OBJECT-ORIENTED STREAM PROGRAMMING USING ASPECTS: A HIGH-PRODUCTIVITY PROGRAMMING PARADIGM FOR HYBRID PLATFORMS

by

MINGLIANG WANG

A dissertation submitted to the
Graduate School – New Brunswick
Rutgers, The State University of New Jersey
In partial fulfillment of the requirements
For the degree of
Doctor of Philosophy
Graduate Program in Electrical and Computer Engineering
Written under the direction of
Professor Manish Parashar
And approved by

..................................................

..................................................

..................................................

..................................................

New Brunswick, New Jersey

JANUARY, 2011
ABSTRACT OF THE DISSERTATION

Object-Oriented Stream Programming using Aspects: a High-Productivity Programming Paradigm for Hybrid Platforms

By MINGLIANG WANG

Dissertation Director:
Professor Manish Parashar

The move to massively parallel hybrid platforms, such as multicore CPUs accelerated with heterogeneous GPU co-processing systems, is significantly impacting software programmers because existing programs have to be properly parallelized before they can take advantage of these advanced processing architectures. However, using current programming frameworks such as CUDA leads to tangled source code that combines code for the core computation with that for device and computational kernel management, data transfers between memory spaces, and various optimizations. In this research, we propose a programming system based on the principles of Aspect-Oriented Programming, to un-clutter the code and to improve programmability of these heterogeneous parallel systems. Specifically, we use a standard Object-Oriented language to describe the core computations and aspects to encapsulate all other support functions, such as parallelization granularity and memory access optimization. An aspect-weaving compiler is then used to combine the core OO program with these aspects to generate parallelized programs. This approach modularizes concerns that are hard to manage using conventional programming frameworks such as CUDA, has a small impact on existing program structure as well as performance,
and as a result, simplifies the programming of accelerator-based heterogeneous parallel systems. Studies on example programs suggest that programs written using this system can be successfully translated to CUDA programs for execution on a CPU + GPU co-processing system with comparable performance. The performance of the translated code achieved ~80% of the hand-coded CUDA programs.

We also introduce a performance model based on Bulk Synchronous Parallel (BSP) to help with quick identification of performance bottlenecks and tuning programs for better performance. This model defines a machine parameter (Machine Characteristic Ratio) and an application parameter (Application Characteristic Ratio) to identify the principle factors that can be used to bound application performance for the hierarchical parallel execution in the GPU co-processing device.
ACKNOWLEDGEMENT

The completion of this research owes it to many people who offered their invaluable help and suggestions. I would like to gratefully thank my advisor Professor Manish Parashar for his continuous enthusiasm, insightful guidance, incessant encouragement, and invaluable advice during the course of this work in The Applied Software Systems Laboratory (TASSL) and the NSF Center for Autonomic Computing (CAC), and my graduate study at Rutgers University.

Many thanks to the members of my committee, Professor Zhang Yanyong, Professor Ulich Kremer, and Professor Dario Pompili for taking time in their busy schedules to provide advice on and feedbacks to my thesis.

I would also like to thank Dr. Hector Klie of ConocoPhillips and University of Texas at Austin for his guidance and help in the work completed at ConocoPhillips.

Thanks to all the great people at CAIP, CAC and the ECE department, Judy Pelicane, Jim Housell, Bill Kish, Lynn Ruggiero, to name a few, for making all administrative and technical issues non-issues.
# Table of Contents

Chapter 1 Introduction .......................................................................................................................... 1

1.1 Moving to Heterogeneous Massive Parallelism ................................................................. 1

1.2 Requirements of Parallel Programming Support ............................................................ 5

1.3 Overview of the Research ..................................................................................................... 8

Chapter 2 Background and Related Work for Stream Programming ................................. 13

2.1 Architectures and Models for Parallel Computation ........................................................ 13

2.2 Support for Stream Programming ....................................................................................... 17

2.2.1 Language Extensions ........................................................................................................ 17

2.2.2 Embedded Languages ....................................................................................................... 19

2.2.3 New Languages .................................................................................................................. 20

2.2.4 Libraries ............................................................................................................................. 21

2.2.5 Compiler Directives ........................................................................................................... 22

2.3 Summary ................................................................................................................................. 23

Chapter 3 Stream Programming in Object-Oriented Programming Using Aspects ............... 26

3.1 Execution Model ..................................................................................................................... 28

3.2 Join Point Model for Accelerator Programming ............................................................... 29

3.3 Intrinsic Data Types ............................................................................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Computation Kernels and Execution</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Data Transfers</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Compiling Process</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>The Compilation Architecture</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>Code Weaving</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Computation Kernel Generation</td>
<td>40</td>
</tr>
<tr>
<td>3.10</td>
<td>Optimization</td>
<td>40</td>
</tr>
<tr>
<td>3.11</td>
<td>Tooling Support</td>
<td>41</td>
</tr>
<tr>
<td>3.12</td>
<td>Examples and Evaluations</td>
<td>42</td>
</tr>
<tr>
<td>3.12.1</td>
<td>N-Body Problem</td>
<td>42</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Evaluations</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Models of Parallel Programming and Processing</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Advantage of CPU + GPU Co-Processing</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Model of CPU + GPU Co-Processing</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>Computation Model within a GPU Co-Processor</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Computation Model within a Thread Block</td>
<td>66</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Computation Model of a Grid of Thread Blocks</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance Prediction for CPU + GPU Co-Processing</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Comparison of existing programming systems ....................................... 23
Table 2: Runtime comparison of n-body problem simulation ................................ 50
Table 3: Comparison of structural changes .......................................................... 51
Table 4: Comparison of n-body, line-of-code ...................................................... 52
Table 5: CPU + GPU co-processing and Amdahl's Law ...................................... 61
Table 6: GPU Co-Processing Considered in the BSP Model ............................... 71
Table 7: Blocked Matrix Multiplication in the BSP Model ................................... 72
Table 8: Machine Characteristic Ratios for two generations of Nvidia GPU’s ....... 77
Table 9: Performance of Dense Matrix Multiplication ........................................ 80
Table 10: Monte Carlo simulation for pricing European Stock Options ............. 84
Table 11: Commonly used optimization technique ............................................. 88
List of Illustrations

Figure 1: Moving to heterogeneous manycore co-processing future...................... 2
Figure 2: A heterogeneous CPU + GPU co-processing system.................................. 3
Figure 3: Requirements of parallel programming systems...................................... 7
Figure 4: Weaving component code with aspect code........................................... 10
Figure 5: Execution model of a heterogeneous CPU + GPU system....................... 28
Figure 6: Parallel methods executed as a grid of thread blocks............................ 29
Figure 7: Compiling architecture of the accelerator programming system.............. 38
Figure 8: Fusing kernels to improve arithmetic intensity......................................... 41
Figure 9: OOSP tooling support .................................................................................. 42
Figure 10: Class diagram of N-body simulation...................................................... 45
Figure 11: Call graph of N-body CPU implementation........................................... 46
Figure 12: Call graph of the OOSP implementation................................................ 47
Figure 13: Performance comparison of options pricing ........................................... 51
Figure 14 Parallel Programming models. ................................................................. 56
Figure 14 Program execution vs. clock frequency, a CPU-bound program............. 57
Figure 15: Achievable speedup of CPU+GPU co-processing................................. 60
Figure 16 CPU/GPU co-processing program structure ........................................... 63
Figure 17: An architectural model of GPU............................................................... 65
Figure 18 A simple parallel addition kernel......................................................... 68
Figure 19: A super step in BSP and cost model...................................................... 70
Figure 20: Matrix Multiplication using blocking................................................... 73
Figure 21: Performance modelling of Nvidia Fermi ............................................... 76
Chapter 1
Introduction

In 1965, Gordon Moor predicted that the number of transistors that can be placed inexpensively on an integrated circuit will double every 18 to 24 months [1]. Until recently, this law translates to increases in clock speed and performance gain of a single processing element and it is considered the free lunch [2] period for software programmers: the same sequential program will automatically gain performance just by running it on a piece of faster hardware. Single thread performance improvement is realized by scaling up the clock speed and automatic exploitation of Instruction Level Parallelism (ILP).

Automatic extraction of ILP from serial programs is rather limited [3] and scaling up clock frequency stopped working in circa 2005 when memory speed could no longer catch up with the increase of processing capability and power consumption became prohibitively high, which are figuratively described as memory wall and power wall, respectively. The new model of performance improvement has since converted to adding more processing cores into a single microprocessor package in order to exploit thread level parallelism (TLP).

1.1 Moving to Heterogeneous Massive Parallelism

Mainstream desktop and notebook computers in 2006 started to being shipped with dual-core processors and higher-end desktops and workstation computers have since 2008 begun to have quad-core processors. Additionally, in the area of specialized computation,
for instance graphics, Graphic Processing Units (GPUs) already contain tens to hundreds of processing cores. CPUs are also evolving into the manycore direction [4]. For better power efficiency and performance, even processors in ultra-portable devices, such as mobile phones, are heading toward multicore processors [7].

Figure 1: Moving to heterogeneous manycore co-processing future.

In addition to the increase in core count, another technology trend in processor is the collocation of heterogeneous cores in a single system. This is driven by the demand to increase power efficiency [5]. As stated in [6], to scale multicore processor design to incorporate a large number of cores, ultra-low EPI (energy per instruction) cores are essential. General purpose CPU cores are designed to manage complex control flows and to maximize the exploitation of ILP, leading to EPI’s unacceptable to scale to the manycore future. For instance, the Core 2 Duo has an EPI of ~10nj [8] and to achieve a 10X performance gain, the power consumption will become prohibitively high (~1kWatt). However, simpler cores, such as the 8-core 32 threads Intel GMA X3000 [9], which is used for graphic computation, has a much smaller EPI of ~0.3nj, making it practical for scaling to manycore design.
This trend has already been verified by actual product releases and plans, for instance, STI Cell B.E. processor is a combination of a general purpose PowerPC core with 8 simpler cores dedicated to floating points, and NVIDIA Tegra processor [10] is a combination of an ARM CPU core with a manycore GPU. Intel Larrabee [11] is also taking on the same approach.

Computing system builders are combining heterogeneous processors to optimize their systems as well. For example, Roadrunner, the fastest computer in the world as of June 2009 adopts a heterogeneous design [12], which includes both Opteron and Cell B.E. processors on each node of the super computer.

Following this line of processor and computer system design evolution, future generations of computer systems will have many processing cores with different processing characteristics. For example, it is highly likely the next major mainstream computer will have a few general purpose cores with simultaneous multithreading for running complex operating system software and task parallelism, and hundreds of simpler cores suitable for simple throughput computation tasks, such as those that appear in graphics and numeric computation. A heterogeneous processing system on a single node can be schematically illustrated in Figure 2.

![Figure 2: A heterogeneous CPU + GPU co-processing system.](image-url)
Challenges

This unavoidable move to massively parallel hardware is making huge impact on software programmers because existing sequential programs will not be able to perform better on parallel hardware unless they are properly parallelized. This is an entirely different picture for computer programmers because in the past, even a poorly written program would speed up when processor clock rates increased. With the manycore parallel processing sea change, a program will not automatically achieve performance gains if more processing cores are added. Instead, it may actually run slower because single core clock rate may reduce for better power efficiency.

In addition to parallelism, programmers also need to take care of the characteristics of the heterogeneity in these processors so that mapping algorithms to these hardware resources are efficient. For example, general purpose cores are suitable for complex control flows but with limited throughput and poor power efficiency; whereas many specialized cores can work together to efficiently do simple computation at orders of higher speed. Depending on different throughput and latency requirements, processing resources can be switched on and off to optimize power consumptions. Furthermore, memory access has a huge impact on computation performance as the hierarchical nature [13] [14] of memory system relies on programs to present good behaviors (locality and suitable working sets) to function efficiently.

Existing mainstream programming systems (models, languages, tools, libraries, runtime) are legacy from the early age of computation where heterogeneity, parallelism and memory access characteristics were not taken into account. Facing the coming massive
parallelism transition, we need to identify requirements to cope with these challenges and then fix them.

1.2 Requirements of Parallel Programming Support

Adoption of parallel programming in software development is driven by several factors and understanding them is the first step to solve these challenges. In the following, we list these drivers and then summarize the requirements for a parallel programming system.

*Programmer Productivity* directly determines the software development process efficiency and affects the overall cost and the time-to-market of software projects. In parallelizing existing software, the extra efforts required in adapting algorithms to parallel hardware, learning new tools, conducting performance tuning and dealing with extra bugs introduced, are all negatively affecting programmer productivity.

*Program Performance* measures how efficient a software program can finish a given task on a given hardware platform. On parallel hardware, if a parallelized program can deliver perceivable performance gain over a sequential one, the parallelization efforts are justifiable and will promote the adoption of parallel programming.

*Legacy Integration* or backward compatibility is crucial to drive the adoption of a particular programming methodology. It is unrealistic to require a development team to completely rewrite programs consisting of hundreds of thousands of or more lines of code. Instead, parallelization should be an incremental process that starts from the portions of programs that have big impact on performance and also bear small risk. The model of programming should also fit in existing development process. For example,
mainstream software development adopts object-oriented methodologies and any new parallel programming system should work in harmony with it.

We discuss the requirements in the context of a programming system, as it offers the interface for programmers to build software systems. It includes programming models, performance model, programming languages, tools (editor, compiler, debugger, profiler, and etc.), libraries, and runtimes. A *Programming Model* [15] is an abstract model of computation used by programmers to reason about how a program executes. It abstracts physical machines to provide a convenient framework to reason about how computations proceed and their associated costs. The purpose of a programming model is to aid in designing and understanding computational tasks. A programming model is embodied in a programming system through programming languages, libraries, runtimes and tools.

In understanding these driving factors, we propose the following requirements for parallel support in a parallel programming system.

*Simple* We need a simple programming model. Programmers reason with a programming model about program executions. In sequential programming, Random Access Machine (RAM) model serves this purpose. It is sequential, safe, deterministic, and easy to understand and communicate. The abstraction of memory, processing elements, and the simple cycles of moving data and instruction between them offer a clear view of the program progress and a cost model which helps characterize program performance. So ideally, a parallel programming model should possess the same set of properties. It should not introduce extra categories of bugs due to parallelization, program executions be deterministic, and offer a sequential view of execution. The sequential view seems to
contradict with parallelism nature but it is feasible if we discuss it in a relatively high level of abstraction.

*Expressive* This programming system should also provide expressive mechanisms to manipulate the aspects that affect program performance. It requires a clear cost model and a handy mechanism in the programming system so that programmers can understand the bottleneck of program performance and tune its performance through proper constructs in a program. In parallel processing, there are two important aspects to performance: *parallelism* and *memory access*. Traditional models in mainstream programming languages lack in properly exposing these two crucial aspects. We argue for *explicit parallelism* as experience has shown that it is far easier to compile programs with explicit parallelism onto parallel hardware than to extract implicit parallelism from sequential programs. Due to the hierarchical nature of memory systems, memory access should also be tunable for a program to adapt to the underlying hardware.

![Figure 3: Requirements of parallel programming systems for mainstream programmers.](image-url)
Compatible Compatibility with existing programming systems is essential: the programming system should fit in existing programming environments, offer smooth integration, and are easy to teach and learn. As the mainstream programming paradigm is the object-oriented approach, we need to come up with a programming system that works in harmony with it.

Scalable An efficient design that does not scale to future generations of hardware will be less important. Forward scaling [16] is a software design approach that promises to deliver excellent performance on current day dual or quad core hardware and will scale up that performance on future manycore processors. Programmers should focus more on writing programs that delivers performance gain when more cores are added than on enhancing program efficiency on today’s hardware. In fact, with exponential increase in core counts, the total cost of computing is constantly lowering and hence squeezing out extreme efficiency with precious programmer effort is not a smart choice in many cases.

In summary, the identified requirements are: a programming system that supports incremental adoption in current object-oriented environments, does not introduce extra categories of bugs due to parallelization, allows expressive handling of parallelism and memory access, and continues to deliver performance gain on future generations of hardware when more and more processing cores are added.

1.3 Overview of the Research

In this research, we target at providing a scalable parallel programming system that works in accord with existing mainstream environments, so that programmers can incrementally
parallelize their programs to tap into the parallel processing power in current and future generations of hardware. We prioritize scalability over efficiency, simplicity over capabilities, and compatibility over completeness.

We adopt a simple data parallel programming model that is compatible with object-oriented paradigm. Specially, in view of the success of stream programming model in developing general-purpose GPUs (GPGPU) applications, we incorporate stream programming into the object-oriented paradigm, introducing parallel classes and parallel methods so that related data and operations are encapsulated as classes for a higher level of abstraction. In stream programming, computation is organized as a sequence of computational steps called kernels, which are independently applied to collections of data elements. Within a kernel, many computations can be combined to increase the arithmetic intensity [30], which is the ratio between the amount of computation performed and the memory bandwidth consumed. Increasing this ratio is crucial in achieving efficient computation because, in modern computers, processors have at least an order of magnitude more computational power than memory bandwidth.

We have come to realize that concerns about parallelism, such as computation granularity, and those about memory access, which highly affects program efficiency through interaction with memory hierarchy and program scheduling, are orthogonal to the main computation portion of the program. Existing programming systems do not deal with them well and the programs often suffer from the problem of tangle-of-concerns. We hence apply the principles of aspect-oriented-programming (AOP) [17] to address this problem, proposing methods to define supporting aspects that modularize parallelism and memory access concerns in stream programming.
AOP modularizes these crosscutting concerns as *aspects* that do not naturally decompose into functional units. The normal code that fits into the functional decomposition scheme is termed as *component* (or *base, mainline*) code. A special compiler - *aspect weaver* then inserts program fragments, as necessary, into the component code. The places of insertion are called *joint points* and are specified by a language construct called *pointcut*, which identifies a set of join points in the component code. The aspect code to be inserted is specified by another language construct called *advice*. The organization of aspect-oriented programs and the process of weaving aspect code into component code are shown in Figure 4.

The final woven program is highly tangled with component and aspect code fragments, but the component and aspect source programs are cleanly modularized, much more amenable to programmers.

![Figure 4: Weaving component code with aspect code.](image)

We seek for compatibility with existing programming environments, so we do not change the syntax of exiting languages. The base component of a program remains exactly the same as conventional programs. The aspect portion of the code is defined separately
using a small aspect language, which is based on extensions in AspectC++ [18]. We also define a set of intrinsic data types and functions to ease specification of parallelism and memory access aspects. The base and aspect code are fed into a source-source translation engine, which understands the correlation among the aspects with the base code and semantics of intrinsic data types and functions. The translation engine then generates another set of conventional source programs to realize the parallel semantics. The translated sources are then fed into conventional compilers for further processing.

The execution system targets at stream processing hardware GPU and general purpose multicore CPU hardware. The current compiler targets include NVIDIA CUDA and OpenCL interfaces. The generated programs are standard C++ programs with invocations to CUDA and OpenCL APIs. This part of the research is explained in Chapter 3.

Another equally important aspect in a high-productivity programming system is the performance model of a computer system. This model is useful in answering questions such as 1) what system component combinations, such as CPU, GPU, memory sub-system, interconnect sub-system, are required to achieve a performance goal, 2) what is the efficient and effective ways to program a given hardware system, 3) how to identify performance bottleneck and tune a program for better performance in software development. In this research, we introduce a performance model that helps answer, in the context of heterogeneous processing environment, the above-mentioned questions.

This model identifies information from the bulk properties of the hardware platform, such as system bus bandwidth, sustained memory bandwidth, and peak instruction execution rates, and properties of applications, such as the ratio of arithmetic operations to memory
access operations of software programs. We then correlate these several pieces of information together to disclose insightful understandings about the performance of this particular combination of hardware components with the applications running on them. These bulk properties are subsumed into two parameters in the model: a machine parameter, Machine Characteristic Ratio (MCR) and an application parameter, Application Characteristic Ratio (ACR), to identify the principle factors that can be used to bound application performance for the hierarchical parallel execution in a heterogeneous platform. The application and usefulness of this model are demonstrated in Chapter 4.
Chapter 2
Background and Related Work for Stream Programming

In this chapter, we briefly discuss the processing models of current mainstream parallel hardware architectures and the programming models developed for them. We then focus on several popular stream programming systems, contrasting their features and discussing their deficiencies.

A *programming model* [15] is an abstract model of computation used by programmers to reason about program execution. The purpose of a programming model is to provide a suitable abstract machine so that programmers can write programs relative to this model, while at the same time programming language compilers can efficiently translate the same computation into lower level machine code for efficient execution. A *processing model* describes how a computer performs computation and is the interface provided by processor vendors to assembly programmers and compiler writers in its instruction set architecture (ISA). A programming model sits on top of processing models and a processing model, on the other hand, can also be regarded as the programming model exposed by an ISA. Due to this layering nature, a lower level programming model could be the processing model of another higher level programming model.

2.1 Architectures and Models for Parallel Computation

Parallel processing models can be categorized into those that exploit *task parallelism* and those that exploit *data parallelism*. Task parallel processing takes the approach of decomposing a program into separate sub-tasks and of scheduling them simultaneously
on different processing cores. In contrast, data parallel processing decomposes data collections and relies on parallel execution of a set of operations on the elements of each data collection.

The current mainstream processors (Intel and AMD) have multiple processing cores in a package, usually with separate level 1 cache units for each core and shared level 2 and level 3 cache units, and some also with on-chip interconnect. This design is derived from the single-core legacy and looks like a symmetric multiprocessor system (SMP) from the programming point of view, though with different performance characteristics. The processing cores in these designs are optimized for single thread latency and devote a large percentage of die areas to exploit ILP and on-chip cache. These designs also distinguish to each other by differences in the design of memory systems and the supported on-chip inter-core connects.

The processing model of these designs is naturally biased toward task parallel computing. Each core or a thread in a core is managed by the operating system and allows for scheduling of a separate process. This is a multiple instruction multiple data (MIMD) processing model and to finish a higher level task, multiple sub-tasks usually need to communicate and synchronize with each other.

In the shared memory architecture as we can find in most standalone systems, sub-tasks communicate through the commonly accessible memory; the basic mechanism for synchronization is locking. Before accessing a region of shared memory, a sub-task needs to acquire a lock that protects that shared memory. A lock allows only one sub-task to acquire it at a time and all other concurrent locking attempts will be blocked until the
lock becomes free. Locking as a synchronization mechanism is very error-prone [20] and programs using it often exhibit problems such as deadlock, livelock, and data races. Even though a lock-based multithreaded program is correct, it can suffer problems such as over-conservatively use of locks, leading to serialization in actual execution. Transactional memory architectures [21] [22] are proposed to provide lock-free atomic operations in a multithreaded environment, but it is limited by scalability, number of operations allowed in an atomic region, and insufficient hardware support [23]. Others also proposed to apply coordination framework to multithreaded programming [20].

Task decomposition suffers from fundamental problems in parallel programming. First, it is often difficult to find many sub-tasks in a typical program. As the core counts scales up exponentially, this mismatch will only become wider. Without enough subtasks, a program will not benefit from the increase in parallel processing power, resulting to a scalability issue. Second, load balancing in task decomposition is hard to achieve as it is almost impossible to have all the subtasks to finish together. The cores that finish earlier will then have to sit idle, leading to imbalance and inefficient use of hardware. Third, a multithreaded program, even though is correct, can be nondeterministic. Determinism is required in some application area, such as financial modeling for repeatability requirement.

Existing popular programming models for these homogeneous systems include Pthreads [53] and OpenMP [52] for programming shared memory system, and Message Passing Interface (MPI) [54] for distributed memory systems. There are also other systems tailored for specific domain applications, such as MapReduce [55] for large-scale web data processing, and [56] for loosely coupled systems.
Another type of processor design, as exemplified by the Graphic Processing Unit (GPU), includes support for coordinating threads of the *Single Program Multiple Program* (SPMD) model. Unlike in a MIMD machine, where threads compete for resources, threads in a SPMD processor are coordinated as a single program in order to reduce contentions and to optimize for throughput. This is the processing model used in typical GPU architectures of current days. To achieve efficiency, threads can be suspended to hide latency in certain expensive operations, such as memory access. Thread scheduling is through hardware and incurs zero cost [24], in contrast to the expensive OS managed scheduling in MIMD processors.

These SPMD GPUs are often modeled as *stream processing*, an instance of data parallel processing. Computation is organized as a sequence of steps called *kernels*, which are independently applied to collections of data elements. Within a kernel, many computations can be combined to increase its *arithmetic intensity* [30], which is the ratio between the number of arithmetic operations performed and the memory bandwidth consumed. Increasing this ratio is crucial in achieving efficient computation because, in modern computers, processors have at least an order of magnitude more computational power than memory bandwidth. This model scales well into the massive parallelism future as a data collection is usually very large and can be simply divided into an appropriate number of chunks to match the number of cores. This model can also be made deterministic as threads executing a kernel do not communicate and collective operations for communication are entirely delegated to system component.

Due to the popularity of stream processing hardware such as GPUs, recent years have seen several programming systems to directly support stream programming. Though
stream programming imposes more restrictions on how computation can be expressed, if assisted with proper collective operations, however, it is as powerful as task parallel programming [25]. This is also verified by some recent mappings of irregular scientific applications [26].

Additionally, stream programming not only maps to stream processing hardware, it is also possible to map stream programming onto general purpose MIMD multicore processors [27]. This makes stream programming a widely implementable programming model to survive architecture changes in the ongoing evolution into heterogeneous manycore future.

In the following subsection we review existing stream programming systems, focusing on the recent systems that are derived from general purpose computing on GPUs and multicore CPUs. For a broader review of stream programming, please refer to [28].

2.2 Support for Stream Programming

We categorize these programming systems based on their mechanisms: language extensions, embedded languages, compiler directives, libraries and new languages. We differentiate them based on mechanisms and level of abstraction, and then evaluate them in terms of the requirements we laid out in Chapter 1: ‘simple’, ‘expressive’, ‘compatible’, and ‘scalable’.

2.2.1 Language Extensions

NVIDIA CUDA is a programming system based on extensions to C language for programming their GPUs as co-processing devices [29]. In this model, a program is divided into sequential portions that run on CPU, and parallel portions that run on the co-
processing GPUs. It supports a set of memory and device management APIs on CPU side through standard library mechanism, and definition of ‘kernel’ functions, which are parallel portions of a program that are executed in parallel on GPUs.

NVIDIA CUDA is by far (2010) the most widely used stream programming system, as it is flexible and widely available on major OS platforms. It has been successfully applied in many fields to speed up numerically intensive computation.

OpenCL is an ongoing effort [31] to construct a ‘foundation-level’ acceleration API for programming GPU like massively parallel hardware from different vendors. Its programming model is similar to CUDA as computation is expressed as C function-like kernels executed in parallel over collection of data stored in memory buffers. It however focuses on cross-vendor compatibility and portability.

Listing 1 is the sample CUDA program that performs matrix addition. We do not provide OpenCL example as it is essentially the same in terms of programming model and also similar in syntax.

Because OpenCL is currently supported by almost all major players in this field and its openness, we anticipate it will become the standard model of programming and API for programming accelerator with low-level mechanism.
2.2.2 Embedded Languages

RapidMind [32] (acquired by Intel in Aug. 2009) is an embedded stream programming system for C++ and was derived from an earlier experimental shading language Sh. The programming model consists of three C++ data types: Value<N, T>, Array<D, T>, and Program. An Array abstracts the concept of data collections and a Value the concept of data elements. A Program stores a sequence of operations that can operate on the Values of Arrays in parallel. RapidMind platform allows computational kernels be specified directly in the C++ language and its metaprogramming feature supports code tuning and parameterization.

The code in Listing 2 is a simple RapidMind program to compute the sum of two matrices. As we can see, the program object is declared after Program and value defined between BEGIN and END. The embedded program is later applied to Array type variables a and b, with result stored in variable a.

```
__global__ void
matAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}

int
main()
{
  // Kernel invocation
  dim3 dimBlock(16, 16);
  dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
               (N + dimBlock.y - 1) / dimBlock.y);
  matAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Listing 1: CUDA program to compute the sum of two matrices. A kernel method is defined with the modifier __global__ to a C function. Syntax is added to C so that parameters to control kernel invocation can be passed for each kernel invocation.
RapidMind platform relies on C++ template and macros mechanism to define the embedded programs. The RapidMind programming model is essentially a procedural language embedded in object-oriented C++; the embedded Program objects are sporadically defined in skeleton C++ programs and are alien to the object oriented programming paradigm.

In the same vein of embedded programming system for C++, Intel is actively developing its Ct [16] or C for throughput computing programming system. Similar to RapidMind, it is an embedded, procedural programming system in C++, with canned data type templates to support data parallel programming. Additionally, it includes direct support for nested parallelism.

```
Valuel i;
rapidmind::init();
const unsigned int w=512, h=512;
Array<2,Valuelf> a(w, h), b(w, h);

Program mul_add = BEGIN {
  In-Valuelf> i1, i2;
  Out-Valuelf> o;
  o = i1 + i2 * f;
} END;

float f;
const unsigned int w = 512, h = 512;
float a[w * h * 3], b[w * h * 3];
for (int y = 0; y < h; y++)
  for (int x = 0; x < w; x++)
    for (int e = 0; e < 3; e++)
      a[(y * w + x) * 3 + e] +=
        b[(y * w + x) * 3 + e] * f;
```

Listing 2: RapidMind program to compute the sum of two matrices. Left: RapidMind Program. Right: the same computation in sequential C.

### 2.2.3 New Languages

Brook [33] is a programming system initially developed for the Merrimac [30] streaming processor and was later adapted for programming general-purpose GPU hardware. The base part of the language is similar to C. To support stream programming, it provides
three abstractions: a ‘*stream*’ construct to represent a collection of data that can be operated on in parallel, a special type of ‘*kernel*’ functions that operate on streams, and also a collective operations ‘*reduction*’ for calculating a single value out of a set of data elements. A sample Brook program is shown in Listing 3.

```c
kernel void saxpy (float a, float4 x<>, float4 y<>,
    out float4 result<>)
{
    result = a*x + y;
}

void main(void)
{
    float a;
    float4 X[100], Y[100], Result[100];
    float4 x<100>, y<100>, result<100> ;
    // ... initialize a, X, Y ...
    streamRead(x, X); // copy data from mem to stream
    streamRead(y, Y);
    saxpy(a, x, y, result); // execute kernel on all elements
    streamWrite(result, Result); // copy data from stream to mem
}
```

Listing 3: A Brook program to compute the standard BLAS level 1 function saxy, or scaled sum of vectors. A kernel function is defined with a modifier ‘kernel’. Kernel functions are called on stream data types in the same as a conventional C function is called.

### 2.2.4 Libraries

Accelerator [34] is a library-based programming system from Microsoft to exploit GPUs as accelerators. Arithmetic intense kernels are inferred by inspecting programs and then offloaded onto GPUs. The code fragment in Listing 4 shows a sample program using Accelerator library.

Accelerator is an SIMD programming model but the compiler dynamically generate stream kernels to be offloaded onto GPU for execution. It is a library based approach hence relies on the pre-defined set of operations available for programmers, which limit flexibility. Additionally, its SIMD based programming model does not directly expose
stream kernels, eliminating the important construct for programmers to increase arithmetic intensity of performance critical programs.

```
using Microsoft.Research.DataParallelArrays;
static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);
    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i);
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }
    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0};
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }
    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Listing 4: Accelerator program in C#. 2-dimensional convolution implemented in Accelerator [34].

### 2.2.5 Compiler Directives

PGI [35] provides a set of compiler directive, similar to OpenMP, for offloading computation to GPU. Similar approach is also taken by the HMPP system [36], which defines its own set of compiler directives to generate native GPU codes. A sample program with these directives is shown in Listing 5.

```
!$acc region
    !$acc do parallel
do j = 1, m
    do k = 1, p
        !$acc do parallel, vector(32)
do i = 1, n
        a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
enddo
enddo
!$acc end region
```

Listing 5: PGI OpenMP like directives for offloading computation to GPU. This is example directives and code snippets for FORTRAN.
2.3 Summary

Using the requirements we identified in section 1.2, we summaries the comparison of these existing programming systems in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Compatible</th>
<th>Expressive</th>
<th>Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA / OpenCL</td>
<td>No</td>
<td>Yes/No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Brook GPU</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RapidMind (Intel Ct)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Compiler Directives</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The insufficiencies of these systems are explained in more details in the following.

- Firstly, there is insufficient support for OO paradigm. In the last several decades, object-oriented design and programming have dominated the software development in both industry and the academia. OO paradigm raises the level of abstraction and increases code reusability. For any programming model to be widely accepted, it is hence a requisite to smoothly integrate with OO paradigm and its programming environments. In systems like CUDA and OpenCL, where support is through extensions to the general-purpose procedural C programming environment, there is not enough abstraction and encapsulation. Additionally, these systems require programmers to handle details such GPU memory allocation/de-allocation, data transfers between GPU and CPU and low-level
device management. In fact, these systems are more suitable for compilation targets than for direct use by application developers. Although RapidMind is a C++ based solution, the embedded programming model itself is still procedural and C++ is only used to skeleton the embedded code. Array<> and Program<> data types can only exists as data members in a class and parallel semantics are sporadically buried in C++ member functions.

• Secondly, systems like Brook and RapidMind are inflexible. Both of them require an explicit conversion to stream data types through ‘read’ and ‘write’ operations. When a data structure does not map naturally to a flat ‘array’, this requirement generates severe problems in both expressiveness and runtime efficiency.

• Thirdly, they are mostly short-term and/or vendor-specific solutions. To survive the evolution of parallel computation, forward-scaling is a necessary requirement for a parallel programming system to last through the current and next few generations of computer hardware [16]. Future hardware will vary in the aspects of number of cores, thread granularity and synchronization overhead. This requires programs dynamically choose computation granularity and synchronization mechanisms. These existing programming systems do not consider this very important aspect. In terms of vendor neutrality, proprietary solutions often scare serious users away as they do not want to risk their future by locking to a single vendor.

• Lastly, we cannot wait for another programming language. A new language takes too long to get mature and accepted in mainstream software development community. Programmers need a programming system that is proved and
integrates well with their existing mainstream environments (C++, FORTRAN, Java, and .Net), but not another immature programming language.

In this following chapter, we describe an aspect-oriented programming system that addresses the afore-mentioned issues. The system uses standard C++ to program the core computational functions of a program. It then introduces a small aspect language for describing the crosscutting concerns in accelerator programming and enforces a formal separation between core functional modules and secondary aspects for supporting purposes. The issues dealt with in the aspect portion of the framework are: (1) deciding layout of the thread hierarchy for a kernel launch, (2) transferring data between memory spaces, (3) mapping data to the thread hierarchy, and (4) the placement of data within a memory hierarchy. This arrangement leads to clean separation and modularization of these crosscutting concerns. Furthermore, the aspect code is high-level, and insulates programmers from dealing with a particular hardware platform.
As discussed in the Introduction, we identified the requirements for a programming system as: “*a programming system that supports incremental adoption in current object-oriented environments, does not introduce extra categories of bugs due to parallelization, allows expressive handling of parallelism and memory access, and continues to deliver performance gain on future generations of hardware when more and more processing cores are added.*” In this section, we propose a programming system satisfying these requirements.

We propose a method to combine stream programming model with existing object-oriented programming systems so that parallel programming can be made safe, expressive, and scalable in mainstream development process.

The programming system described in the following supports accelerator programming using conventional C++ and a small aspect language. The small aspect language is largely based on AspectC++ [18] but specially customized for expressing aspects that arise from accelerator programming. This system provides the following features to help with programming GPU accelerators in a high-level and clean way, and as a result addresses the code-tangling problem.

- Computation kernels are organized as integral part of the source code, and they can be class member methods. As class members, they can also be virtual functions so as to support polymorphism of computation kernels.
• Better modularization of the program code. Auxiliary functions such as device management, determining thread hierarchy for kernel execution and data transfers are expressed in separate aspect code, leading to clean and better modularized code.

• GPU main memory is automatically managed through intrinsic data types. Memory transfers code is managed by compiler. Programmers give declarations about memory synchronization requirements and hints about changed memory buffers in computation kernels, and then the compiler derives code to start data transfers when required. This reduces programming effort and improves code quality.

The code listed in Listing 6 is a simple *Vector* class\(^1\) in C++. We will use it to illustrate the language concepts in the proposed system.

```c++
class Vector {
  public:
    void increase(float inc);
    void shift(int offset);
  private:
    unsigned int m_size;
    MemBlock<float> m_data;
    // other members omitted . . .
};

Listing 6: A C++ Vector class (incomplete) for parallel execution on an accelerator.
```

The *increase* method increases each element of the class by a constant *inc*; the shift method rotates the elements by an *offset*. The computations in these methods are data-parallel and can be offloaded to an accelerator. The data member *m_data* is the memory buffer for the elements. It is a class template provided by our programming system,

---

\(^1\) This is to illustrate the concepts of our programming system, and does not represent a good design of a full-fledged vector class.
featuring managed memory across memory spaces. Details of this mechanism will be discussed in section 3.3.

In the following subsections, we first briefly introduce the execution model of a CPU-GPU accelerator system, and then focus on the important mechanisms of the system for stream programming with aspects.

3.1 Execution Model

An execution model conceptually describes how a program makes progress at runtime. In the CPU+GPU heterogeneous parallel processing system, a computation is composed of a sequence of serial actions on the CPU and parallel computational kernels on the GPU. This execution model is illustrated in Figure 5.

![Figure 5: Execution model of a heterogeneous CPU + GPU system.](image-url)

The execution of a computation kernel is organized in a hierarchy of threads. In a two level hierarchy, as is typically seen in a GPU accelerator, threads are firstly organized as thread groups which may have shared memories and synchronization mechanism among threads in the group.
Thread groups are further organized into a grid. Threads across local groups cannot communicate directly. A thread grid may be much larger than the actual number of available processing elements on a particular GPU accelerator, so that multiple threads often need to be temporarily multiplexed. This is actually a *desired property* as it allows the program to scale up performance when more number of processing cores is available in future generations of hardware.

A two-level thread hierarchy as is commonly found in modern GPUs is illustrated in Figure 6. The sizes of the grid and the thread groups are determined during the launch of each kernel method. Within each kernel method, this geometric information and the index of the current thread executing a kernel instance are available, so that each kernel can determine its share of data elements to process.

![Figure 6: Parallel methods executed as a grid of thread blocks.](image)

### 3.2 Join Point Model for Accelerator Programming

A Join Point Model (JPM) is central to any AOP framework and it defines three components. 1) What can be the join points? This determines what points in a running
program where aspect code can be inserted. 2) How are join points specified? A pointcut defines a set of join points and a language is required to specify it. 3) How is the advice code to run at join points specified? Our JPM is based on the model of AspectC++ [18], but modified to facilitate accelerator programming.

In this niche AOP system, the JPM is relatively simple. The only join points are function executions. Semantically, we define a function execution join point to encompass all the actions that comprise a function, including argument evaluations, the actions in the function body, and passing the return values back to the call sites.

AspectC++ provides a fairly complex language to specify pointcuts, and these details are available in [18]. Since the only allowed join points in our system are function executions, we only need to use a subset of the existing pointcut specification mechanism in AspectC++. An example of pointcut designation is shown in the following.

```
pointcut pt_inc() = execution(
    "Vect::increase(float)" ||
    "Vect::shift(int)" );
```

This example defines a pointcut named “pt_inc”, which includes the execution of member methods ”increase(float)” and “shift(int)” of the class Vect. More complex pointcut expressions are allowed, and for details, please refer to [18].

Advice code for GPU programming needs to define the following behaviors for a computational kernel: the thread layout of a kernel launch, data transfers between memory spaces, and data mapping. For these very specific needs, we provide two types of advice to describe them: kernel and memory. All other generic advice types in
AspectC++ are not allowed. We also do not allow constructs to modify a program’s static structure, such as members of its classes and the relationship between classes.

In this system, a *kernel* advice for a pointcut specifies that at each execution of a join point in this pointcut, the computation in the function should be offloaded to an accelerator device, and additional code in the advice specifies launch parameters, such as the device to use, thread layout, and shared memory size for thread groups.

A *memory* advice specifies data synchronization information for non-scalar variables referred in a computation kernel. Note that memory advice is declarative and statements in it do not correspond to actual actions.

The following shows definitions for kernel and memory advice, respectively. In the kernel advice, it is specified that the thread layout be a 1-D array of thread block with size 100, and within each thread block, the threads are organized as a 1-D array of size 32.

```
//kernel advice
advice pt_cknls(): kernel() {
    grid.x = 100;
    thread.x = 32;
}
```

The memory advice on the other hand, specifies that memory buffers should be synchronized when this kernel finishes.

```
// memory advice
advice pt_cknls(): memory() {
    sync_mem(ON_EXIT);
}
```
In many cases, advice code of kernel and memory needs to access the context of a joint point that is currently activated, and we provide this information through pointcut parameters and access to data members of the activated class instance.

The following example shows utilization of this context information in a kernel advice to determine the thread layout for a computation kernel. Borrowing the idea of this pointer, that is the pointer to the current object when the join point is activated.

```cpp
advice integrate(): kernel() {
    thread.x = that->m_p;
    thread.y = that->m_q;
    grid.x = that->m_numBodies/that->m_p;
    grid.y = 1;
}
```

Aspects are language constructs that modularize units of crosscutting concerns. An aspect is similar to a class in that it can include the regular members allowed in a class definition, in addition to pointcut and advice definitions. The example in Listing 7 defines an aspect for the Vector class.

In this example, the kernel advice chooses the device to use for this kernel launch, and sets up the dimensions of the 2-level thread hierarchy. The memory advice specifies that memory synchronization is required on both entry and exit of this kernel, and that the data member m_data is changed in the execution of the kernel and the new version is on the GPU side. With this piece of information, proper data transfer requests can be inserted by the aspect weaving compiler.
3.3  Intrinsic Data Types

Memory regions that are accessed in a computation kernel should be treated differently from regular CPU memory as they live in a separate address space and cannot be accessed directly from a CPU side thread. We provide a C++ class template to model this memory system. The type is however understood by our compiler, as an intrinsic type. Its usage will be translated by our compiler.

The following code declares a GPU memory region to be an array of float4.

```
MemBlock<float4, N> m_Pos;
```

It is a class template for 1-D arrays of basic data types, with overloaded subscript operators and automatic memory management. The following shows its usage in a computation kernel.

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
float4 pos = m_Pos[index];
```
In the CPU side code, these types must be accessed through an access method \( T^* \) \( 
\text{MemBock}<T>.\ptr() \). This access method returns the CPU side of the data type and may initialize a data transfer from GPU memory to CPU memory to update the stale data. An example of this usage is shown below.

\[
\text{float}^4* \ h\_ptr = m\_Pos.\ptr();
\]

The on-chip shared memory is also represented as intrinsic data types, and exposed as another C++ class template \textit{SharedMem}. A usage of this shared memory in a kernel function is shown in the following.

\begin{verbatim}
SharedMem<float> cache_Pos;
int l_index = threadIdx.x + blockDim.x*threadIdx.y;
cache_Pos[l_index] = m_Pos[index];
oosp::syncThreadsBlock();
\end{verbatim}

Note that the size of shared memory is controlled by thread launch parameters in the \textit{advice} code to this kernel method.

\textit{SharedMem} is a simpler data type for programmer as it is only usable in a computation kernel. It also has overloaded subscript operators in order to be used as a random access 1-D array. Its purpose is to tag data types so that compiler can map it to on-chip shared memory on a GPU accelerator.

\subsection{Computation Kernels and Execution}

Computation kernels are defined as regular C++ functions with added semantics from aspect code. They can be global or namespace functions or class member methods. They are organized as in standard C++, without requiring being placed in separate compilation
units. Kernel definitions require no language extensions in the component portion of the code. An example of a kernel definition is shown below. It defines the body of the member \texttt{void Vector::increase(float )}.

\begin{verbatim}
void Vector::increase(float val) {
  int index = blockIdx.x * blockDim.x
              + threadIdx.x;
  m_data[index] += val;
}
\end{verbatim}

Notice that in each kernel method, as in CUDA, a programmer has access to the variables \texttt{blockIdx}, \texttt{blockDim}, and \texttt{threadIdx}. They refer to the index of the current thread block, the dimension of the thread block, and the local index in the thread block, respectively.

Calling a kernel starts a hierarchy of threads with each executing an instance of this kernel. The actual layout of this thread hierarchy is specified in its kernel advice.

Kernels can access variables from a variety of places. The scoping rules of the conventional C++ language apply, but the data types and their allowed accesses are restricted. The allowed types and accesses are listed in the following.

- Scalar data types, which can be variables defined in a kernel, global or namespace scope, or class members. Note however that only scalar variables defined within a kernel is writeable, all others being read only.
- Intrinsic data types for global memory of a GPU accelerator. Read and write accesses are allowed.
- Intrinsic data types for shared memory within a thread block. Read and write accesses are allowed.

Programs that do not conform to this rule will trigger compilation errors.

The following is an example aspect specification for the previous computation kernel.
Note that in this example, the shared memory size is determined from the context parameters of the active join point.

### 3.5 Data Transfers

We define data transfer requirements using intrinsic functions in the memory advice code. The specification includes: 1) if synchronization is required, 2) when it is required, and 3) what data should be synchronized.

In the following example, we declare synchronization requirements both on entry and exit of the computational kernel. The memory regions affected is a data member `m_data`.

```java
aspect ACC_Vect {
    pointcut pt_inc() = execution(
        "Vect::increase(float)"
    )
    advice pt_inc: kernel() {
        device = dev;
        grid.x = 100;
        thread.x = 32;
        sharedMemSize = that->m_p * that->m_q
                        * sizeof(float4);
    }
}
```

Memory synchronization requirements are declaratively specified. Programmers do not specify memory transfer operations; instead, they declare if memories need to be synchronized and which memory space has the latest version on a kernel completion.

### 3.6 Compiling Process
The compiler is a source-source translator for C++ programs. The compiler has two high-level tasks: 1) identify all the kernel function execution join points and then weave into the kernel and memory advice code; 2) generate computation kernels for target accelerator APIs and insert in the CPU side program these APIs calls.

As discussed earlier, the JPM in our programming system is simpler than a generic AOP. The only type of join points is function execution and the advice types are restricted to kernel and memory advices. The interaction of advices at a join point is also simple and well-defined. The correctness of the coordination due to aspect code is relatively easy to maintain.

The following discusses the details of the compiling process.

### 3.7 The Compilation Architecture

The compilation architecture is built on the PUMA code analysis and transformation framework [48], which is also used in the AspectC++ aspect weaving compiler.

The flow of this compilation architecture is illustrated in Figure 7. The pointcut expressions in the aspect code are evaluated to identify all join points for each pointcut. For each joint point, the aspect weaver takes in the C++ mainline code and the advice code in the aspect code to generate standard C++ host side programs. The kernel code generator also takes in C++ mainline code and aspect code, but use them to generate GPU kernel source.
The exiting PUMA system is a source-to-source compiling system and it builds its proprietary abstract syntax tree representations of source programs. Due to the added keywords for expressing tailored aspect entities, the parsing module was modified to accept the new aspect language. We then implemented two passes to implement the code weaving functions and kernel code generation functions.

3.8 Code Weaving

The weaving process is relatively simple as it only involves function execution join points. For each identified kernel function, the changes are restricted to its body. The original body is replaced with the following.

- Code to setup kernel launch parameters, including all the actions in the kernel advice code.
- Invocation of the offloaded kernel functions. This includes loading the kernel code to the device, setting up parameters to be passed in, and actually starting the execution of a kernel.
• Code fragments to initiate memory transfers. This code is derived from the memory advice code. These memory transfer code fragments appear both before and after kernel invocations.

The memory management functions for GPU main memory also rely on a runtime library. We built a C++ template library to help with this purpose. This reduces the amount of code to be directly inserted in the weaving process and also eases re-targeting to different accelerator APIs. With this convenience, the weaver is allowed to insert the same code for both the OpenCL and CUDA targets. This library also helps with tracking the update status of the memory buffers for the intrinsic data types to model GPU global memory. These data types require maintenance of memory buffers in both the CPU and GPU address spaces.

The code shown in Listing 8 is an example output of the code weaving process. It is from one of the kernels in the n-body simulation problem that we will discuss later.

```cpp
void BodySystemOOSP::_integrateNbodySystem(
    float deltaTime) {
    //OOSP_START
    int sharedMemSize = m_p * m_q *
        sizeof(float4);
    dim3 threads(m_p, m_q, 1);
    dim3 grid(m_numBodies/m_p, 1, 1);

    BodySystemOOSP__integrateNbodySystem
        <<<grid, threads, sharedMemSize >>>
        (deltaTime, m_oldPos.d_ptr(),
         m_numBodies, m_softeningSquared,
         m_oldVel.d_ptr(), m_damping,
         m_newPos.d_ptr(), m_newVel.d_ptr());

    m_newPos.setDeviceHasNew();
    m_newPos.syncMem();
    m_newVel.setDeviceHasNew();
    m_newVel.syncMem();
    //OOSP_END
}
```

Listing 8: Code weaving for CUDA target. The original function body is replaced with code to set up kernel launch parameters, to call the generated GPU kernel function, and to call memory transfer functions.
3.9 Computation Kernel Generation

A piece of kernel advice for a pointcut identifies all the kernel methods to be transformed. For each kernel method identified, its function body is used to generate the body of a kernel function for a given target, e.g., CUDA. The parameter list of the function is changed to allow passing parameters from host thread to a GPU kernel. The following lists the major points in kernel generation.

- All references to non-local scalar variables are converted to references to parameters to the generated kernel function. Additional parameters to a kernel function are generated to pass in non-local scalars.
- All references to intrinsic data types for GPU global memory is translated to references to array data types. Additional array parameters are added to a kernel function.
- Usage of intrinsic types for on-chip shared memory is transformed into raw arrays. Kernel launch parameters in the host side code decide shared memory sizes.
- Index variables and geometric sizes variables of the thread hierarchy are directly translated to corresponding variables in CUDA or function calls in OpenCL, as both CUDA and OpenCL directly support them.

3.10 Optimization

It is desirable to do some optimization to improve the performance of the final generated code. We discuss as an example the technique to fuse kernels to increase arithmetic intensity. During the compilation process, if we notice consecutive calls to kernel methods that are compatible for kernel fusion, we can apply this technique. Because we work on source code level, we require kernel methods declared as ‘inline’ in C++ to qualify them as fuse targets.
The kernel fusing optimization is very much like the loop fusion [47] technique in conventional compiler literature. In kernel fusion, we move the operations of one kernel to another one, while in loop fusion, from one loop kernel into another one.

There are also other possibilities for optimization. Note however that, we can delegate most of the optimizations to the subsequent compiling process as we are using a source-source translation scheme.

3.11 Tooling Support

It is necessary to build the support for OOSP into an integrated development environment for easier management of the development process.

Each OOSP source code unit can be viewed in three different perspectives: ‘computation’, ‘invocation control’, and ‘memory management’. As shown in Figure 9, each source code editor window has three pages, corresponding to the three perspectives. A programmer can simply choose each individual page to write code for each aspect. With this intuitive environment, the task of writing code for OOSP can be simplified and productivity improved.
3.12 Examples and Evaluations

In this section, we describe two example applications implemented using the OOSP programming system. We provide comparison to CUDA systems in terms of abstract level, structural change to existing program, lines-of-code statistics, and runtime performance.

3.12.1 N-Body Problem

3.12.1.1 Description

An N-body problem numerically simulates the continuous interactions among a system of bodies. The classical example is an astrophysical system where each body is an individual star and they interact through the gravitational force. Other problems in computational science use n-body simulation as well, for instance, turbulent fluid flow simulation and global illumination computation in computer graphics.
A direct and simple method of solving n-body problem is the all-pairs method, which is a brute-force method that evaluates all the pair-wise interactions among N bodies. This method has a computational complexity of $O(N^2)$, so it is not usable in simulation of a systems with a large number of bodies. More advanced methods based on far-field approximation of longer-range forces is often practical, but they still use the all-pairs method in computation of interactions among close-range entities. A survey of the methods developed for n-body problems is available here\(^2\).

Since all-pairs method is also used in advanced far-field methods to compute close-range interactions, improving its performance will also benefit the far-field methods. Additionally, since all-pairs method is compute-intensive ($\frac{\text{computation flops}}{\text{data size}} = \frac{O(N^2)}{O(N)} = O(N)$), it is scalable over the number of bodies or input size, and hence a good candidate for offloading it to accelerators.

In the following, we will first mathematically formulate the all-pairs method and then introduce a serial implementation on traditional CPU platform. We then implement it using our proposed OOSP framework for offloading it onto GPU accelerators. We also compare it with a NVIDIA CUDA version in terms of modularity, impact on exiting CPU code, and adaptability to platform changes. Optimization techniques are also discussed and compared with CUDA version.

### 3.12.1.2 All-Pairs Method for N-Body Simulation

We take the n-body problem in simulating the gravitational force between stars to illustrate the formulations. The location of a body is denoted by vector $\mathbf{x}_i$, and velocity

\(^2\) http://www.amara.com/papers/nbody.html
of that by $\mathbf{v}_i$. The force vector $\mathbf{f}_{ij}$ describes the gravitational force on body $i$ from body $j$. The actual magnitude and direction of this force is formulated in the following equation.

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|}. \tag{1}$$

The total force $\mathbf{F}_i$ on body $i$ is the result of its interaction with the remaining bodies in the system, which is computed by summing them together, as in the following equation.

$$\mathbf{F}_i = \sum_{1 \leq j \leq N}^{j \neq i} \mathbf{f}_{ij} = G m_i \cdot \sum_{1 \leq j \leq N}^{j \neq i} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2}. \tag{2}$$

A softening factor $\epsilon^2 > 0$ is often added to the denominator in the above equation to exclude the situation where 2 bodies approach each other too close, the forces becoming infinite and leading to a collision, which is an undesirable situation for numerical computation. This is also reasonable because a body often represents a galaxy which can pass through each other without actual collision. Adding this factor, the above equation is rewritten as follows:

$$\mathbf{F}_i \approx G m_i \cdot \sum_{1 \leq j \leq N}^{j \neq i} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2 + \epsilon^2} \tag{3}$$

The acceleration of this body can be expressed as follows:

$$\mathbf{a}_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2 + \epsilon^2} \tag{4}$$

This acceleration is used to integrate over time to update the position and velocity of body $j$. 

This example is implemented in three versions: CPU, CUDA and OOSP. The CPU and CUDA versions are in an example project from the CUDA SDK and the OOSP version is developed to evaluate our proposed framework. The class diagram is illustrated in Figure 10.

![Class diagram of N-body simulation](image.png)

Figure 10: Class diagram of N-body simulation. Three versions are provided: CPU, CUDA, and OOSP. They all implement the same interface BodySystem.

### 3.12.1.3 CPU Implementation

The CPU implementation is from the n-body project shipped with CUDA SDK version 2.1. It is briefly discussed here to understand its source code structure and the computation involved. We take it as the model when discussing the structural changes to programs due to moving computation to OOSP.

The CPU version of all-pairs n-body solution implementation is separated into a few C++ functions. Each time step of the integration is invoked by the `update` method, which calls the actual integration method `integrateNBodySystem` for one time step. This method calls function `computeNBodyGravitation` to compute the new force for each body due to attraction to the remaining N-1 bodies. The force of each body is then used to compute its acceleration, and then velocity and the new position after the given time step passed in from `update`. Function `bodyBodyInteraction` computes the interaction given any 2 bodies and
it is used by `compueNBodyGravitation`. The call graph of this computation is shown in Figure 11.

![Call graph of N-body CPU implementation. The C++ class BodySystemCPU implements the CPU version. The code is from NVIDIA CUDA SDK.](image)

The CPU implementation is used as the basis for parallelization in OOSP and CUDA versions of the program. The division of responsibilities among the functions reflects the design choice of a programmer made. `update` is the public interface of the class and all the remaining functions are protected from accessing them directly by class users. Other public methods in the interface are for exchanging data with the visualization component of the program and for setting parameters to control the n-body integration process.

### 3.12.1.4 OOSP Implementation

The OOSP implementation is based on the CPU implementation and offloads computation to a GPU accelerator.

To utilize the GPU global memory, the OOSP implementation utilizes the OOSP class template `oosp::MemBlock<class T>` for to store data that is to be used in the parallel kernel
methods. This type can be accessed from host side, by a few access methods. This is an *intrinsic data type* that is understood by the compiler.

To provide optimized program that fully utilize the accelerator hardware, the OOSP implementation also exploits the fast on-chip memory. Another intrinsic data type `oosp::SharedMem<class T>` is used to model the on-chip shared memory within a thread block.

Another optimization technique involved including loop unrolling and optimized global memory access.

Functionalities are distributed among 4 accelerator methods, which are shown in Figure 12.

![Call graph of the OOSP implementation.](image)

**3.12.2 Evaluations**

In this sub-section, we evaluate the proposed OOSP framework in several aspects by empirical study. We compare the OOSP model implementation with the CPU and CUDA implementations. As the CUDA and CPU programs are from NVIDIA CUDA SDK
serving as examples for GPU developers, we could take them as model programs representing current engineering practice.

We intend to use the following set of metrics to evaluate OOSP programming framework. *Abstraction level:* OOSP raises the level of abstraction and insulates programmers from details of hardware problems. More fundamentally, its support for Object-Oriented programming achieves a higher level of abstraction than the current CUDA, OpenCL and other programming systems. We will compare the structure and line-of-code of the N-Body simulation program with its corresponding GPU/CPU version.

*Expressiveness:* OOSP should not restrict programmers from applying common optimization techniques in accelerator programming. OOSP exposes on-chip shared memory to programmers for programming to exploit data locality in a group of correlated threads, which is supported by low-level systems, such as CUDA and OpenCL.

*Structural changes:* When moving to accelerator programming, it is a desired trait if structural changes to program source code can be minimized and localized. OOSP helps with programmers in this respect by confining accelerator programming related details to aspect code and maintaining compute kernel code with other CPU code that logically belong together.

*Execution Efficiency:* OOSP programs should provide similar execution efficiency when compared to hand-coded code with low-level APIs. Significant performance degradation at runtime will greatly restrict the acceptance of OOSP. We will empirically compare the OOSP implementation with the CUDA implementation in our case studies.

3.12.2.1 Abstraction Level
OOSP has a higher level of abstraction as it directly supports Object-Oriented Programming paradigm. Procedural frameworks, such as CUDA, OpenCL, lie at a lower procedural level. Compared to RapidMind or Intel Ct, we still provide higher level of abstraction as they are actually embedded procedural languages within C++, whereas we support computation kernel directly as C++ member methods. Additionally, the aspect code abstracts and modularizes the accelerator related details into separate compilation units, eliminating the clutter introduced when using the other conventional methods.

The global memory available on GPU accelerators is modeled as an intrinsic C++ data type that is higher level than the raw memory interface available in CUDA and OpenCL. Managed memory allocation/free member functions, typed element type, and other convenient member methods provide a higher level interface. Furthermore, the synchronization of memory between GPU global memory and CPU main memory is done through a set of more advanced declarative directives. It then relies on compiler to automatically generate imperative actions to make actual memory transfers. This is a higher level mechanism than directly coding memory operation by programmers.

OOSP also support the common concept of on-chip fast shared memory through intrinsic data types. This is a C++ type that has convenient members, so it is a higher level structure than the procedural C interface available in CUDA and OpenCL.

As an example, the N-body program we developed can be compiled for both NVIDIA GPU and multicore CPU. The NVIDIA target is through generation of CUDA source code and multicore CPU through the AMD OpenCL implementation.

3.12.2.2 Execution efficiency
To investigate the overhead induced due to using OOSP framework, we compared the performance of our OOSP version with the CUDA version.

**N-Body Simulation**

The result in the previously discussed n-body simulation example is shown in Table 2. From this table we observed roughly 20% degradation of performance compared with the CUDA version.

Table 2: Runtime comparison of n-body problem simulation. GTX 280 NVIDIA GPU card is used, CUDA SDK version 2.1, Linux 2.6, x86_64.

<table>
<thead>
<tr>
<th>Number of Bodies</th>
<th>GFLOPS /s</th>
<th>Interactions /s</th>
<th>Perf Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>30720</td>
<td>436.0</td>
<td>21.8 billion</td>
</tr>
<tr>
<td>OOSP</td>
<td>30720</td>
<td>358.6</td>
<td>17.9 billion</td>
</tr>
</tbody>
</table>

**Options Pricing**

The Black-Scholes model for options pricing [50] provides a partial differential equation for the evolution of an option prices. We use the same computation method used in the example project BlackScholes from CUDA SDK [51]. The original code is a procedural CUDA C program and OOSP version is a C++ program.

The options parameters and pricing data are stored as class members and are of the intrinsic type `MemBlock<float>`. The runtime performance of the OOSP version is compared against the CUDA SDK version and the result is shown in Figure 13.

In this example, the OOSP version achieved 79% to 84% of the performance (giga-options/sec.) of the version in the CUDA SDK.
3.12.2.3 Structural Changes

Structural changes to the original C++ n-body program are investigated from several aspects and they are listed in Table 3.

Table 3: Comparison of structural changes of using OOSP programming framework for n-body problem

<table>
<thead>
<tr>
<th>Changes to External Interface</th>
<th>Number of C++ Classes</th>
<th>Class Members</th>
<th>Non-Member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU n.a.</td>
<td>1</td>
<td>7 data members</td>
<td>4 non-public methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 computation function</td>
<td></td>
</tr>
<tr>
<td>CUDA no</td>
<td>1</td>
<td>10 data members</td>
<td>2 non-public methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 CPU side functions</td>
<td></td>
</tr>
<tr>
<td>OOSP no</td>
<td>1 + aspect</td>
<td>12 data members</td>
<td>6 non-public methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 aspect advice functions</td>
<td></td>
</tr>
</tbody>
</table>

The OOSP program improves on the CUDA version in that it 1) reduces number of non-member functions from 14 (10CPU + 4 GPU) to regular C++ member functions and 2) additional aspect advice functions, which is enclosed in an AspectC++ aspect. The non-member functions in CUDA are GPU kernels and wrappers to them.
The OOSP version keeps all the computation functions as class members and adds 2 aspect advice functions to manage memory transfers and thread grid layout. The mainline code is very similar to the original C++ program and contains only computation code. With OOSP, additional concerns are in additional source file, resulting in a clean program structure.

We also show the Line-Of-Code (LOC) metric in Table 4.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>CUDA</th>
<th>OOSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>.h</td>
<td>40</td>
<td>55</td>
<td>190</td>
</tr>
<tr>
<td>.cpp</td>
<td>200</td>
<td>190</td>
<td>180</td>
</tr>
<tr>
<td>.cu</td>
<td>N/A</td>
<td>Kernel: 170</td>
<td>Wrapper: 100</td>
</tr>
<tr>
<td>aspect</td>
<td>N/A</td>
<td>N/A</td>
<td>40</td>
</tr>
<tr>
<td>Total</td>
<td>~240</td>
<td>~510</td>
<td>~410</td>
</tr>
</tbody>
</table>

Clearly, OOSP has a smaller number of codes compared to CUDA version. More importantly, however, the program structure is greatly improved. The final OOSP program is very similar to the original CPU version, and the clutter due to offloading computation to GPU device is removed from the mainline code, yielding a much cleaner structure of program.
Chapter 4
A Performance Model for Heterogeneous Co-Processing

4.1 Introduction

In adopting heterogeneous systems to solve various computational problems, we have different choices to build a hardware system for a particular application. For example, in building a system that exploits CPU + GPU co-processing, we need to decide CPU models, memory modules, the interconnect technology between the CPU and GPU subsystems, and the model/number of GPU device. For efficiency reasons, we want to build systems that are balanced and yield reasonable, expected performance, within a budget. The minimum question we need to answer is: does converting to co-processing yield enough benefits for that computational problem?

The other equally important concern is from the standpoint software programmers. We are provided with a heterogeneous system and are to design and implement an application to solve a computational problem; it will be very helpful if there are some principles/guidelines that help us in making conscious choices in various stages of software development. For example, in early design stage, we need to choose/design algorithms that fit with the target platform, because one particular piece of hardware may give us very different tradeoffs comparing to another one. In programming stage, we need guidelines and knowledge in choosing different programming constructs, library calls, instructions, and etc, to efficiently implement an algorithm. In the later debugging and optimization stages, we will need to be able to consciously identify the bottlenecks,
to understand the asymptotic performance limits, so as to make judicious decisions to improve performance.

In fact, being able to analyze, predict or bound the performance for a particular system is invaluable to application programmers, system builders, compiler writers and hardware architects; this help them to identify problems, to improve their respective designs and then to meet target efficiency and performance or concerned system parameters.

In this chapter, we propose a performance model that is usable in the context of heterogeneous processing environment for the above-mentioned audience. We, however, do not strive to provide a model that precisely predicts performance—it’s hard if not impossible; instead, we are dedicated to providing a model that is easy to understand, communicate, and apply, hence providing useful and insightful information in helping with achieving performance, in the aspects of system building, algorithm design, programming and debugging activities. The model is imperfect, imprecise and subject to adjustments and refinements under different contexts, but it is widely applicable. We achieve this by identifying information from the bulk properties of the hardware platform, such as system bus bandwidth, sustained memory bandwidth, and peak instruction execution rate, and these properties of applications, such as the ratio of arithmetic operations to memory access operations of software programs. We then correlate these pieces of information together to disclose insightful understandings about the whole combination of hardware components and the applications running on them.

In the following, we first apply the well-known Amdahl’s Law [37] to analyze the usefulness and effectiveness of choosing heterogeneous processing system, showing the
bounds of efficiency benefits, given constraints in power consumption or chip area. The performance of the massively parallel co-processing devices (GPUs) are modeled using the Bulk Synchronous Parallel [38] model; we study the mapping of this model to GPU processing and then derive the model parameters for the existing processing model in the current generation of GPU co-processing devices. Relating a machine’s model parameters with a particular program, we are then able to bound / predict the performance of that program executing on that machine. We can also compare the observed values with those obtained in the model to help with improving program performance.

To begin with, we first briefly review the models of parallel programming and processing and the importance of a model that plays the central role of a conceptual framework and reference.

4.2 Models of Parallel Programming and Processing

A programming model is an abstraction used by programmers to reason about the execution of programs. Due to the high level abstraction and for the purpose of portability and simplicity, a programming model cannot expose all the details of the underlying implementation hardware. However, a well-designed programming model should allow manipulation of the most important aspects in the hardware processing models to achieve efficiency.

A programmer can change the expression of a program relative to the high-level programming model and the compiler should be able to understand this information so as to generate machine code to achieve the desired performance improvement. In this process, it is essential to define a clear performance model so that programmers can
easily evaluate the performance impact of various program structures, in order to make adjustment for optimization.

The evaluation and success of a model for parallel computation should present properties that are useful in various stages of the software development. Firstly, it should be able help with the design and analysis of algorithms. It should provide a framework to dictate the structure of the program. For instance, a message passing model of computation literally structures a program into processors that receive various types of messages and sending corresponding result messages. The basic programming task is to correctly pair send and receive operations, which is shown in the left half of Figure 14. On the other hand, a shared-memory parallel programming model provides a uniform address space where various tasks can write to and read from it. The parallel threads are designed to reading data from assigned memory regions and writing back to particular memory regions, which shown in right half of Figure 14. The import consideration is to maintain the constraints of these writing and reading accesses to shared memories, so that race conditions are avoided while achieving best efficiency.

Figure 14 Parallel Programming models. Message passing model is shown on the left and shared memory on the right.
Additionally, a useful model should also be able analytically predicate the performance of a program, if certain parameters of the machine and the required characteristics of that program are provided. For instance, in the familiar sequential model using imperative paradigm, if all other conditions remain unchanged, a CPU-bound program finishes in proportional to the clock frequency of the processor it is scheduled to run on, as shown in the following equation.

\[ T_{\text{execution}} = \frac{N \cdot \text{CPI}}{R_{\text{clock}}} \]  

(1)

where \( N \) is the number of instructions in a program,

\( \text{CPI} \) is \textit{clock per instruction},

\( R_{\text{clock}} \) is the clock rate on that machine.

An example plot of this execution time model is also shown in Figure 15.

\[ \text{Execution time vs. CPU clock frequency for a CPU-bound program} \]

This is a simple relationship and it works very well for the past decades for predicating the performance of sequential programs. This model ignores the memory hierarchy and
the working set size of a given program, but it is useful in practice since it yields useful insights for the execution time of a given program on a particular machine. In the multicore and heterogeneous computation era, we would like to have a model that is similar in complexity, and also as useful and effective in predicting and estimating performance.

4.3 Advantage of CPU + GPU Co-Processing

Amdahl’s Law [37] is useful in finding the maximum overall improvement in a system when only a part of the system can be improved. In application to parallel processing, it can be used to predict the theoretical upper limit of achievable speedup when parallel processing is used. Amdahl’s Law has a simple formulation and is expressed in equation (2), where $f$ is the fraction of the computation that can be run in parallel, and $N$ is the number of processors available for parallel execution. Applying this law, the upper bound in speedup improvement for converting a computation with $f$ fraction of parallel computation is $1/(1 - f)$. For example, if 50% of a given computation can be executed in parallel, no matter how many processing elements we have, the maximum speedup achievable is 2.

\[
\text{Amdahl’s Law:} \quad S = \frac{1}{1 - f + \frac{L}{N}} \quad (2)
\]

In this subsection, we show the application of this simple yet powerful law in arguing for heterogeneous co-processing using CPU and GPU technologies.

In microprocessor design, the die area is a limiting factor for a given generation of technology because when the required die area becomes large, the yield (due to defects,
only a fraction of the chips meets the production standard) of a given semiconductor manufacturing technology will decrease. The die area is hence chosen to meet a particular yield rate to maximize profits. The size of this die area, in together with the area of each transistor, determines the number of transistors can be put into a single chip. Given a limited size of die area and amount of transistor counts, it hence makes sense to choose a design that maximizes the overall processing efficiency if processing cores with different performance characteristics can be built. For example, GPU and CPU take different design approaches because they have different goals in performance optimization [40]. CPU design aims to optimize the sequential thread latency by putting together a large amount of transistors into complex and clever control logic. On the other hand, GPU design chooses to optimize the aggregate throughput efficiently execute massively parallel tasks.

The number of transistors in a given processor directly determines the power it needs to draw from the electricity source. In a given system design, the maximum power is specified and hence can be used to determine the group of components that can be put together. We can use the Amdahl’s law to show the advantage of GPU+CPU co-processing in terms of power efficiency. Given a maximum power package, and the power requirement for each core, we can predict the maximum speedup achievable.

Firstly, we keep the power package constant, and then build as many CPU and GPU cores as possible under this constraint. We are then able to calculate the achievable speedups by applying the Amdahl’s Law. For instance, taking the typical numbers from current generation of technology, we can plot a surface of this speedup in Figure 16.
In computing this figure, we assume the maximum power in a system is 160W, each CPU core consumes 40W, each GPU core consumes 1W, and a CPU core is 5 times faster than a GPU core. These numbers are realistic values from the current (2009) generation of semiconductor technology.

Table 5 below shows a few typical points taken from the graph. We compare two programs that have different portions of parallel work. In the first program, we assume a 20% parallel work, which is a mostly sequential program. The second one is a very parallel-intensive work, with a parallel fraction of 99.5%. The result listed in the table clearly shows the advantage of GPU+CPU co-processing: it is the fastest in both tasks, both for a mostly sequential one and a mostly parallel one.

Figure 16: Heterogeneous CPU+GPU co-processing speedup achievable given a maximum power package.

Notice that the high percentage (99.5% used in Table 2) of parallel work is realistic to many important and real-life loads appeared in engineering and scientific computing. For
example, in the benchmark applications (the Seven Dwarfs) presented in [42], there are plenty of parallelisms to map them onto massively parallel hardware. In fact, parallelism is a norm but an exception.

Another interesting result in this simple analysis is that it is not a good idea to put more than 1 CPU core for performance reason: when there is parallel work, it is always more efficient to execute it on throughput oriented GPU co-processor. Of course, in practical environments, there are other factors to consider, but we illustrate here that the model of GPU+CPU co-processing is generally applicable to various tasks and is also the most efficient choice.

Table 5: CPU + GPU co-processing advantage, predicted using Amdahl’s Law. In this table, we assume a fixed power package of 160W is stipulated and a program on a CPU core runs 5x faster than the same program on a CPU core. The numbers of GPU and CPU are calculated by assuming we build as many as possible cores under the power constraint.

<table>
<thead>
<tr>
<th>% of Parallel Work</th>
<th># of CPU</th>
<th># of GPU</th>
<th>Processing Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1.0</td>
</tr>
<tr>
<td>20.0</td>
<td>1</td>
<td>120</td>
<td>80.80</td>
<td>1.24</td>
</tr>
<tr>
<td>20.0</td>
<td>4</td>
<td>0</td>
<td>85.00</td>
<td>1.18</td>
</tr>
<tr>
<td>99.5</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>99.5</td>
<td>1</td>
<td>120</td>
<td>22.32</td>
<td>4.48</td>
</tr>
<tr>
<td>99.5</td>
<td>4</td>
<td>0</td>
<td>25.37</td>
<td>3.94</td>
</tr>
<tr>
<td>99.5</td>
<td>0</td>
<td>160</td>
<td>5.60</td>
<td>17.82</td>
</tr>
<tr>
<td>99.5</td>
<td>1</td>
<td>120</td>
<td>4.48</td>
<td>22.32</td>
</tr>
</tbody>
</table>

We can use the Amdahl’s law to find the bounds of a particular computation mixed with a parallel and a sequential portion. This limit can guide us in answering the yes-or-no
question: if the upper limit $1/(1 - f)$ is within the design goal, we can proceed to converting the parallel portion of the program to run on parallel hardware; otherwise, we should either abandon the effortless attempt, or identify potential parallelism that has not been discovered and then start with this new piece of information a new iteration of application of this bound analysis.

### 4.4 Model of CPU + GPU Co-Processing

It is useful and necessary to establish the model of a heterogeneous processing system built from CPU and GPU devices. Fortunately this is straightforward and familiar picture of computation as the number of components is limited and interactions are well understood.

At a higher level, when considered in the context of a system composed of CPU and GPU processing, an application that utilizes GPU co-processing is a composition of activities that are listed in the following.

1) Computation on CPU for sequential processing
2) Memory transfers between GPU and CPU DRAM’s
3) Computation on GPU coprocessing device for parallel processing

Each invocation of a GPU device function or memory transfer also implies a point of synchronization. In simple systems, at any given time, only one of the three computation can happen; more complex systems allows asynchronous execution of these three stages and hence yield better utilization of the system hardware. This flow of program control and hardware activity is schematically shown in Figure 17.
The runtime of a program of this style is composed of 3 parts.

- Time spent in executing the sequential portion of the code on a CPU, denoted by $T_{cpu}$. This is the sum of all the sequential executions that are interleaved by GPU executions and memory copies.
- Time spent in executing the parallel portion of the code on the GPU co-processor, denoted by $T_{gpu}$. This is the sum of all the parallel executions on the GPU co-processor.
- Time spent in copying data between CPU and GPU DRAM’s, denoted by $T_{mem}$. This is also the aggregated sum of all memory transfers. It is possible that a CPU and a GPU are put into the same system socket and hence are able to have shared access to a single memory region, but we here consider the more general case of a separate system interconnect.

The total runtime is simply the sum of these 3 terms.
The CPU time is a familiar term to programmers and can be estimated using all the conventional techniques. An individual memory transfer time can be divided into the constant startup time, and the time that is proportional of the actual size of memory transfers.

\[ T = T_{cpu} + T_{gpu} + T_{mem} \]  
\[ T_{cpu} = \sum_{all \; sequential \; cpu} t_i \]  
\[ T_{gpu} = \sum_{all \; parallel \; gpu} t_i \]

The memory transfer bandwidth can be easily estimated by looking at the specification of the system connect, or empirically measurement. Sizes of memory transfers can be obtained inspecting program code or from algorithm design. Thus, the aggregate time spent in memory transfers is easily calculated.

\[ t_{mem} = t_{startup} + t_{transfer} \]
\[ t_{transfer} = \frac{\text{Size}}{\text{Bandwidth}} \]
\[ T_{mem} = \sum_{all \; mem \; trans} \left( t_{startup} + t_i \right) = n \times t_{startup} + \frac{1}{\text{Bandwidth}} \sum_{all \; mem \; trans} \text{Size}_i \]  

The parallel execution time is not easily derived; we need to look at more details of the GPU architecture and understand its model of execution. In the next subsections, we will dive into the details of GPU computation in order to further understand the performance characteristics this relatively new hardware architecture. Fortunately, we do not need to
invent new tools to understand its behavior, we are lucky enough to adopt and adapt existing result in parallel architecture to achieve our goals.

### 4.5 Computation Model within a GPU Co-Processor

GPU as a co-processing device to a conventional CPU introduce factors that are not seen in a homogeneous multicore system. Firstly, GPU itself is a hierarchically organized set of computation elements. In a model GPU package, for instance, NVIDIA Tesla series, it consists of an array of multithreaded processors organized in a communication network, which can be abstracted in Figure 18.

![Figure 18: An architectural model of GPU. It consists of an array of processing cores arranged in a 2-level hierarchy. Cores within the same processor have a small but very fast shared memory region for local cooperation; they also have synchronization mechanisms for threads within the same processor. The on-chip cache is relatively small and usually not suitable for long-term reuse.](image)

The first tool we can use is the Parallel Random Access Machine (PRAM) [43] model. It is a simple extension to the Random Access Machine that we are familiar with in the sequential world. We are going to use it to model the computation in each thread block.

We can also apply the Brent’s Theorem [44] to deal with the situation where the number of physical threads is less than the number of threads in a thread block.
In the GPU computation model, thread blocks are further organized into a grid structure and this grid of thread blocks fully describes the layout of a parallel computation kernel, which is the unit of computation when seen from the CPU control thread. At this level, memory accesses to the off-chip GPU main memory play a pivotal role in determining the performance of a particular computation kernel. Recognizing, describing and then accounting for this component in an easy to manipulate way is crucial to the success of a performance model. In this aspect, we found the Bulk Synchronous Parallel (BSP) [38], [39] model meets our requirements and provides a holistic method to describe these important parameters and can provide performance estimation based on program texts and a few machine parameters.

In the following, we will detail the models for these two levels, and will also use examples to illustrate their usefulness.

4.5.1 Computation Model within a Thread Block
As shown in Figure 18 and discussed in earlier chapters, the current GPU architecture is a 2-level hierarchy. Concurrently executing threads are organized as a grid of thread blocks. Each thread block has a block of shared memory that is accessible to all threads in that thread block but not by threads in a different processor. Additionally, all threads have access to a global memory through the memory interface. Each thread has uniform access time to memory cells in both the fast shared memory and slower global memory. These features prompt using the framework of Parallel Random Access Machine (PRAM) [43] model to study the performance behavior within a thread block. The impact of limited number of processing cores within a processor can be resolved by applying the Brent’s Theorem [44]
A simple example to illustrate the computation at this level can be given by using the parallel element-by-element addition. This computation kernel is shown in Figure 19. If executed on a theoretical PRAM machine, this computation kernel has a step complexity of $O(1)$. In a real system with limited number of processing cores, we can apply Brent’s Theorem to get the actual step complexity.

Brent’s Theorem states that If an algorithm involving a total of $N$ operations can be performed in time $T$ on a PRAM with sufficiently many processors, then it can be performed in time $T + (N - T)/P$ on a PRAM with $P$ processors.

In a concrete example, suppose this kernel is executed in with a configuration of $num\_threads$ of threads in a thread block and the number of processing cores on a given processing is $P$, we can compute the actual execution time of this kernel on a GPU that has $P$ number of processing cores in each of its multiprocessor.

$$T = 1, \quad P = num\ of\ procesing\ cores, \quad N = num\ of\ threads\ to\ run, \quad N > P$$

$$real\_parallel\_time = T + \frac{N - T}{P} = 1 + \frac{N - 1}{P} \approx \frac{N}{P} \quad (9)$$

This is an easy to remember and use formula; it captures the nature of this execution model when the number of processing cores is less than ideal: the parallel operations are executed in serial.
We have shown in the above that PRAM model can be used to estimate the execution time of a thread block. At a higher level of the GPU computation, i.e., a grid of thread blocks that is started by an invocation of a GPU computation kernel, a more suitable model of computation, however, is needed.

### 4.5.2 Computation Model of a Grid of Thread Blocks

In current generations of processor architecture, the latency of a main memory access instruction is about 2 orders that of a typical ALU / FPU instruction [45], and thus analysis of program performance must not only counting the number of floating / integer operations, but also the number of memory access operations to main memory. It is more so in current day GPU co-processing model of computing because the cache is so small compared to the large number of data accessed in a typical computation [40]. GPU computation is targeting at throughput computation and the number of data read from/written to the main memory is inherently huge. Even though the GPU architecture is moving toward adding more cache modules to its hardware design, the rational is not to hide latency by using cache but rather to optimize memory access patterns in order to better utilize memory bandwidth [40]. These performance characteristics hence necessitate a performance model that better deals with this situation.

```c
__global__ void parallel_add (double *g_a, double *g_b, double *g_c, unsigned int n)
{
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        g_c [i] = g_a[i] + g_b[i];
}
```

Figure 19 A simple parallel addition kernel. The step complexity of this piece of code is $O(1)$ in the PRAM model. With limited number of processing cores, the step complexity can be computed using Brent’s Theorem.
The desired model needs to enable easier capturing of the main aspects of GPU throughput computation. As discussed in earlier chapters, the first hallmark of this style of computation is organization of a program into computation kernels. An algorithm at a higher level can be expressed as a sequence of computation kernels executed on GPU. In current software engineering practice, this ‘kernel’ structure serves as a unit of compilation unit, and interestingly it also constitutes the framework to analyze and design parallel algorithms.

Secondly, as the on-chip cache in GPU computing is not intended for long term reuse, accounting for and then reducing main memory traffic become extremely important in characterization and optimization of the performance of GPU algorithms. In the other words, we need explicit expression of memory access in the performance model.

In searching for this ‘silver bullet’ for GPU throughput computation, we found that the earlier work in studying Bulk Synchronization Parallelism (BSP) [37] very helpful and nearly meet all the requirements we ask for.

BSP computation is composed of a sequence of super steps. Each super step has three stages: local computation, global communication, and barrier synchronization, as illustrated in Figure 20 [39]. In the local computation stage, each process proceeds individually without conducting any communication with other processes. In the stage of global communication, each process sends messages to other processes to share data. The last stage is a barrier synchronization to guarantee all the processors finish before exiting this super step.
The model of computation on a GPU can be described in this BSP framework. First of all, an instance of GPU parallel kernel execution is a super-step in BSP. In executing a GPU parallel kernel, it involves loading data from off-chip main memory to processing cores and each GPU core runs separately until finish. After all the threads on the GPU exit, the GPU kernel finishes its execution, this is a global barrier. Because each thread can read / write to any memory cells in the main memory, this provides global communication mechanism among the massive number of threads in a GPU kernel.

In the classical model of BSP, the vertical dimension in Figure 20 represents the three phases: local computation, global communication, barrier synchronization; the horizontal structure abstracts concurrency: each vertical `bar’ represents one of the threads in the concurrent local computation. In applying BSP to GPU computation, we, however, find it more suitable to let each vertical `bar’ represent one thread block, instead of one thread in a thread block. We argue for this mapping because the following.
• The threads within a thread block always start and finish together, and on the same processor. As long as a thread block is scheduled to run on a processor, they are there until all the threads finish. This is regulated in the current programming model and its mapping to physical GPU processors.

• Threads within the same thread block can interact and have access shared memory, but they do not communicate with threads not in the same thread block. This model is more suitably described by the PRAM model, as we have discussed in the previous sub-section, than uncomfortable fitting it to the BSP model.

Consequently, looking from outside of a thread block, the computation can be agnostically treated as a unit of ‘sequential’ process in the BSP model, even though there is plenty of concurrency within it. If more knowledge of the execution within a thread block is desirable, we can `open it up’ and apply the familiar PRAM model. This not only simplifies the discussion but in fact better grasp the crux of the GPU computation model: the GPU threading model is not flat, but a 2-layer hierarchy. This framing of GPU computation into the BSP model of computation is shown in Table 6.

Table 6: GPU Co-Processing Considered in the BSP Model.

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BSP</strong></td>
<td>Local computation</td>
<td>Global communication</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Local computation in each thread block, PRAM model. Thread blocks are parallel.</td>
<td>Access to off-chip globally shared DRAM is global communication.</td>
</tr>
</tbody>
</table>
We can further illustrate this model by analysis of a concrete example in matrix multiplication using blocking. We divide the matrix into small tiles to improve data use and this algorithm can be neatly presented in the BSP framework for GPU we have discussed, shown in Table 7. We take the implementation from the CUDA SDK\(^3\) and list the computational kernel in the following, Figure 21. Each thread block of this kernel computes the product of a sub-matrix of size BxB. Given a matrix A of size m x p, B of p x n, product C = A*B is of size m x n. Each thread block loads two BxB sub-matrix from A and B, respectively, multiply them, and then accumulate the result to get a sub-matrix of BxB of matrix C.

Table 7: Blocked Matrix Multiplication in the BSP Model.

<table>
<thead>
<tr>
<th></th>
<th>Local computation</th>
<th>Global communication</th>
<th>Barrier Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>Compute the sub-matrix BxB of the product matrix C. There are (\frac{m}{B}, \frac{n}{B}) thread blocks organized in a 2-D grid of size (\frac{m}{B}, \frac{n}{B}).</td>
<td>Each thread block reads data from global memory for matrices A and B, and write result for matrix C. This access to global memory could be considered a complete exchange pattern in BSP model.</td>
<td>Implicit after the matrix multiplication kernel finishes.</td>
</tr>
</tbody>
</table>

Looking at this matrix multiplication example, however, careful readers may have noticed that this is not completely conformant the BSP model in that the communications

\(^3\) http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html
(DRAM memory access in this MM kernel) do not take place in the last phase. In fact, there are several phases of communication: one phase of memory read for each iteration of the loop and a final DRAM write access to store the result sub-matrix.

Another difference is that the execution of the computation instructions may overlaps with the memory access communications. This is especially true if considering that there are multiple thread blocks resident in the same processor and the computation instructions are only executed when the data from DRAM is ready, stalled thread blocks waiting for data (communication time) overlap with computation instructions for which data is satisfied.

```c
// Example code from CUDA SDK
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    // Multiply the two matrices together;
    // each thread computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Figure 21: Matrix Multiplication using blocking, compute C = A * B. Each thread block compute a sub-matrix of size BxB of matrix C. This illustrates interactions (data sharing and thread synchronization) among threads within the same thread block. From outside, however, a thread block can be opaquely viewed as a sequential process, abstracted as the vertical `bar' in the BSP model.
But fortunately, these are not fatal. The first item on multiple phases is harmless because aggregating the multiple memory accessed and only accounting for them in one place does not change the sizes of the communication. This is not only simplifies analysis by focusing on the ‘bulk’ properties of a program, but also is supported by the actual execution model in a GPU device. The writing to global DRAM is not guaranteed visible to other threads until the next device kernel is launched. This coincides with the BSP super-step model and exhibits the reasonable engineering choices in practical design.

The overlapping of communication with computation can also be remedied by taking the maximum of the computation and communication parts, instead of summing them up. This is actually a common issue in applying BSP to other architecture [39].

4.6 Performance Prediction for CPU + GPU Co-Processing

In the previous sections, we put the execution of programs on a GPU into the BSP model, treating the execution of a thread block as the individual sequential process in that model, accesses to the global DRAM as the communication phase, and the implicit synchronization at kernel finish time the global barrier. Adding to this the time spent in transferring data between CPU DRAM and GPU DRAM, and the time spent in the sequential CPU thread, we can easily get the total runtime of a program running on this hybrid platform.

In this section, we further develop this proposal by introducing two parameters that describe the ‘bulk’ properties of the machine (GPU device) and the application. Matching these two parameters, we are able to quickly identify the bounds of performance and bottlenecks of a given application.
The first parameter is **Machine Characteristic Ratio (MCR)**, which is the ratio of peak floating point operation execution rate to the peak memory bandwidth from the processors to the DRAM (equation 9). We here explicitly consider the off-chip traffic because this is the critical quantity that introduces stalls for memory read/write instructions. This is a fixed number and do not change with the applications that are running on that hardware.

\[
mcr = \frac{\text{peak FLOPs (Flop/sec)}}{\text{peak Bandwidth (Bytes/sec)}}
\]  

The second parameter is **Application Characteristic Ratio (ACR)**, which is the ratio, given a particular program, of the number of floating point operations to off-chip memory access (equation 10).

\[
acr = \frac{\# \text{ of Flops / thread block}}{\# \text{ of DRAM accesses / thread block}}
\]  

Using the first number, we can plot a curve about the peak performance of applications running under this crucial constrain. This is derived from execution model of a GPU, which neatly fits into the 3-phase BSP model of computation: local computation, access latency to DRAM, and barrier synchronization. For non-trivial computation, the first 2 phases dominate, and because the concurrent nature of a memory access units and arithmetic units, the execution time is can be calculated using the following formula.

\[
T = \max(\text{local computation}, \text{DRAM access}) + \text{barrier cost}
\]  

We can plot a figure to illustrate this characteristic (Equation 11) of the GPU execution model. In the [20], the X-axis represents the ratio of arithmetic operations over off-chip memory operations (DRAM access), the Y-axis on the left represents achievable FLOPS
(red curve), and the Y-axis on the right achievable bandwidth (blue curve). On all parts of the curve, except for the very exceptional point, the machine is running at either peak FLOPS or peak Bandwidth mode, or imbalanced in terms of the memory unit and arithmetic units. The exceptional point is the point of Machine Characteristic Ratio.

![Graph of Application Characteristic Ratio vs. Machine Characteristic Ratio: Nvidia Fermi](image)

**Figure 22:** Performance of Nvidia Fermi, modeled using the machine characteristic ratio and application characteristic ratio. When $acr < mcr$, off-chip bandwidth is the dominating factor and afterwards, flops is the bottleneck.

This curve is simple but provides very useful insight into the execution of a GPU co-processing device. Given its machine parameters (MCR) and application characteristics (ACR), this method quickly identifies reachable peak performance, the bottleneck to performance, and ways to improve on the current status. If an ACR lies to left of the MCR in the graph, it is bandwidth-dominate program, and the optimization goal is to achieve peak bandwidth; if to right of the MCR, the goal is to achieve peak FLOPS. These numbers are ‘bulk’ properties of machines applications, requiring no detailed knowledge of machine architecture; hence it can be useful across generations of hardware evolution and a broad spectrum of applications. We list this rate for two generations of Nvidia GPUs in Table 8.
MCR and ACR are two ends that we need to keep in mind in investigating the performance of application running on a particular piece of hardware. They determine how well the machine is suited to run that application or vice versa, how well the program has been designed or optimized for a particular architecture.

Table 8: Machine Characteristic Ratios for two generations of Nvidia GPU’s. Notice that even though these two generations of GPU’s have very similar MCR (27.8) for single precision floating point computation, they have drastically different ratio for double precision, due to improvement in Fermi architecture. Additionally, even they offer similar peak FLOP and MCR for single precision computation, improvements (for instance, on-chip memory management) in Fermi may make the application runs closer to peak than in GTX 280.

<table>
<thead>
<tr>
<th></th>
<th>Fermi</th>
<th>GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Processors</td>
<td>32*14 = 448 CUDA cores</td>
<td>8*30 = 240 CUDA cores</td>
</tr>
<tr>
<td>Peak off-chip Memory Bandwidth</td>
<td>148 GB/sec</td>
<td>141.7 GB/sec</td>
</tr>
<tr>
<td>Peak GFlops/sec (single precision)</td>
<td>1030 TFlops</td>
<td>933 GFlops</td>
</tr>
<tr>
<td>Peak GFlops/sec (double precision)</td>
<td>515 GFlops</td>
<td>78 GFlops</td>
</tr>
<tr>
<td>Flops / Bandwidth (single precision) [Oper./Byte]</td>
<td>1030 / (148/sizeof(float)) = 27.8</td>
<td>933 / (141.7/sizeof(float)) = 26.3</td>
</tr>
<tr>
<td>Flops / Bandwidth (double precision) [Oper./Byte]</td>
<td>515 GFlops / (148/sizeof(double)) = 27.8</td>
<td>78 / (141.7/sizeof(double)) = 4.4</td>
</tr>
<tr>
<td>DRAM Type</td>
<td>GDDR5</td>
<td>GDDR3</td>
</tr>
</tbody>
</table>

They are useful in the following several aspects.

- Performance bounding. Identification of MCR of a given machine can be done by looking up the speciation table from the manufacture or by running benchmarks. ACR can be estimated by looking at the source code of the computational kernel, or by theoretical analysis in the early algorithm and program design stages. These
two numbers immediately tell the performance bounds of an application and yield manageable expectation in software engineering practice.

- **Performance debugging and optimization.** Peak performance is not easily achievable and the first cut of a program usually runs below peak performance. This model is helpful in identifying directions and aspects for possible improvements. For instance, if we can reduce the DRAM traffic from a thread block by increasing the inter-thread block data use, i.e. using the fast on-chip shared memory, we can move up the ACR and improve its achieved FLOP/sec, even under the constant peak bandwidth. Actually, most of the optimization techniques can be organized into this framework.

- **Hardware Design Optimization.** This model is also useful for hardware/architecture designers as it provide a quick guidance toward providing more efficient hardware design. For instance, in collecting the typical ACR’s for the target market, the manufactures could adjust their designs to implement machines with MCR’s that match their applications. On the other hand, designers can also facilitate application designers so that they can easily improve their ACR’s. One example could be the introducing larger on-chip shared memory or providing automatic management by adding a new level of cache memory (Nvidia Fermi offers this feature, an improvement from previous generation of GPU design).

### 4.6.1 Example Application: Dense Matrix Multiplication
We use the proposed model to analyze the dense matrix multiplication program in the following. Assuming computing $C = A \times B$, where $A$ is of size $m \times p$, $B$ of size $p \times n$, the production matrix $C$ is then of $m \times n$.

In the simple case where there is no reuse, computing the matrix requires:

**DRAM Access:** $M_{\text{no \ reuse}} = m \cdot n \cdot (2p + 1) \approx 2mnp$, because we need to read one row from $A$ and one column from $B$, and writing the result to $C$ for each element of $C$.

**Computation:** $m \cdot n \cdot 2p = 2mnp$ (Flops), because we need to do $p$ multiplication, $p$ addition for each of the elements of $C$.

If we assume $m = n = p$, and single precision (4 bytes) computation, we get the AMR for MM: 0.25. This is a very bad number and far below the peak FLOPs available in a GPU. Actually, if theoretical peak best ACR can be obtained by assuming complete data use. In this case, the DRAM access is $(m^2p + p^2n + m^2n)$, and the ACR is $O(n)$ for $m=O(n)$, and $p=O(n)$, which means it can go as high as needed if we have matrixes that is large enough.

**Arithmetic Intensity:**

$$r_{\text{no \ reuse}} = \frac{A}{M_{\text{no \ reuse}}} = \frac{2mp}{2mnp \cdot \text{sizeof (float)}} = 0.25 \left(\frac{\text{flop}}{\text{byte}}\right)$$

However, on-chip memory is limited and complete data re-use is not achievable for non-trivial matrices. We hence need to dividing the matrices into tiles to improve data reuse and hence a higher ACR. Here we take the blocked MM code from CUDA SDK and analyze it.
Blocked Matrix Multiplication with BLOCK size $= B \times B$

**DRAM Access:**

$$M_{\text{reuse}} = \frac{n}{B} \cdot \frac{m}{B} \cdot \left(\frac{p}{B} \cdot 2 + 1\right) \cdot B^2 = \frac{m \cdot n \cdot (2 \cdot \frac{p}{B} + 1)}{B} \approx \frac{2mnp}{B},$$

since we divide A and B into small block of size BxB and data is re-used in computing the $B_A \times B_B$, where $B_A$ is from A and $B_B$ from B. So, with reuse, 1/B of off-chip traffic is required.

**Computation:**

$$\approx \frac{n}{B} \cdot \frac{m}{B} \cdot \frac{p}{B} \cdot (2B^3 + B^2) \approx 2mnp(Flops),$$

since computation for each block is $2B^3$ for computing matrix multiplication for each block and $B^2$ for accumulating the result for each block in the production matrix C.

**Arithmetic Intensity:**

$$r_{\text{reuse}} = \frac{A}{M_{\text{ reuse}}} = \frac{2mnp}{B \cdot \text{sizeof(float)}} = 0.25 \cdot B \left(\frac{\text{flop}}{\text{byte}}\right), \text{ or } O(B)$$

(13)

Comparing Equation 12 with Equation 13, we can immediately notice that the ACR is improved by a factor of B. The actual measured performance is reported in Table 9. The best performance so far is reported in [46].

**Table 9: Performance data obtained from experiments for Dense Matrix Multiplication.**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>SP Peak, GFlop/s</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel 1: no reuse</strong></td>
<td>74.5, on Fermi</td>
<td>7.23% of Peak Flop</td>
</tr>
<tr>
<td><strong>Kernel 2: reuse within a (16 x 16) thread block.</strong></td>
<td>173.3, on Fermi</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Reported Best:</td>
<td>624, on GTX280</td>
<td>Volkov 2008 SC08 [46]</td>
</tr>
<tr>
<td></td>
<td>60% of peak</td>
<td></td>
</tr>
</tbody>
</table>
4.6.2 Example: Monte Carlo Simulation

Monte Carlo methods are a class of computational algorithms that use random sampling to compute the results of physical simulations or mathematical equations. It randomly generates inputs from the problem domain using a specified probably distribution suitable for that problem domain, uses these inputs to perform computation for the physical/mathematical system, and then aggregate the results of these individual computations into final result. This method is widely used in physical science, finance and business, engineering, statistics and etc.

In the finance domain, for instance, it is used to evaluate the prices and risks of a portfolio of securities and derivatives under uncertain market conditions. It is especially useful when there are no analytical solutions for the equations used to describe the values and risks.

Monte Carlo methods based methods rely on computation of a very large number (hundreds of thousands) of integration paths to reach reasonably accurate result, and since computation of each path is independent, it presents a very good chance for parallelization. Furthermore, since the computation of each path relies on a minimal input and the computation can be arbitrarily complex, the Application Characteristic Rate tends to be very high, meaning a FLOP bound computation is generally expected.

We take an example of computing the prices of stock options. The price of a call option (the right to sell a stock at a particular price) is a function of several factors.

- $S$: the price of the underlying stock.
• $T$: the time to expiration of the option. The time in the future to buy or sell a stock.

• $X$: the strike price of the option. The price in the future at which the stock is can be bought.

• $R$: the risk-free rate of return. Usually interest rate from Treasury Bond is used.

• Exercise restrictions: there are several styles of restrictions to limit the time options can be exercised. For instance, European Options can be only exercised at the stock expiration date, while American Options are more flexible and may be exercised at any time before the expiration date.

In computing the prices of these stock options, the underlying stocks are assumed to be a geometric Brownian motion with drift $\mu$ and volatility $\nu$.

The price $S_t$ is then described in a stochastic differential equation

$$dS_t = \mu S_t dt + \nu S_t dW_t$$

(14)

where $W_t$ is the Wiener random process: $X = W_T - W_0 \approx N(0, T)$.

The solution of this equation is

$$S_t = S_0 e^{\mu T + \nu (W_T - W_0)} = S_0 e^{\mu T + \nu \sqrt{T} N(0,1)}$$

(15)

The expected future value is

$$E(S_t) = S_0 e^{\mu T} E\left(e^{\nu \sqrt{T} N(0,1)}\right) = S_0 e^{\mu T} \cdot e^{0.5 \nu^2 T}$$

(16)

On the other hand, assuming the stock investment returns the same profit as the risk-free interest rate investment, the expected return in risk-free interest rate investment yields the
value of stock as \( S_0 e^{rT} \), we can hence determine that \( \mu = r - 0.5\nu^2 \). Subsequently the stock price can then be expressed

\[
S_T = S_0 e^{(r-0.5\nu^2)T + \nu\sqrt{T}N(0,1)} \tag{17}
\]

This price depends on the random sample \( N(0,1) \) and if we evaluate enough individual samples in the input, we are able to get an accurate value of the expected price of this underlying stock.

The prices of derivatives based on this stock can then be evaluated. For instance, the price of a European Option is \( V_{call}(S,T) = \max(S_T - X, 0) \).

Using Monte Carlo method, we can estimate the expected value of the call option \( V_{call}(S,T) \). We can generate a large number of conforming random samples, use them to get the prices of the underlying stock at time \( T \), and then compute the expected value of the price by averaging over all the prices

\[
V_{expected} = \frac{1}{n} \sum_{i=1}^{n} V_i(S,T) \tag{18}
\]

The current price of this option can then be computed by discounting its future value at time \( T \).

\[
V_{current} = V_{expected} e^{-rT} \tag{19}
\]

The cost of this computation is largely in computing the value \( V_i(S,T) \) of each path, which boils down to

\[
S_T = S_0 e^{(r-0.5\nu^2)T + \nu\sqrt{T}N(0,1)} \tag{20}
\]
Computing this function requires, if written in C programming language can be shown in the following.

```c
// compute the call value of a Stock Option
float call_value(float S, float X, float f1, float f2, float r) {
    float call_value = S*exp(f1+f2*r) - X
    return call_value = call_value > 0? call_value: 0;
}
```

We can then estimate the number of FLOPs in this kernel. There are 4 floating operations and an evaluation of the exponentiation function ‘exp’. The cost of this ‘exp’ function will dominate the cost.

The ACR(call_value) is expected to be large and peak flops is the expected optimization target.

Table 10: Monte Carlo simulation for pricing European Stock Options. The FLOP is estimated based on the lower bound of 5 FLOPs for evaluating one path.

<table>
<thead>
<tr>
<th>SP Peak</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo simulation for</td>
<td>Fermi</td>
</tr>
<tr>
<td>pricing the European Options.</td>
<td>&gt; 150 Gflop / s, if using the lower bound of 5 flops.</td>
</tr>
</tbody>
</table>
co-processing. The model we proposed is useful and insightful in that these existing optimization methods neatly fit into this model and can be correlated with the simple parameters in the model.

The target of optimization is to achieve peak bandwidth or peak FLOPs. As we have discussed earlier in this chapter, the Machine Characteristic Ratio (MCR) is a trait of a particular machine and do not change. To achieve desired FLOP or bandwidth, we will need to change that program, to achieve either peak bandwidth or FLOP. Ideally, the machine should operate at either peak FLOPs or bandwidth. In reality, however, the machine works in under-optimal state, for a variety of reasons. In the following, we discuss the commonly available techniques available for program optimization and put them into the context of the performance model we developed in this chapter.

4.7.1 Coalesced memory access and message granularity
Memory transactions from the processing cores to the DRAM are expensive operations and efficient using this operation is extremely important. For example, the DRAM should be viewed as aligned segments of 16 and 32 words [49], and if the threads within a thread block present enough locality, DRAM accesses from a thread block can be organized into a smaller number of memory transactions that better use the memory bandwidth. The effect is directly reflected in our model as it increases the effective bandwidth of the communication interconnect.

4.7.2 Utilization of on-chip Shared Memory
Shared memory is fast on-chip memory for communication and sharing data among threads within the same thread block. It can be also regarded as the software managed L1 cache. Effective use of this memory can reduce the DRAM access traffic, hence
effectively increases the Application Characteristic Ratio. Recall that we defined this ACR as the ratio of arithmetic operations over off-chip memory operations. Higher ACR for memory bound programs effectively reduce program runtime because the total number of memory operations is reduced. We have seen this effect in the previous example of dense matrix multiplication.

4.7.3 Shared Memory Bank Conflicts
This is a technique for in-thread-block optimization. Shared memory banks conflicts arise when multiple threads within a thread block try to address the same memory bank in the shared memory module. If this happens, instructions have to be serialized to resolve confliction. This does not change the traffic to the DRAM; it instead reduces execution time of a thread block. The effect on the model parameter is reduction in ACR because a thread block can now finish faster.

4.7.4 Divergent Warps
In the current Nvidia GPU architecture, the threads within the same thread block are executed in batched called warps. If the threads within the same warp take difference paths of the code, or diverge, these different paths will be executed in a serialized fashion. This divergence effectively increase the finish time of a thread block and hence decrease the ACR of that program.

4.7.5 Occupancy Improvement
Occupancy is the ratio of the number of actually resident threads (or warps, if considered in Nvidia context) to the maximum number of possible threads allowed to be remain resident. Low occupancy is a problem because it may introduce hardware under-utilization. For instance, in a bandwidth-bound program, if there are not enough threads
resident in a processor, the memory unit will not be fully utilized, resulting in longer than necessary program runtime. Low occupancy is caused by resource constraint. For example, if a thread block requires a large amount of on-chip shared memory, it is possible that there is no way to put enough thread blocks onto a processor for execution.

4.7.6 Instruction Optimization
This category of optimizations involves interchanging instructions that run longer with those that are less accurate, have shorter latency but is deemed enough. This is a low-level optimization and requires understanding of both the needs of application and the hardware details of executing those related instructions. One commonly used optimization that falls into this category is to swapping double precision mathematics with single precision, or mixing them. This optimization has effects on both the DRAM memory access time and the time spent in arithmetic operations.

These optimization techniques are also summarized and correlated to the model parameter in the following Table 11: Commonly used optimization technique and their relation with the model parameters.
Table 11: Commonly used optimization technique and their relation with the model parameters.

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Effect on Model Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coalesced Access to global memory</td>
<td>Smaller numbers of memory transaction required, due to more effective usage of memory bus bandwidth, or avoiding waste of memory bandwidth, towards peak bandwidth.</td>
</tr>
<tr>
<td>Utilization of Shared Memory</td>
<td>Less memory transaction required, due to reuse, or elimination of redundant loads from the slower global DRAM. Application Characteristic Ratio increased, towards peak FLOPS.</td>
</tr>
<tr>
<td>Avoid bank conflicts in shared memory</td>
<td>Reducing serializing memory access to the on-chip shared memory, towards peak FLOPS</td>
</tr>
<tr>
<td>Improve Occupancy</td>
<td>Toward achieving peak bandwidth.</td>
</tr>
<tr>
<td>Hiding Register Dependency</td>
<td>Improvement in instruction execution; towards achieving peak FLOPs.</td>
</tr>
<tr>
<td>Instruction Optimization</td>
<td>This optimization has effects on both the DRAM memory access time and the time spent in arithmetic operations</td>
</tr>
<tr>
<td>Control Flow optimization</td>
<td>Improvement in control flow in a thread blocks; towards achieving peak FLOPs.</td>
</tr>
</tbody>
</table>

4.8 Summary

In this chapter, we first discussed the pressing challenges to performance necessitates heterogeneous coprocessing for efficiency reasons given the constraints in silicon die area and maximum power package. We applied this Amdahl’s law to show the benefits of converting to heterogeneous computation under different composition of computation loads. The unanimous conclusion is that as long as there is parallel computation, even though a small faction, it is profitable to move these loads to cores that are optimized for throughput, rather than leaving them on the CPU core that is optimized for sequential thread latency.
The runtime of a program designed to exploit CPU + GPU coprocessing can be split into three portions: the time spent in sequential CPU, the time spent in parallel GPU coprocessing, and the time to move data between CPU and GPU DRAMs. Breaking the runtime into these three components helps with measuring this execution time. We then focused on characterizing the GPU coprocessing runtime because the CPU time term is a familiar one and the memory transfer time can be estimated using memory transfer size divided by system interconnect (PCI-Express) bus bandwidth. We noticed that the hierarchical thread model of GPU device and applied the PRAM to describe the threads that are in the same thread block; the behavior of a grid of thread block is modeled using the Bulk Synchronous Parallel model. BSP is applicable to GPU device execution because the thread blocks of a grid do not communicate, execute independently and concurrently on their own: this is local computation in BSP; the paradigm of loading from global DRAM to a locally (within the same thread block) shared fast on-chip memory: this is global communication in BSP; and finally the implicit barrier for each device kernel becomes the global barrier in BSP.

The application of BSP model to describe parallel GPU execution is useful in several aspects. Firstly, we introduced the notion of Machine Characteristic Ratio (MCR) to express the balance between the execution units and the memory sub-system in a parallel execution machine. We then introduced the Application Characteristic Ratio (ACR) to describe the ratio of arithmetic operations over off-chip traffic for a given program. Combining these two ratios, we can quickly decide if we put a program with a given \(acr\) onto machine having a given \(mcr\), what will be the peak performance and the potential
bottleneck. We showed the application of this idea to dense matrix multiplication and
Monte-Carlo Simulation for stock option pricing.

The model we proposed is also useful in performance debugging and tuning. Using the
model established in this chapter, we are able to put the existing optimization approaches
into this framework. This is not only conceptually aesthetic, but also establishes a logical
connection between the tuning techniques with the parameters that affect the
performance, giving a conscious guidance in taking actions for improving application
performance.
Chapter 5
Conclusion and Future Directions

We are undergoing one of the most important transitions in computing: single-thread performance is flating out and raw machine performance improvements must be largely derived from thread-level parallelism; parallel programming becomes unavoidable. Due to the efficiency (power and silicon area) in throughput-oriented hardware such as GPU, parallel computing systems consisting of heterogeneous processing cores become very appealing in power-sensitive environments such as mobile devices and large-scale data centers. These factors introduce additional complexities to the software programming activities. In this thesis, we proposed a system, Object-Oriented Stream Programming (OOSP), to support programming these heterogeneous CPU + GPU co-processing systems. We noticed that a significant amount of complexities in existing programming frameworks, e.g., CUDA, OpenCL, are due to the crosscutting concerns such as managing GPU devices, memory allocation/de-allocation, data transfers between host and device memories, and management of thread hierarchies of kernel functions. We hence propose to apply the principles of Aspect-Oriented Programming to modularize them as aspects. This method reduces clutters in the source code and hence helps to achieve better modularization. To help with programming the very specific aspects arising from heterogeneous co-processing systems, we provided a dedicated aspect language for efficiency. Additionally, in view of the complexity resulted from manually managing computational kernel code of GPU, we created a code transformer that automatically extract kernel code from the component code, further reducing structural changes to the original programs.
We also conducted case studies with an n-body simulation program. We programmed the OOSP versions this program and successfully translated it into source programs for CUDA targets. Comparison was made and we observed reduced program structure changes, uncluttered code, and runtime performance close to hand-coded CUDA version.

To further complete the programming system, we proposed a *performance model* for massively parallel heterogeneous co-processing systems. The model helps programmers with understanding the performance characteristics of their programs. We adopted the *Bulk Synchronous Parallelism* (BSP) model for the parallel portion of a program, because this model neatly subsumes the hierarchical execution model of computation that runs on GPU coprocessors. With the aid of this model, programmers can quickly bound and evaluate performance of various program structures, obtain insights into the hardware machine and the application, and consciously tune program performance, in a convenient and practical manner.

In the future, this work can be extended in several directions. Firstly, the proposed OOSP system currently concentrates on programming a single node system where GPU and CPU entities co-exist in the same system and communicate through the interconnect available in the same system board. OOSP, however, can be extended to a heterogeneous clustered environment, where cross-node communications and node management are also crosscutting concerns to the main computation and hence are well-suited to the aspect-oriented programming paradigm. Current practices use MPI to co-ordinate processes running on different computation nodes, requiring an additional layer of programming framework. A unified programming system is preferred. Secondly, the code weaving model is static, meaning advice code can only be changed with a recompilation / linking
of the whole project. This can be problematic because high-performing programs are often sensitive to hardware configurations and this often results in hardware-specific advice code, which necessitates a recompiling / linking for each deployment instance under current static weaving model. A more flexible scheme would be adopting a dynamic weaving model.
References


[52] Rohit Chandra et al., Parallel Programming in OpenMP, 1st ed. (Morgan Kaufmann, 2000).


Curriculum Vitae

Mingliang Wang

EDUCATION
PhD in Computer Engineering, Dept. of ECE                  Rutgers University
Master of Engineering, Dept. of ECE                      National University of Singapore
BS, Dept. of CS                                             Peking University

PROFESSIONAL EXPERIENCE
Bloomberg L.P.  NYC, NY
05/2010 -- present               Senior Software Developer

ConocoPhillips Houston, TX
06/2008 -- 08/2008               R. & D.  Intern

BMC Software Singapore
06/2001 -- 07/2005               Software Developer

PUBLICATIONS
“Object-Oriented Stream Programming using Aspects,” M. Wang and M. Parashar, 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Atlanta, Georgia, USA, 2010


