© 2011 Deepthi Nandakumar

AUTOMATIC TRANSLATION OF CUDA TO OPENCL AND COMPARISON OF PERFORMANCE OPTIMIZATIONS ON GPUS

BY

DEEPTHI NANDAKUMAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Wen-mei W. Hwu

ABSTRACT

As an open, royalty-free framework for writing programs that execute across heterogeneous platforms, OpenCL gives programmers access to a variety of data parallel processors including CPUs, GPUs, the Cell and DSPs. All OpenCL-compliant implementations support a core specification, thus ensuring robust functional portabiity of any OpenCL program. This thesis presents the CUDAtoOpenCL source-to-source tool that translates code from CUDA to OpenCL, thus ensuring portability of applications on a variety of devices. However, current compiler optimizations are not sufficient to translate performance from a single expression of the program onto a wide variety of different architectures. To achieve true performance portability, an open standard like OpenCL needs to be augmented with automatic high-level optimization and transformation tools, which can generate optimized code and configurations for any target device.

This thesis presents details of the working and implementation of the CUDAtoOpenCL translator, based on the Cetus compiler framework. This thesis also describes key insights from our studies optimizing selected benchmarks for two distinct GPU architectures: the NVIDIA GTX280 and the ATI Radeon HD 5870. It can be concluded from the generated results that the type and degree of optimization applied to each benchmark need to be adapted to the target architecture specifications. In particular, the different hardware architectures of the basic compute unit, register file organization, on-chip memory limitations, DRAM coalescing patterns and floating point unit throughput of the two devices interact with each optimization differently.

To my parents, for making me who I am. To my husband, for his unwavering support. And to my son - my life, my pride and my joy.

ACKNOWLEDGMENTS

I would like to wholeheartedly thank my adviser, Dr. Wen-mei W. Hwu, for his guidance during my master's program. His constant support and encouragement kept me energized and enthusiastic about my work. His stimulating ideas and thoughtful feedback were invaluable to me in defining my research. Being counted as one of his students is a privilege that I am grateful for.

I would also like to thank the members of the IMPACT group for the great teamwork and constant flow of ideas. In particular, I would like to thank I-Jui (Ray) for all those intellectually stimulating discussions, John for sharing his research insights and Nady for all the fun and good food he shared. I am also deeply grateful to Marie-Pierre for her prompt help in all official matters and for making life so much easier, especially in my last semester.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 4

CHAPTER 3 FINE-GRAINED SPMD PROGRAMMING MODELS: CUDA AND OPENCL 6
- 3.1 Platform Model 7
- 3.2 Memory Model 9
- 3.3 Device Execution Model 10
- 3.4 Code Development and Programming 11

CHAPTER 4 CUDATOOPENCL: IMPLEMENTATION 15
- 4.1 Kernel Program Transformations 15
- 4.2 Host Program Transformations 17

CHAPTER 5 GPU ARCHITECTURE EVALUATION: NVIDIA GTX 280 AND ATI RADEON 5870 22
- 5.1 Common Architectural Features 22
- 5.2 Architectural Differences 24

CHAPTER 6 PERFORMANCE RESULTS OF OPTIMIZATIONS . 28
- 6.1 Improving Performance of Compute-Bound Kernels on GPUs . 28
- 6.2 Improving Performance of Memory-Bound Kernels on GPUs . 33

CHAPTER 7 CONCLUSION 37

REFERENCES 38

CHAPTER 1

INTRODUCTION

In the last decade, graphics processing units (GPUs) have gradually evolved into highly parallel processors with extremely high computational throughput. The addition of support for floating point arithmetic, programmable shader pipelines and arbitrary memory addressing have enabled GPUs to be used not only as powerful graphics engines but also as programmable, high-performance, massively parallel engines for scientific and general-purpose computing. Programming this set of applications for the GPU (termed GPU computing) had historically been a challenging exercise, because the application had to be restructured in terms of the graphics pipeline using graphics APIs such as OpenGL and DirectX. The introduction of NVIDIA's CUDA (Compute Unified Device Architecture) [1] enabled better productivity and performance of GPU computing applications by eliminating the need for graphics application programming interfaces.

The CUDA programming model is based on fine-grained SPMD threads, with limited inter-thread communication, controlled by a centralized process [2]. CUDA has been demonstrated as an effective programming model for porting a variety of applications to GPUs, such as molecular dynamics [3], medical imaging [4], and bioinformatics with significant gains in performance and functionality. However, as modern processor architectures evolved into highly parallel heterogeneous systems, there was a definite need to enable software developers to take full advantage of the compute power of heterogeneous CPUs, GPUs and other devices from a single, multi-platform codebase. OpenCL (Open Computing Language), managed by the Khronos OpenCL Working Group, emerged as an open, royalty-free standard for portable, parallel programming of heterogeneous CPUs, GPUs, Cell, DSP and other processors [5].

OpenCL supports a wide range of applications, from embedded software to HPC solutions, through a low-level, high-performance, portable hardware

abstraction layer. OpenCL is thus poised to form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. Many semiconductor and OEM partners of the Khronos working group, including Apple, NVIDIA, IBM and AMD, have released OpenCL conformant implementations, and an extensive, diverse group of companies have been contributing towards the evolution and development of the OpenCL specification. The core specification that every OpenCL-compliant implementation is required to support abstracts the ISA and driver model of a compute device with a standard interface, thus allowing one source codebase to be ported to different platforms with assurance of correct functionality.

With widespread acceptance of OpenCL, there is significant interest in converting the large body of applications written in CUDA to OpenCL, and evaluating their performance on other OpenCL-compliant parallel systems, such as GPUs from different vendors, CPUs and Cell. To this end, this thesis presents a language translator tool, CUDAtoOpenCL, that translates CUDA code into valid OpenCL code, while still preserving the kernel decomposition and configuration of the original CUDA program. Thus, CUDAtoOpenCL can be utilized to easily generate a portable application codebase for parallel programming on a variety of devices. However, the performance of an OpenCL application is not nearly so portable across multiple architectures. Some of this is expected, since each hardware architecture provides different strengths and weaknesses to applications. For instance, the balances of DRAM bandwidth, cache sizes and cache bandwidth, single-precision floating-point throughput, and special function unit throughput all affect performance significantly. But some performance differences are, if not unexpected, at least unfortunate in that they are a casualty of the current program expression. For instance, different hardware have different SIMD width requirements, thus causing code written in a different width or scalar mode to perform well below peak throughput. Some architectures may provide good performance for a certain instruction mix ratio while others may require a different ratio for better yield. This lack of systemic support for performance tuning in OpenCL leads serious software developers to maintain multiple high-performance codebases for multiple devices.

To achieve the goal of portable and efficient execution of OpenCL programs on diverse devices, more aggressive optimizations need to be

built into OpenCL compilers. Thus, every platform can use these advanced optimization techniques, in the configuration best suited for the target devices(s), to generate performance from a single expression of the program. This thesis details the challenges of performance portability in OpenCL-compliant architectures and explains why the solution to these challenges is not straightforward. Examples of sophisticated transformations that give the best known performance results on key benchmarks were analysed, and their performance portability was evaluated on similar yet significantly different many-core architectures. This work focuses on two GPU platforms from different vendors, in particular the NVIDIA GeForce GTX 280 and the ATI Radeon HD5870, to analyze similarities and differences in program behavior.

Chapter 2 describes previous related work and background material relevant to this thesis. Chapter 3 illustrates the general characteristics of fine-grained SPMD programming models and articulates the similarities and differences between CUDA and OpenCL. Chapter 4 describes the implementation details and transformations within the CUDAtoOpenCL source-to-source translator. The salient architectural similarities and differences between the NVIDIA GTX280 and ATI Radeon HD5870 are discussed in Chapter 5, followed by the performance evaluation of selected benchmarks in Chapter 6.

CHAPTER 2

RELATED WORK

There has been significant research interest in converting CUDA code to OpenCL, most notably the Swan project [6]. However, Swan uses a perl-based regular expression replacement method in kernels and a common runtime library for the host API calls. This involves rewriting of CUDA code to use the Swan runtime library, which CUDAtoOpenCL avoids by using a more reliable AST-based source-to-source translator for host and kernel code. More importantly, the regular expression substitution method employed in the Swan library has many repercussions in terms of the tool capability. Some transformations can be very cumbersome or even infeasible when implemented with string substitution. This is especially true in the case of transformations that require adjustment of scope or collection of information from multiple positions in code in order to implement a transformation. The AST-based CUDAtoOpenCL translator has a complete view of the relevant source code and is thus capable of handling real-world applications effectively and requires very little restructuring of code.

Several previously published works have done extensive studies on GPU computing optimization. For modern GPU computing languages, some of the earliest optimization studies were performed by Ryoo et al. for NVIDIA's CUDA language on the GeForce 8800 GTX [7]. Jang et al. analysed architecture and optimizations using ATI's Brook+ language on the Radeon HD3870 [8]. However, there are few detailed publications that analyze optimizations based on a fine-grained hierarchically organized SPMD model like OpenCL/CUDA for the ATI Radeon/Firestream architectures. Also, the need for a single codebase with efficient and portable access to different architectures like the NVIDIA and ATI GPUs requires us to understand the effect of performance optimizations on these architectures on a common platform like OpenCL.

A large portion of application optimizations for GPU computing in

general, and OpenCL programming in particular, is still applicable and relevant in driving this analysis: data reuse transformations such as the use of local/constant memory, efficient use of registers, trading off resource usage with effective memory latency hiding (which is implemented by time-multiplexing across work-items), and effective use of off-chip bandwidth. However, our major contribution lies in investigating optimization techniques whose effect on performance differs significantly across GPU architectures.

Performance portability has been an issue for even sequential systems for quite some time, although to a lesser degree. Historically, the autotuning community has done significant work addressing this issue for sequential machines, and recently applied some of the same techniques to some basic GPU computing optimizations [9] [10]. However, autotuning fundamentally relies on a space of parameters to explore, hence, augmenting compiler technology with advanced optimization techniques is the only clear solution to avoid generating multiple codebases.

CHAPTER 3

FINE-GRAINED SPMD PROGRAMMING MODELS: CUDA AND OPENCL

A programming model that is also portable across multiple parallel computing platforms has many challenges. The model must be capable of expressing a wide variety of applications, yet have enough flexible parameters to enable a wide variety of architectures to be supported. The fine-grained SPMD (Single Program Multiple Data) threaded model with limited thread cooperation, controlled by a centralized process has gained popularity in many-core parallel programming, and we describe in this section, two models based on this paradigm: CUDA and OpenCL.

CUDA, released in 2006 by NVIDIA [1], was introduced as a proprietary technology comprising a software architecture, language, API and tools for GPU compute programming. The CUDA specification also describes the ISA as well as the hardware architectural view of the GPU device. CUDA tools and SDK are provided solely by NVIDIA, specifically targeting only the NVIDIA GPUs. However, as an open, royalty-free standard, with a defined API and language specification, OpenCL is equipped to support multiple device classes (CPUs, GPUs, DSPs, Cell, etc). SDKs and tools are provided by corresponding device vendors [5]. With widespread industry-wide support, OpenCL is projected to form the foundation for portable parallel programming. Since the CUDA programming model has a larger and more varied set of application codebases with a stronger technical support group, an easy conversion scheme for programs written in CUDA to OpenCL code is of great interest to the community. This necessitates a clear understanding of the similarities and differences between the CUDA and OpenCL programming models, with special emphasis on the device, memory, execution and code development models as well as the respective toolchains.

Table 3.1: A comparison of terminology across OpenCL and CUDA and their description

OpenCL Terminology	CUDA Terminology	Description
Compute Unit	Streaming Multiprocessor	
Processing Element	Scalar Processor	
Local Memory	Shared Memory	Software managed memory shared between threads/work-items in a block/work-group
Global Memory	Global Memory	Baseline memory for kernel inputs and outputs
Constant Memory	Constant Memory	Read-only memory
Image Buffer	Texture Memory	Memory related to 2-D and 3-D data structures
Private Memory	Registers	Local to each thread/work-item

3.1 Platform Model

As shown in Figure 3.1, the CUDA device model features a hierarchical, scalable system with a number of streaming multiprocessors (SM), each consisting of multiple processors [2], [11]. All the processors in a single SM share resources like shared memory and constant and texture caches; however, each processor has its own private register space. OpenCL shares almost all features of the CUDA device model [5]; however, the specification adopts a more generic, vendor-agnostic terminology as shown in Figure 3.2. While the CUDA device model assumes an NVIDIA GPU and a host CPU, the OpenCL specification refers only to a host and a compute device, which could be a GPU, multicore CPU, Cell, DSP, etc. Table 3.1 shows the mapping between OpenCL and CUDA for terminology pertaining to the platform model details. Both CUDA and OpenCL support querying of actual device capabilities and features using special API functions.

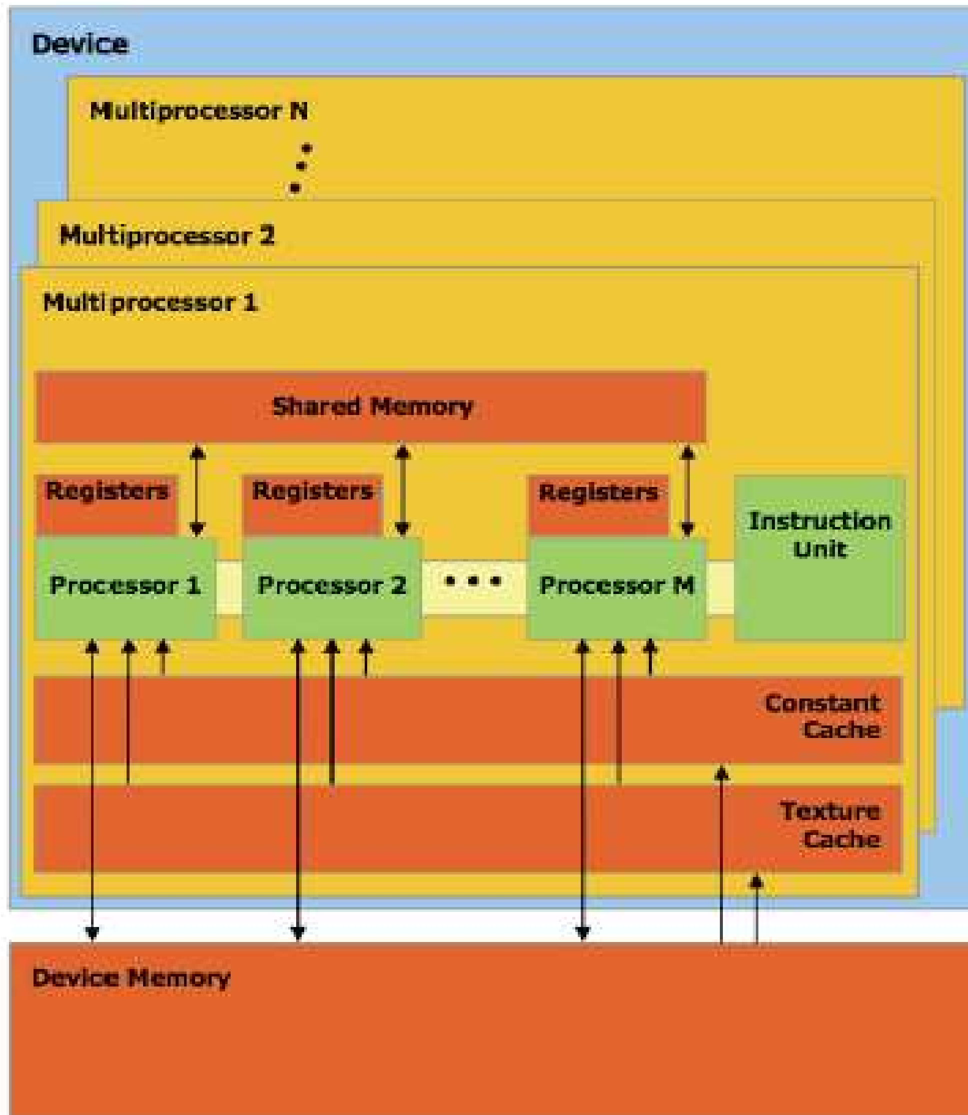

Figure 3.1: CUDA: Platform Model and Memory Model

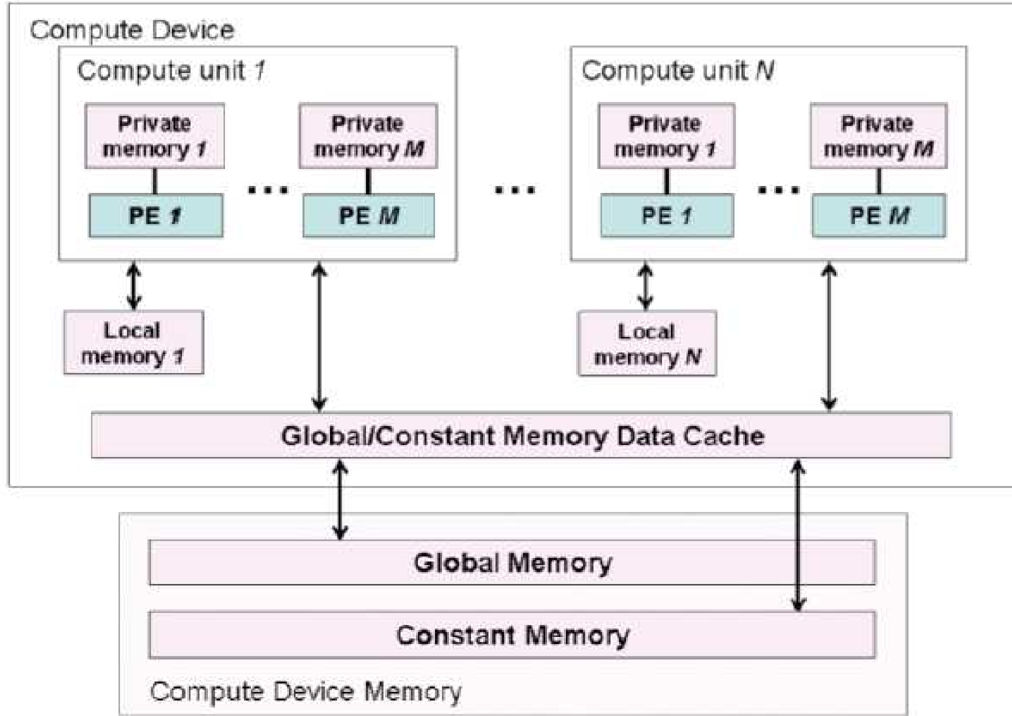

Figure 3.2: OpenCL: Platform Model and Memory Model

3.2 Memory Model

CUDA and OpenCL feature a multi-tiered, hierarchical memory model with separate host and device memories. Memory is explicitly defined and managed by the programmer as five distinct regions in both CUDA and OpenCL: global, constant, local (OpenCL)/shared (CUDA), image (OpenCL)/texture (CUDA) and private (OpenCL)/register (CUDA). The memory regions and how they relate to the respective platform models are described in Figure 3.1 and Figure 3.2. Table 3.1 supplies a mapping between the terminology as applied to memory model details in both CUDA and OpenCL as well as a summary of the memory space capabilities. As described in the platform model description, OpenCL uses more generic terminology and abstracts away differences introduced due to implementation characteristics. For example, CUDA explicitly defines constant and texture memories as cached whereas OpenCL considers these details implementation/vendor-dependent. Another difference between the CUDA and OpenCL memory definitions is that there is no OpenCL equivalent of CUDA-defined local memory, as the OpenCL specification

Table 3.2: A comparison of execution model terminology and kernel programming keywords/built-in functions across OpenCL and CUDA

OpenCL Terminology	CUDA Terminology
Grid	N-Dimensional Index Space
Block	Work-group
Thread	Work-item
threadIdx	get_local_id
blockIdx	get_group_id
blockDim	get_group_size
gridDim	get_num_groups

essentially leaves details about register overflow-handling to the device vendor. Both CUDA and OpenCL support querying of actual memory sizes and capabilities using special API functions.

3.3 Device Execution Model

An OpenCL/CUDA program is typically divided into host and device components; the host program performs system-level processing and launches the device component of the program, called a kernel function. When the host invokes a kernel function, it launches a *grid* (CUDA)/*N-dimensional index space* (OpenCL) of *threads* (CUDA)/*work-items* (OpenCL) that begin their execution at the start of the kernel function. The parallel grid/NDRange is composed of groups of threads/work-items called thread *blocks* (CUDA)/*work-groups* (OpenCL), which are developer-defined groups that may perform barrier synchronization and locally share data. Thus, both CUDA and OpenCL provide very similar hierarchical decomposition of the computation index space, which is supplied to the device on kernel invocation. The execution model is also linked with the device and memory models. The CUDA programming model defines individual threads such that they map directly onto hardware threads; however, since the OpenCL specification leaves the implementation details to the device vendor, each work-item may or may not map directly to a hardware entity. Individual threads/work-items and thread blocks/work-groups can be uniquely identified within the kernel

program using keywords; Table 3.2 shows the mapping between OpenCL and CUDA for terminology relating to the execution model as well as keywords that uniquely identify each entity within the parallel region of execution.

3.4 Code Development and Programming

Due to the similarities in their programming models, there are some similarities between the OpenCL and CUDA code development model. However, there are also significant differences between them, particularly within the compilation toolchain and runtime support. A CUDA/OpenCL application consists of a *host* program, which activates computation kernels or data-parallel routines in the *device* program. While only data-parallel execution is supported in CUDA, OpenCL also supports task-parallel and hybrid models.

3.4.1 Code Development Model

In OpenCL, the host program interacts with the device using the OpenCL C programming language; in CUDA, the host program uses either C runtime for CUDA or CUDA Driver API. The popularly used CUDA runtime API provides a higher level of abstraction than the OpenCL C API or the CUDA Driver API and is, therefore, less verbose. CUDA provides deep host and device program invocation support, with very efficient CUDA-specific kernel invocation syntax. On the other hand, the OpenCL C API and CUDA Driver API provide finer-grained control. CUDA also specifies a device program intermediate language, whereas the OpenCL specification recognizes this as an implementation-specific detail, which may or may not be present.

3.4.2 Toolchain

The CUDA and OpenCL toolchain differ significantly in their capabilities and limitations. Within CUDA, the entire program is statically compiled by the CUDA-SDK provided *nvcc* compiler. CUDA custom kernel invocation syntax and deep host and device program integration requires the use of the *nvcc* compiler for host program compilation. For device code, *nvcc*

emits either CUDA intermediate code, called PTX assembly or device-specific binary code. PTX code can be further compiled and translated by the device driver to actual device binary code. The *nvcc* compiler enables seamless code generation; device program files can be compiled separately or mixed with host code. If required, the device and host program can also be separately compiled and the output C code integrated with the host toolchain.

The OpenCL toolchain, on the other hand, relies on a compiler provided by the OpenCL implementation to translate OpenCL C to supported device executable code. The compiler must necessarily support a minimal standard set of OpenCL defined options. OpenCL kernels can be compiled either at build-time (statically), like CUDA device programs, or run-time (dynamically). In runtime compilation, the OpenCL API accepts the source text as a string from the host program and returns compilation errors, if any. This means typically the kernel program is included as a file separate from the host program. The kernel source code is included in the application binaries. The host program is compiled with the default host toolchain and OpenCL is used through its C API. Thus, the OpenCL toolchain with dynamic compilation is much more flexible than the C runtime for CUDA toolchain for the final application.

3.4.3 Host API Programming

Host programming in CUDA offers two options: the higher-level C for runtime CUDA API and the lower-level CUDA Driver API. The CUDA runtime eases device code management by providing implicit initialization, context management and module management. In contrast, the CUDA driver API requires more code and is harder to program and debug, but offers a better level of control. The driver API requires the kernel program to be loaded as a CU-binary file.

The OpenCL API is as low-level as the CUDA Driver API, requiring explicit context initialisation, context creation and deletion, command queue management, manage device memory, dispatch kernels on devices, etc. Though it is similar in design to the CUDA Driver API, all API function calls and built-in library functions are different. Table 3.3 compares a typical host program written using the OpenCL C API, CUDA Driver API

Table 3.3: A comparison of host programming across the OpenCL C API, the CUDA Driver API and CUDA Runtime API

OpenCL C API	CUDA Driver API	CUDA Runtime API
Setup		
Initialize platform Get devices Choose device Create context Create command queue	Initialize Driver Get devices (Choose device) Create context	
Device and host memory buffer setup		
Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result
Initialize kernel		
Load kernel source Create program object Build program Create kernel object bound to kernel function	Load kernel module (Build program) Get module function	
Execute the kernel		
Setup kernel arguments Setup execution configuration Invoke kernel	Setup kernel arguments Setup execution configuration Invoke kernel	 Setup execution configuration Invoke kernel (using kernel invocation syntax) and pass kernel arguments
Copy results to host		
Copy results from device memory	Copy results from device memory	Copy results from device memory
Cleanup		
Delete memory objects Delete context	Delete memory objects Delete context	Delete device memory pointers

and CUDA Runtime API. OpenCL also does not support stream offsets at the API/kernel invocation level. Offsets must be passed in as a parameter to the kernel and the address of the memory computed inside it. CUDA kernels may be started at offsets within buffers at the API/kernel invocation level.

3.4.4 Kernel Programming

OpenCL kernel programming (technically referred to as the OpenCL C programming language) and CUDA kernel programming (technically referred to as C for CUDA) are based on the C99 and C programming languages, respectively, with extensions and limitations as well as an extensive library of built-in functions. CUDA kernel programming also supports limited C++ features. Some differences between OpenCL and CUDA kernel programming include access to work-item/thread indices; while OpenCL uses built-in functions, CUDA uses built-in variables as detailed in Table 3.2. Multiple pointer traversals are allowed with C for CUDA, but must be avoided on OpenCL, as the behavior of such operations is undefined. In OpenCL, pointers must be converted to be relative to the buffer base pointer and only refer to data within the buffer itself. CUDA defaults all kernel pointer arguments to global memory, while OpenCL requires address space qualification for kernel pointer arguments. CUDA includes support for both voting functions and atomic functions, whereas OpenCL supports atomic functions only as extensions. Asynchronous memory copying and prefetch functions are supported only in OpenCL.

CHAPTER 4

CUDATOOPENCL: IMPLEMENTATION

The CUDAtoOpenCL source translation framework is implemented within the Cetus source-to-source compilation framework [12] with slight modifications to the IR and preprocessor to accept ANSI C with the language extensions of the CUDA version. The Cetus compilation framework implements an abstract sytax tree (AST) intermediate representation as a Java class hierarchy. A high-level representation provides a syntactic view of the source program, making it easy to understand, access and transform the input program [13]. The CUDAtoOpenCL tool is implemented as a transformation pass on the kernel program and host program separately within the Cetus infrastructure. The transformations in the host program are relatively more sophisticated and advanced compared to the kernel program transformations.

4.1 Kernel Program Transformations

The transformation passes in the kernel program can be divided into data extraction passes, function header and function body transformations.

4.1.1 Extraction of Constant Memory Declarations

Many of the transformations involving kernels require propagation of information extracted from the CUDA source to different parts of the OpenCL program. In particular, this involved the extraction of constant memory declarations and kernel function headers.

The CUDA syntax allowed constant memory declarations to be made anywhere within the source, as long as it was in the scope of the kernel program. As mentioned in Section 3, the OpenCL host and kernel program

are typically in separate files, unlike a CUDA program; thus the scope of the OpenCL and CUDA program could be different. To maintain consistency, therefore, the CUDAtoOpenCL translator transforms the kernels such that the constant memory declarations are passed as arguments into the kernel. The entire program is scanned through to check for constant memory declarations.

4.1.2 Function Header Transformations

In general, a typical CUDA kernel header can be represented as: __global__ void cenergy (int numatoms, float gridspacing, float * energygrid) while a typical OpenCL header is represented as __kernel void cenergy (int numatoms, float gridspacing, __global float * energygrid, __constant float4 atominfo[]). The following transformations are applied to the CUDA function header to generate valid function code.

- The *__global__* function qualifier in a CUDA kernel program is replaced by the *__kernel* qualifier in the OpenCL program.
- The *__device__* function qualifier in CUDA is invalid in OpenCL and is to be removed.
- All memory pointers are, by default, considered as global memory pointers in a CUDA kernel. The OpenCL syntax requires explicit use of the *__global* keyword for global memory pointers.
- Add constant memory declarations extracted from the program to the parameter list.

4.1.3 Function Body Transformations

Many CUDA built-in functions are invalid in OpenCL and are to be replaced by their OpenCL equivalents. The following replacements are applied to CUDA kernel code to generate valid OpenCL code.

- The barrier synchronization function *__syncthreads()* is replaced with its OpenCL equivalent *barrier (CLK_LOCAL_MEM_FENCE)*;

- Native math-specific functions in CUDA such as *__sqrtf, __sinf, __cosf, __tanf*, etc. are to be replaced with their OpenCL native equivalents *native_sqrt, native_sin, native_cos, native_tan*, etc. A more complete list can be obtained in [5].

- While CUDA uses built-in structures like threadIdx, blockIdx, etc. to access thread/block indices, OpenCL defines built-in functions for the same. Table 3.2 lists all required translations, with the OpenCL functions accepting integer arguments for the corresponding elements of the structure variable.

4.2 Host Program Transformations

As explained in Section 3.4.3, host programming in CUDA offers two options: the higher-level C for runtime CUDA API and the lower-level CUDA Driver API. Although OpenCL bears significant similarities with the CUDA Driver API, most current CUDA programs are written in the higher-level and more programmer-friendly C for runtime CUDA API. The CUDAtoOpenCL tool translates source code written in the C runtime API to the OpenCL C API. The following sections detail each step associated with the host program transformation.

4.2.1 OpenCL Environment Setup

The OpenCL C API requires extensive environment setup details, which are absent in the C for CUDA runtime API. For a single compute device and a single kernel file, however, the environment setup is program-invariant and can be abstracted as a library function that gets called when the host program is launched. The following are the tasks which are a part of the environment setup in a generic OpenCL program.

- Create context based on type of compute device (CPU, GPU, ACCELERATOR, DEFAULT, ALL).

- Populate context structure and device details by querying the device details using the OpenCL API.

Table 4.1: CUDA memory handling functions and their equivalent OpenCL API functions

CUDA memory handling function	OpenCL memory handling function
cudaMalloc	clCreateBuffer, Options: CL_MEM_READ_WRITE
cudaMemcpy, Options: cudaMemcpyHostToDevice	clEnqueueWriteBuffer
cudaMemcpy, Options: cudaMemcpyDeviceToHost	clEnqueueReadBuffer
cudaMemcpyToSymbol	clCreateBuffer, Options: CL_MEM_READ_ONLY — CL_MEM_COPY_HOST_PTR
cudaFree	clReleaseMemObject
cudaMemset	clEnqueueWriteBuffer

- Create command queue and set command queue properties.
- Create OpenCL program executables with the kernel source code.
- Build the kernel program.
- If build not successful, get program build info and display errors.

This library function requires only the kernel and header file names to be passed as parameters. This step also includes extraction of information regarding the number of kernels, which are then used to declare the kernel variables globally.

4.2.2 Memory Handling Functions

The C for CUDA runtime API includes functions for memory allocation and memory copy, which can be directly translated into OpenCL memory allocation and copy functions. Table 4.1 gives the CUDA memory handling functions and the equivalent OpenCL functions with the relevant options.

All CUDA memory handling functions are, by default, implemented as blocking API calls. However, the implementation of an OpenCL memory copy/allocation function is vendor-dependent; hence, all OpenCL memory

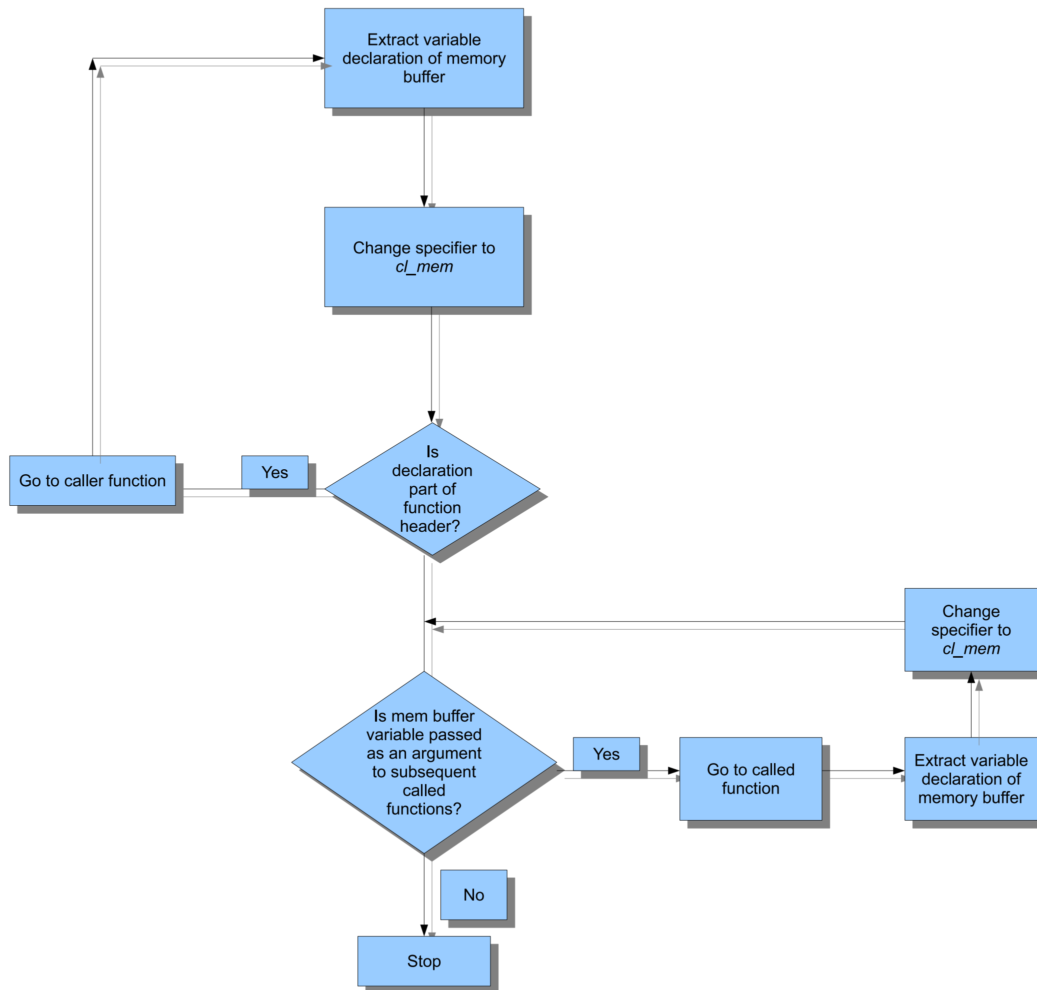

Figure 4.1: Memory Functions: Propagating datatype changes across functions

functions are best followed by a “wait” command to ensure the previous function has been completed. A conservative method to ensure this in OpenCL is to use *clWaitForEvents* after each memory handling function. The CUDAtoOpenCL translator is not yet capable of detecting asynchronous memory handling function calls.

While the CUDA API deals completely with C-based pointers, the OpenCL API uses a wrapper structure referred to as *cl_mem* to denote memory buffers. This necessitates a change of datatype in the source code. A change in datatype of the memory buffer requires the variable declaration to be changed and this change to be propagated across the function call stack in both directions (towards called function up the function call stack, and towards calling functions down the function call stack). The subroutine that contains the *cudaMalloc* function call is the current subroutine; a subroutine with the memory buffer variable passed as an argument that occurs further up in the function call stack (a called function) is a terminal function if the memory buffer argument is not passed as an argument in any subsequent called functions. The datatypes of the function argument corresponding to the memory buffer in each of the function headers is replaced with the *cl_mem* specifier until the terminal function is reached. Similarly, a subroutine that occurs further down the function call stack is a terminal function if the subroutine contains the variable declaration statement of the memory buffer in the function body. The datatypes of the function argument corresponding to the memory buffer in each of the function calls is replaced with the *cl_mem* specifier until the variable declaration in the terminal function is reached. Figure 4.1 shows the block diagram of the recursive handling of the function call stack.

The CUDAtoOpenCL translator is thus fundamentally superior to regular expression-based substitution mechanisms. Adjusting the scope position and propagating datatype changes across function boundaries either cannot be handled or can be managed only with great difficulty in substitution-based translator mechanisms. The above recursion-based mechanism handles inter-procedural transformations with high accuracy and eliminates the need for source code changes.

4.2.3 Modifying Kernel Execution Calls

The C for CUDA API features deep host and device program integration support with very efficient syntax; however, the OpenCL C API uses separate kernel invocation function calls. The setup for the execution configuration of the kernel also differs across CUDA and OpenCL; while CUDA uses structures of type *dim3*, OpenCL uses arrays of type *size_t*. The CUDAtoOpenCL library defines variables of type *size_t*, which are then initialised to the values corresponding to the *dim3* variables within the source code. The CUDA kernel execution statement is then replaced with a number of OpenCL API function calls as detailed below.

- Create OpenCL kernel with the name of the kernel function to be launched as argument.
- Modify kernel execution configuration parameters.
- Set kernel arguments (parsed from the CUDA kernel launch statement) one after another.
- Add all constant memory declarations as kernel arguments to the end of the existing argument list.
- Launch the kernel using *clEnqueueNDRangeKernel* and the initialised execution configuration parameters.
- Wait for launched kernel to be completed using *clWaitForEvents*. This is equivalent to a *cudaThreadSynchronize* command in CUDA.

CHAPTER 5

GPU ARCHITECTURE EVALUATION: NVIDIA GTX 280 AND ATI RADEON 5870

This section describes the salient architectural features of two GPU devices: the NVIDIA GTX 280 and ATI Radeon 5870. The main goal here is to understand the similarities and contrasts between these devices, and how these architecture variations affect performance optimizations from a developer point of view.

5.1 Common Architectural Features

Most of the basic features of the OpenCL programming model from Figure 3.2 are realised in hardware by both GPU devices [14], [15]. As such, these GPUs share many common features. We consider here the case of a single device only, as the effect of all performance optimizations of interest is felt at this level, and can easily be extended to multiple devices.

The desire to use GPUs for both graphics and general-purpose computation motivated GPU vendor companies to develop a new unified graphics and compute GPU architecture and the OpenCL programming model. The basic massively multithreaded array of processors is organised into Streaming Multiprocessors/SIMD engines in the NVIDIA/AMD architectures, respectively, referred to as *compute unit* in OpenCL parlance. Each SM/SIMD engine consists of a core (a complete processing unit consisting of ALUs and supporting hardware).

Transparent scaling across a wide range of parallelism is a key design goal of both the GPU architecture and the OpenCL programming model. When an OpenCL program launched on the host CPU invokes a kernel, a centralized unit distributes the work-groups to SMs/SIMD engines with available execution capacity. Each of these compute units contains primary hardware structures for data: storage for work-item contexts, register file

for private data storage of work-items, and local memory storage, typically implemented as banked SRAM capable of servicing multiple simultaneous requests from different SIMD lanes. Local storage has very low access latency and high bandwidth; thus OpenCL programs can copy data from global memory to local memory for memory regions that are accessed multiple times by a block. Each of these resources is divided equally among all the work-groups assigned to that compute unit. To instantiate a kernel, a work-group must be provided with a work-item context and enough private storage for every constituent work-item, and a region of local storage as large as the requested amount of local memory. When a work-group uses more resources than what is available inside a compute unit, this means all work-items in a work-group cannot be launched simultaneously due to hardware limitations and the OpenCL runtime returns an error. If the kernel uses too many registers, some OpenCL implementations may decide to promote a few registers to higher levels of memory, thus leading to significant performance degradation, depending on how the particular vendor driver handles the situation. Thus, the size of the storage structures (context storage, register file and local memory) places an upper limit on the total number of work-groups that can be launched within a compute unit; this is different across NVIDIA and AMD GPUs, and, sometimes, even across different devices from the same vendor.

Typically, a compute unit on the NVIDIA and AMD GPUs also implements the barrier synchronization intrinsics in a single hardware instruction. Lightweight thread creation is also a common feature of these devices; also, hundreds of threads can be launched with zero thread scheduling overhead. The SM/SIMD engine is responsible for mapping each thread to a core, which then executes independently with its own instruction address and register state. Because all work-items in a work-group launch the same kernel, different work-items will share a large portion of their dynamic instructions. By exploiting this pattern, architecture can achieve higher performance with reduced complexity by implicitly executing groups of work-items simultaneously in SIMD-like fashion. In both the NVIDIA and AMD GPUs, the SM selects a statically determined group of work-items (called warp in an NVIDIA GPU and wavefront in an AMD GPU) at every instruction issue time that is ready to execute and issues the next instruction to the active work-items in that group. This issue-group width is unrelated to

the SIMD width; both the NVIDIA and AMD architectures time-multiplex a single issue-group onto SIMD execution lanes of narrower width. If threads of a single group diverge via a data-dependent conditional branch, each branch path taken is serially executed via predicated execution, and when all paths complete, all threads converge back to the same execution path.

Global memory, which is implemented in external DRAM, coalesces individual accesses from parallel work-items, from issue-groups or relevant subunits, into fewer memory block accesses when the addresses fall in the same block and meet alignment criteria. Severe bandwidth degradation can occur if addresses cannot be bundled together into a single memory block or if the alignment criteria are not met. More details can be found in [15] and [14]. The large thread count in each SM/SIMD engine, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM.

5.2 Architectural Differences

Although the NVIDIA GTX 280 and ATI Radeon 5870 GPUs share many common architectural features as seen in Section 5.1, there are also many primary differences which directly affect program optimization from a developer point of view. Table 5.1 lists various parameters that differ across the two architectures.

One of the most significant differences is in the architecture design of the basic core of each compute unit. While the NVIDIA GPU relies significantly on deep instruction pipelining from multiple issue-groups to hide instruction latency, the AMD GPU relies more heavily on instruction-level parallelism within a single work-item to achieve the same arithmetic throughput. Thus, each core within a compute unit in the AMD GPU is a 5-way VLIW design, while the NVIDIA GPU simply includes a scalar core, with requisite ALUs and other supporting hardware. As mentioned in Section 5.1, the vendor defines an upper limit on the number of work-items that can simultaneously be scheduled on a single compute unit (depending on the available context storage). Since the NVIDIA architecture relies on instruction pipelining across multiple issue-groups to hide latency, developers need to make sure that the actual scheduled number of work-items on each compute unit is

Table 5.1: GPU Architecture Parameters Comparison

	NVIDIA GeForce GTX 280	ATI Radeon HD5870
No. of compute units	30	20
Work-item execution units	8 scalar units (double-clocked w.r.t. system)	16 5-wide VLIW units
Peak SP throughput	933 GFLOPS	2720 GFLOPS
Peak global memory bandwidth	141.7 GB/s	154 GB/s
Register file size (per compute unit)	64kB scalar registers	256kB 4-wide vector registers
Work-item issue-group width	32	64
Global memory burst (coalescing) size	64 bytes / 16 words	256 bytes / 64 words
Work-item contexts (per compute unit)	1024	1587.2 (Centralized, GPU-wide limit of 31744)
Local scratchpad capacity	16kB	32kB
Local scratchpad banks	16	32
Work-items accessing local memory banks	16 per cycle	16 per cycle

as close to the theoretical maximum as possible for best performance. On the AMD GPU, on the other hand, developers need to focus on exposing as much instruction-level parallelism to the compiler as possible; excellent performance can be achieved even if the scheduled number of work-items on each compute unit is significantly below the theoretical maximum that can be accomodated. For this very same reason, OpenCL programs on the AMD GPUs are liable to heavy performance degradation if the compiler is incapable of extracting ILP within a kernel. Note, however, that ATI GPUs still time-multiplex different issue-groups to effectively hide global memory latency.

Another major difference between the NVIDIA GPU and the ATI GPU is in the available private storage per compute unit. The ATI GPU has significantly higher register file sizes as compared to the NVIDIA GPU, as shown in Table 5.1. This indicates that the ideal design of a kernel for best performance on the ATI GPU should include larger kernels, or a coarse granularity of task decomposition. More importantly, kernels of coarser granularity would expose sufficient independent instructions for the compiler to identify and extract ILP, thus generating efficient VLIW code. The vector nature of the register file and register read/write ports on an ATI GPU also indicates the importance of data vectorization of OpenCL code for best performance. The NVIDIA GPU on the other hand has a significantly lower register file size, but needs to accomodate as many work-items per compute unit as possible to hide global memory/instruction latency; therefore, a more finely decomposed kernel would deliver the best performance.

The design of the local memory banks also differs significantly across both architectures. As shown in Table 5.1, the NVIDIA GPU local memory implementation services a 4-byte word to each scalar core per cycle if bank conflicts are avoided. This means that for each core, the design supports a local memory bandwidth that is roughly a third of the operand consumption rate for three-operand instructions. The AMD GPU, on the other hand, implements a local data store capable of supplying two 4-byte words to each core of the compute unit per cycle under conflict-free access patterns. However, since each core of the compute unit on an AMD GPU is a 5-way vector (unlike the scalar units of the NVIDIA GPU), this means that the architecture can supply operands only at the rate of 2/15 of the core's operand consumption rate. Thus, kernels with heavy use of local memory

may be disadvantaged on the AMD GPU, and need to be modified to make much heavier use of the larger register file instead. Conversely, the NVIDIA GPU architecture encourages more prudent use of its smaller register file. Another significant difference affecting performance optimizations includes the best form of indexing for conflict-free accesses. The NVIDIA GPU local store architecture consists of 16 banks supplying 16 4-byte words at a time to 16 execution units. This indicates that the best pattern of access is for work-items to make consecutive memory accesses of type `float`. On the other hand, the ATI GPU has 32 banks supplying 16 execution units with two 4-byte words each. This implies that to avoid bank conflicts, each work-item make `float2` accesses. This difference in the best addressing for conflict-free access patterns is an important concern for developers writing portable OpenCL kernels.

The global memory systems of these devices also differ slightly in their response to short-vector loads and stores. Based on the documentation from AMD, the effective bandwidth of coalesced `float4` accesses within a task can be as much as 128 GB/s, which is almost 30% higher than the effective bandwidth achieved by perfectly coalesced `float1` (95 GB/s) accesses. The NVIDIA GPU, on the other hand, demonstrates almost equal bandwidth for coalesced `float4` and `float1` accesses. Thus, memory vectorization is an important optimization for good performance on the ATI GPU, whereas it is a dispensable one for the NVIDIA devices.

CHAPTER 6

PERFORMANCE RESULTS OF OPTIMIZATIONS

We study four GPU computing benchmarks, chosen from application fields including bio-molecular physics, BLAS and fluid dynamics. All benchmarks are translated from CUDA using the CUDAtoOpenCL translator. The benchmarks chosen are a careful blend of scientific computing applications limited by floating point unit throughput (direct summation Coulombic potential and the MRI-Q computation) and memory-intensive applications limited by arithmetic intensity and off-chip memory bandwidth (7-point stencil and Dense Matrix Multiply). We apply several optimizations targeting many-core GPU architectures, and evaluate their effect on performance on the NVIDIA GTX 280 and ATI Radeon 5870 architectures, with the goal of identifying the major differentiating factors for high performance. The benchmarks were executed on the NVIDIA GPU Computing SDK v3.1 and the ATI Stream SDK v2.2, both environments configured on a Linux-64 platform. Some performance profiler counters on the Radeon 5870 were generated using the ATI Stream Profiler on a Windows-7 platform as well.

6.1 Improving Performance of Compute-Bound Kernels on GPUs

The direct summation Coulombic potential algorithm [16] and the MRI-Q computation [4] share similar features; both algorithms calculate the output dataset (either as a regular lattice of points or as a linear array) by summing the contributions from all elements of a constant input dataset. Each parallel work-item loops over all input data and maintains a running sum for single/multiple output elements.

Figures 6.1 and 6.2 show the effects of many specific optimization

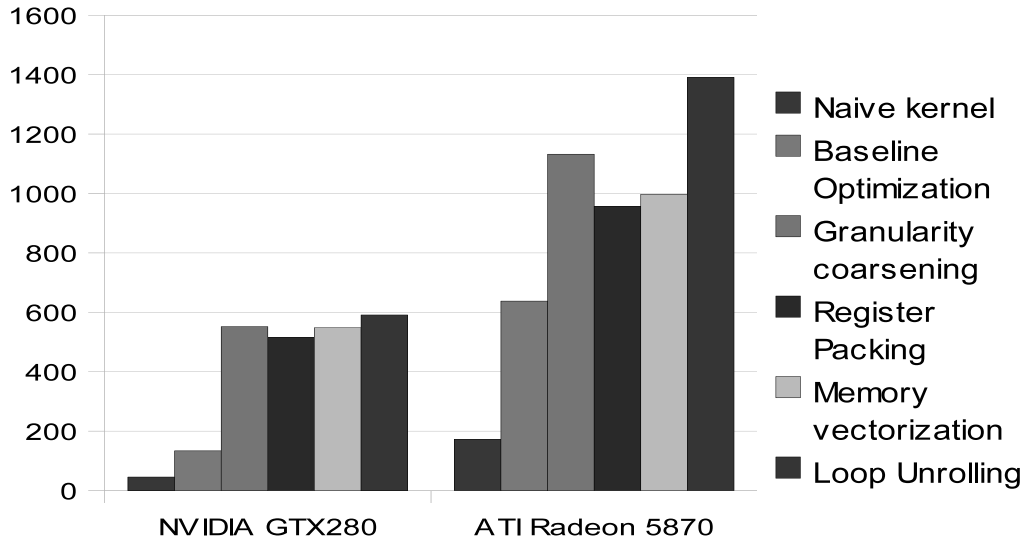

Figure 6.1: Performance Optimizations on the CP kernel on NVIDIA GTX280 and ATI Radeon HD5870. The best performance is achieved at a granularity of 8 for GTX280 and 16 for HD5870.

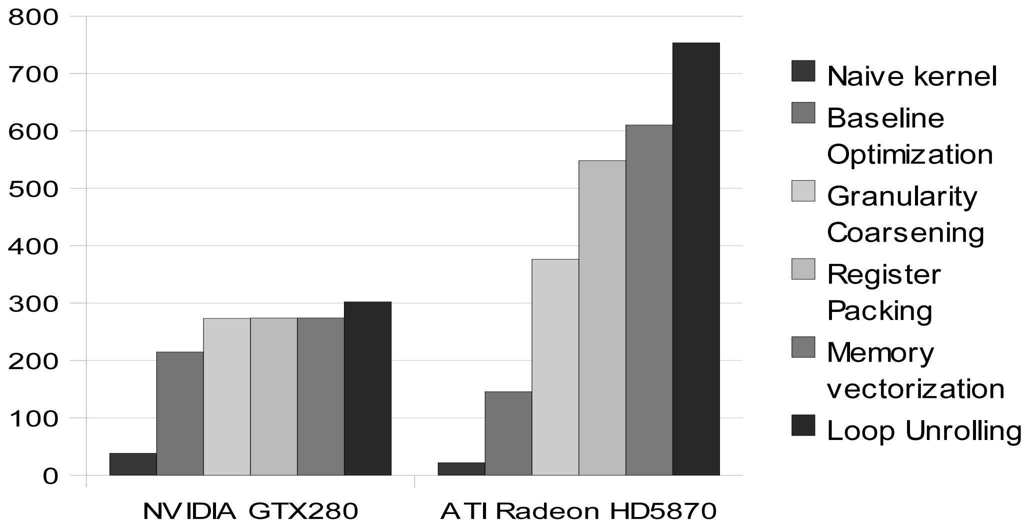

Figure 6.2: Performance Optimizations on the MRI kernel on NVIDIA GTX280 and ATI Radeon HD5870. The best performance is achieved at a granularity of 4 for GTX280 and 8 for HD5870.

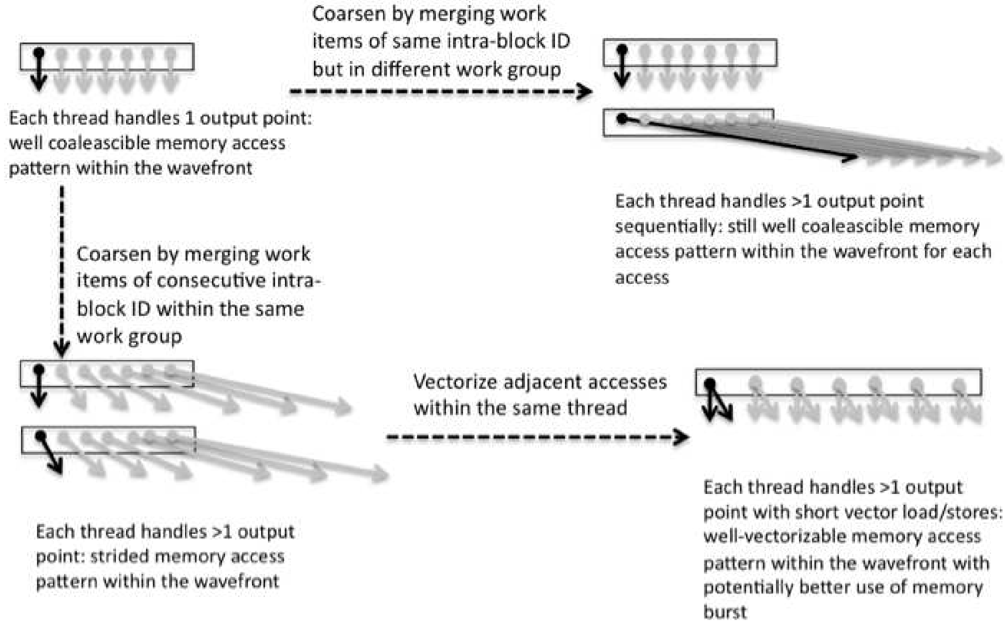

Figure 6.3: Different thread coarsening strategies

techniques on the CP and MRI-Q kernel on both the G280 and Radeon 5870 architectures. We first consider what we call the baseline optimization set for GPU compute programming, which includes well-understood techniques to improve use of memory bandwidth and utilize special function units [7], like the use of constant and/or local memory, native functions and tuning work-group sizes. Note that performance improves comparably on both devices; CP improves by a factor of 3.69 on the Radeon HD 5870 and by a factor of 2.91 on the GTX 280, while the MRI kernel improves by a factor of 6.71 on the HD 5870 and by a factor of 5.63 on the GTX 280.

We now apply granularity coarsening on the kernels, i.e. computing multiple output points per work-item, and observe the effects on both the architectures. Thread coarsening can be implemented by merging work-items within a work-group, or by merging multiple work-groups as shown in Figure 6.3. Apart from improving memory and data reuse, thread coarsening also increases the availability of independent instructions for VLIW scheduling. When consecutive work-items in a work-group are merged, this could cause a strided memory access pattern within an issue-group, thus resulting in poor memory coalescing. Converting strided accesses into short vector loads and stores, however, may improve performance. On the other

hand, when multiple work-groups are merged (by merging work-items with the same local ID from each work-group), the memory coalescing pattern would be conserved.

On the Nvidia GTX280, assuming memory coalescing requirements are satisfied, thread coarsening improves memory reuse by increasing the number of computations per memory access and also enables reuse of intermediate compute data. Hence, we see that the performance of CP increases almost by a factor of 4.15 for a granularity of 8, and further granularity coarsening causes only negligible performance gains. The MRI kernel, however, gives only a modest 27% improvement for a granularity of 4, due to register pressure balancing out any reuse benefits. On the Radeon 5870, however, granularity coarsening has significantly more impact. The performance improvement for CP is 46% and 56% , for granularities of 4 and 8, respectively. More significantly, for the MRI kernel, the performance improves by a factor of 2.59 and 3.1 for granularities of 4 and 8. This is reflected in the ALU efficiency metric, which increases from 33.33% to 62% and then 71%. Additionally, owing to the significantly larger register file of the Radeon 5870, performance penalty due to register pressure usually occurs at higher granularities than the NVIDIA kernels in almost all applications. This is reflected in Figure 6.1, which shows the best thread-coarsened performance of CP on NVIDIA GTX280, for a granularity of 8 and on Radeon HD5870 for a granularity of 16. Similarly, in Figure 6.2, MRI gives best performance on the GTX280 at a granularity of 4, and on the HD5870 at a granularity of 8. This reflects the fundamental problem of performance portability, in that the degree of optimization applied to a kernel differs fundamentally across target devices, and aggressive compiler tools are required to manage and automate these transformations.

Our next optimization is register packing, which packs data into `float4` registers to potentially improve performance by generating SIMD-ized instructions, for the VLIW compiler on the HD5870. CP suffers from an increased instruction count of almost 22.5%, not sufficient to offset an improvement in ALU efficiency of 4%. However, the MRI kernel shows clear gains of 45.2% with register packing. The NVIDIA kernels, on the other hand show a consistent (although slight) reduction in performance on implementing register packing, as this introduces additional `MOV` instructions for data packing pulling down the functional unit throughput. This is

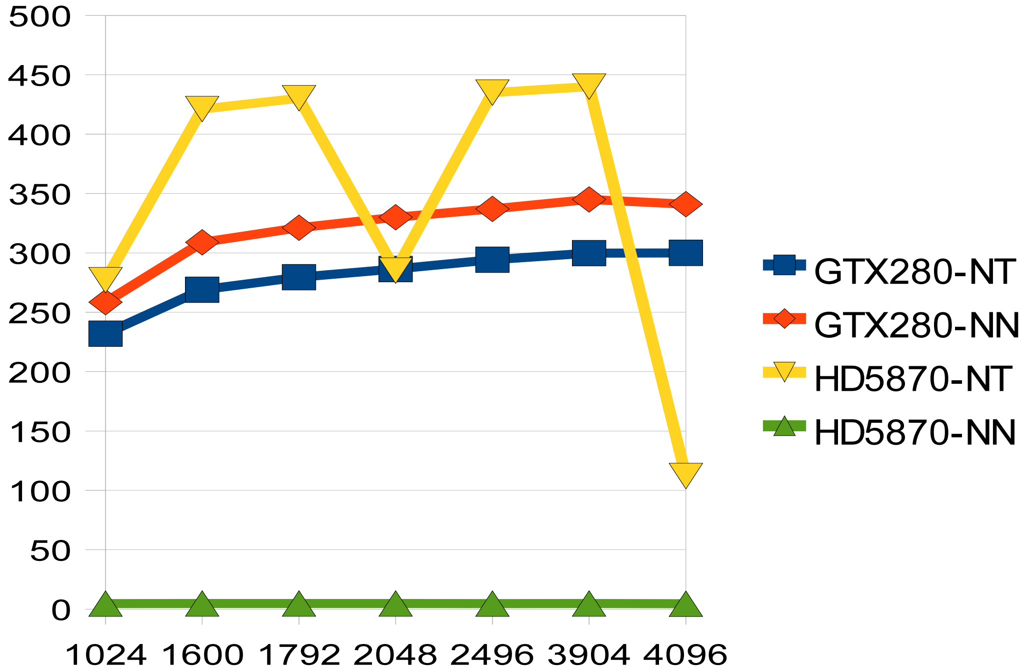

Figure 6.4: Performance of SGEMM on NVIDIA GTX280 and ATI Radeon HD5870 for different sizes

an important example of a transformation that has differing performance impacts on the HD5870 and GTX280.

The next most important optimization is on improving sustained global memory bandwidth through memory access vectorization. Note that for the ATI HD5870, the sustained bandwidth reaches almost 128 GB/s when accessing `float4` data types, as compared to 95 GB/s for `float` datatypes. In comparison, for the same data transfer on NVIDIA GTX280, the bandwidth is 98 GB/s, 101 GB/s and 79 GB/s using `float`, `float2` and `float4` data types. This is demonstrated in the MRI kernel which achieves about 11% performance gains on the HD5870, as compared to less than 1% on the GTX280. The CP kernel, on the other hand, does not gain much benefit as constant caches are already being used. Another classical optimization technique that turns out to be very important for the Radeon HD5870 is loop unrolling, which greatly benefits VLIW packing efficiency and reducing control instructions. It leads to particularly large gains as demonstrated in MRI (24% on HD5870 and 10.2% on GTX280), and CP(39.5% on HD5870 and 7.8% on GTX280).

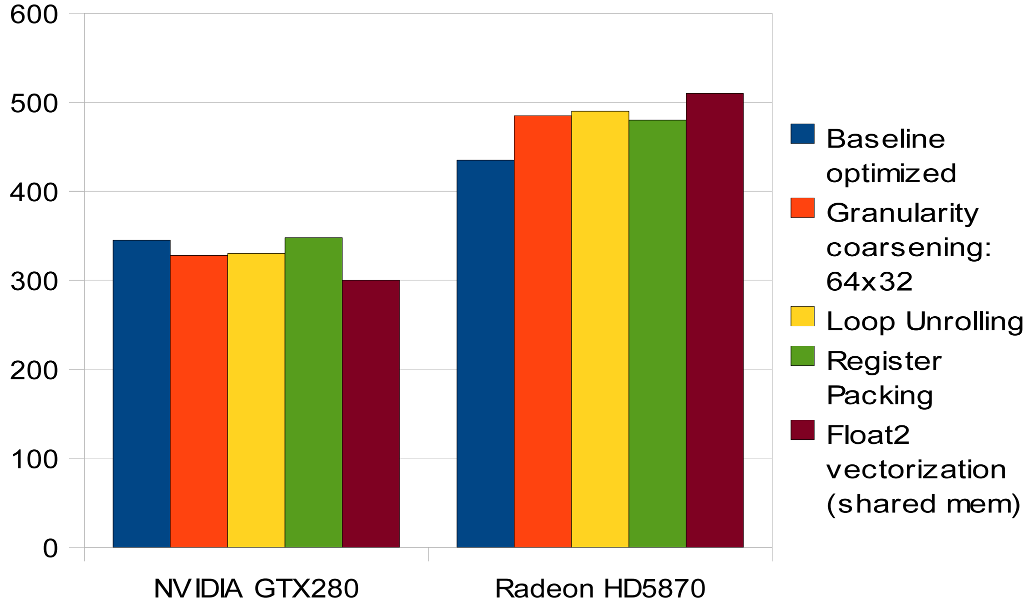

Figure 6.5: Performance optimization of SGEMM on NVIDIA GTX280 and ATI Radeon HD5870

6.2 Improving Performance of Memory-Bound Kernels on GPUs

We now analyze the performance of the SGEMM kernel, specifically focusing on the operation $C = \alpha * op(A) * op(B) + \beta * C$, with the 'N, N' configuration denoting $op(A) = A$; $op(B) = B$ and the 'N, T' configuration denoting $op(A) = A$; $op(B) = B^T$. To evaluate the performance portability of SGEMM on the two devices under study, we use Volkov and Demmel's implementation [17] and analyze the performance on both the devices. This implementation of SGEMM utilizes registers to store elements of A, and local memory to store the B matrix with each work-group (consisting of 64 work-items) computing a 64 x 16 tile of the output matrix. This kernel is considered the baseline optimized kernel for the matrix multiply kernel.

As shown in Figure 6.4, on the GTX280, and the NVIDIA GPU Computing SDK v3.1, a direct port from CUDA into OpenCL achieves about 345 GFLOPs for the 'N, T' configuration and 300 GFLOPs for the 'N, N' configuration for a matrix of size 3904 x 3904. On the HD5870, however, while the 'N, T' configuration achieves about 430 GFLOPs, the 'N, N' configuration achieves merely 4.7 GFLOPs. This is likely due to poor

utilization of the AMD GPU's larger burst size when accessing matrix B and channel/bank conflicts due to workgroups accessing memory at strides of matrix width. Only 64 bytes of every 256 byte burst are utilized [14], thus resulting in 4 times as much DRAM traffic as consumed data, saturating channel capacity. We also notice sharp drops in performance of the 'N, T' kernel at matrix sizes which are powers of 2, 1024, 2048 and 4096, because the lack of any interleaving mechanism to randomize requests to channels and banks within the HD5870 memory system exposes it to complete channel conflicts and, therefore, to drastic drops in performance.

Modifying the 'N, T' kernel to improve register packing and VLIW efficiency, as shown in Figure 6.5, boosts performance on the HD 5870 to 480.2 GLOPs, an improvement of 11.6% due to improved ALU packing efficiency alone (from 42.2% to 51.4%). As expected, on the NVIDIA device though, this leads to less than 1% improvement in performance. Vectorization of local memory accesses to `float2` datatypes improves performance to 510.3 GFLOPs on the HD5870. Given the 32 banks of local memory performing reads/writes at the granularity of a quarter-issue-group, `float2` accesses perform best on local memory. `float` accesses are subject to VLIW packing constraints, often utilizing only half of the bandwidth, while `float4` accesses reliably cause bank conflicts. The GTX280, on the other hand, experiences a reduction in performance of more than 8% with `float2` vectorization due to two-way bank conflicts on local memory accesses. Further granularity coarsening to 64 x 32 tiles does not give major performance gains on either device because the kernel experiences zero data reuse, and memory reuse benefits are offset by increased register pressure. Also, increased instruction fetches leads to higher effective latency in terms of the stalls experienced by the fetch unit. Thus, it can be concluded that while a small subset of the major data reuse transformations for SGEMM [17] are performance portable and applicable on both the GTX280 and the HD5870, the highest performance numbers can be achieved only with finer architecture-aware tuning, possibly even causing detrimental performance on the other architecture.

The next application of interest is a 3-D finite difference computation of order 2 [18], also called the 7-point stencil. Optimization of similar heavily memory-bound kernels for the GPU architecture is often extremely challenging, and depends significantly on the specific memory coalescing

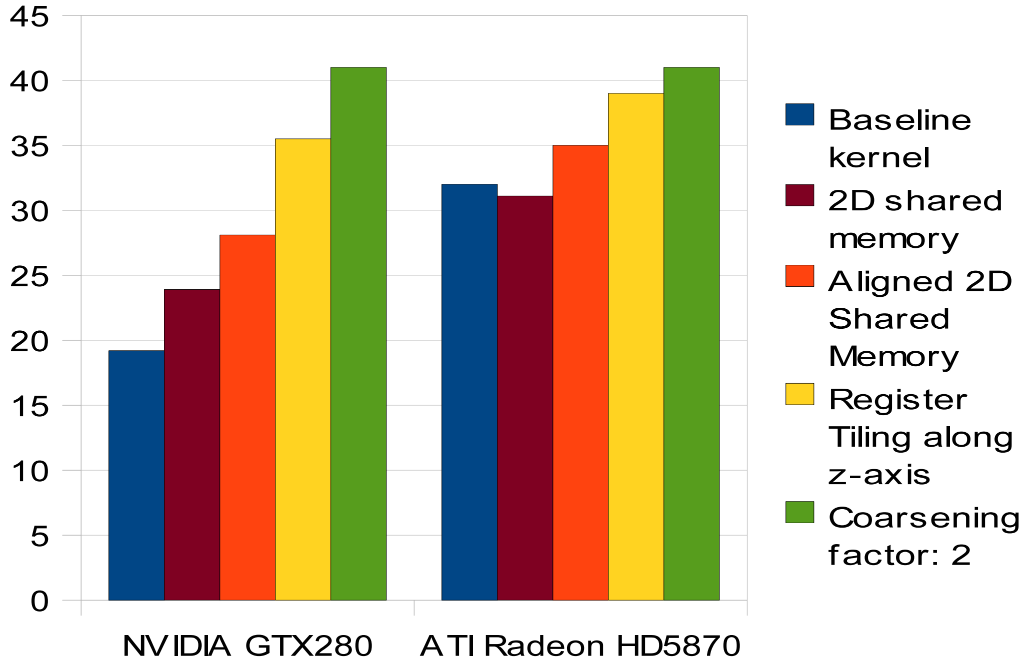

Figure 6.6: Performance optimization of 7-point stencil on NVIDIA GTX280 and ATI Radeon HD5870

hardware, channel and bank interleaving schemes, and behavior of unaligned memory access patterns within each device. Figure 6.6 shows the optimizations applied to the stencil kernel on both the GTX280 and HD5870. Techniques to enable data reuse across work-items in a workgroup like local memory tiling are studied. This is implemented in two ways: by loading unaligned blocks of memory with the boundary elements included or by loading an aligned 2-D block into local memory, with tile boundary elements being loaded directly into registers. Further improvements like register tiling for element reuse in the third dimension yield more performance, of up to 36 GFLOPs which is very close or better than the best performance reported by previous works [10], [18]. Additionally, we apply thread coarsening on the stencil kernel, with each work-item computing two output elements, which reduces local memory read bandwidth requirements by 20%, thus boosting performance to 41 GFLOPs. The same optimizations exhibit moderate performance gains on the ATI Radeon HD5870. The peak performance achieved is about 42 GFLOPs, with non-power-of-2-size grids to dodge the severe penalties of channel and bank conflicts. This can be partially explained by the poor burst utilization on the HD5870, especially on the 2-D aligned

local memory blocking. For a 16 x 16 tile of elements, each workgroup accesses 18 consecutive words. However, due to the ordering of these memory accesses, they are grouped into 3 bursts of 64 words each, leading to a burst utilization of 18/192, less than 10% efficiency. The NVIDIA device on the other hand experiences much better bandwidth utilization, as the burst size is only 16 words; with 3 bursts per row of accesses, this leads to a burst utilization of 18/48, about 37% efficiency. However, it is unclear why unaligned local memory blocking or register tiling along the z-dimension do not give more substantial improvement in performance. More details about the memory subsystem than is known at this point need to be revealed before targeted optimizations for the HD5870 can be applied to improve performance of this important application.

CHAPTER 7

CONCLUSION

With the release of OpenCL, developers hope to utilize the wide array of codebases in CUDA to generate readily available OpenCL code. To this end, the CUDAtoOpenCL source-to-source tool translates CUDA code to OpenCL, while being able to adjust scope position and handle inter-procedural transformations, unlike other regular expression-based substitution methods. This thesis describes the implementation details and working of the CUDAtoOpenCL tool within a source-to-source translation framework. This work also illustrates the lack of capability within OpenCL to provide performance portability among devices. Due to the fundamental architectural differences between devices, aggressive optimizations are necessary in an OpenCL implementation to provide performance portability from a single source code instance. This work presents some of those essential transformations, such as thread coarsening, register packing and vectorization, and demonstrates that the extent to which those optimizations are applied have drastic performance effects on performance, with the optimal configuration for different devices diverging significantly. With the necessary infrastructure in place, performance portability using OpenCL will be much more realistic and satisfactorily available to developers.

REFERENCES

[1] NVIDIA, "NVIDIA CUDA," 2010. [Online]. Available: http://www.nvidia.com/cuda

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[3] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors." *Journal of Computational Chemistry*, vol. 28, no. 16, September 2007.

[4] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2008, pp. 261–272.

[5] Khronos OpenCL Working Group, "The OpenCL Specification," May 2009. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[6] M. Harvey and G. D. Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093 – 1099, 2011.

[7] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, February 2008.

[8] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-aware optimization targeting multithreaded stream computing," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 62–70.

[9] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Proceedings*

of the International Parallel Distributed Processing Symposium, April 2010, pp. 1–12.

[10] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[11] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 3.2*, NVIDIA Corporation, August 2010.

[12] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, pp. 36–42, 2009.

[13] S. Lee, T. Johnson, and R. Eigenmann, "Cetus - An extensible compiler infrastructure for source-to-source transformation," in *16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2003)*, 2003. [Online]. Available: citeseer.ist.psu.edu/article/lee03cetus.html

[14] Advanced Micro Devices Inc., *ATI Stream Computing OpenCL Programming Guide*, Advanced Micro Devices Inc., June 2010.

[15] NVIDIA, *OpenCL Programming Guide for the CUDA Architecture: Version 3.1*, NVIDIA Corporation, May 2010.

[16] C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W.-M. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *Proceedings of the ACM International Conference on Computing Frontiers*, May 2008, pp. 273–282.

[17] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1–11.

[18] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 79–84.

[19] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, Tech. Rep. GIT-CERCS-09-01, 2009.

[20] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th International Conference on Machine Learning*, June 2008, pp. 104–111.

[21] Intel, "Threading Building Blocks," 2007. [Online]. Available: http://threadingbuildingblocks.org

[22] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for Brook streaming applications on multiprocessors," in *Proceedings of the 4th International Symposium on Code Generation and Optimization*, March 2006, pp. 196–207.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March/April 2008.

[24] OpenMP Architecture Review Board, "OpenMP application program interface," May 2005.

[25] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-M. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[26] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-8-474

[27] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 434–444, 2005.

[28] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, "Is the schedule clause really necessary in OpenMP?" in *Proceedings of the International Workshop on OpenMP Applications and Tools*, June 2003, pp. 147–159.

[29] C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1, pp. 3–25, September 2000.

[30] W.-Y. Chen, "Building a source-to-source UPC-to-C translator," M.S. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 2004.

[31] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, “MCUDA: An effective implementation of CUDA kernels for multi-core CPUs,” in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, July 2008, pp. 16–30.

[32] A. Aiken and D. Gay, “Barrier inference,” in *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, 1998, pp. 342–354.

[33] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, “Parallel languages and compilers: Perspective from the titanium experience,” *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 266–290, 2007.

[34] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[35] S. Lee, S.-J. Min, and R. Eigenmann, “OpenMP to GPGPU: A compiler framework for automatic translation and optimization,” in *Proceedings of 14th ACM Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 101–110.

[36] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, “Chunking parallel loops in the presence of synchronization,” in *ICS ’09: Proceedings of the 23rd International Conference on Supercomputing*, 2009, pp. 181–192.

[37] D. Nuzman and A. Zaks, “Outer-loop vectorization: Revisited for short SIMD architectures,” in *PACT ’08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 2–11.

[38] S. Larsen and S. Amarasinghe, “Exploiting superword level parallelism with multimedia instruction sets,” *SIGPLAN Not.*, vol. 35, no. 5, pp. 145–156, 2000.

[39] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, “CUDA-Lite: Reducing GPU Programming Complexity,” in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, 2008, pp. 1–15.

[40] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU compiler for memory optimization and parallelism management,” in *PLDI*, 2010, pp. 86–97.

[41] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008, pp. 225–234.

[42] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 513–522.

[43] G. Wang, X. Yang, Y. Zhang, T. Tang, and X. Fang, "Program optimization of stencil based application on the GPU-accelerated system," *International Symposium on Parallel and Distributed Processing with Applications*, pp. 219–225, 2009.