMPP variant 2	0.574	6.00
HMPP variant 3	0.367	9.39

Table 5.20: Experiment 2: Performance results after HMPP implementation in variant 3, matrix size: 2048

The baseline translation is a basic OpenMP-to-CUDA conversion, and the input OpenMP code will be translated into the CUDA program with no performance optimization options. According to OpenMPC compiler feedback, the initial OpenMP `omp parallel` region is successfully translated into one CUDA kernel function.

- Variant 2: Translation with all safe optimization

  In this variant, all OpenMPC beneficial safe environment variants are applied to assist the output CUDA generation process to obtain better performance. Four compiler options are applied in this variant: useGlobalGMalloc, useMallocPitch, useParallelLoopSwap and assumeNonZeroTripLoops.

- Variant 3: Translation with user-assisted tuning

  The third variant implementation of OpenMPC focuses on its user-assisted tuning system. A set of various CUDA programs can be obtained from the input OpenMP program with OpenMPC compiler generated tuning configuration files. For the input OpenMP program, 96 tuning-configuration files were created, and 12 configuration files are selected to generate tuned output CUDA programs. As shown in Table 5.21, the tuning configuration file number 20 has a better performance than the others.

Configuration File Nr.	0	1	10	20	30	40
Execution time (sec.)	11.154	9.894	Error	2.274	11.125	Error
Configuration File Nr.	50	60	70	80	90	95
Execution time (sec.)	11.053	2.275	Error	11.147	2.277	11.182

Table 5.21: Execution time of matrix multiplication using OpenMPC tuning configuration files

- Variant 4: Translation with a modified input program and all safe optimization

In the last variant, the input OpenMP program is manually modified with the OpenMP `collapse` clause to fuse nested loops into iteration space. In addition, all OpenMPC safe optimization options are applied too, and the thread block size is set to 256.

After the OpenMPC implementation, the performance measurements are shown in Table 5.22. The results demonstrate that the performance of the baseline transformation (OpenMPC variant 1) is poor. The translation with all safe optimization (OpenMPC variant 2) and with user-assist tuning (OpenMPC variant 3) can bring better performance, but these performance improvements are very limited. However, in the best case, the OpenMPC variant 4 achieves a speedup of 9.38x, which is very similar to the HMPP variant 3.

Version	**Execution time (second)**	**Speedup**
Sequential reference (icc -O3)	3.445	-
OpenMP (16 threads)	1.029	3.35
CUDA	0.125	27.50
PGI variant 1	0.274	12.57
PGI variant 2	0.269	12.83
PGI variant 3	0.216	15.96
HMPP variant 1	0.605	5.70
HMPP variant 2	0.574	6.00
HMPP variant 3	0.367	9.39
OpenMPC variant 1	10.637	0.32
OpenMPC variant 2	2.284	1.51
OpenMPC variant 3	2.274	1.52
OpenMPC variant 4	0.367	9.38

Table 5.22: Experiment 2: Performance results after the OpenMPC implementation, matrix size: 2048

5.3 Analysis of different problem sizes for experiment 1

The experiments are also performed with different problem sizes. Detailed overall execution times including data transfers are shown in Table 5.23. The execution times decrease step-by-step with a different level of optimization. The optimization techniques that each approach provides allow these continual performance improvements.

These first variants are basic implementations without optimizations. In general, they do not deliver very good performances. For example, the PGI variant 1has the longest execution times because of its large data transfer overhead, especially for higher matrix dimensions. These second variants deal with reducing the data movement cost in PGI and HMPP or using all safe optimization options in OpenMPC, and they decrease the execution times at different level. The PGI variant 3 and HMPP variant 3 are their respective tuning processes, which present the best-case execution times. As observed from the data, the execution time of the last variants of

PGI Accelerator compiler, HMPP Workbench, OpenMPC and CUDA implementations are very close.

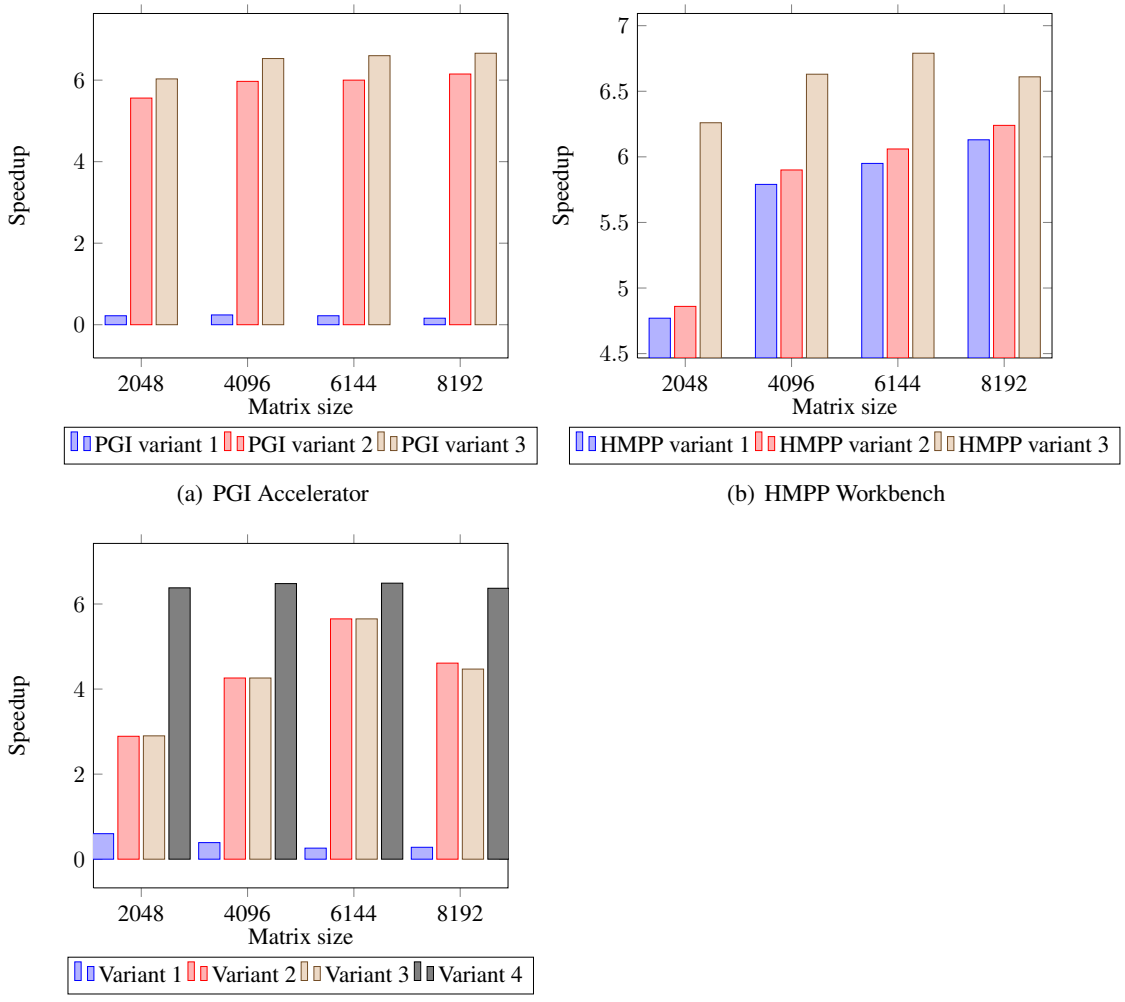
Matrix size	2048	4096	6144	8192
Sequential C	3.800	15.160	34.076	60.689
CUDA	0.641	2.273	5.233	9.112
PGI variant 1	17.467	62.843	152.178	376.057
PGI variant 2	0.684	2.540	5.678	9.861
PGI variant 3	0.630	2.321	5.157	9.108
HMPP variant 1	0.797	2.618	5.728	9.900
HMPP variant 2	0.783	2.568	5.627	9.723
HMPP variant 3	0.607	2.286	5.017	9.175
OpenMPC variant 1	6.371	38.521	132.710	214.028
OpenMPC variant 2	1.317	3.559	6.031	13.160
OpenMPC variant 3	1.312	3.563	6.035	13.562
OpenMPC variant 4	0.595	2.340	5.248	9.526

Table 5.23: Experiment 1: Execution time in seconds

Figure 5.30 presents the detailed speedups obtained through different variants using different approaches in the first experiment. Figure 5.30(a) indicates that the implicit data movements managed by the PGI Accelerator can cause a big drop in performance, and the data transfer overhead between the CPU and the accelerator is an obvious performance bottleneck. As can be seen from Figure 5.30(b), the HMPP variant 1 has better performance than the PGI variant 1 and OpenMPC variant 1. The performance gap is mainly due to different targets or code regions of different high-level supports. For PGI Accelerator and OpenMPC compiler, loops within structured blocks specified by their respective directives are the fundamental targets that will be compiled into kernel functions. Nevertheless, for HMPP Workbench computationally intensive functions are the main targets, and all function arguments must be transferred between the CPU memory and the accelerator memory. Furthermore, it can also be seen from their third variants that the best performance of the PGI Accelerator compiler and HMPP Workbench is achieved through controlling the loop mapping process, a suitable vector width or block size with higher occupancy rates can improve the performance. Figure 5.30(c) shows that these baseline translations without any optimizations do not achieve good performance with OpenMPC source-to-source compiler. However, with its optimization options and user-assist tuning, the speedup of the second and third variants increases significantly. The variant 4 achieves the best performance due to more active threads.

A hand-written CUDA program is implemented for comparison. This offers an opportunity to compare the basic support CUDA version with other high-level support versions. The best performance obtained by the high-level programming approaches, which are used to present its performance that a high-level programming approach can achieve.

Figure 5.31 illustrates the performance comparison from the first experiment. In general, the first application JACOBI method is not a GPU-friendly algorithm. The cause of this performance problem seems to be the uncoalesced memory access to global memory on the hardware



(a) PGI Accelerator

(b) HMPP Workbench

(c) OpenMPC compiler

Figure 5.30: Speedups of the PGI Accelerator, HMPP Workbench, and OpenMPC compiler implementations of the experiment 1. Speedups are over CPU serial implementations.

accelerator. As can be seen from the data in this figure, all three approaches can achieve very similar performance to the CUDA version, sometimes ever better.

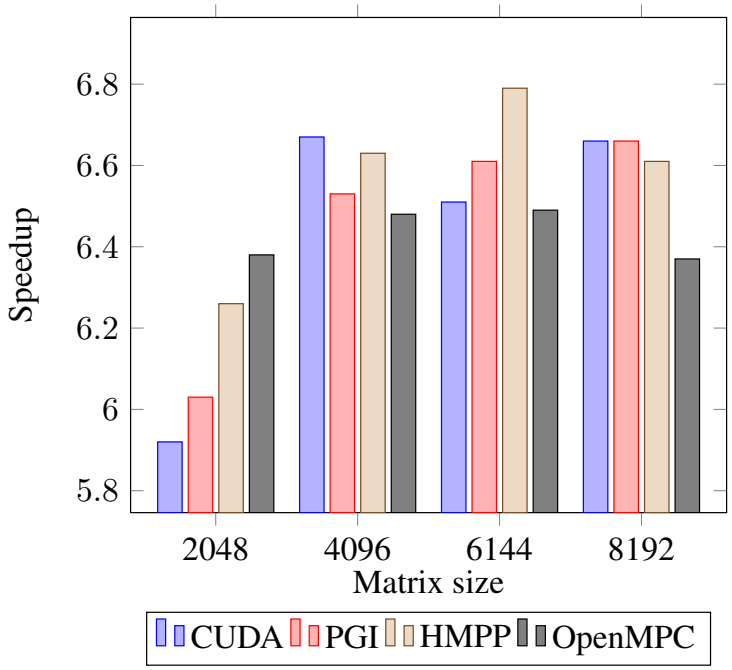


Figure 5.31: Performance comparison between of hand-written CUDA, PGI Accelerator, HMPP Workbench, and OpenMPC compiler implementations of the experiment 1. Speedups are over CPU serial implementations.

5.4 Analysis of different problem sizes for experiment 2

The second experiment is done by considering different input matrix sizes as well, and the overall execution times in the second experiment are presented in Table 5.24. Please note that all sequential C programs are compiled with Intel icc compiler with optimization level -O3, during the experiments, our results show a huge performance gap between intel icc -O3 and GNU gcc -O3.

In general, the last variants implemented by each approach always have the shortest execution time just like the first experiment. Next to the PGI Accelerator compiler and the HMPP Workbench, there is an apparent performance improvement between the second and third variants, the reason for the improvement is the more appropriate mapping of nested loops into a kernel function. There is a large performance difference between the OpenMPC's first three variants and the last variant. In the first three variants, the OpenMPC compiler has parallelized only the outermost loop. After collapsing the nested loops, the performance is remarkably increased.

Matrix size	512	1024	2048	3072
Sequential C	0.057	0.468	3.445	11.394
CUDA	0.004	0.021	0.125	0.401
PGI variant 1	0.007	0.035	0.274	0.810
PGI variant 2	0.006	0.033	0.270	0.801
PGI variant 3	0.006	0.032	0.216	0.634
HMPP variant 1	0.157	0.193	0.605	1.649
HMPP variant 2	0.142	0.183	0.574	1.610
HMPP variant 3	0.007	0.047	0.367	1.197
OpenMPC variant 1	0.535	2.322	10.637	56.658
OpenMPC variant 2	0.136	0.548	2.284	5.376
OpenMPC variant 3	0.137	0.549	2.274	5.375
OpenMPC variant 4	0.008	0.048	0.367	1.556

Table 5.24: Experiment 2: Execution time in seconds

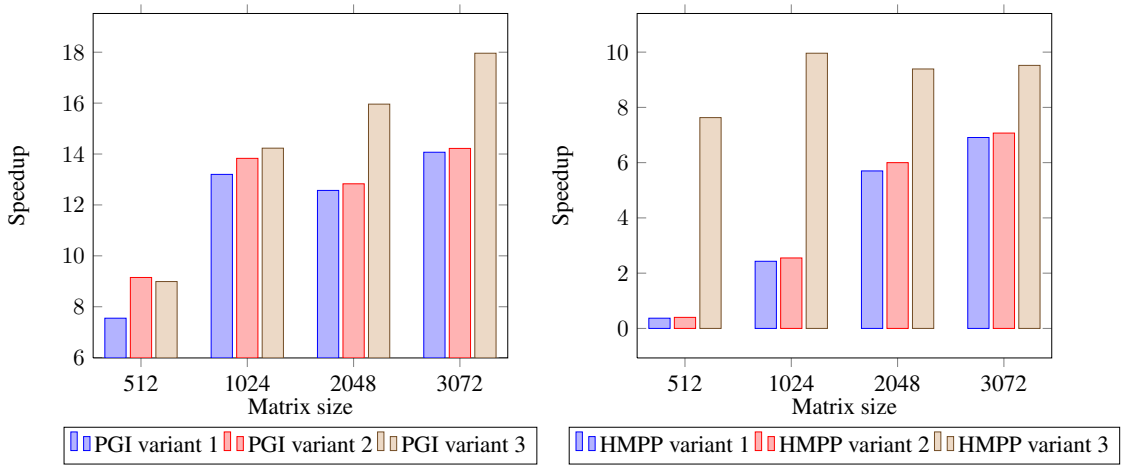
Figure 5.32 illustrates the detailed speedups achieved through three step-wise variants, which are implemented with different approaches during the second experiment.

As Figure 5.32(a) and Figure 5.32(b) illustrate, there is a slight speedup increase between their first and second variants, as was pointed out, a better performance can be obtained by lower data transfer costs. In addition, with managing their loop mapping process, the performance from the second variant and the third variant grow by a significant amount. It can be seen from Figure 5.32(c) that the first OpenMPC variant achieves poor performance in comparison to the other OpenMPC variants. The second and third variants do improve their respective performance, but these improvements are limited. With a modified input program, the speedup grows remarkably in the last variant.

Figure 5.33 illustrates a performance comparison between the hand-written CUDA program and other implementations. As in the first experiment, the best performances of these three high-level programming supports are chosen in order to compare their performance. As can be seen from this figure, the hand-tuned CUDA program achieves the best performance in every matrix size. The PGI Accelerator compiler can get a better performance than the HMPP Workbench and OpenMPC compiler, it achieves up to 68.83% (63% on average) of the performance of the CUDA program, and the HMPP Workbench achieves up to 58.39% (42.7% on average) of the performance of the hand-written CUDA program. The OpenMPC has a very similar performance to HMPP, and it achieves up to 57.89% (40.5% on average) of the hand-tuned CUDA program.

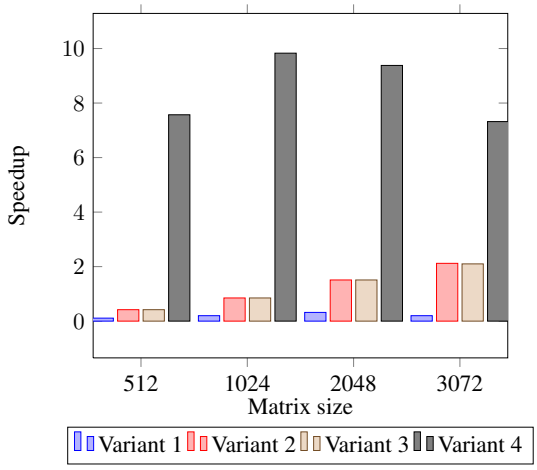
5.5 Discussion

In the experiments, three high-level compiler directive-based programming approaches were evaluated step-wise in order to achieve better performance and discover their programming features. Although the hand-written CUDA versions have the best performance as expected, this



(a) PGI Accelerator

(b) HMPP Workbench



(c) OpenMPC compiler

Figure 5.32: Speedups of the PGI Accelerator, HMPP Workbench, and OpenMPC compiler implementations of the experiment 2. Speedups are over CPU serial implementations.

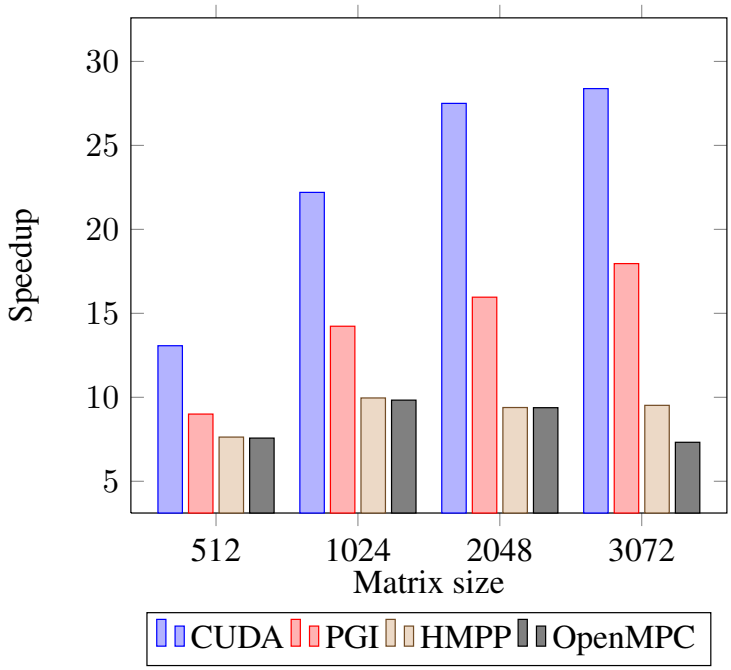


Figure 5.33: Performance comparison between of hand-written CUDA, PGI Accelerator, HMPP Workbench, and OpenMPC compiler implementations of the experiment 2. Speedups are over CPU serial implementations.

high performance can be only obtained with good programming experience and expert hardware knowledge. In addition, the time spent on the CUDA implementations is much longer than the high-level programming models, especially, by manual performance tuning.

As can be seen from the experimental results, all three high-level programming models - PGI Accelerator, HMPP Workbench, and OpenMPC - can achieve good performance. They do not require any knowledge of CUDA or OpenCL, however, this extra knowledge could help finding an appropriate optimization strategy because of their close relationship with this low-level programming support.

The experiments present two main ways to improve performance: reducing data movement costs and tuning loop schedules. As can be seen from the experiments, by default, the data movement between the CPU and GPU memory is implicit and managed by the PGI and HMPP compiler. However, this is not always efficient. This can cause unnecessary data copying. This movement is costly and the memory transfer overhead could be a performance bottleneck. Thus, minimizing the data movement can lead to better performance. In the PGI Accelerator programming model, explicit data regions or data clauses can assist developers in optimizing the data traffic. The PGI data region is a critical feature in its high-level concept. It can contain other compute regions or data regions. The data region can keep data on the accelerator memory to exploit data reuse. HMPP Workbench does not have this data region concept, but with its `advancedload` and `delegatedstore` compiler directives can be used to explicitly control

the data transfer between memories, and its concept of a group of codelets can improve data reuse.

In general, the loop mapping task is much more complex than managing data movement. The PGI Accelerator compiler does not have enough control over this mapping process. Although the compiler will determine an appropriate schedule to map the loop level parallelism onto accelerator parallelism, this is not always optimal. In contrast, the HMPP Workbench programming model offers a set of codelet generator directives that can more explicitly control this process, but it is still hard to find an optimal solution. In some cases, the optimization process needs to be repeated in order to find an appropriate value, for example, the vector width in PGI or the 'blocksize' in HMPP. According to the experimental results, such values, which achieve higher occupancy, can lead to better performance.

OpenMPC is a relatively different programming model compared to the PGI and HMPP programming model. It takes an OpenMP program as the input and translates the source code into a CUDA program. It takes advantage of OpenMP and it can improve the usage scope of the OpenMP programming model. When comparing with PGI and HMPP, OpenMPC can generate a readable CUDA program. Another advantage of OpenMPC is that it provides a tuning framework based on its compiler directives and environment variables. At the moment, this tuning system does not work fully automatic, and it generates a number of configuration files that will be used by programmers to manually create an output program. In OpenMPC, the knowledge of CUDA programming is required in some levels. Currently, the OpenMPC does not exploit the shared memory on the accelerator. The OpenMP shared and threadprivate data are mapped directly on global memory, thus, its performance depends deeply on the global memory access pattern. Uncoalesced global memory access means performance degradation. OpenMPC has developed some optimization options, such as parallel loop-swap or matrix transpose, which may reduce memory access. In addition, one of the things that can also help improve performance is the adding of OpenMP directive clause `collapse` before multiple nested loops in order to create large iteration space.

From above, Table 5.25 gives an overview of the main features of these evaluated high-level programming models.

	PGI Accelerator	HMPP Workbench	OpenMPC
Category	Directive-based, commercial product	Directive-based, commercial product	Directive-based/ source-to-source compiler, open source
Hardware targets	CUDA-enabled GPUs, Intel Xeon Phi co-processors(2013)	Multiple targets, including CUDA-enabled GPUs, AMD GPUs, Cell processor etc.	CUDA-enabled GPUs
Strength	Concept of data region for data reuse	More explicit HMPPCG kernel optimization directives	Generate CUDA program
User community, documentation and reference materials	Growing, more detailed	Small, limited information	No user community, limited information
More suitable for use in	Small programs	Well-structured programs	Both
Outputs	Executables, CUDA kernel functions(optional)	Executables, kernel functions	CUDA programs
Maintenance	Easy	Easy	Average
Data movements	Implicit/explicit	Implicit/explicit	Implicit/explicit
Loop mapping	Implicit/explicit	Implicit/explicit	Implicit/explicit
Compiler feedback & Profile information	More detailed	Detailed	Less detailed

Table 5.25: Overview over the main features of the evaluated high-level programming models

CHAPTER 6

Other related high-level programming approaches

This chapter reviews six different high-level programming models that are related to the topic of this thesis.

6.1 hiCUDA

The hiCUDA (for high-level CUDA) project defined a high-level directive-based language for programming for NVIDIA's CUDA-enabled GPUs [Han and Abdelrahman, 2011]. The University of Toronto first introduced this programming model in 2009, and it is an ongoing research project. It provides a set of compiler directives specified by using the `#pragma` mechanism. According to the different purpose, the hiCUDA directives can be divided into two groups: directives used in its computation models (kernel, loop_partition, singular and barrier) and in its data model (global, constant shared) [Han and Abdelrahman, 2011]. These directives are used to assist the hiCUDA compiler to identify a code region that should be offloaded to the GPU, and to specify variables in different GPU memory spaces (texture memory is not supported yet). One of the advantages of this programming model is that it allows programmers to explicitly exploit the GPU shared memory to improve performance.

However, in comparison to other high-level programming models, hiCUDA's API does not reach the same level of abstraction as other high-level approaches. With the hiCUDA, in many cases, developers can find a one-to-one mapping between the original usual CUDA operations and hiCUDA directives. Its annotations are still low-level and very closely coupled with the CUDA programming model.

6.2 accULL

The accULL compiler framework [Reyes et al., 2012] is an implementation of the new OpenACC standard, which was introduced in Chapter 3. The accULL programming model may be the first implementation of the OpenACC with support for both CUDA and OpenCL platforms [Reyes et al., 2012]. Supporting OpenCL is an important feature in this programming model. Thus, accULL allows programmers to easily target both multi-core CPUs and GPUs using an environment variable. It is a high-level multi-target programming model, based on the source-to-source compiler YaCF with a runtime library, called Frangollo. Currently, it has not yet fully implemented all OpenACC constructs, but non-expert user can already apply all the basic features of the OpenACC standard. As its developers have noted, they are not going to turn this research tool into a commercial product. Thus, accULL is a good choice for programmers who want to test the OpenACC API. A commercial compiler with support for OpenACC, such as the last version of PGI or HMPP compiler, is still too expensive.

6.3 Mint

The Mint [Unat et al., 2011] directive-based programming model allows the non-expert to take the advantage of the heterogeneous architecture at a high level of abstraction. Similar to the OpenMPC programming model, it includes a source-to-source translator that can translate a normal C program into an optimized CUDA program. It is built on the ROSE project[1], an open source compiler infrastructure. The Mint programming model uses a small set of mint compiler directives to assist the Mint compiler during the translation and optimization process. Unlike the OpenMPC's automatic OpenMP-to-CUDA translator, the Mint's C-to-CUDA translator needs additional assistance from programmers to identify a code region that will be accelerated on the GPU. In addition, explicit managing of data transfers between memories is always required. One of the advantages of this approach is its on-chip memory optimizer for exploiting on-chip memory, such as shared memory and registers to hide the high global memory access latency. This optimizer enables a thread block to load a block of data into in-chip shared memory. However, this programming model does not have the same level of abstraction as PGI or HMPP programming models, and it is still closely related to the low-level CUDA programming model.

6.4 Intel Array Building Blocks

Intel Array Building Blocks (ArBB) is a retargetable dynamic compilation framework [Newburn et al., 2011]. This flexible parallel programming model is designed not only for traditional multi-core architecture, but also for heterogeneous many-core architecture. Intel ArBB is one of the newest components of the Intel Parallel Building Blocks (PBB) besides Intel Cilk Plus and Threading Building Blocks (TBB).

[1] Http://rosecompiler.org

Intel's ArBB can run data-parallel vector computations on a possibly heterogeneous system [Diaz et al., 2012]. This approach supports both thread and data parallelism, and consists of a standard C++ library interface and a runtime. It is based on two programming models: the RapidMind [McCool et al., 2006] and the Intel Ct [Ghuloum et al., 2007] programming model. The most unique features of Intel ArBB are the retargetability and dynamic compilation, which are defined by its runtime system. This efficient runtime system is based on a virtual machine (VM) with a just-in-time (JIT) compiler. By the dynamic code generation, the VM can target different architecture. It supports the code generation for various targets, including SSEn, AVX and the ISA for MIC architecture [Newburn et al., 2011].

6.5 WootinJ

WootinJ [Ioki et al., 2012] is a Java-based programming system for CUDA, developed by the Tokyo Institute of Technology. Unlike other Java to CUDA approaches, such as JCUDA [Yan et al., 2009] with a CUDA-still notation like the angle brackets $<<<$ and $>>>$ to identify a kernel function call, or the Jcuda [jcuda.org, 2013] requires hand-written CUDA kernels, WootinJ allows programmers to write their applications in pure Java mode. WootinJ consists of Java-to-CUDA runtime translator that can generate a CUDA code from a Java method at runtime. First, the WootinJ translator builds an abstract syntax tree (AST) from the Java bytecode at runtime, and then from this AST, this translator will create the output CUDA code. This programming model has defined the method `CUDAKicker.run` to tell the compiler which kernel method should be offloaded to the GPU. Unfortunately, this `run` method requires the same details as an execution configuration in CUDA programming model, and the data transfers before and after launching a kernel method on GPU are not fully automatically, programmers need always to call the method `CUDAData.get` to get data from GPU memory to CPU memory.

6.6 PAR4ALL

PAR4ALL [HPC Project, 2013] has positioned itself as a fully automatic approach that can generate not only OpenMP program for multi-core processors but also CUDA or OpenCL program for hardware accelerators from C and Fortran source code. PAR4ALL is a non-commercial source-to-source compiler which is mainly based on PIPS [Amini et al., 2011] compiler framework. Unlike other approaches which require the use of complier directives, for example, in PGI or HMPP programming model, this approach performs an automatically translation process. The PAR4ALL compiler `p4a` is completely independent of target architecture. Programmers can use a different compiler option `-openmp`, `-cuda` or `-opencl` for automatic parallelization with OpenMP, CUDA or OpenCL generation. This compiler can produce a corresponding output program source with an executable. This approach is very easy to apply compared to other high-level programming models.

CHAPTER 7

Conclusions

Heterogeneous systems can achieve good performance in many application fields. Generally, there are three practical reasons that HPC has a keen interest in using various accelerators: high-performance, low-cost with less power consumption and the increased programmability. In practice, to obtain the performance benefits, it is still difficult and expensive. In fact, the lack of a suitable programming model is the source of the difficulty. Basic programming support for heterogeneous systems, such as CUDA or OpenCL, is very popular and widely used, but writing an optimized CUDA or OpenCL program is really complicated, even for experienced developers.

In this thesis, different high-level parallel programming models for heterogeneous systems are presented. They represent the direction of the development of programming support for these systems. Three of them, PGI Accelerator, HMPP Workbench and OpenMPC, were evaluated through two different experiments, and hand-written CUDA implementations were used as references. As these two experiments show, the high-level programming approaches for heterogeneous systems allow developers to more easily and efficiently to implement applications on those systems due to their higher level of abstraction. Programmers, who want to get the parallel computational power of these systems, do not have to spend a lot of time studying the hardware characteristics and dealing with low-level programming details, such as CUDA or OpenCL. For now, they can focus on the hardware-independent application code itself without increasing the burden on them.

As can be seen from the experiments, this high-level support for heterogeneous systems can offer a high productivity solution with decent performance. With a few compiler directives or compiler options, a native program, which can previously only be run on a single core or multi-core, can then be easily accelerated on the hardware accelerator such as GPUs. With a few new lines, directive-based programming models PGI Accelerator and HMPP Workbench compiler can automatically port a native C program to be run on an accelerator. Without modification, an original OpenMP program can be translated by the source-to-source compiler OpenMPC into a CUDA program. This shows an important aspect of improving reusability of many existing

codes. With the high-level programming support, a traditional C or OpenMP program can easily be ported to run on a heterogeneous hardware environment without changing the original code structure. In this thesis, the step-wise experimental design shows two critical factors for improving performance: minimizing data movement costs and controlling the loop mapping. For the first critical performance factor, all the tested high-level approaches can provide appropriate optimization methods to avoid unnecessary copying. For the second factor, although they also allow many optimization opportunities, they might need further development in the future.

PGI accelerator, OpenACC programming model, OpenMPC and other related programming models offer a high-level API for programming on heterogeneous systems, but they are still in the stage of initial development. They exploit only a subset of capabilities of the underlying hardware. In addition, the hardware accelerator cannot be completely invisible to programmers yet. Although the knowledge of basic support approach, like CUDA or OpenCL, is not required, a basic knowledge of them will help developers to improve the performance by using these high-level programming models. A good programming model should stay hardware-independent. The current high-level programming models have noticed it too. However, unfortunately, the fine-tuning the performance is still application and hardware dependent. The performance portability cannot be always guaranteed.

For most developers, high-level programming support for heterogeneous systems should be more attractive. The experimental results show that there is a small performance gap between these high-level programming models and low-level CUDA implementations. However, the performance of high-level approaches will be for many users acceptable because of their higher usability and productivity.

Bibliography

[AMD, 2013] AMD (2013). Amd accelerated processing units. `http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx`. Accessed: 04.2013.

[Amini et al., 2011] Amini, M., Ancourt, C., Coelho, F., Creusillet, B., Guelton, S., Irigoin, F., Jouvelot, P., Keryell, R., and Pierre, V. (2011). Pips is not (just) polyhedral software. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, Chamonix, France.

[Barker et al., 2008] Barker, K., Davis, K., Hoisie, A., Kerbyson, D., Lang, M., Pakin, S., and Sancho, J. (2008). Entering the petaflop era: The architecture and performance of roadrunner. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –11.

[Branover et al., 2012] Branover, A., Foley, D., and Steinman, M. (2012). Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37.

[Buck et al., 2004] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA. ACM.

[CAPS enterprise, 2012] CAPS enterprise (2012). Hmpp directives, reference manual. HMPP Workbench 3.0.

[Daga et al., 2011] Daga, M., Aji, A. M., and Feng, W.-c. (2011). On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 141–149, Washington, DC, USA. IEEE Computer Society.

[Damaraju et al., 2012] Damaraju, S., George, V., Jahagirdar, S., Khondker, T., Milstrey, R., Sarkar, S., Siers, S., Stolero, I., and Subbiah, A. (2012). A 22nm ia multi-cpu and gpu system-on-chip. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 56 –57.

[Diaz et al., 2012] Diaz, J., Munoz-Caro, C., and Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Trans. Parallel Distrib. Syst.*, 23(8):1369–1386.

[Fang et al., 2011] Fang, J., Varbanescu, A., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225.

[Farber, 2011] Farber, R. (2011). *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Ghuloum et al., 2007] Ghuloum, A., Sprangle, E., Fang, J., Wu, G., and Zhou, X. (2007). Whitepaper, ct: A flexible parallel programming model for tera-scale architectures. Online available: `http://www.intel.com/content/www/us/en/research/intel-labs-ct-parallel-programming-paper.html`. Accessed: 04.2013.

[Glaskowsky, 2009] Glaskowsky, P. N. (2009). Nvidia's fermi: The first complete gpu computing architecture. White paper, prepared under contract with NVIDIA Corporation.

[Han and Abdelrahman, 2011] Han, T. D. and Abdelrahman, T. S. (2011). hicuda: High-level gpgpu programming. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):78–90.

[Heaney, 2012] Heaney, B. (2012). Dac 2012 keynote: Designing a 22nm intel architecture multi-cpu and gpu. Online available: `http://www.dac.com/App_Content/files/49/Brad_Heaney_slides_DAC%202012.pdf`. Accessed: 04.2013.

[HPC Project, 2013] HPC Project (2013). Par4all automatic parallelization. `http://www.par4all.org`. Accessed: 04.2013.

[INTEL, 2013] INTEL (2013). Intel many integrated core architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`. Accessed: 04.2013.

[Ioki et al., 2012] Ioki, M., Hozumi, S., and Chiba, S. (2012). Writing a modular gpgpu program in java. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS '12, pages 27–32, New York, NY, USA. ACM.

[Jang et al., 2011] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. (2011). Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105 –118.

[jcuda.org, 2013] jcuda.org (2013). Java bindings for cuda. `http://www.jcuda.de`. Accessed: 04.2013.

[John L. Hennessy, 2011] John L. Hennessy, D. A. P. (2011). *Computer Architecture, 5th Edition A Quantitative Approach*. Morgan Kaufmann.

[Kirk and Hwu, 2010] Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Kogge et al., 2008] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., and Others (2008). Exascale computing study: Technology challenges in achieving exascale systems. TR-2008-13.

[Kumar et al., 2005] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. (2005). Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38.

[Lee and Eigenmann, 2010] Lee, S. and Eigenmann, R. (2010). Openmpc: Extended openmp programming and tuning for gpus. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –11.

[Lee and Eigenmann, 2012] Lee, S. and Eigenmann, R. (2012). Openmpc: Extended openmp for efficient programming and tuning on gpus. *International Journal of Computational Science and Engineering*.

[Lee et al., 2009] Lee, S., Min, S.-J., and Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA. ACM.

[Linderman, 2009] Linderman, M. D. (2009). *A programming model and processor architecture for heterogeneous multicore computers*. PhD thesis, Stanford, CA, USA. AAI3351459.

[Mark et al., 2003] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907.

[Marroquim and Maximo, 2009] Marroquim, R. and Maximo, A. (2009). Introduction to gpu programming with glsl. In *Proceedings of the 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI-TUTORIALS '09, pages 3–16, Washington, DC, USA. IEEE Computer Society.

[Mattson et al., 2004] Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for parallel programming*. Addison-Wesley Professional, first edition.

[McCool et al., 2004] McCool, M., Du Toit, S., Popa, T., Chan, B., and Moule, K. (2004). Shader algebra. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 787–795, New York, NY, USA. ACM.

[McCool et al., 2006] McCool, M. D., Wadleigh, K., Henderson, B., and Lin, H.-Y. (2006). Performance evaluation of gpus using the rapidmind development platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA. ACM.

[Newburn et al., 2011] Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., Guo, P., Liu, Z., and Zhang, D. (2011). Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA. IEEE Computer Society.

[Nickolls and Kirk, 2008] Nickolls, J. and Kirk, D. (2008). *Graphics and Computing GPUs. Appendix A of Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[NVIDIA, 2009] NVIDIA (2009). Whitepaper, nvidia's next generation cuda compute architecture: Fermi. Online available: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`. Accessed: 04.2013.

[NVIDIA, 2012] NVIDIA (2012). Nvidia geforce 8 series. Online: `http://www.nvidia.com/page/geforce8.html`. Accessed: 12.2012.

[NVIDIA, 2013] NVIDIA (2013). The world's first gpu. Online: `http://www.nvidia.com/page/geforce256.html`. Accessed: 04.2013.

[OLCF, 2013] OLCF (2013). Oak ridge leadership computing facility - titan user guide. `https://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/`. Accessed: 04.2013.

[Oneppo, 2007] Oneppo, M. (2007). Hlsl shader model 4.0. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 112–152, New York, NY, USA. ACM.

[OpenACC, 2011] OpenACC (2011). The openacc application programming interface, v1.0. Online available: `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`. Accessed: 04.2013.

[Patterson and Hennessy, 2008] Patterson, D. A. and Hennessy, J. L. (2008). *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition.

[Reyes et al., 2012] Reyes, R., Lopez, I., Fumero, J. J., and de Sande, F. (2012). accull: An user-directed approach to heterogeneous programming. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 654–661, Washington, DC, USA. IEEE Computer Society.

[Rogers, 2011] Rogers, P. (2011). Amd fusion developer summit 2011 keynote: The programmer's guide to the apu galaxy. Online available: `http://amddevcentral.com/afds/assets/keynotes/Phil%20Rogers%20Keynote-FINAL.pdf`. Accessed: 04.2013.

[Rotem et al., 2012] Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A., and Weissmann, E. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro, IEEE*, 32(2):20 –27.

[Sakaugen, 2010] Sakaugen, K. (2010). Isc 2010 keynote: Petascale to exascale. extending intel's hpc commitment. Online available: `http://download.intel.`

com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf. Accessed: 04.2013.

[Sakaugen, 2011] Sakaugen, K. (2011). Isc 2011 keynote: Accelerating the path to exascale. Online available: http://newsroom.intel.com/docs/DOC-2152. Accessed: 04.2013.

[Schaa and Kaeli, 2009] Schaa, D. and Kaeli, D. (2009). Exploring the multiple-gpu design space. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA. IEEE Computer Society.

[Seiler et al., 2008] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. (2008). Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA. ACM.

[Spafford et al., 2012] Spafford, K. L., Meredith, J. S., Lee, S., Li, D., Roth, P. C., and Vetter, J. S. (2012). The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 103–112, New York, NY, USA. ACM.

[The Portland Group Inc., 2010] The Portland Group Inc. (2010). Whitepaper, pgi accelerator programming model for fortran and c, v1.3.

[Thompson et al., 2002] Thompson, C. J., Hahn, S., and Oskin, M. (2002). Using modern graphics architectures for general-purpose computing: a framework and analysis. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Top 500, 2013] Top 500 (2013). Top 500 supercomputer sites. http://www.top500.org. Accessed: 04.2013.

[Unat et al., 2011] Unat, D., Cai, X., and Baden, S. B. (2011). Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA. ACM.

[Vuduc and Czechowski, 2011] Vuduc, R. and Czechowski, K. (2011). What gpu computing means for high-end systems. *Micro, IEEE*, 31(4):74 –78.

[Wang et al., 2007] Wang, P. H., Collins, J. D., Chinya, G. N., Jiang, H., Tian, X., Girkar, M., Yang, N. Y., Lueh, G.-Y., and Wang, H. (2007). Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 156–166, New York, NY, USA. ACM.

[Wolfe, 2010] Wolfe, M. (2010). Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA. ACM.

[Yan et al., 2009] Yan, Y., Grossman, M., and Sarkar, V. (2009). Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg. Springer-Verlag.

[Yuffe et al., 2011] Yuffe, M., Knoll, E., Mehalel, M., Shor, J., and Kurts, T. (2011). A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264 –266.

Lebenslauf

Name:	Bo Wu
Staatsangehörigkeit:	China

Ausbildung

2000-2004	Bachelorstudium Communication Engineering, Hunan University, China
2005-2006	Deutsch-Intensivkurse, Vorstudienlehrgang der Wiener Universitäten
2006-2009	Bakkalaureatsstudium Technische Informatik, Universität Wien, TU Wien
seit 2009	Masterstudium Scientific Computing, Universität Wien