NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: An Environment to Support GPU and Multicore Programming for Rapid, High Performance, Application Deployment

Author: James Laurence Brock

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Doctor of Philosophy

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Gunar Schirner	Date
Thesis Reader: Dr. Sanjeev Mohindra	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Sara Wadia-Fawcetti	Date

An Environment to Support GPU and Multicore Programming for Rapid, High Performance, Application Deployment

A Dissertation Presented

by

James Laurence Brock

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

August 13, 2012

Abstract

Homogeneous multicore processors, heterogeneous multicore processors, high performance accelerators, and other heterogeneous architectures have significant computing potential over traditional single core processors. Computer systems comprised of these specialized processing elements are increasingly common. Due to the increased complexity of these architectures, programming them has become increasingly complex and error prone. Each of these architectures have different memory systems, programming languages and development environments. This has driven the need for portable programming APIs and tools that allow developers to easily exploit all of the computational power of these platforms and effortlessly move their programs between different computing systems. To deal with these challenges MIT Lincoln Laboratory developed the Parallel Vector Tile Optimizing Library (PVTOL) to simplify the task of portable programming for complex systems. The PVTOL Tasks and Conduits framework provides a set of high-level programming constructs for writing high performance code that is portable across a range of traditional and heterogeneous architectures. This research extends PVTOL to include support for Graphics Processing Units (GPUs) and heterogeneous computing architectures using both the NVIDIA Compute Unified Device Architecture (CUDA) and Open Compute Language (OpenCL), while maintaining simplicity of programming and portability. We have demonstrated the utility of this framework by porting both a quantum Monte Carlo simulation and 3D cone beam image reconstruction application to different

systems consisting of various heterogeneous architectures. These applications have been ported from single CPU/GPU systems up to heterogeneous cluster architectures with as many as 24 nodes containing GPUs, showing significant speed up and scalability with minimal developer effort. Using this framework, we have achieved total application run time speed ups of quantum Monte Carlo simulations of 115x on 24 distributed GPU nodes and speed ups of 3D cone beam image reconstruction of 315x on 16 distributed GPU nodes compared to multi-threaded C code.

Acknowledgements

I would first like to thank my advisor Prof. Miriam Leeser. This work would not be possible without her incredible support and guidance. I would like to thank my other committee members Prof. Gunar Schirner and Dr. Sanjeev Mohindra for their direction and expertise.

This dissertation would not have been possible without MIT Lincoln Laboratory, which initiated the project that formed the basis for, and provided me with the software to continue my research. I would like to specifically thank Hahn Kim, Sanjeev Mohindra, Jeremy Kepner, Edward Rutledge, and Robert Bond for their help with along the way. I would like to acknowledge Prof. Mark Niedre for dedicating his knowledge and time to helping me develop a biomedical imaging application that would yield meaningful results. I gratefully acknowledge the support of the National Nanotechnology Infrastructure Network (NNIN) Computation Project and

use of the Orgoglio GPU cluster at Harvard University. Also, the support of the Northeastern University Computer Architecture Research (NUCAR) group for the use of the Medusa cluster.

A very special thanks goes to my wife Brigette, my family, and my friends for their unending support and constant encouragement. Thanks also goes to all of the professors and teachers I have had that have helped me get to this point.

Contents

Chapter 1

Introduction

Homogeneous multicore processors (Intel Xeon, AMD Athlon, ARM Cortex), heterogeneous multicore processors(Intel Sandy Bridge, AMD Fusion APU), dedicated high-performance accelerators (GPUs, DSPs), and other system-on-chip (SoC) architectures (NVIDIA Tegra, Texas Instruments OMAP) have all demonstrated significantly increased processing performance over traditional single core processors in recent years. However, with the increased complexity of these architectures, programming for these devices has also become increasingly complex and error prone. This problem is only exacerbated when having to develop applications for heterogeneous computer systems containing one or more of these processing elements, which may have different memory systems, programming languages and development environments. Additionally, each of these architectures is more adept at different types of computation and different workloads. New generations of different processors are released at different rates. Finding the best combination of programming model, application organization, and computing architecture has become a moving target.

With ever changing platforms and programming paradigms, being able to write code that is portable across various systems has become important in the development of high-performance computing applications.

When developing a high performance application, the top priority is the core algorithm's performance and consistency, followed by secondary concerns like device initialization, memory management, and data synchronization. There has been much work in the area of converting code meant for one platform into code meant for another, and automatically exploiting parallelism in serial code, but they all fail to achieve the performance of algorithms programmed for a dedicated platform and may, in fact, make algorithmic decisions that are counter-productive. Thus, these solutions require the developer to intervene and adjust the code in order to achieve their desired results. The value of a program developer is in their ability to make complex, high-level decisions to develop these core algorithm functions better than automated methods. Alleviating the burden of the lower priority, platform-specific challenges, like memory management, allows the developer to concentrate their effort in the more important aspects of the algorithm development process. This was the primary motivation behind MIT Lincoln Laboratory developing the Parallel Vector Tile Optimizing Library (PVTOL) as a means of writing high performance applications that are portable across a large number of multicore general purpose computing platforms. Extending PVTOL to support graphics processing units using the Compute Unified Devices Architecture (CUDA) and Open Compute Language (OpenCL)

supported architectures enables a programmer to develop a portable application capable of executing on many modern heterogeneous computing platforms with minimal effort.

In this research, we have extended the functionality of the PVTOL tasks and conduits framework to include support for heterogeneous architectures supported by CUDA and OpenCL. The contributions of this work are:

CUDA Support

This contribution consisted of extending the PVTOL tasks and conduits framework to support CUDA-capable graphics processing units. This required support for multiple asynchronous kernel launches, and heterogeneous conduit support for data transfers to and from both global and constant memory on one or more CUDA devices. This also necessitated a new means of mapping tasks to heterogeneous architectures that fits in to the existing tasks and conduits framework structure.

OpenCL Support

We have further extended the tasks and conduits framework to support OpenCL supported architectures, which includes the OpenCL equivalent of all CUDA GPU support. This means that we have developed constructs for encapsulating OpenCL platforms, devices, and command queues while retaining all the existing PVTOL tasks and conduits functionality. To the best of the author's knowledge, this is the first such OpenCL framework.

CUDA/OpenCL Abstraction and Interoperability

We have added support for multiple parallel programming languages and developed an example application for the tasks and conduits framework that demonstrates interoperability between NVIDIA CUDA devices and OpenCL (NVIDIA, ATI, Intel, etc.) device. Our extensions include other features that have not been demonstrated in previous work, including support for constant memory, multi-point conduits, and maintaining a consistent and accessible C/C++ based programming environment. This is also novel. CUDA and OpenCL yield better performance on different architectures, and being able to use them together enables a developer to use the best choice for each architecture rather than having to choose one for the implementation of an entire system.

Heterogeneous Utility Functions

In order to add support for heterogeneous programming models while, an interface between the different programming model APIs and the platform-independent tasks and conduits framework is needed. This keeps app platform-specific code and API calls separate from the higher level behavior of the tasks and conduits. We have developed a set of heterogeneous utility functions that create an API interface for the tasks and conduits code to interface to the appropriate programming model based on the application mapping. This also has benefit of establishing an API for which other programming models can be easily added in the future.

Heterogeneous System Portability

This work also demonstrates application execution using PVTOL tasks and conduits

on a variety of architecturally diverse systems. Previous work has demonstrated platform portability by executing the same program on different accelerators connected to the same host architecture. An example of this is moving an application from using CPU host and FPGA accelerator to using the same CPU host and a GPU accelerator[37]. We have ported an application between systems without any of the same processing components. This demonstrates an application's ability to be ported between systems with completely different heterogeneous configurations.

Use of PVTOL to develop an accelerated Fluorescence Mediated Tomography (FMT) application

In order to exercise as many of the features and possibilities of the framework as possible, two GPU-accelerated applications have been developed. First, we adapted a well known algorithm for Monte Carlo simulations of photons through biological tissue from an optimized single-threaded C application[41] to a massively parallel CUDA application. This algorithm is additionally modified for real-life application scenarios using time resolved tracking of photons[28], utilizing the tasks and conduits framework to achieve the best parallelization and execution speed up possible.

Use of PVTOL to develop an accelerated and portable 3D Cone Beam Computed Tomography (3D CBCT) application

The second application is 3D Cone Beam Computed Tomography (3D CBCT) reconstruction using the popular Feldkamp-Davis-Kress algorithm[14]. This application has been developed in OpenCL and ported among numerous architectures to

determine the conditions that achieve the best application speed up.

Our framework for heterogeneous application development and portability is unique in its comprehensive support for CUDA as well as support for OpenCL, its support for interoperability between programming paradigms (CUDA, OpenCL), and its complete portability across different heterogeneous systems consisting of a wide variety of architectures.

Chapter 2

Background

In this chapter, we present the background of the original Parallel Vector Tile Optimizing Library (PVTOL) framework, other heterogeneous programming models and constructs, the medical imaging algorithms we use as example applications within our extended framework, and other work related to this research. We first discuss the various heterogeneous and parallel programming models and their prominent features, outlining their use cases and limitations. Then we will introduce the PVTOL framework's primary software abstractions (*Task* and *Conduit*), PVTOL application construction and execution, and the framework's supported computing architectures. This will describe the initial state of the framework prior to the implementation of the research presented in this dissertation. This will be followed by descriptions of the example applications used to test the various features of the extended framework and their basic functionality. The applications have been chosen for their relevance and varying computational problems. Lastly, related work is discussed in relation to the capabilities of the extended tasks and conduits frameworks.

2.1 Heterogeneous and Parallel Programming

The proliferation of new and varied computing architectures in commodity computer systems has driven the need for programming platforms, languages, and tools that support heterogeneous systems. These specialized processing architectures, such as graphics processing units (GPUs), digital signal processors (DSPs), field programmable gate arrays (FPGAs) and others require programming paradigms that extend beyond those designed for general purpose processors (CPUs). A number of programming languages and language extensions like NVIDIA's Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) standard have been developed to address this need, but lack sufficient tools and higher-level functionality for exploiting task parallelism and distributed memory systems. This section presents an overview of currently available parallel programming paradigms and their capabilities for addressing the current issues in heterogeneous programming.

2.1.1 Graphics Processing Unit (GPU)

Advancements in graphics processing unit (GPU) hardware have been led by the commodity games market, requiring an ever increasing amount of computing power to handle ever more complex 3D graphics. The computing power of dedicated GPUs, whose peak performance is typically measured by single precision floating point operations per second, have gone beyond that of state-of-the-art CPUs. Additionally, the low cost of commodity graphics hardware has made the GPU even more appeal-

ing for general-purpose computation[35]. A general comparison of a multicore CPU architecture versus a GPGPU architecture can be found in Figure 2.1. Note that the GPU uses much more area for computation and less for control and memory hierarchy.

Figure 2.1: Comparison of CPU and GPU architectures.[31]

The current generation of general purpose graphics processing units (GPGPU) are extremely flexible and powerful processors that include highly programmable single instruction multiple data (SIMD) computational cores. However, there are some limitations to programming and performance on GPU architectures. The memory hierarchies of graphics architectures contain a number of memory spaces with different features (read-only, read-write, thread local, global, etc.). This makes programming more difficult than architectures with a single unified memory model, such as C/C++. GPUs also require re-writing functions for massively parallel execution that maps to the small processing cores of the GPU architecture, and re-thinking the distribution of processing among elements within an application.

2.1.2 NVIDIA Compute Unified Device Architecture (CUDA)

The NVIDIA Compute Unified Device Architecture (CUDA) is both a family of hardware designs and a set of extensions to the C/C++ programming languages to interface to an NVIDIA graphics processing unit (GPU) enabling the execution of general purpose computing programs on a GPU device (GPGPU) as a coprocessor to a host CPU[31]. The CUDA API provides functions to configure devices, allocate and copy data between host and device, and executing Single Processor Multiple Data (SPMD) functions, called kernels, on the GPU device, as well as other functionality. The CUDA-supported architectures provide a thread hierarchy for massively parallel execution of kernel code, as well as a memory hierarchy for providing threads access to necessary data. The thread hierarchy is designed to allow for potentially tens of thousands of threads to be launched concurrently on a single device. This level of concurrency can provide significant increases in performance over traditional serial code. In particular, GPGPU programming using CUDA has been shown to yield significant performance improvements in many scientific computing applications[23].

The thread execution and memory hierarchy of the NVIDIA CUDA programming model can be seen in Figure 2.2. The execution threads are grouped into one or two dimensional thread blocks, and those *thread blocks* are further organized into one or two dimensional groups of thread blocks, called *grids*[31]. This thread execution organization corresponds to the various types of memories that are shared among thread blocks and block grids. The CUDA memory hierarchy provides threads three

types of read/write memory spaces. Each thread has access to its own local registers, each thread block has access to *shared* memory, and all threads within all thread blocks and grids have access to *global* memory. There are also two types of read-only memory spaces, *constant* and *texture* memory, both of which act as high-speed cached memories accessible by all threads. Each of the various GPU memory types have different access times and are best used for different types of computation. Properly managing the memory accesses and data flow through one's algorithm is crucial to achieving good performance in a CUDA application.

CUDA provides both synchronous and asynchronous API functions for launching GPU kernels and performing memory operations, such as allocation, deallocation, and copying data between the CPU host and the GPU device. Synchronous functions execute just as a C/C++ function would, returning to the calling function only after it has completed its execution. Asynchronous functions return to the calling function immediately, launching their operations concurrently. In order to query and synchronize code with asynchronous functions, the CUDA programming model provides *events* that can monitor asynchronous function execution status.

The functionality in the CUDA API is very closely tied to the proprietary NVIDIA GPU thread and memory hardware architectures. Execution features such as floating point support, floating point standard compliance, atomic functions, concurrent data copy and kernel execution, and others may vary depending on which generation of hardware is being used. All of these differences between the CUDA programming

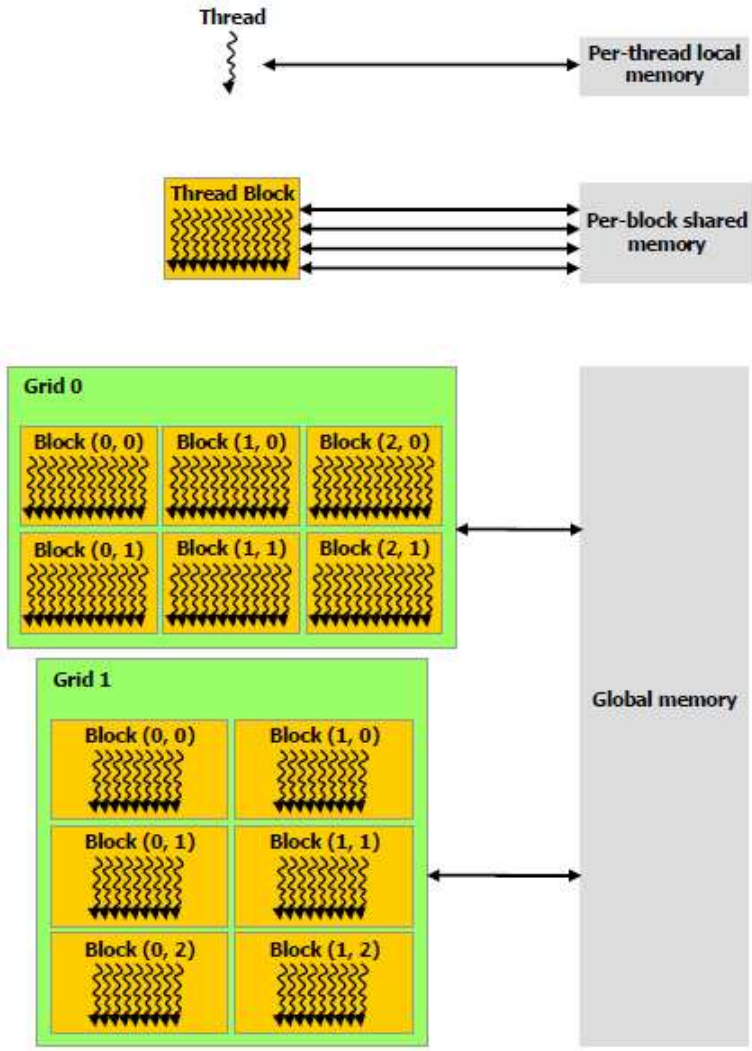

Figure 2.2: NVIDIA CUDA thread execution model and memory hierarchy[31]

and memory models and the traditional unified shared memory model of general purpose processors makes developing applications to GPUs in a way that is portable and adaptable difficult. This is the primary reason for needing an intuitive and efficient abstraction of the interface between programmer and NVIDIA CUDA API that works with other heterogeneous computing architectures well.

2.1.3 Open Computing Language (OpenCL)

The Open Computing Language (OpenCL) is a standard programming model for parallel and heterogeneous computation that is meant to be portable across different heterogeneous architectures and devices for a wide range of applications[10]. OpenCL is a C-based language that models execution and memory management in a way that is similar to the NVIDIA CUDA architecture but is architecture independent. The OpenCL architecture model consists of a *Platform model*, *Memory model*, *Execution model* and *Programming model*. The Platform model, shown in Figure 2.3, consists of a host device (typically a CPU) connected to one or more OpenCL devices[19]. An OpenCL program is executed on some computational device (CPU, GPU, DSP, or other processor architecture), and each device contains one or more processor cores which is then comprised of one or more processing elements capable of executing single instruction multiple data (SIMD) code.

OpenCL's programming interface includes functionality for enumerating available platforms and devices, managing memory allocations and transfers among devices, compiling OpenCL kernels, launching kernels on targeted devices, querying execution progress and error checking[38]. Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution[19]. Much like the CUDA programming model, the executing device threads will each run an instance of the kernel to which they are assigned.

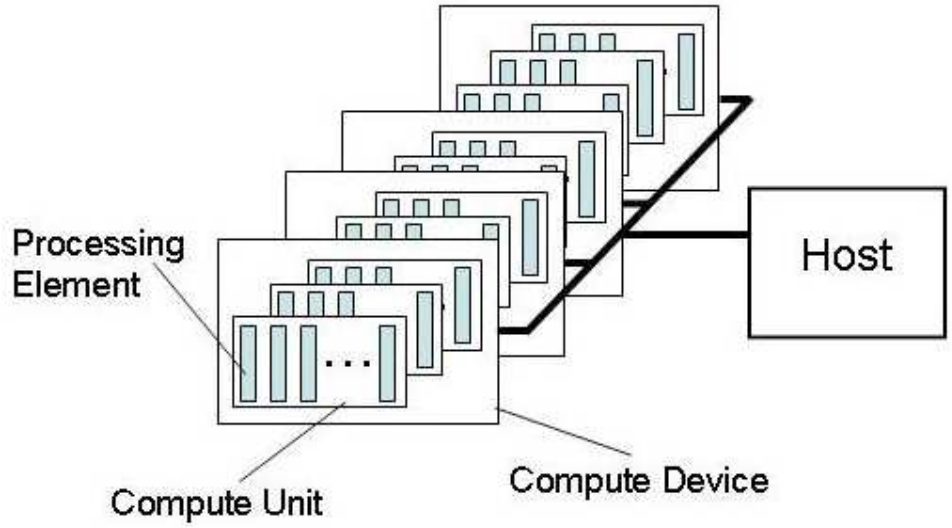

Figure 2.3: OpenCL processing architecture model[19]

These threads are called *work items* and are organized into work groups. The work-items in a given work group execute concurrently on the processing elements of a single compute unit.

One major difference between OpenCL and previous heterogeneous programming models like CUDA and Brook+[8] is that it enables runtime compilation of kernel code for execution on devices. This enables OpenCL applications to take advantage of whatever hardware devices are available without having to recompile the full application. The OpenCL memory model consists of four different memory spaces as shown in Figure 2.4. This is similar to the CUDA memory model, but with some subtle differences to make it more applicable to other computer architectures. OpenCL defines global memory, constant memory, shared local memory, and private memory. Just as with kernel execution, synchronization, runtime compilation and other as-

pects of OpenCL, the actual implementation of the different types of memory is up to the individual hardware vendors.

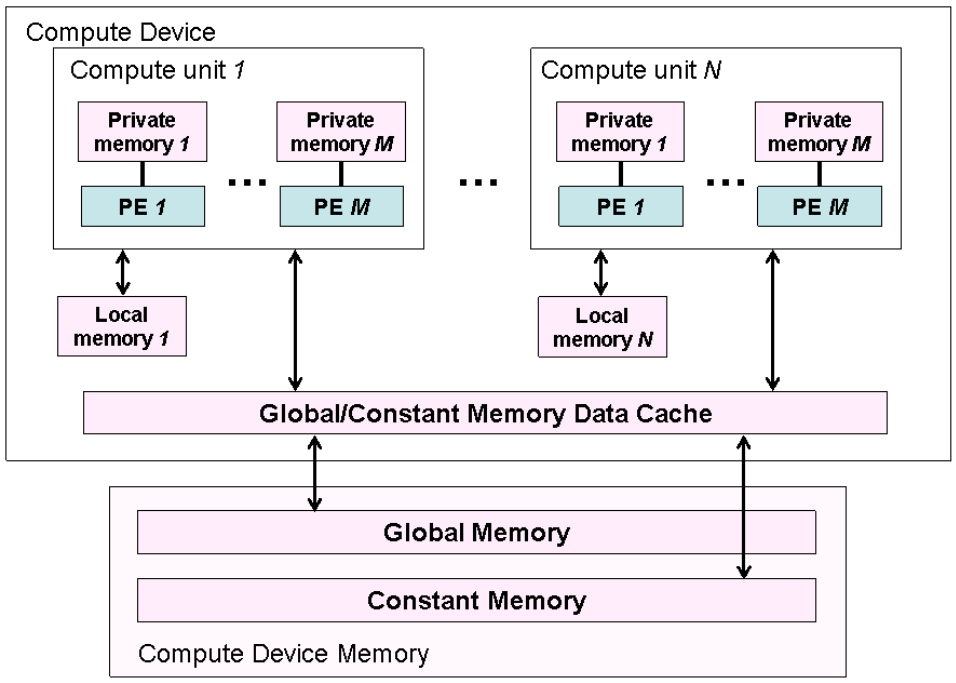

Figure 2.4: OpenCL memory and computational organization[19]

All OpenCL actions, such as memory management operations and kernel execution are enqueued in command queues associated with each device and performed asynchronously according to the vendor's implementation. Synchronization functions, similar to those found in CUDA, are available to force barriers within command queues for coarse-grain control over application synchronization. While the OpenCL standard is portable across different computing platforms, it is designed for use at the same level as CUDA applications and does not include simple abstractions for task parallelism and constructing applications with concurrent host threads. This necessitates simpler abstractions for common functionality that enable developers

to focus on algorithm development and performance improvement. The PVTOL heterogeneous tasks and conduits framework provides that abstraction.

2.1.4 Other Parallel Programming Models

A parallel programming model is an abstraction of the computer system architecture, and is not specific to any particular machine type[22]. However, there are many possible models for parallel computing because of the different ways several processors can be put together to build a parallel system[11]. In addition to the GPU-based parallel programming models described above, there are a number of other parallel programming models that have been developed for a variety of architectures, including shared memory, distributed memory, heterogeneous memory, and combinations of those models as well. This section will present an overview of the most widely used and applicable models to this dissertation.

POSIX Threads (Pthreads)

The POSIX Threads (Pthreads) C extension (IEEE Std 1003.1c-1995) is a set of functions for creating, destroying, and running independent threads. A thread is a lightweight process having its own program counter and execution stack[3]. The model is very flexible but low level, and is usually associated with shared memory and operating systems[11]. This model has locks and condition variables for managing access to shared memory locations and uses the fork/join parallel programming pattern[22]. Because the global memory is shared between threads in this model,

the programmer must be aware of and developed code to account for race conditions, deadlocks, and memory access patterns. Additionally, the low-level nature of Pthreads makes it extremely difficult to create programs that are easy to maintain and scalable to a large number of processors[11].

Open Multiprocessing (OpenMP)

The Open Multiprocessing (OpenMP)[36] programming model is a multi-threaded programming API that is implemented as compiler directives, pragmas, and a run-time library. OpenMP is slightly higher level than Pthreads, since the compiler directives and pragmas designate to the compiler how to create threads, manage thread synchronization and access shared memory[11]. OpenMP also follows the fork/join model, allowing a single control thread to split into a number of independent tasks. Simply designating a parallel region enables a single task to be replicated across a set of threads. In order to distribute different tasks across threads, the model provides a set of work sharing directives enabling each thread to execute a different task[36]. OpenMP allows for high level parallel abstraction, making it simple to convert serial code to parallel code and very well suited to shared memory high performance computing applications. The model is limited in that it does not support distributed or heterogeneous memory systems, or where more control over thread behavior is needed.

Message Passing Interface (MPI)

The Message Passing Interface (MPI)[26] is a parallel programming model for distributed memory systems where communication between systems is handled through exchanging messages. This is implemented as a library that specifies the names and functionality of the routines used to pass and receive messages amongst threads[26]. One of the most popular implementations of the message passage interface is OpenMPI[34]. A message is a shared portion of data that is copied from one concurrent process' memory space to another process' addressable memory. The communication can only take place when the first process executes a send operation and the second process executes a receive operation[11]. This can be done in point-to-point communication between tasks or in a broadcast-subscribe style of communication where there is a single sending task and multiple receiver tasks. This synchronization has to be done by the programmer, as well as assigning computation to tasks, making the difficulty of programming and level of granularity similar to that of Pthreads. MPI is currently the de facto standard for HPC applications on distributed architectures. Thus, this model is well suited to the SPMD and Master/Worker program structure patterns.[22]. Because the message passing interface is so general, OpenMPI is also useful for applications where portability is important.

2.2 PVTOL Tasks and Conduits Framework

The original version of the Parallel Vector Tile Optimizing Library (PVTOL) was developed by MIT Lincoln Laboratory as a means of writing high performance signal and image processing code that is portable across multi and many core general purpose computing architectures[20]. The goal of PVTOL is to provide a set of consistent, portable C/C++ abstractions of computation (*tasks*) and data management and synchronization (*conduits*), thus allowing programmers to develop applications on serial processors and port them to many different heterogeneous and multicore systems with minimal effort. The PVTOL API provides a consistent, portable programming model that hides the complexity of the underlying processor configuration and memory hierarchies. Using widely supported libraries, such as POSIX threads (`pthreads`), `MPI`[26], and `Boost`[33], the PVTOL tasks and conduits framework provides a set of high-level programming constructs for task and data parallelism capable of dealing with the heterogeneity and complexity of different computing platforms and systems[25]. The PVTOL program uses a task manager to launch tasks as independent processes and monitors their progress and completion. The tasks and conduits use C++ templates so that specialized versions of each can be created for different processor and co-processor architectures and still maintain the same structure and interface. Tasks are connected via conduits, which oversee data transfer and synchronization. Tasks can have any number of input and output conduits connected to them. Conduits can have two or more endpoints, and can also support

a broadcast/subscribe type of interaction. Tasks and conduits can be connected in virtually any configuration, allowing for a tremendous amount of versatility in which applications can be constructed.

2.2.1 PVTOL Tasks

PVTOL tasks are hierarchical, modular structures that isolate and abstract data processing that can then be mapped to one or more processing elements in a system. Data are sent to and received from the task through a common conduit interface. Tasks support data parallelism by being able to encapsulate and launch single program multiple data (SPMD) code. Instantiating multiple tasks to run concurrently in a PVTOL application allows the programmer to employ task parallelism. Maps are used to assign tasks and data to various processing elements. A task map designates on what processing elements a task will execute its code, while a data map is used to distribute the data being operated on within a task to the particular processing elements. For example, take a system with 4 processing elements and executing on a data set of 512 elements. A task map may assign a task to processing elements 0, 2, and 3, and that task's data map may break up the 512 elements into arrays of elements 0-63, 64-127, and 128-511. These maps will cause the data elements 0-63 to be implicitly mapped to processing element 0, elements 64-127 to be mapped to processing element 2, and elements 128-511 to be mapped to processing element 3. This separation of data and task mapping enabled PVTOL to support task and data parallelism independent of the system architecture and each other. The task

constructs in PVTOL use C++ templates to allow maps to be passed as template arguments at initialization. This isolates the developer's code within each task from how it is mapped to the system's hardware, thus maintaining a separability between the application and the underlying architecture.

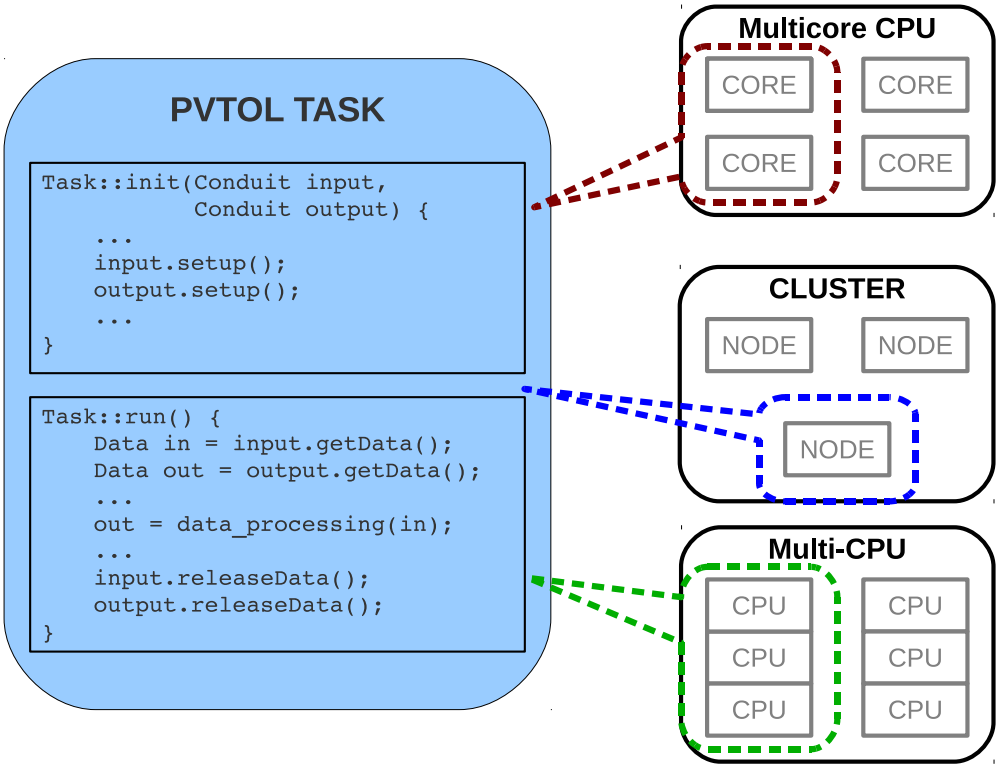

Figure 2.5: Example structure and possible resource mappings for a task.

A PVTOL task must have at least two functions, `init` and `run` that perform any initialization and processing, respectively. Task initialization can be as simple as setting a variable, or as complicated as performing multiple platform and device initializations. Typically, conduit endpoints are established in the `init` function, so that data can be transferred to and from the task during computation. The `run`

function, when called, is launched asynchronously and is responsible for obtaining access to input and output data buffers within connected conduits, executing computations on the data, and then releasing control of the data buffers back to the conduits. Figure 2.5 shows a simplified task and different ways it may be mapped to the processing elements within various computer systems.

2.2.2 PVTOL Conduits

PVTOL conduits are programming constructs that are responsible for isolating and abstracting data management, transfers, and synchronizing data communication between two or more tasks processing the data. This is accomplished with the use of threading libraries for tasks executing on the same platform and MPI for tasks executing on multiple machines[20]. Conduits can manage data communication between two tasks utilizing separate or shared memory systems. In the case where a conduit's endpoint tasks use separate memory systems, separate buffers are allocated on each and data is transferred from source to destination as soon as it is available at the source. If the conduit's two endpoints share the same memory system, a single data buffer is allocated, and control of the buffer is managed amongst the connected tasks. Conduits are templated with two arguments, a conduit type and a data type, which enable the conduit to instantiate any data communication interface for which a conduit model exists. The data type specifies any C/C++ data object to be transferred through the conduit, which can be specified as multi-dimensional data, such as arrays and matrices. Multi-buffering is supported to allow for conduits to act as a FIFO

memory system between tasks. This greatly simplifies the process of establishing appropriately sized memory buffers between tasks processing data at different rates.

Conduits are interacted with through their `Reader` and `Writer` interfaces. The `Writer` interface is used by one or more source tasks to inject data into the conduit. Conversely, the `Reader` interface is used to extract data from the conduit by one or more destination tasks. Both of these interfaces have the same three primary functions, `setup`, `getData`, and `releaseData`. Figure 2.6 depicts an example of how a conduit is implemented across different memory systems. The `setup` function is used for specifying information important to the initialization of the conduit, such as the dimensions of the data and the number of buffers. The conduit will then establish data buffers within the source and destination memory systems.

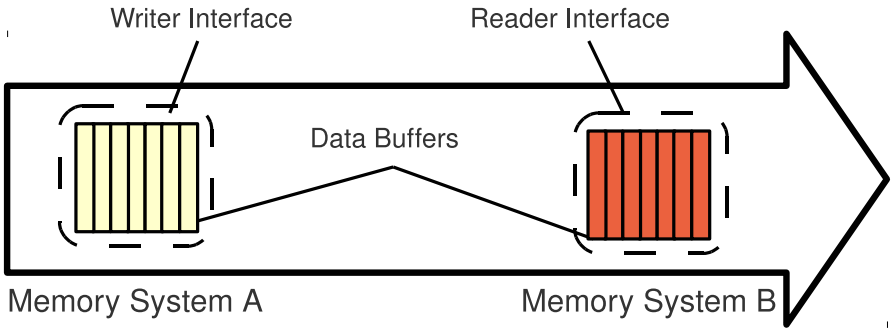

Figure 2.6: Example structure for a conduit interfacing between distributed memory systems

The `getData` and `releaseData` functions abstract the complicated hand-shaking and synchronization that takes place when transferring data between memory systems and concurrent processes. The `getData` function is called by a task when it wants

to obtain access to some memory from a conduit, and the task will wait until that memory is available. When it becomes available, the function returns a pointer to the data and the task is free to operate on that memory as it sees fit. When processing is complete, the task then calls `releaseData` to signal that memory is again available to other elements in the system. Once a task has obtained access, written data and released access at the source of a conduit, the data is immediately transferred to an available buffer in the destination memory system and made available there, completely transparent to the user. The initial version of PVTOL supported this type of behavior across any type of general purpose computing platform[25].

2.2.3 PVTOL Applications and Architectures

PVTOL applications are constructed at the highest level by connecting different computational tasks within an algorithm with conduits in order to produce the desired data processing pipeline. Because this is a set of C/C++ constructs, the application elements are declared, mapped, and the pipeline constructed within the `main` function. This isolation and centralization of the application organization and mapping creates a separation of any aspects of the application specific to the system architecture and the data processing, memory management, and synchronization. This separation enables an application developer to experiment with different processing element mappings, data flow configurations, and system architectures with little or no programming effort[20]. The simple application example shown in Figure 2.7 shows how tasks and conduits can be used to construct a common image or signal

processing pipeline. This example demonstrates a number of advantages to the tasks and conduits framework in addition to the separation of data processing and memory management. This example isolates the different functions of the core algorithm and data I/O, allowing for data I/O interfaces or algorithm components to be easily swapped with others. Figure 2.8 shows how the example application in Figure 2.7 is constructed and mapped. In this example, lines 3-6 initialize the PVTOL program and instantiate the task map with a given node of processing elements. Each task will then be launched as a thread on that node, and execute concurrently. For the purposes of this example, the data input task (`DIT`) is assigned to CPU core 0, the FFT task is assigned to CPU core 1, the FILTER task is assigned to CPU core 2, the IFFT task is assigned to CPU core 3, and the DOT task is assigned to CPU core 4. Lines 9-17 declare each task with their corresponding task map and each conduit with its corresponding name and data type. Lines 20-30 show the application initializing, running, and waiting for the completion of each task. When tasks are initialized, they are passed the endpoint interfaces of the conduits connecting them. This initializes the pipeline organization and data flow depicted in Figure 2.7.

2.3 Fluorescence Mediated Tomography (FMT)

In order to demonstrate the portability and versatility of the PVTOL tasks and conduits framework, we need an example application that exercises the full feature set available. We use a medical imaging application called Fluorescence Mediated

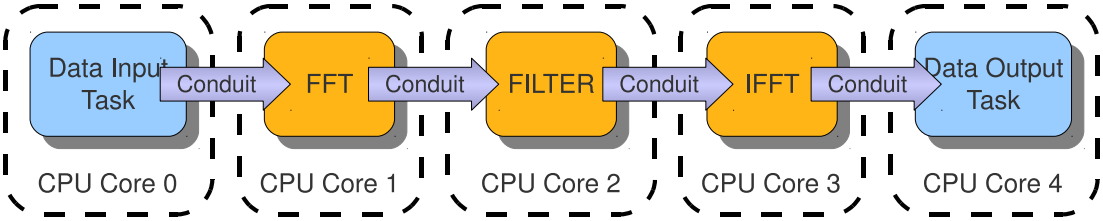

Figure 2.7: Example PVTOL application with a data input task feeding an FFT-FILTER-IFFT task pipeline with the result being stored by a data output task.

```
1  int main(int argc, char *argv[]) {
2     // Initialize PVTOL and each task map
3     PvtolProgram prog(argc, argv);
4     rank.push_back(0);
5     RankList ranks(rank);
6     TaskMap taskMap(ranks);
7
8     // Declare each task and conduit
9     Task<DIT>       dit( "DIT",    taskMap);
10    Task<FFT>       fft( "FFT",    taskMap);
11    Task<FILT>      filt("FILTER", taskMap);
12    Task<IFFT>      ifft("IFFT",   taskMap);
13    Task<DOT>       dot( "DOT",    taskMap);
14    HeterogeneousConduit<float> cdt0("DIT to FFT");
15    HeterogeneousConduit<float> cdt1("FFT to FILTER");
16    HeterogeneousConduit<float> cdt2("FILTER to IFFT");
17    HeterogeneousConduit<float> cdt3("IFFT to DOT");
18
19    // Initialize the tasks with their conduits
20    dit.init(cdt0.getWriter());
21    fft.init(cdt0.getReader(), cdt1.getWriter());
22    filt.init(cdt1.getReader(), cdt2.getWriter());
23    ifft.init(cdt2.getReader(), cdt3.getWriter());
24    dot.init(cdt3.getReader());
25    // Run the tasks
26    dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();
27    // Wait until tasks complete
28    dit.waitTillDone(); fft.waitTillDone();
29    filt.waitTillDone(); ifft.waitTillDone();
30    dot.waitTillDone();
31 }
```

Figure 2.8: Code to construct and execute a basic PVTOL application.

Tomography (FMT). FMT utilizes fluorescent indicators to highlight particular types of tissue and molecules, in order to make them more responsive to the wavelengths

of light being transmitted through them. Image reconstruction in FMT involves three steps: i) optical measurement of the fluorescence intensity transmitted through an animal between light source and detector pairs, ii) accurate modeling of light propagation between source and detector pairs to yield system weight functions (i.e. the forward problem), and iii) inversion of the resulting system of equations to yield the fluorescence image. Figure 2.9 depicts an example experimental set up of imaging a mouse body with two fluorescent indicators embedded in it and the output of transmitting light from a source to detector through the mouse body in a rotation around it.

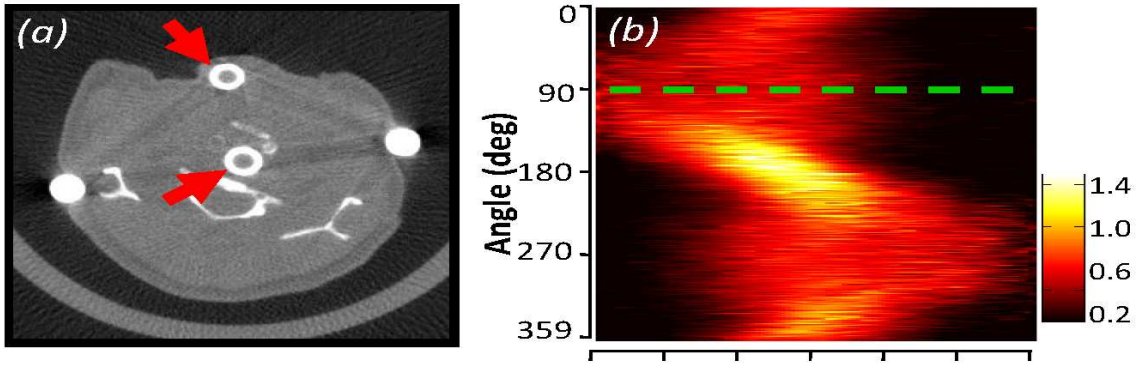

Figure 2.9: Example fluorescence mediated tomography scenario. (a) Mouse body cross section with two fluorescent indicators indicated with arrows and light source and detector (full circles). (b) The transmitted photon intensity over time (x axis) at each transmission angle (y axis).

The FMT algorithm has a number of different functional blocks that make it easy to exploit task parallelism. The second part (accurate modeling of light propagation) will be the focus of our examples because it is massively data parallel and adaptable to GPU and multicore architectures. The simulation of each photon through

the biological media is independent of other photons, making it a good candidate for data parallelization. In our case, the important information is the path each detected photon takes through the media. The paths of every detected photon are then accumulated to a 3D grid to give the final sensitivity function from the light source to the detector.

Previous works have shown the major challenge in FMT is the high degree of light scatter through biological tissue which limits the potential imaging resolution of the technique[16]. In particular, it's been shown that early-arriving photons can be accurately modeled using Monte Carlo simulations[28]. Wang et al.[41] developed the Monte Carlo Multi Layer (MCML) software package to perform Monte Carlo simulations of photons propagating through biological tissue. As Monte Carlo simulations are well suited towards parallelism, this work was the basis for Monte Carlo extreme (MCX), which performs photon propagation simulations on GPUs and shows significant speed up over the serial version[12]. Both of these software packages track the amount of energy absorbed and transmitted throughout the medium. In the application presented by Niedre et al.[28], the absolute path of photons traveling through the tissue is the significant piece of data. The example FMT application presented in this work adapts the MCML code to record photon paths instead of energy absorption[6].

2.4 3D Cone Beam Computed Tomography (3D CBCT)

The algorithm for cone beam image reconstruction was originally proposed by Feldkamp et al.[14], and has many useful applications in medical imaging. This technique generates 3D data from a series of 2D projections that have been acquired by a computed tomography (CT) scan. Figure 2.10(a) shows the experimental configuration of a conventional 3D cone beam CT scan with a flat panel detector. In the process of acquiring scanned data, the x-ray source moves in an orbital path and the detector panel moves in the same motion along with the source. The detector plane lies perpendicular to the rotational axis of the x-ray source, and it produces a set of 2D projections at discrete positions within the path of rotation. Figure 2.10(b) shows the 3D CBCT coordinate system. The xyz space is the volume and uv represents the projections that are to be back projected to the volume.

The 3D object being imaged is reconstructed from the 2D projections in two stages, weighting and filtering and then a final back projection. In the first step, the raw projections are individually weighted and filtered to produce filtered projections. The weighting and filtering may utilize many types of filters (ramp, hanning, hamming, etc.). The reconstructed 3D volume is then generated by applying the values of the weighted projections to the volume according to Equation 2.1[4].

$$F(x,y,z) = \frac{1}{2\pi t}\sum_{i=1}^{t} W_2(x,y,i)Q_i(u(x,y,i),v(x,y,z,i)) \tag{2.1}$$

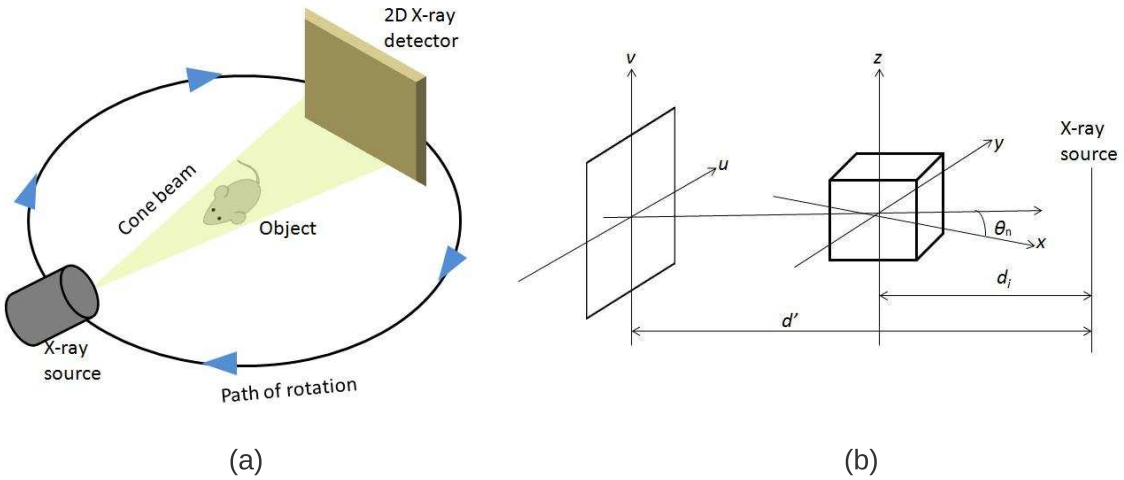

Figure 2.10: (a) 3D Cone Beam Computed Tomography experimental setup. (b) 3D Cone Beam Computed Tomography coordinate system.

where $W_2(x, y, n)$ represents the weight value, $u(x, y, n)$ and $v(x, y, z, n)$ represent the projection and volume coordinates, respectively.

$$u(x, y, i) = \frac{d'(-x \sin \theta_i + y \cos \theta_i)}{d_i(-x \cos \theta_i - y \sin \theta_i)} \tag{2.2}$$

$$v(x, y, z, i) = \frac{d'z}{d_i(-x \cos \theta_i - y \sin \theta_i)} \tag{2.3}$$

$$W_2(x, y, i) = \frac{d_i}{d_i(-x \cos \theta_i - y \sin \theta_i)} \tag{2.4}$$

Many serial implementations of the algorithm exist in many programming languages. For reference, we will use both the MATLAB and C based solutions included in the Image Reconstruction Toolbox provided by [15]. Other previous works have shown that each output voxel of the reconstructed volume can be computed independently, making this algorithm easily adaptable to the GPU platform using CUDA[30][32]. These implementations, and others, have shown that the 3D cone beam image reconstruction algorithm is both computationally intensive, as well as

suited towards a number of architectures. This makes it an excellent candidate for adapting to the OpenCL platform and utilizing it for demonstrating interesting features of the heterogeneous PVTOL tasks and conduits framework.

2.5 Related Work

With the proliferation of heterogeneous computing architectures and the ever increasing pace at which new architectures are created, developers and engineers need to introduce different techniques to provide concurrent processing and portability in their applications. A number of different methodologies have been employed attempting to solve this issue, including the development of programming languages, compilers for extracting available parallelism in existing source code, and higher level software frameworks for heterogeneous computing platforms.

2.5.1 Programming Languages

There have been many attempts to create programming languages and environments that enable developers to create portable code[37][13][2][38]. These works all propose methods and frameworks for writing code that can be compiled and executed on a wide variety of distinct computer architectures. The Partitioned Global Address Space (PGAS) model was developed by Aggarwal et al. to simplify the use of memory hierarchies in heterogeneous systems. PGAS uses the SHMEM+ API[2] to map all the various memory resources in a system into a single global address space. This model abstracts basic memory allocation, movement, and synchronization across

different architectures within a system, but does so using a non-standard interface (SHMEM+), and does not alleviate any of the difficulty of managing memory across concurrently executing programs.

OCCAM is an API for writing parallel applications[13]. The primary drawbacks to the OCCAM platform are that it requires the developer to specify data management and optimizations, and it currently only supports homogeneous multicore CPU systems. The Accelerator library is a set of language constructs designed to allow a developer to write code describing parallel data computation that can be compiled to execute on GPU, FPGA and multicore CPU architectures[37]. One drawback to this method is that the Accelerator programming model requires the developer to deal with the complexity of data management, transfers, and synchronization. This model also only includes support for a set number of data array types and operations that can be performed on accelerators like FPGAs and GPUs.

While the OpenCL standard[19][38] provides a consistent programming model for writing portable code, there are a number of reasons it is difficult to use in heterogeneous systems. Using OpenCL requires a vendor implementation of the OpenCL API, capable of recognizing platforms, devices, drivers, and compiling OpenCL code for them. Vendor OpenCL libraries, such as those currently released for graphics processors by NVIDIA and AMD only support the vendor's own platforms and devices and necessary host devices (general purpose processors, like CPUs). Thus, OpenCL applications are only able to execute on systems containing architectures supported

by a particular vendor's implementation. OpenCL kernels must be rewritten in order to achieve the best performance on different platforms. This severely limits the heterogeneity of the systems on which OpenCL can be used, since any one vendor's implementation will support only their own platforms. Additionally, platform specific programming languages like NVIDIA's CUDA continue to yield better performance in a number of cases than equivalent OpenCL implementations[18].

2.5.2 Compilers and Interpreters

Many previous works have tried to address the issues inherent in parallel programming by enabling code to be adapted to different architectures for better data parallelization using the single process multiple data (SPMD) model [42][21][5]. These methods try to extract and classify data parallelism in serial algorithms written in a commonly used programming language, such as `C`, `C++`, or `FORTRAN`.

In [42], the authors develop the PGI Accelerator programming model. This model attempts to split the responsibility of compiling standard `C` or `FORTRAN` code into CUDA code that can execute on GPUs with the programmer. This model employs the use of `pragma` style directives to allow the user to specify which computations (typically loop bodies) and data objects are to be held on the GPU board. The model uses a planner to determine how to map the designated code loops onto the accelerator architecture and then attempts to optimize the generated accelerator code in order to achieve greater performance and occupancy. This model has limitations in that it can only unroll the innermost loops, can't optimally order parallel loops in

different CUDA blocks, and code generation for reductions becomes very complex. The work presented by Liu et al.[21] also uses `pragma` style directives, but to specify an input parameter search space. This framework then attempts to generate GPU kernels that are adaptable to different input values using a two-part approach. The first part uses an iterative heuristic-based empirical search of the user specified input space, measuring the performance of each configuration and tracking the differences based on the changes in inputs. The second step then generates code that is adaptable to differences in inputs while attempting to yield good performance. This work focuses specifically on GPUs, relies on the user to define the input parameter search space for the framework, and requires the algorithm to already be written in native GPU code (i.e. CUDA) to evaluate it. The CuPP framework[5] was developed as a C++ based API for performing data management and CUDA kernel executions. This framework provides support for using C++ classes and data structures that can be used in GPU kernels. However, this work currently only includes support for a single vector data structure, and includes caveats like not being able to support classes containing pointers to other data. Additionally, the framework only uses C++ constructs, and does not support C, which is how many CUDA applications are written. Data structures used in CUDA kernel calls are copied to/from the device memory with each kernel call. This does not allow for data remaining on the device for consecutive kernel calls, exacerbating the memory bandwidth bottleneck in modern GPGPU computing. CuPP also does not provide any mechanism for exploiting task

parallelism or synchronization among multiple threads of an application. All of these works implement data parallelism on GPU architectures in a way that attempts to achieve the greatest thread parallelism and resource occupancy. However, greater GPU occupancy through thread parallelism has been shown to not always yield the best data throughput versus some methods employing instruction level parallelism and data parallelism with less occupancy[40].

2.5.3 Software Frameworks

A small number of works have developed frameworks to abstract out data management and task scheduling and communication on heterogeneous systems[1][9][17][29]. In `PFunc`, the authors develop a C/C++ API for task specification and scheduling. The framework provides means of specifying thread scheduling policies, thread groups, and thread ordering functions[17]. This API has a few shortcomings in that it still requires the developer to perform all data synchronization using programming structures that are not any simpler than writing traditional multi-threaded C/C++ applications, and it currently only supports multi-threaded applications on CPU-based architectures. Helios[29] is an operating system designed to allow heterogeneous programming using satellite kernels with GPU and NUMA architectures in mind. However, many architectures like current generation GPUs and other dedicated accelerators are not capable of running the Helios satellite kernel because they lack sufficient hardware resources like timers and interrupt controllers. Auto-Pipe and the *X* language is a development environment for pipelined applications ex-

ecuting on architecturally diverse computing platforms[9]. This framework uses a coordination language to construct streaming applications in a way that is similar to the original PVTOL tasks and conduits[25]. Computational blocks are declared, and then data communication between them are described by edges that connect to input and output ports of the blocks. Each block is then mapped to a processing element in the system, assuming an implementation of the block exists for the platform it is mapped to. Edges have two distinct end points and are mapped to pre-defined interfaces for communicating between two computational blocks. The Auto-Pipe system does not allow for completely arbitrary task mappings and communication beyond pipelining. Edges are not able to perform communication among more than two blocks at any one edge. The PVTOL Tasks and Conduits framework has the capability of supporting communication between multiple endpoints. Additionally, the Auto-Pipe tools and libraries require the use of the proprietary *X* language to construct applications and specify task and communication properties, deviating from widely known programming languages.

Another framework being developed is the Open Component Portability Infrastructure (OpenCPI)[24]. This project attempts to provide an infrastructure for component-based applications using general purpose processors, DSPs, FPGAs and GPUs on embedded systems. This is achieved by creating *authoring models* of computational components that describe their input and output interfaces, so that they can be updated or replaced without affecting any other component in the system.

This framework, like PVTOL, requires kernels and core functions to be written in their native language. However, it requires the developer to define the component and data interfaces that PVTOL provides.

Aggarwal et al. propose a heterogeneous application framework, the System-Level Coordination Framework (SCF)[1]. This framework takes a different approach to providing similar features as PVTOL Tasks and Conduits. SCF attempts to abstract out data communication and synchronization among different architectures with edges and processing with tasks, as well as providing a means of device management very much like the original Tasks and Conduits framework[20][25][6]. There are some significant differences between the frameworks. An application written using SCF consists of a variety of files of varying types and functions. Tasks are not mapped using C/C++ structures, but instead specified in a separate file that declares the task function name, architecture, and IDE used to compile or implement it. Message passing along an edge is implemented using the authors' System Coordination Library (SCL). The input and output data structures of each task are assigned to an edge in an SCL file[1]. Tasks and edges are then constructed together into an application using a task graph. The application uses all of these specifications to create an application at compile-time as long as it can compute a communication interface between all platforms and devices. The PVTOL Tasks and Conduits framework contains all of this information in the application's `main` function, using standard C/C++ constructs. The SCF model also uses an architectural hierarchy similar to

that of OpenCL, where the application consists of a number of platforms that themselves consist of one or more devices. Each of these devices must be SCF-compliant, which means that it is capable of supporting communication between multiple tasks mapped to it. This framework supports general purpose processors, but many GPUs can not support the SCF-compliant type of inter-kernel communication. The authors also do not provide an example of this framework executing on a GPU architecture.

2.5.4 Summary of Related Work

PVTOL tasks and conduits have the added advantage over other approaches of supporting a wide variety of heterogeneous platforms and programming models, maintaining a consistent C/C++ based programming environment, minimizing additional development time learning proprietary languages, and alleviating the burden of adapting applications to extremely complicated memory models.

Chapter 3

Methodology and Design

Here we present the research that has been done to extend the PVTOL tasks and conduits framework to support heterogeneous architectures, including the support for the CUDA and OpenCL programming models, the specifics of the heterogeneous task and conduits structures, and other supporting structures and information.

3.1 Heterogeneous Tasks and Conduits

The original version of PVTOL Task and Conduits contains support for general-purpose homogeneous processing platforms (Intel Xeon, AMD Athlon, etc.), but had not been extended to include other accelerator architectures. In my research, support for heterogeneous architectures that include graphics processing units have been added to the PVTOL tasks and conduits framework. This provides abstractions for allocating memory, transferring data between the host and GPU-based tasks, and executing kernels. Support for heterogeneous architectures is seamlessly integrated with the existing PVTOL tasks and conduits constructs using the NVIDIA Compute Unified Device Architecture (CUDA) and the Open Compute Language (OpenCL).

Heterogeneous tasks and conduits interface to the CUDA and OpenCL APIs through a set of utility functions.

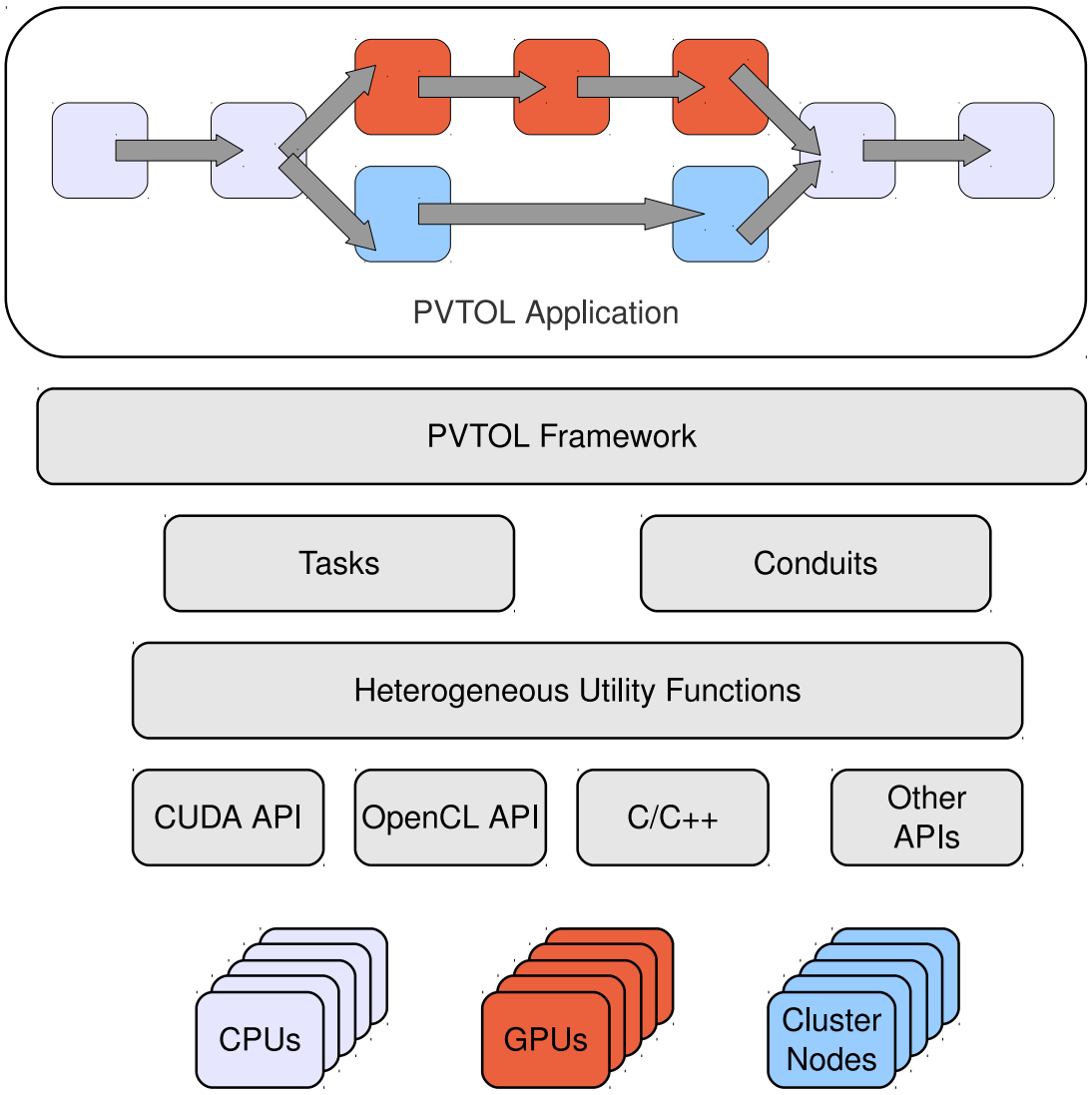

Figure 3.1: PVTOL heterogeneous tasks and conduits framework software organization

Figure 3.1 depicts the software organization of the heterogeneous tasks and conduits framework, including the user application, pvtol system, task and conduit ab-

stractions, and heterogeneous utility functions. The rest of this section describes the utility functions and their API, as well as the design and implementation changes that were made to integrate heterogeneous support into the PVTOL tasks and conduits framework. We then discuss heterogeneous tasks and heterogeneous conduits and their applications.

```
1  int main(int argc, char *argv[]) {
2     // Declare variables
3     cufftComplex *hostIn, *hostOut;
4     cufftComplex *devIn,  *devOut;
5     cufftHandle planFwd;
6     int N = 1024, batchSize = 10;
7     int idx, nBytes;
8
9     // Allocate host-side memory
10    nBytes = sizeof(cufftComplex)*N*batchSize;
11    hostIn = (float*)malloc(nBytes);
12    hostOut = (float*)malloc(nBytes);
13
14    // Initialize input data
15    for (idx = 0; idx < N*batchSize; idx++) {   //
16       hostIn[idx].x = sinf(idx);               // Data Input Function
17       hostIn[idx].y = cosf(idx);               //      inFunc
18    }                                           //
19
20    // Allocate device-side memory
21    cudaMalloc((void **)&devIn, nBytes);
22    cudaMalloc((void **)&devOut, nBytes);
23
24    // Copy input data from host to device
25    cudaMemcpy(devIn, hostIn, nBytes, cudaMemcpyHostToDevice);
26
27    // Execute kernels to process data
28    cufftPlan1d(&planFwd, N, CUFFT_C2C);
29    for (idx = 0; idx < batchSize; idx++) {
30       cufftExecC2C(planFwd,                    //
31                    devIn[idx*N],               // Forward FFT
32                    devOut[idx*N]);             //
33    }
34
35    // Copy output data from device to host
36    cudaMemcpy(hostOut, devOut, nBytes, cudaMemcpyDeviceToHost);
37
38    // Process output data (compare, write to disk, etc.)
39    double maxErr = 0.0;                        //
40    double tmp = 0.0;                           //
41    for (idx = 0; idx < N*batchSize; idx++) {   //
42       tmp = hostOut[idx].x/N - sinf(idx);      // Data Output Function
43       maxErr = max(fabs(tmp), maxErr);         //      outFunc
44       tmp = hostOut[idx].y/N - cosf(idx);      //
45       maxErr = max(fabs(tmp), maxErr);         //
46    }                                           //
47    printf("Max FFT error = %g\n", maxErr);     //
```

```
48
49      // Free memory buffers
50      cufftDestroy(plan);
51      free(hostIn);
52      free(hostOut);
53      cudaFree(devIn);
54      cudaFree(devOut);
55      return 0;
56  }
```

Listing 3.1: Simple application initializing data, executing a CUFFT kernel, and analyzing the output data

As a demonstration of the benefits of the tasks and conduits framework for heterogeneous applications, we look at the example of using the CUFFT library shown in Listing 3.1. In this example, memory buffers are allocated on both the host CPU and GPU device, host side data is initialized, input data is copied down to the device, the CUFFT function is executed on each batch of data, output data is copied back up to the host, the output data is post processed, and finally all of the data buffers are deallocated. All of this happens sequentially, with no data processing and memory operations happening concurrently, and all of the memory management being the responsibility of the application developer. Later in this chapter, we show this application written using the heterogeneous tasks and conduits framework and discuss the benefits of its use.

3.1.1 Heterogeneous Utility Functions

The heterogeneous utility functions are an API that allow the task and conduit code to perform their functionality using the appropriate parallel programming model, while keeping the platform independent code isolated from the programming model specific code. The utility functions provide a means for allocating data, moving

data, initializing devices, executing kernels (including third party library functions), deallocating data, and cleaning up platform-specific variables. These functions take in parameters from tasks and conduits related to the dimensions, type, and location of the data, and in an intelligent manner, configure the appropriate arguments for function calls to the appropriate API. The API and code for the utility functions can be found in the appendixes in Appendix A.2.1.

One of the key pieces of information required by each utility function is the heterogeneous task info data structure `HTaskInfo`, shown in Listing 3.2. This is the data structure that contains the information necessary to identify the heterogeneous device to which a task is mapped and how to properly interface with it. The structure contains four pieces of information described in Table 3.1.1. From this information, the utility functions can execute with respect to the proper heterogeneous device.

```
1  // Heterogeneous Task Location Type
2  typedef enum{ LOC_INVALID  = 0,
3                LOC_CPU      = 1,
4                LOC_CUDA     = 2,
5                LOC_OCL      = 3,
6                LOC_NLOCS    = 4 } hTaskLoc;
7  // Heterogeneous Task Platform Type
8  typedef enum{ PLAT_INVALID = 0,
9                PLAT_C       = 1,
10               PLAT_NVIDIA  = 2,
11               PLAT_AMD     = 3,
12               PLAT_INTEL   = 4,
13               PLAT_NPLAT   = 5 } hTaskPlat;
14 // Heterogeneous task information structure
15 typedef struct {
16    hTaskLoc  location; // Task location
17    hTaskPlat platform; // Platform identifier
18    int       device;   // Task device rank
19    int       process;  // Task process rank
20 } HTaskInfo;
```

Listing 3.2: Heterogeneous task information data structure and types

In order to develop the utility functions, we first had to determine which function-

Variable Name	Description
Location	Specifies the type of device (CPU, CUDA, OpenCL)
Device Index	The index of the device to be used
Process Index	The index of the process (CUDA stream ID, OpenCL command queue)
Platform	The OpenCL platform to utilize (not used for CUDA devices)

Table 3.1: Heterogeneous task information structure variables and descriptions.

ality the tasks and conduits framework would require with respect to heterogeneous platforms. The PVTOL system itself needs to be able to initialize any system-wide information and variables required to use the CUDA and OpenCL programming models. The functions `initSystem` and `closeSystem` take a set of flags as input indicating which programming models the application will use. These functions will allocate and initialize and global variables or interfaces to supported third party libraries, and then deallocate and close down those variables, respectively. The tasks, as an abstraction for data processing, need to be able to initialize devices, build kernels at run-time, and then execute those kernels. The `initDevice`, `build`, `safeBuild`, and `launchKernel` functions accomplish this. `initDevice` establishes variables necessary to execute functions on a device, such as contexts and command queues. The `build` and `safeBuild` functions build kernels for a specified device. The difference between the two is that `safeBuild` will output compilation errors, build logs, and other compilation information at run-time so that the user can edit and re-compile kernels without having to rebuild their entire application. This is primarily for debugging kernel compilation, but can be useful for the case where different OpenCL

implementations use different compilers across various systems. The `launchKernel` function takes in a kernel name, lists of kernel parameters and parameter sizes, and launches the specified kernel on the specified device.

The last set of utility functions are utilized by the conduits and pertain to heterogeneous memory operations. The functions `initMem`, `freeMem`, `clearMem`, and `moveData` will allocate, deallocate, zero-fill, and transfer data from source to detector respectively. `initMem` uses the data dimensions, data type, and device information provided by the conduit to allocate memory on the appropriate device. `freeMem` uses the same information to de-allocate that memory, typically when the conduit destructor is called. In the case of the `clearMem` function, if no specific API call to clear memory exists for a particular programming model a very general kernel is provided. For example, CUDA has a `cudaMemset` function, but the equivalent function in OpenCL (`clEnqueueFillBuffer`) is only supported in OpenCL v1.2 or higher. Thus, for OpenCL implementations at v1.1 or lower, a generic memory fill kernel is built and executed. The final memory utility function is `moveData`. This function takes in data pointers, data dimensions, data type, and device information for both a source and destination buffer. The appropriate API call is then made to copy the data from the source to the destination.

The utility functions will also perform some parameter checking at run-time to ensure operations the user wants to perform are supported by the hardware. There is error checking to ensure that the operations execute on the hardware without issue,

printing out diagnostic information if there is one. The tasks and conduits framework can be compiled to quit the application when an error is encountered, or to just print out information about the error and continue. The utility functions can also handle interfacing to any third party libraries, such as CUFFT, CUBLAS, GPUFFTW, and others through this common API. This reduces the amount of developer effort needed for many high performance applications, and is done transparent to the user's interaction with the tasks and conduits framework.

Heterogeneous Map Structure

The first structure necessary for adapting the tasks and conduits framework to heterogeneous platforms is to designate a way to map tasks to heterogeneous architectures. One constraint of all heterogeneous programming models currently is that there is a host device (typically a CPU) and an accelerator device. Because of this distinction, each heterogeneous task will run as a CPU thread interacting with accelerator devices, and there will be a difference between its host mapping and its heterogeneous device mapping. This necessitates a heterogeneous device map for tasks in addition to the existing task map. We developed the heterogeneous task map structure, which can be found in Appendix A.1.1, as a collection of heterogeneous task information structures. This allows tasks, at instantiation, to be mapped to one or more heterogeneous devices. Each device will execute an independent heterogeneous task on each of the devices it is mapped to.

CUDA Support

To add CUDA support to the framework, we needed to implement each of the utility functions using the CUDA API. For simplicity and to make the basic functionality of the framework easier to change, we decided to utilize the CUDA Runtime API, as opposed to the lower-level CUDA driver API. The CUDA version of the heterogeneous utility functions can be found in Appendix A.2.2. The `cudaInitSystem` function will query the properties of all of the CUDA-enabled devices on the system or compute node and then initialize any third party libraries, such as CUFFT, CUBLAS, CURAND. The `cudaInitDevice` function will initialize the device and CUDA stream (equivalent to a device execution thread) specified by the device and process respectively of the heterogeneous task information data structure.

The memory functions (`cudaInitMem`, `cudaFreeMem`, `cudaClearMem`, and `cudaMoveData`) support a number of CUDA memory types. These functions support CUDA *global*, *constant*, and page-locked host-side memory that is device accessible. Both *register* and *shared* memory must be managed within a kernel, so the framework does not attempt to support them. The dimensions of the data are known by the conduits, enabling these functions to support 2D and 3D memory structures with both unity and non-unity strides. Because the conduits handle data synchronization, the utility functions use asynchronous API calls. The final piece of functionality in the CUDA utility functions is for launching kernels, which will set up the kernel arguments and then launch the kernel execution. The CUDA utility functions are written to sup-

port versions of CUDA 4.0 and above, including the latest version at the time of this writing, CUDA 4.2.

OpenCL Support

OpenCL support for the framework is more complicated than adding CUDA support because it is a more general parallel programming model that allows for different heterogeneous configurations and functionality, such as host-device unified memory and runtime kernel compilation. In addition to the extra utility functions, there is some common OpenCL functionality that is supplementary to the utility functions and can be useful to algorithm development. This is provided in the OpenCL helper functions in Appendix A.2.4. These OpenCL helper functions print information about OpenCL constructs like platforms and devices in a human readable format, decode the various OpenCL data types in strings, and can search for platforms and kernels based on input strings.

The OpenCL version of the heterogeneous utility functions can be found in Appendix A.2.3. The OpenCL version of the utility functions have to maintain information about the system on which it is being used, such as available platforms, devices, contexts, command queues, and compiled kernel objects[19]. The `oclInitSystem` function will query all of the OpenCL platforms and devices in the system and allocate memory for storing information about them. The `oclInitDevice` function will initialize the context and command queue specified by the device and process respectively of the heterogeneous task information data structure. There are two

OpenCL utility functions that are not found in the other supported programming models, `oclSafeBuild` and `oclBuild`. Both of these functions are used for compiling kernels at runtime, but the `oclSafeBuild` function is interactive. It will print out compilation error and warning messages and prompt the user to attempt to fix and re-compile the kernel. This allows the user to correct the kernel and re-compile at runtime if needed. This is a useful debugging tool, but different platforms will use different compilers and may yield different compilation results, so it may be useful in porting OpenCL applications as well. The `oclBuild` function simply compiles the specified kernel and indicates whether it was successful or failed. Building of kernels typically takes place at initialization to avoid compiling while executing time-sensitive code.

The OpenCL memory utility functions (`oclInitMem`, `oclFreeMem`, `oclClearMem`, and `oclMoveData`) support the OpenCL memory types equivalent to the supported CUDA memory types, *global* and *constant*, as shown in Figure 2.4. All OpenCL commands are queued asynchronously until the command queue is flushed. The OpenCL utility functions launch kernels in the same way as the CUDA utility functions. The OpenCL utility functions are written to support OpenCL versions 1.0, 1.1, and 1.2, and take into account when a platform is using a specific version.

C/C++ Support

The C/C++ general purpose processor model was the original platform for which the PVTOL tasks and conduits framework was developed. When the framework was

adapted for heterogeneous architectures, the C/C++ support had to be moved behind the heterogeneous utility functions API. The CPU utility functions can be found in Appendix A.2.5. These functions have very straight-forward implementations using the standard C memory functions `malloc`, `memset`, and `free`, and enable the heterogeneous tasks and conduits framework to maintain CPU support while using the same utility function API as the GPU architectures.

3.1.2 Heterogeneous Task

The heterogeneous task is a software abstraction for processing on any heterogeneous processing elements, including CPUs, GPUs, accelerated processing units (APUs), and cluster compute nodes. The task construct relies on two critical pieces of information, the processing element(s) it is mapped to, and the function(s) it is meant to execute. All heterogeneous tasks rely on the same information and maintain the same API functions (`init` and `run`). The original task class was implemented as a C++ template with a local variable to keep track of the CPU core it is mapped to. This variable has been replaced with a heterogeneous task information structure to keep track of which processing element, platform, and thread index a heterogeneous task is mapped to. The only difference in how the algorithm developer uses a task is that instead of being mapped to CPU cores with a list of thread indices, a heterogeneous task is mapped with a list of heterogeneous task information structures. This mapping is still done when the task is declared in the `main` function. An example of additional heterogeneous task mappings and example implementation can be found

in Figure 3.2 (changes highlighted). Compared to the original task implementation shown in Figure 2.5, little change is needed to move to a heterogeneous task.

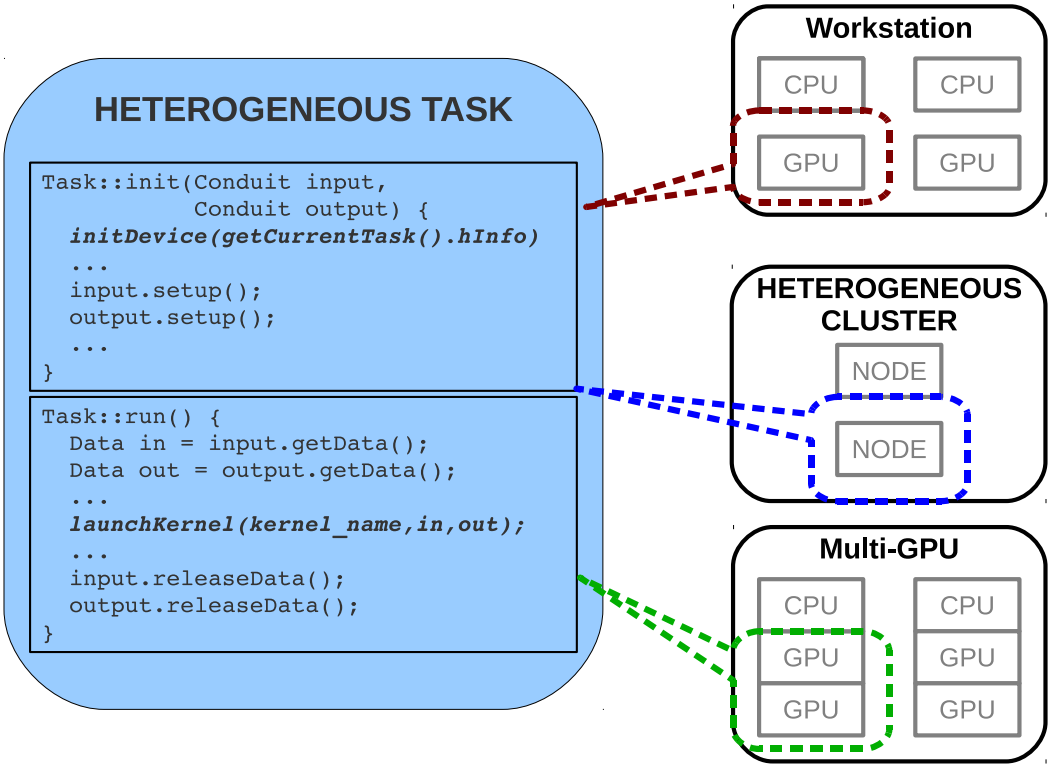

Figure 3.2: Example processing element mappings for a heterogeneous task

A task is implemented as a C++ template that accepts as a template argument a class implementing the `init` and `run` functions. The implementation of these functions changes very little between the CPU-based task and the heterogeneous task. The only alteration to the `init` function is a call to the heterogeneous utility function `initDevice`. The change to the `run` function is a call to the `launchKernel` in place of simply calling a C/C++ function. Neither of these changes requires user interaction, keeping all of the developer programming in the `main` function. Tasks

can be mapped to one or more CUDA streams or OpenCL command queues. This allows a singel heterogeneous task to execute on one or more GPU devices, and for one or more heterogeneous tasks to execute on a single GPU device.

3.1.3 Heterogeneous Conduit

The heterogeneous conduit is a software abstraction for data allocation, movement, and synchronization between two or more heterogeneous tasks. Conduits maintain the same `Writer` (insertion) and `Reader` (extraction) interfaces as the original conduits, but implement all of the memory functions by interfacing with the heterogeneous utility functions. The crux of adapting the conduits to heterogeneous architectures is determining the proper interface between the end points of the conduit. Thus, when a task calls the `setup` function on one of the conduit interfaces to initialize an end point, the heterogeneous task information structure is passed to the conduit. This allows the conduit, once initialized, to recognize the configuration needed to move and synchronize data between endpoints. A list of the supported conduit configurations can be found in Table 3.1.3. Each endpoint or immediate interface is a location for a memory buffer to be allocated. At each location, the conduits can allocate multiple buffers to establish a *queue\FIFO* of buffers that can be used to accumulate multiple data sets. This accounts for the case where a source task may be producing data at a greater rate than a destination task is consuming it.

As can be seen from the list, some of the conduit configurations require an in-

Source Endpoint	Intermediate Interface	Destination Endpoint
CPU		CPU
CPU		CUDA Device
CPU		OpenCL Device
CUDA Device		CPU
OpenCL Device		CPU
CUDA Device A	CPU	CUDA Device B
OpenCL Device A	CPU	OpenCL Device B
CUDA Device	CPU	OpenCL Device
OpenCL Device		CUDA Device
CUDA Device A	*shared*	CUDA Device A
CUDA Device B	*shared*	CUDA Device B
OpenCL Device A	*shared*	OpenCL Device B

Table 3.2: All supported non-distributed memory heterogeneous conduit configurations

termediate set of memory buffers to move data between devices. This is due to constraints of the CUDA and OpenCL programming models. The heterogeneous conduits can also recognize when the source and destination endpoints are on the same device and use a single set of shared memory buffers to eliminate data copies by managing access to the shared set of buffers. All of the data buffers on CUDA or OpenCL devices are allocated in *global* memory by default, but can be designated for *constant* memory by passing a text *string* name to the conduit constructor. Another way of avoiding unnecessary data transfers across conduits, is to inject a data set into the conduit and lock it using the `lock` conduit function. If for example, a conduit carrying input data is streaming new data every iteration of an algorithm but the configuration variables transmitted on another conduit never change, the developer can lock the configuration variables conduit so that the variables are only transmitted once and are not unnecessarily using bandwidth. This is particularly useful

in the case of GPU architectures, where memory bandwidth is often a performance bottleneck. The heterogeneous conduits maintain the same thread and memory synchronization guarantees as the original conduit structures. This ensures that all data allocation, movement, and manipulation occur asynchronously and conflict-free.

3.2 Heterogeneous Applications

In order to construct an application using the heterogeneous tasks and conduits framework, four things are needed in the application's *main* function; initializing the system; declaring your mapped tasks and conduits; initializing the tasks with the appropriate conduit endpoints; and finally, running the tasks and waiting for them to complete execution. An example of the benefits of the tasks and conduits framework is demonstrated by using it to implement the CUFFT application in Listing 3.1. Breaking this application up into independent data initialization, FFT, and data output tasks and using conduits for communication between them results in a multi-threaded application that is much easier to program. Listing 3.3 shows the `main` function for this application.

```
1  int main(int argc, char *argv[]) {
2     // Declare variables
3     cufftHandle plan;
4     int N = 1024, batchSize = 10;
5     int idx, nBytes;
6
7     // Initialize system
8     PvtolProgram prog(argc, argv);
9     initSystem(CUDA_SYS_FLAG);
10    cufftPlan1d(&plan, N, CUFFT_C2C);
11
12    // Create task maps
13    rank.push_back(0);
14    RankList ranks(rank);
```

```
15    TaskMap taskMap(ranks);
16    HeterogeneousMap gpuMap(HTaskInfo{LOC_CUDA, 0, 0, PLAT_NVIDIA});
17
18    // Declare tasks with functions, names, and maps
19    Task<void>  inTask(&inFunc, "IN", taskMap);
20    Task<void>  fftTask(&cufftExecC2C, "FFT", taskMap, gpuMap);
21    Task<void>  outTask(&outFunc, "OUT", taskMap);
22
23    // Declare conduits
24    HeterogeneousConduit<cufftComplex> cdtIn;
25    HeterogeneousConduit<cufftComplex> cdtOut;
26
27    // Initialize tasks with conduit end points
28    inTask.init(N, batchSize, cdtIn.getWriter());
29    fftTask.init(N, batchSize, cdtIn.getReader(),
30                               cdtOut.getWriter());
31    outTask.init(N, batchSize, cdtOut.getReader());
32
33    // Run tasks
34    inTask.run(); fftTask.run(); outTask.run();
35
36    // Wait for tasks to complete
37    inTask.waitTillDone(); fftTask.waitTillDone();
38    outTask.waitTillDone();
39
40    // Cleanup system
41    closeSystem(CUDA_SYS_FLAG);
42    return 0;
43 }
```

Listing 3.3: Simple application initializing data, executing a CUFFT kernel, and analyzing the output data using the heterogeneous tasks and conduits framework

There are a number of benefits to this version of the application separate from the accelerated FFT that the CUFFT library provides. Separating the functional parts of the application into concurrent tasks makes what was a sequential, single threaded application into a streaming multi-threaded application. Each of the tasks will execute concurrently, looping over the `batchSize` sets of data. Due to the fact that the data is now being streamed in smaller blocks, smaller memory buffers on both the host and GPU device are needed, saving memory space. Another advantage to the pvtol tasks and conduits application is that it isolates the data processing from the memory management so that they can be abstracted separately. The use of the

`HeterogeneousConduit` structure to represent data movement and synchronization alleviates the burden of data management from the developer.

Constructing and mapping a task and conduit application to several CPU cores is shown in Figure 2.7. In order to accelerate this application by executing the FFT, FILTER, and IFFT tasks on a GPU device, those tasks must be re-mapped to available CUDA-enabled or OpenCL-enabled devices, as shown in Figure 3.3. The code to construct and map this accelerated application is shown in Listing 3.4. Note that the only changes required are to remap the task to a different platform (in this case CUDA), and point the task to the CUDA kernel function rather than the C/C++ function. This amounts to a change of only 5 source lines of code (SLOC) in addition to the GPU kernel code. Also, mapping the FFT, FILTER, and IFFT functions to the same GPU device will cause the conduits to use a single set of shared data buffers in the conduit between them eliminating the need to perform data copies.

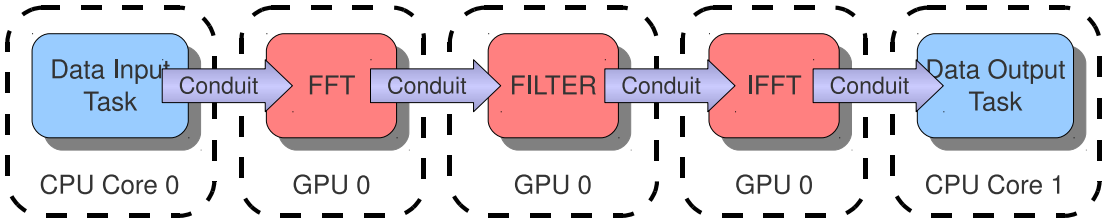

Figure 3.3: Depiction of heterogeneous task and conduit application running an FFT-FLITER-IFFT task pipeline on a GPU

```
1 int main(int argc, char *argv[]) {
2    // Initialize PVTOL and each task map
3    PvtolProgram prog(argc, argv);
4    rank.push_back(0);
5    RankList ranks(rank);
```

```
6     TaskMap taskMap(ranks);
7     HTaskInfo hInfoGPU = {LOC_CUDA, 2, 0, PLAT_NVIDIA};
8     HeterogeneousMap gpuMap(&hInfoGPU, 1);
9
10    // Declare each task and conduit
11    Task<DIT>      dit( "DIT",    taskMap);
12    Task<FFT>      fft( "FFT",    taskMap, gpuMap);
13    Task<FILT>     filt("FILTER", taskMap, gpuMap);
14    Task<IFFT>     ifft("IFFT",   taskMap, gpuMap);
15    Task<DOT>      dot( "DOT",    taskMap);
16    HeterogeneousConduit<float> cdt0("DIT to FFT");
17    HeterogeneousConduit<float> cdt1("FFT to FILTER");
18    HeterogeneousConduit<float> cdt2("FILTER to IFFT");
19    HeterogeneousConduit<float> cdt3("IFFT to DOT");
20
21    // Initialize the tasks with their conduits
22    dit.init(cdt0.getWriter());
23    fft.init(cdt0.getReader(), cdt1.getWriter());
24    filt.init(cdt1.getReader(), cdt2.getWriter());
25    ifft.init(cdt2.getReader(), cdt3.getWriter());
26    dot.init(cdt3.getReader());
27    // Run the tasks
28    dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();
29    // Wait until tasks complete
30    dit.waitTillDone(); fft.waitTillDone();
31    filt.waitTillDone(); ifft.waitTillDone();
32    dot.waitTillDone();
33 }
```

Listing 3.4: Code to construct and execute the heterogeneous task and conduit application in Figure 3.3

The re-mapping of an application shown in Figures 2.7 and 3.3 is just one example of how to port a task and conduit application to a different computing architecture. In this case some tasks are ported from a CPU to a GPU. Tasks can also be ported from a single instance to multiple parallel instances. In Figure 3.4, the FFT-FILTER-IFFT sequence of tasks are each mapped to two CUDA GPU devices present on the system. These tasks will execute concurrently on two GPU devices, and can either execute on twice as much data in approximately the same amount of time (weak scaling) or execute on the same amount of data in approximately half the time (strong scaling). Regardless of the type of scalability employed, the SLOC needed to make this change is minimal (3 SLOC), as shown in Listing 3.5.

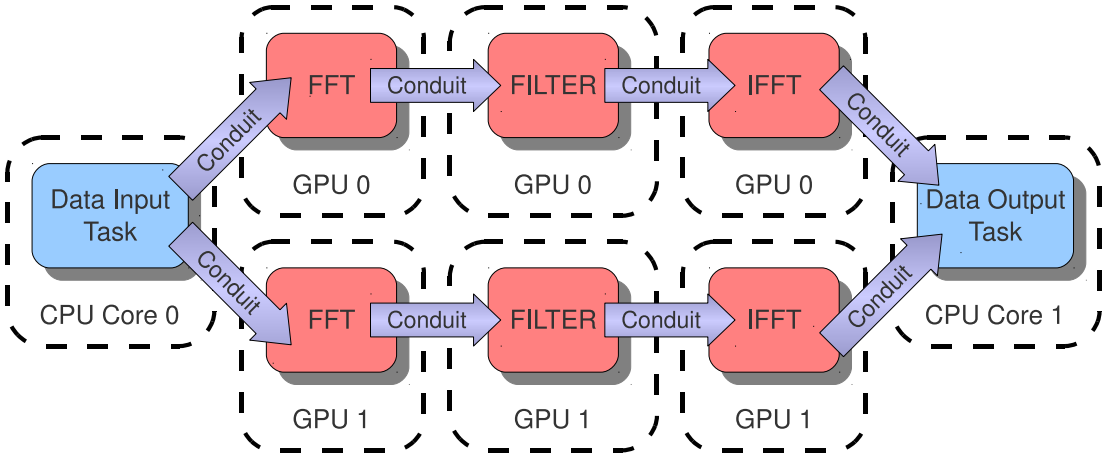

Figure 3.4: Previous FFT-FLITER-IFFT task pipeline mapped to use multiple parallel GPU tasks

```
1  int main(int argc, char *argv[]) {
2     // Initialize PVTOL and each task map
3     PvtolProgram prog(argc, argv);
4     rank.push_back(0);
5     RankList ranks(rank);
6     TaskMap taskMap(ranks);
7     HTaskInfo hInfoGPU1 = {LOC_CUDA, 1, 0, PLAT_NVIDIA};
8     HTaskInfo hInfoGPU2 = {LOC_CUDA, 2, 0, PLAT_NVIDIA};
9     HTaskInfo * hInfos[2] = {hInfoGPU1, hInfoGPU2};
10    HeterogeneousMap gpuMap(hInfoGPU1, 2);
11
12    // Declare each task and conduit
13    Task<DIT>      dit( "DIT",    taskMap);
14    Task<FFT>      fft( "FFT",    taskMap, gpuMap);
15    Task<FILT>     filt("FILTER", taskMap, gpuMap);
16    Task<IFFT>     ifft("IFFT",   taskMap, gpuMap);
17    Task<DOT>      dot( "DOT",    taskMap);
18    HeterogeneousConduit<float> cdt0("DIT to FFT");
19    HeterogeneousConduit<float> cdt1("FFT to FILTER");
20    HeterogeneousConduit<float> cdt2("FILTER to IFFT");
21    HeterogeneousConduit<float> cdt3("IFFT to DOT");
22
23    // Initialize the tasks with their conduits
24    dit.init(cdt0.getWriter());
25    fft.init(cdt0.getReader(), cdt1.getWriter());
26    filt.init(cdt1.getReader(), cdt2.getWriter());
27    ifft.init(cdt2.getReader(), cdt3.getWriter());
28    dot.init(cdt3.getReader());
29    // Run the tasks
30    dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();
31    // Wait until tasks complete
32    dit.waitTillDone(); fft.waitTillDone();
33    filt.waitTillDone(); ifft.waitTillDone();
34    dot.waitTillDone();
35 }
```

Listing 3.5: Code to construct and execute the application in Figure 3.4

Another means of parallelizing tasks is to map them to different distributed compute nodes in a cluster. This mapping functionality was present in the original PVTOL tasks and conduits framework. In order to map a task to two compute nodes, it only needs a task map with two ranks as opposed to the one (assigned in lines 4-6 of Figure 3.5). Adding ranks to the task maps of heterogeneous tasks will launch concurrent instances on distributed compute nodes of a cluster, and those tasks will implement their heterogeneous map on each of those compute nodes. This extends the framework's capabilities to map to a wide range of heterogeneous compute nodes.

Chapter 4

Experimental Setup and Results

This section presents the experimental setup and results of our research, first describing the acceleration of the example applications using the PVTOL tasks and conduits framework, then going over the various computing architectures that were used to run the applications utilizing the tasks and conduits framework. We then present the results regarding performance, portability, and framework overhead related to those applications in various configurations on each of the applicable computing architectures.

4.1 Accelerated FMT Application

We have applied the heterogeneous tasks and conduits framework to the Fluorescence Mediated Tomography application, where Monte Carlo simulations have shown to yield excellent approximations of early photon propagation[28]. The application organization is shown in in Figure 4.1, where the FMT algorithm has been adapted for our purposes of tacking photon paths and ported for execution on an NVIDIA GPU using CUDA[6]. The original MCML code accurately models photon propaga-

tion through biological tissue, recording the amount of energy absorbed or reflected at each step[41]. For our purposes, we do not need to know the energy absorbed by the tissue, but the absolute path traveled by the photons that are successfully transmitted from the source to the detector. We altered the MCML code to track photon paths instead of light absorption using different global data structures with little to no effect on the CPU-based algorithm's performance.

In order to adapt this algorithm to the CUDA platform, we first had to establish how it would be parallelized. There are two primary considerations when parallelizing an algorithm of this type, computation and data access. The Monte Carlo based FMT application simulates millions, or even billions, of independent photons whose propagation through a medium is based off pseudo random numbers that are generated on the GPU device. Because of this, each photon merely needs a set of static input parameters and random number generator seeds to begin execution. The photons can all execute independent of each other, so there are no data dependencies amongst threads. These two conditions make this application easy to parallelize. An efficient parallel pseudo random number generator (PRNG) can be plugged into the implementation to avoid developing one from scratch[23]. The final step of the algorithm is writing the paths of valid photons to the output data structure. This requires storing both the output data structure and the photon paths. The simplest solution to this problem was to isolate all of the different data objects into their most appropriate CUDA memory spaces and execute each photon as a separate GPU

thread. Table 4.1 depicts the GPU memory requirements of the FMT application.

Data Structure	Data Type	Quantity	Size	GPU Memory
Input Parameters	read-only structure	1	120B	Constant
PRNG Seeds	read-only structure	1 per thread	16B	Global
Photon Paths	read-write array of 15K	1 per thread	60KB	Global
Output Grid	read-write 3D array	1	Varies	Global
Photon Data	read-write structure	1 per thread	36B	Local
Local PRNG Seeds	read-write structure	1 per thread	16B	Local

Table 4.1: CUDA-based 1-stage FMT application memory requirements

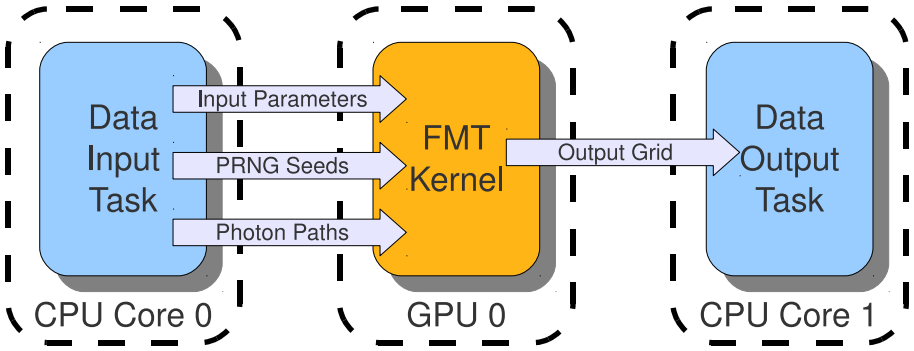

Figure 4.1: Depiction of the one stage CUDA-based FMT application

The tasks and conduits version of the FMT application is shown in Figure 4.1. Because the two input conduits *Input Parameters* and *PRNG Seeds* are read-only and don't change once the application is launched, these conduits are locked to avoid additional memory transfers. This GPU implementation requires that each concurrently executing thread (of `nthreads` total threads) write its path to the *Photon Paths* data structure as it propagates through the medium. Then as a wave of `nthreads` threads finishes, the paths of any photons that have reached a detector are accumulated to the *Output Grid*. This methodology requires a significant number of writes to GPU

global memory, which has a high latency. Many of these writes, for photons whose paths will end up not being accumulated to the global data writes, are unnecessary. Additionally, these writes will necessarily be to non-contiguous locations within the *Photon Paths* and *Output Grid*. This means the memory operations will not be coalesced and thus the latency of GPU global memory accesses will not be hidden. While this application yielded good results (as described in Section 4.4.1), but could be greatly improved by addressing the memory bottleneck.

To address this memory issue, eliminating unnecessary writes to the photon paths was the highest priority. In order to eliminate tracking paths for non-valid photons, the decision was made to first simulate photons to completion and then save the root PRNG seeds of the detected photons. These detected photons are then re-simulated from those root PRNG seeds, accumulating their paths to the *Output Grid* as they propagate through the medium. This eliminates having to write paths to global memory in exchange for some redundant processing. This trade-off of processing versus memory operations will achieve speedups unless an extremely large percentage of simulated photons are detected, which is unlikely for this application domain. Figure 4.2 depicts the two stage FMT application, with its memory requirements listed in Table 4.1. Both stages are located on the same GPU so that the photon packages do not have to be transferred between devices, which would greatly impact the overall application performance[7].

Data Structure	Data Type	Quantity	Size	GPU Memory
Input Parameters	read-only structure	1	120B	Constant
PRNG Seeds	read-only structure	1 per thread	16B	Global
Photon Det Packet	read-write array	1 per detection	24B	Global
Output Grid	read-write 3D array	1	Varies	Global
Photon Data	read-write structure	1 per thread	36B	Local
Local PRNG Seeds	read-write structure	1 per thread	16B	Local

Table 4.2: CUDA-based 2-stage FMT application memory requirements

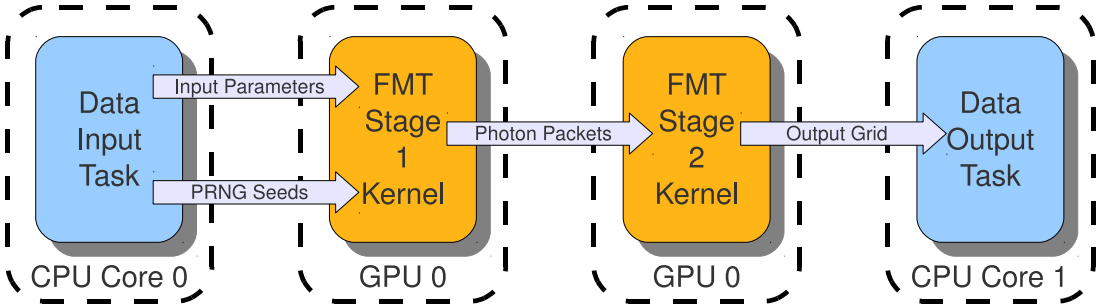

Figure 4.2: Depiction of the two stage CUDA-based FMT application

4.2 Accelerated 3D CBCT Application

The second application adapted for the heterogeneous tasks and conduits framework is written in OpenCL. This algorithm is based on the popular Feldkamp-Davis-Kress algorithm[14], and consists of weighting, filtering, and back projection phases. The algorithm takes a set of 2D projections and parameters describing the geometry of the scan and filtering as inputs, and produces a reconstructed 3D object as output. The weighting phase generates and applies a weighting to each pixel in each projection. The filtering phase applies a 1D filter across each of the 2D projections in the frequency domain, which requires executing and FFT and IFFT in the process. The final phase iteratively applies the filtered projections to the final 3D object. Figure

4.3 shows the tasks and conduits application developed for the OpenCL 3D CBCT algorithm[27], while Table 4.2 depicts the memory requirements for the application.

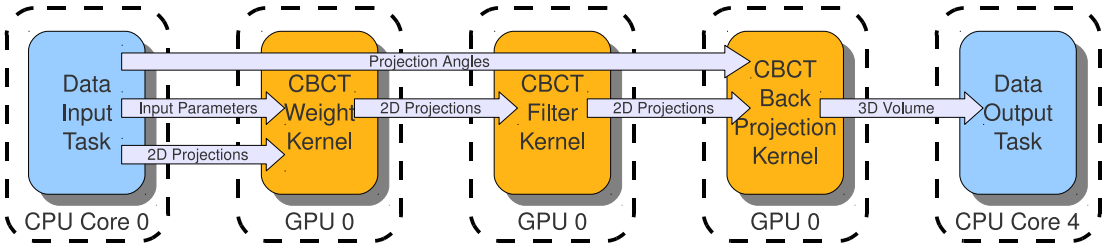

Figure 4.3: Depiction of the OpenCL 3D CBCT application

Data Structure	Data Type	Quantity	Size	GPU Memory
Input Parameters	read-only structure	1	88B	Constant
2D Projections	read-write structure (3x)	1 per angle	varies	Global
Projection Angles	read-only structure	1	1.5KB	Constant
3D Volume	read-write structure	1	varies	Global
Filter Buffer	read-write structure	1	varies	Local
FFT Buffer	read-write structure	1	varies	Local

Table 4.3: OpenCL-based 3D CBCT application memory requirements

Parallelizing the first two phases are relatively easy, since they require little memory and operate on many independent pixels of the 2D input projections. For these phases, we launch as many threads as there are pixels in a single projection so each thread processes independent information for each pixel. These first two phases require some temporary arrays in memory for building filters and FFT output. These arrays are the size of the FFT (equal to the next highest power of two of one of the 2D projection dimensions) and can be stored in local memory for all reasonable data sizes. For the back projection kernel, each voxel of the 3D output volume is processed independently by an OpenCL thread. For each voxel, all of the 2D projections are

looped over to find the values to apply to that voxel as described in the equations of Equation 2.1.

4.3 Experimental Computing Architectures

In this section, we describe the computing architectures used in our experiments. An overview of each of the architectures can be found in Table 4.4, with more detailed descriptions of the architectures below.

System Name	GPU Architecture	Release Year	Number of GPUs	Cores	Software Support	Memory
9800 GX2 Workstation	NVIDIA 9800GX2	2008	2	128	CUDA 4.2 OpenCL 1.1	512 MB per GPU
GTX560 Ti Workstation	NVIDIA GTX560 Ti	2011	1	384	CUDA 4.2 OpenCL 1.1	1 GB
Medusa NVIDIA Node	NVIDIA S1070	2009	8	240	CUDA 4.2 OpenCL 1.1	4 GB per GPU
Medusa AMD Node	AMD Cypress 5870	2010	1	20	OpenCL 1.2	512 MB
Harvard NNIN Cluster	NVIDIA S1070	2009	2 per Node 24 Nodes	240	CUDA 4.2 OpenCL 1.1	1 GB per GPU

Table 4.4: Overview of experimental computing architectures

4.3.1 NVIDIA 9800GX2 Workstation

The first of the computing architectures was used to develop and debug the applications. It contains both CPU and GPU processing elements. This architecture is a single workstation with an Intel®Core™2 Duo CPU running at 3.00GHz, 3.0GB of memory, and running 32-bit Ubuntu 10.04. The workstation has an NVIDIA GeForce 9800GX2 board equipped in it via the PCIe bus, which was released in 2008. The

9800GX2 GPU card has two GPUs, each with 128 CUDA processing cores, and a single 512-bit wide GDDR3 memory interface capable of bandwidth of 128GB/sec (64 GB/sec per GPU) with a total capacity of 512MB. The workstation also has NVIDIA CUDA v4.2 installed, which includes OpenCL v1.1 support for NVIDIA GPU devices.

4.3.2 NVIDIA GTX560 Ti Workstation

This computing platform is a slightly more powerful workstation than the first. The CPU is an Intel®Core™i7-2600K running at 3.00GHz, 16.0GB of memory, and running 64-bit CentOS 5.6. This workstation has an NVIDIA GeForce GTX 560 Ti board equipped in it via the PCIe bus, which was released in 2011 and one of NVIDIA's latest generation GPUs. The GTX 560 Ti GPU card has a single GPU with 384 Fermi (NVIDIA's latest architecture) processing cores, and a 256-bit wide GDDR5 memory interface capable of bandwidth of 128GB/sec with a total capacity of 1.0GB. This workstation has the NVIDIA CUDA v4.2 and AMD APP SDK v2.7 installed, which have support for OpenCL v1.1 and OpenCL v1.2 respectively.

4.3.3 NEU Medusa NVIDIA S1070 Cluster Node

The Medusa cluster is hosted by the Northeastern University Computer Architecture Research Group (NUCAR) and contains a number of heterogeneous cluster nodes that are applicable to the GPGPU computing of the tasks and conduits framework. The first is a compute node with an NVIDIA Tesla S1070 card, which was released in

2009. The node has four Intel®Xeon®quad-core CPUs running at 2.27GHz and two NVIDIA S1070 cards on the PCIe bus. Each Tesla S1070 card contains four Tesla T10 GPUs, each with 4.0GB of memory over a 512-bit wide GDDR3 memory interface capable of a 128GB/sec. The Tesla T10 GPUs each have 240 Tesla processing cores. This node utilizes NVIDIA CUDA v4.2 with OpenCL v1.1 support.

4.3.4 NEU Medusa AMD Cypress 5870 Server

The other Medusa cluster node utilized for testing is the AMD Cypress 5870 node. This node contains the same four Intel®Xeon®quad-core CPUs as the NVIDIA S1070 node, but with an AMD Cypress 5870 GPU card. This card was released in 2010 and contains a single AMD Radeon HD 5870 GPU which has 20 compute units, 512MB of memory over a 256-bit wide GDDR5 memory interface capable of 153GB/sec. This node has the AMD APP SDK v2.7 installed with OpenCL v1.2 support.

4.3.5 Harvard NNIN Cluster (NVIDIA Tesla C1060)

The final computing architecture is a large cluster that is part of the National Nanotechnology Infrastructure Network (NNIN) Computation Project. This cluster contains a large number of compute nodes, but for our purposes we only used the 24 heterogeneous nodes containing GPUs. Each node contains a single Intel®Xeon®quad-core CPU running at 3.00GHz, 16GB of memory and two Tesla T10 GPUs. Each Tesla T10 GPU has the same characteristics as the GPUs in the Medusa NVIDIA

node. This yields a total of 24 nodes and 48 GPUs. These nodes have NVIDIA CUDA v4.0 with OpenCL v.10 support installed in them.

4.4 Results

This section presents the data sets, processing performance, application portability, and framework overhead results of utilizing the tasks and conduits framework with the example applications of Fluorescence Mediated Tomography (FMT) and 3D Cone Beam Image Reconstruction (3D CBCT).

4.4.1 FMT Application

The FMT application tracks photons propagating through a biological medium, such as mouse bodies or slabs of homogeneous medium of various shapes. The scattering behavior of the photons is directly related to the optical properties, size of the medium being used, and the tracking time window in the experiment. This means that the execution time of simulating a set of photons can vary greatly depending on the input parameters of the data set. To exercise the framework with this algorithm, we have chosen a data set that has optical properties typical of living tissue and sizes that are applicable to medical imaging experimentation.

The data sets chosen are homogeneous slabs of tissue with each combination of the optical properties and geometric dimensions listed in Table 4.5. The n value is the refractive index, g is the scattering anisotropy, μ-a is the absorption coefficient, and μ-s is the scattering coefficient. For each pair of optical properties and set

of dimensions, photons were tracked from a single point source to a small (radius = 0.10cm) detector at the opposite end of the slab in the z direction. The photons were tracked for 5.0ns, effectively an infinite amount of time in these scenarios, to reduce the dimensions of freedom of the data sets. The detected photons were collected in bins of 100ps. This variation in geometric size and optical properties gives sufficient variance to the data set such that we can make a reasonable estimate of overall performance of the framework for this application.

Optical Properties (n,g,μ-s,μ-a)	**Geometric Dimensions (x,y,z)**
(1.40, 0.85, 113.333cm^{-1}, 0.150cm^{-1})	(4cm, 4cm, 2cm)
(1.40, 0.85, 87.000cm^{-1}, 0.200cm^{-1})	(8cm, 8cm, 4cm)
(1.40, 0.90, 100.000cm^{-1}, 0.180cm^{-1})	(6cm, 6cm, 6cm)
(1.40, 0.90, 95.000cm^{-1}, 0.200cm^{-1})	(4cm, 4cm, 8cm)

Table 4.5: Optical properties and geometry dimensions of the data sets used for FMT tasks and conduits application testing. The geometries all used 1.0mm voxel sizes.

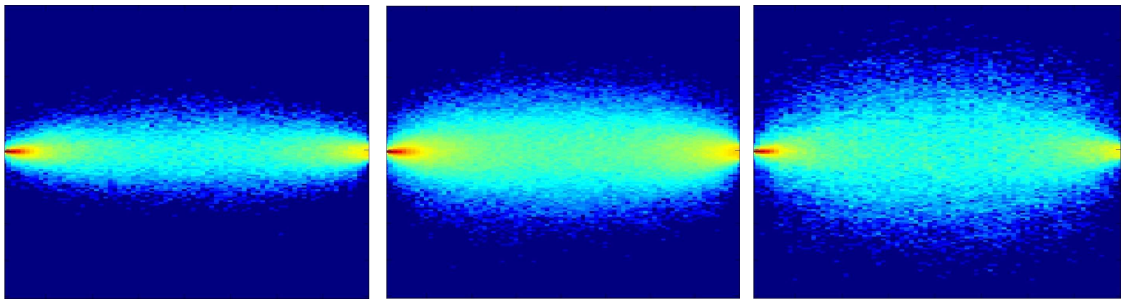

Figure 4.4: Photon propagation output of the PVTOL GPU FMT application for early(a), middle(b) and late(c) time bins through a homogeneous medium

Some example output of the FMT algorithm is shown in Figures 4.4 and 4.5. In each figure, the photon paths from source to detector are shown for early, middle, and late arriving photons, showing the expansion of the path lengths as more time

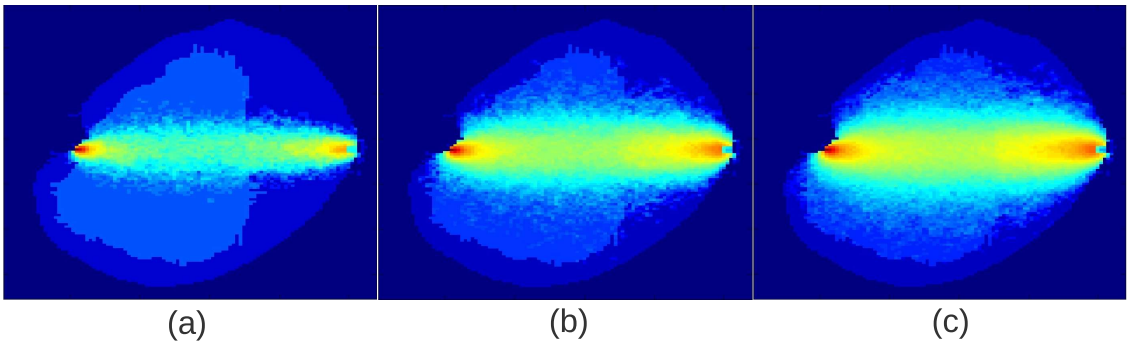

Figure 4.5: Photon propagation output of the PVTOL GPU FMT application for early(a), middle(b), and late(c) time bins through a mouse body

is allowed for them to propagate. The intensity images show a 2D slice of the 3D space along the line from the source to the detector based on the number of photons passing through each voxel of the space. A higher intensity indicates that more of the photons arriving at the detector passed through that voxel. The output images show that the earliest arriving photons at the detector, such as in Figure 4.4(a), take a very direct route from source to detector. Photons arriving at the detector in later time bins, take more indirect routes to the detector. The outputs from the GPU and CPU versions of the algorithm are nearly identical, with some very small variance due to the different random number generators used in the two kernels, supporting the consistency of the algorithm implementations across different platforms.

Portability and Performance

The FMT CUDA application (depicted in Figure 4.1) has been applied to the computing architectures listed in Section 4.3. In each case, the application executed 100 streaming iterations of the FMT datasets, each simulating 10e6 photons. In the

cases where multiple parallel instances of tasks are instantiated, the datasets were divided up among the parallel instances. Table 4.6 contains the results of execution of the first attempt at accelerating the FMT application, which consisted of a single stage. The SLOC changes shown in the table are all relative to the optimized single-threaded C version of the algorithm, and do not include the developed kernel code. All times shown are the total application run time, including kernel execution, memory transfer time, and file I/O.

Architecture	**Application Configuration**	**Execution Time (HH:MM:SS)**	**Speedup**	SLOC change
9800GX2 Workstation	1 CPU	85:56:20	1x	N/A
	1 GPU	03:53:41	22x	5
	2 GPU	01:56:14	43x	7

Table 4.6: Photon propagation execution time and SLOC change for one stage FMT algorithm

As can be seen from the table of results, we are able to achieve good speedups with the tasks and conduits framework. On the 9800GX2 architecture, adapting the FMT algorithm to our single stage GPU version obtained speedups of 22x on a single GPU, and 43x on two concurrently executing GPUs. This port from the CPU version to the GPU version required a change in the framework of just 5 SLOC in addition to the GPU kernel. In order to extend this to a second concurrent GPU required only 2 additional SLOC. For porting this application from being CPU-based to being GPU-based, the majority of the speedup came from the implementation of the GPU kernel. There are very important benefits derived from moving the application to the tasks and conduits framework. First, by breaking up the whole application into

different functional tasks the applications achieved task parallelism, executing tasks concurrently that were previously executing serially. Second, the tasks can now be parallelized as in the two GPU case in Table 4.6. Lastly, the only portion of the application that requires development for the CUDA platform is the GPU kernel. All of the memory management and synchronization code does not have to be written by the developer. In this case, this saves approximately 50-100 lines of code and the time needed to design, code, and debug them that would otherwise be necessary to achieve the results shown.

Architecture	**Application Configuration**	**Execution Time (HH:MM:SS)**	**Speedup**	SLOC change
9800GX2 Workstation	1 CPU (1 Thread)	85:56:20	1x	N/A
	1 CPU (4 threads)	09:57:24	1x	N/A
	1 GPU	01:54:10	5.23x	5
	2 GPU	00:58:09	10.27x	8
GTX560 Ti Workstation	1 CPU	06:28:37	1.53x	0
	1 GPU	00:34:44	17.20x	5
NEU Medusa S1070 Node	2 CPU	05:23:38	1.84x	2
	1 GPU	01:10:50	8.43x	5
	2 GPU	00:38:16	15.61x	8
	4 GPU	00:19:58	29.91x	8
	8 GPU	00:11:52	50.34x	8
Harvard NNIN S1070 Cluster	1 GPU	01:26:20	6.92x	5
	2 GPU	00:53:09	11.24x	8
	4 GPU	00:26:20	22.68x	8
	8 GPU	00:13:37	43.87x	8
	16 GPU	00:07:23	80.91x	8
	24 GPU	00:04:35	115.34x	8

Table 4.7: Photon propagation output of the PVTOL GPU FMT application

The step of breaking up the FMT code into a two stage algorithm and running it on the same CPU and GPU architectures yielded even better speedups, as can be seen

from Table 4.7. Very little change to the kernels was needed to develop this version of the algorithm, and it has produced an application with four concurrently executing tasks. The CPU-based version of the two stage FMT tasks and conduits application performs much better than the original single-threaded version. As can be seen from the first two rows of Table 4.7, the multi-threaded version of the 2-stage application produces an almost 10x speedup over the single threaded 1-stage version on the CPU. Thus, all speedup results presented in Table 4.7 are relative to that multi-threaded version. With regards to expanding the application to run concurrent instances of the accelerated FMT tasks, the number of SLOC the developer is required to change is minimal, not exceeding 8 for any case, since a loop can be created to assign tasks to many different GPU devices in the case of the NEU Medusa architecture or heterogeneous nodes in the case of the Harvard NNIN cluster. It should be noted that the changes in SLOC take place in the creation of heterogeneous map and nowhere else in the code.

With regards to speedup there are a number of aspects of the various system architectures that affect the application run time. In the case of the single GPU mappings on all of the architectures with local GPUs, the primary factor in execution time is how well the CUDA kernels execute on the specific GPU architecture. The fastest single GPU execution time is on the GTX560 Ti workstation which contains the most recent iteration of the NVIDIA CUDA architecture. The slowest local GPU execution time is on the 9800GX2 workstation which had the oldest GPU architecture.

The theoretical peak performance of this application is directly proportional to the number of threads that are launched concurrently up to the maximum allowed by the GPU. In this case the FMT application executes 2048 threads in stage one and 32 threads in stage two. The maximum number of threads that can be launched in stage one is dependent on the size of the geometry being used, and how many threads worth of data can fit onto a given device. The greatest performance bottleneck over the entire application is in stage two, where the atomic global memory writes take place. The maximum number of threads capable of executing in stage two is related to the number of detected photons in stage 1, which is related to the scenario parameters and number of photons being simulated. However, some performance improvements that could be made are to launch more threads for stage two, utilizing concurrent thread execution to mask the latency of atomic global memory operations. Another possible solution, since the global memory writes are not coallesced, is to move memory operations to a third stage where the global memory write operations are re-organized and coallesced before being executed.

Another significant factor in application run time is the latency of data transfer between tasks, specifically the host (CPU) tasks and the device (GPU) tasks. When the heterogeneous conduit connects two tasks with a shared memory system, there is only one set of data buffers, and therefore no latency. When a heterogeneous conduit connects two tasks without a shared memory system, the data is copied. Between all of the system architectures utilized here, there are two memory interfaces, the

PCI express bus (PCIe) and InfiniBand networking. The PCIe bus is used in all the interfaces except the cluster, which uses both the InfiniBand networking between nodes and the PCIe bus within each node. This adds significant latency to the remotely accessed GPUs of the Harvard NNIN cluster, preventing linear speedup when doubling the number of concurrent GPUs in the application.

Comparing the speedup of the architectures containing local GPU devices, we see near linear speedup when expanding the application to run concurrent GPU tasks. The incremental speedup becomes less than linear when utilizing the maximum number of devices available on the system due to contention on the PCIe bus for transferring data between the various tasks. We achieve the maximum speedup for local GPU devices, 50.34x using the NEU Medusa S1070 node and running on 8 concurrent GPU devices, with each executing 13 iterations of the FMT data sets. Similarly, as we approach the maximum number of nodes available on the Harvard NNIN cluster, the speedup does not scale linearly achieving a maximum speedup of 120.34x when executing 24 concurrent GPUs. In both cases, scaling up the number of parallel task pipelines requires a change of only 8 SLOC in addition to the GPU kernel code from the CPU version. In addition to scaling the application to execute a set amount of data in less time, the application could just as easily be adapted to operate on much larger data sets and maintain its performance. The speedups achieved from executing on parallel GPU devices within the single NEU Medusa S1070 node are slightly greater than the same number of parallel GPU devices on

the cluster because of the additional overhead incurred from the latency of moving data between nodes using MPI.

4.4.2 3D CBCT Application

The 3D cone beam computed tomography OpenCL application reconstructs a 3D object from a set of 2D projections. In order to test this algorithm, we generated a number of sample Shepp-Logan style phantoms, and then created projections in a rotation around them. These six distinct sets of phantom projections make up the input data sets for the cone beam image reconstruction algorithm, and consist of a wide range of ellipses that are meant to model biological objects, such as organs, bones, etc. Figure 4.6 shows a 2D cross section of three of the phantoms, while Figure 4.7 shows 2D cross sections of the projections corresponding to phantoms. Each of the 3D phantoms have dimensions (64,60,50), the projections have dimensions (64,60,72), and then the output images will attempt to reconstruct the original (64,60,50) phantoms.

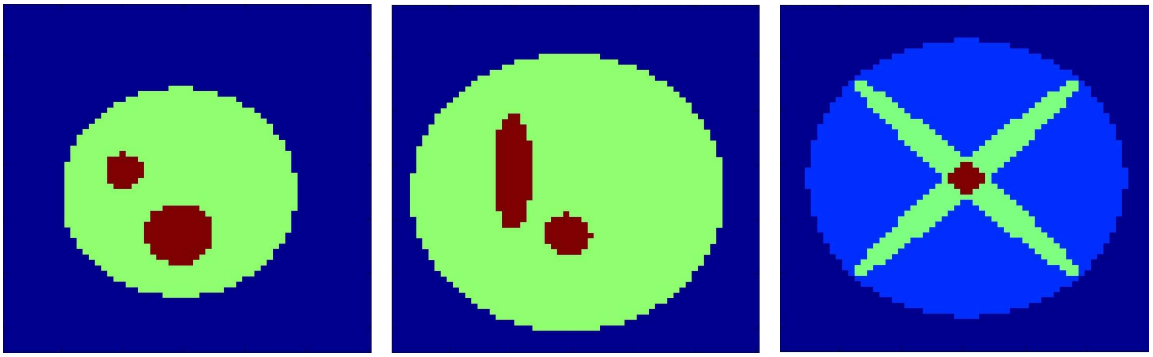

Figure 4.6: Cross sections of 3D cone beam phantoms

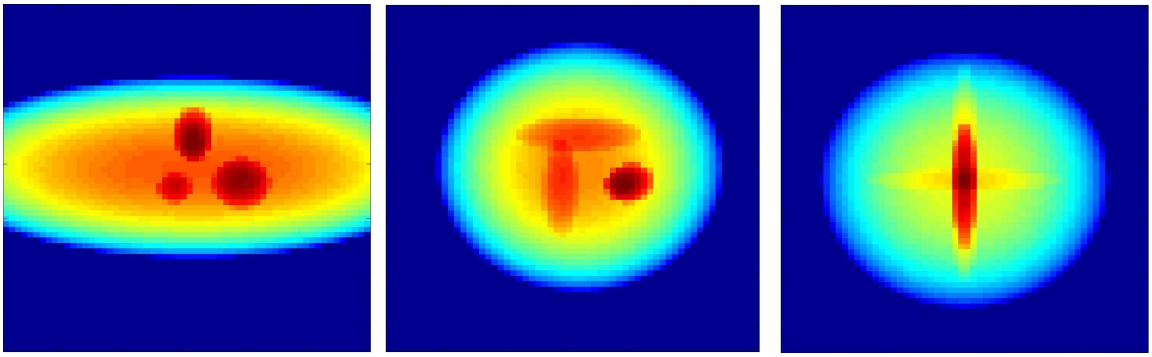

Figure 4.7: Cross sections of 3D cone beam phantom projection data sets

In order to stress the heterogeneous tasks and conduits 3D CBCT application, each run of the application streams 1000 data sets, cycling over the six phantoms and processing them. Figure 4.8 shows example output slices of the corresponding reconstructed objects. While the color scales appear different from the original phantoms, the floating point error of the reconstructed images is $< 0.01\%$ and well within reasonable bounds to consider the reconstruction successful.

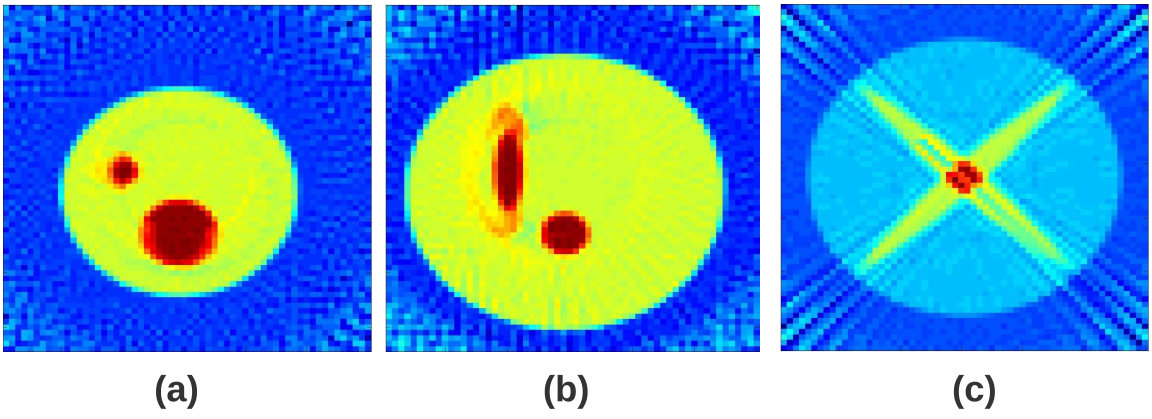

Figure 4.8: Cross sections of the image reconstructions of 3D cone beam phantoms

Portability and Performance

The 3D CBCT OpenCL application shown in Figure 4.3 has been applied to the computing architectures listed in Table 4.3. Similar to FMT results, when parallel instances of the 3D CBCT algorithm are instantiated the 1000 total data sets are divided amongst the multiple parallel instances. Table 4.8 contains the results of executing the CBCT application, where all three processing stages (weighting, filtering, and back projection) are assigned to the same GPU to take advantage of shared memory buffers between kernels. The SLOC changes shown in the table are all relative to the single-threaded C version of the algorithm, and all times shown are the total application run time, including kernel execution, memory transfer time, and file I/O.

The different tasks of the 3D cone beam CT algorithm can be ported from a multi-threaded CPU implementation to an OpenCL GPU implementation with a change of just 6 SLOC in addition to the OpenCL kernels. Again, only changing lines of code in the heterogeneous mapping where the application is constructed. From the table it can be seen that doing this, we have achieved a 20x speedup when porting to an older generation NVIDIA GPU (9800GX2), and 44x speedup when porting the application to the latest generation NVIDIA GPU (GTX560 Ti). An additional 3 SLOC change is needed to expand this application to run multiple parallel GPU instances. Increasing the number of GPUs running the application, we have shown speedups of 197x for 8 local GPUs and 315x for 16 distributed GPU nodes.

Architecture	**Application Configuration**	**Execution Time (HH:MM:SS)**	**Speedup**	SLOC change
9800GX2 Workstation	1 CPU	01:45:03	1x	N/A
	1 GPU	00:05:10	20x	6
	2 GPU	00:03:03	34x	9
GTX560 Ti Workstation	1 GPU	00:02:22	44x	6
NEU Medusa S1070 Node	1 GPU	00:02:46	38x	6
	2 GPU	00:01:35	66x	9
	4 GPU	00:00:57	110x	9
	8 GPU	00:00:32	197x	9
NEU Medusa 5870 Node	1 GPU	00:02:01	52x	6
Harvard NNIN S1070 Cluster	1 GPU	00:03:01	35x	6
	2 GPU	00:01:52	56x	9
	4 GPU	00:00:59	107x	9
	8 GPU	00:00:34	185x	9
	16 GPU	00:00:20	315x	9

Table 4.8: Photon propagation output of the PVTOL GPU 3D CBCT application

The speedup results of the 3D CBCT algorithm exhibit much the same behavior as the speedups of the FMT application when the number of parallel instances are increased. This is due to the framework overhead as well as the latencies and capacities of the PCIe and InfiniBand interfaces used in the NEU Medusa S1070 node and the Harvard NNIN Cluster, respectively. The task parallelism in these experiments comes with minimal developer effort, requires many fewer lines of code, and have the heterogeneous conduits manage data movement and synchronization in the application.

In this application, the most compute intensive part is the back projection kernel. The weighting and filtering kernels simply apply a weight and frequency-domain filter to each 2D projection. These kernels could see some performance improvement in

performing the FFTs in batches rather than for each individual row of each 2D projection, but all of the global memory operations are coallesced and performed as infrequently as possible. The back projection kernel currently contains three nested loops. The outer loop iterates over the slices in the z direction of the output image, while the second iterates over each pixel per slice. This second loop is parallelized across the execution threads, and ensures that there will be coalesced writes to the output image memory buffer. The inner-most loop of the kernel iterates over each 2D projection to apply them to the output image independently. The efficient use of memory operations has shown us that the application is compute bound and limited by the number of concurrent threads that can be launched on the device without overflowing the register file. The application has good speed up relative to the theoretical peak, but could exhibit greater speed up with larger data sets to operate on. Other possible performance improvements that could be explored are moving the weighting and filtering kernels to the CPU for computation to free up GPU resources, and by removing the inner-most loop of the back projection kernel and streaming the 2D projections.

4.4.3 Framework Overhead

With regards to the framework's overhead, it is strongly dependent on a number of aspects of the application a developer develops with the framework. The data set size, number of processing iterations, number of concurrent tasks, distribution and locality of tasks and memory buffers, as well as other factors will greatly affect the

amount of processing overhead incurred from using the tasks and conduits framework. It is designed to only utilize programming models and constructs necessary to the application, and incur a minimal amount of processing overhead in addition to those. For example, for a CUDA-based application containing tasks that are on a single compute node, the only programming models that are required are pthreads and CUDA, so the other programming models like OpenCL and MPI will not be initialized and used. For an application that uses distributed compute nodes the overall overhead will increase by the overhead incurred from using the MPI interface.

In order to demonstrate the overhead incurred by the framework, we compared running the FMT application versus an asynchronous version of the FMT application written with CUDA and `pthreads` and using the same kernels as the tasks and conduits application. The difference in processing between the two should be in the PVTOL tasks and conduits framework, and not in the GPU kernels, memory operations, or `pthreads` library. Figure 4.9 graphs the percent overhead of the framework versus the number of data processing iterations, each of which simulates 1K photons. We chose a relatively small number of photons as to expose the framework overhead. The amount of overhead incurred will vary a great deal for different applications, but should level off at some approximate value for a large enough number of iterations. As can be seen from Figure 4.9, the overhead of the framework becomes an additional 2% of the total execution time when running greater than 100 iterations of 1K photons. This is where the overhead levels off as it asymptotically approaches

the percent overhead incurred per iteration.

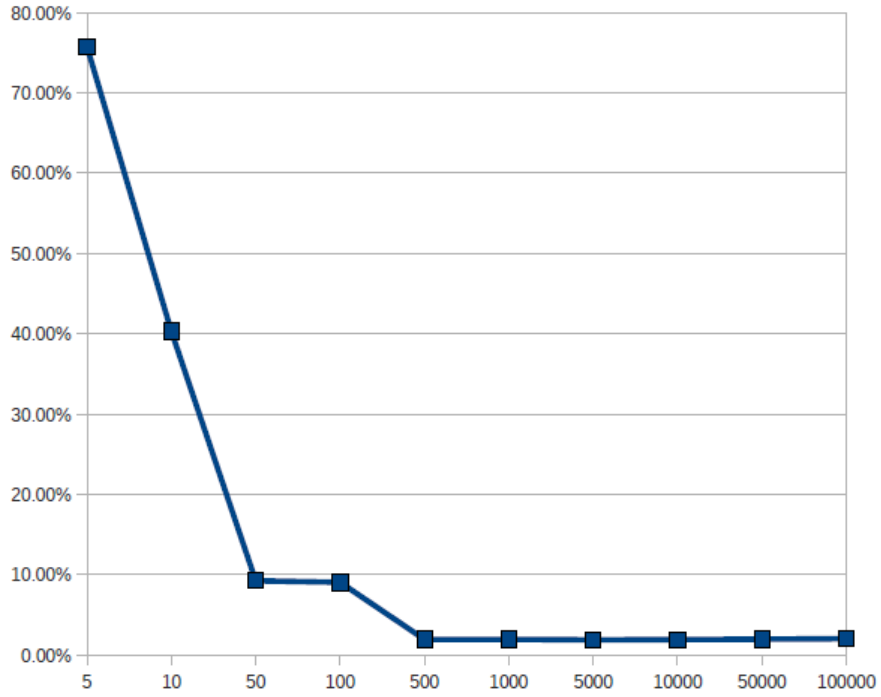

Figure 4.9: Percent difference in execution time of the two asynchronous versions of the FMT application versus number of data processing iterations, running 1K photons per iteration.

It should also be noted that while 2% overhead is incurred, this simplest version of the FMT application written using the tasks and conduits framework requires writing 200 fewer lines of code and the developer needs no knowledge of the `pthreads` programming model or the CUDA memory operations. For more complicated examples with parallel tasks, the SLOC savings of using the tasks and conduits framework increases greatly without requiring knowing any additional programming models. We

further show the behavior of the framework overhead in Figure 4.10, where we ran iterations of 10K photons. In this plot, the overhead reaches a low at 1% due to the increase in the CUDA kernel run time relative to the framework overhead incurred. This indicates that for sufficiently large data sizes, the framework overhead will approach 0%.

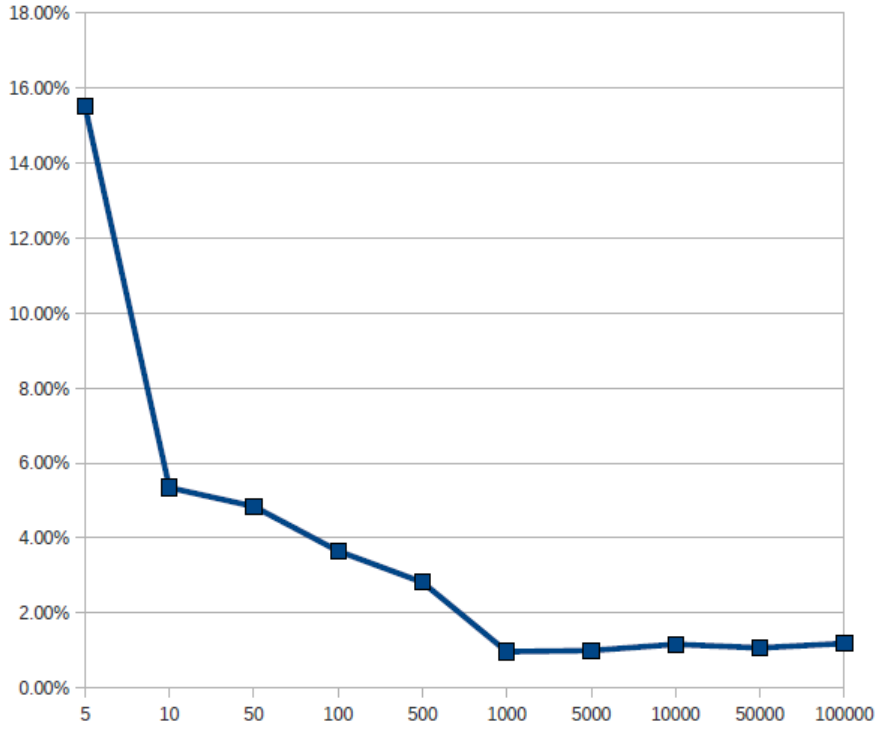

Figure 4.10: Percent difference in execution time of the two asynchronous versions of the FMT application versus number of data processing iterations, running 10K photons per iteration.

The differences in run times shown in Figure 4.9 and Figure 4.10 are the trade off associated with the other benefits of the tasks and conduits framework. This

minimal amount of overhead saves significant amounts of development and debugging time and effort by abstracting away the programming models and code that are necessary to create an asynchronous multi-threaded application. Also, the tasks and conduits framework exposes the task mapping that allows developers to easily parallelize and scale up applications without writing additional code. This makes changing the organization, distribution, and mapping of heterogeneous applications extremely simple. Enabling the different aspects of the frameworks behavior, such as multi-buffering and removing unnecessary data transfers to easily explore a very large application design space and achieve large speedups with minimal consideration of the underlying memory hierarchy and computing architecture.

Chapter 5

Conclusions

This research has successfully extended the PVTOL tasks and conduits framework to support heterogeneous processing architectures through the Compute Unified Device Architecture (CUDA) and Open Compute Language (OpenCL) programming models. A common interface between the tasks and conduits framework and the underlying programming models (heterogeneous utility functions) has been developed to keep all platform-dependent functionality separated from the higher-level behavior of the framework. This enables the use of many different programming models and libraries, the extensibility of the framework to add more programming models as needed, and the maintainability of the software's platform independent features.

In addition, the heterogeneous utility function interface allows tasks and conduits to perform proper and efficient interfacing between different programming models in a manner that is completely transparent to the user. The utility functions perform many platform-specific functions and error checking, while maintaining an API that makes it easy to include interfaces to additional programming models in the future.

Also, the addition of the Heterogeneous Map structure enables the developer to map heterogeneous tasks to the different types of processing elements and programming models available in a system. The support of heterogeneous programming models has further extended the portability of applications written using the tasks and conduits framework, enabling the types of versatile portability shown in the applications of Figure 4.1, Figure 4.2, and Figure 4.3.

We have demonstrated significant speedups and portability of two applications using the tasks and conduits framework. Both the Monte Carlo FMT photon transport and 3D cone beam image reconstruction algorithms exercised a great deal of the supported configurations and features of the tasks and conduits framework, using CUDA and OpenCL respectively. Accelerating the FMT application, we have achieved speedups of 50x and 115x on computing platforms containing 8 local GPUs and 24 distributed GPUs respectively with no more than 8 SLOC in addition to the CUDA kernel code required when porting from the original CPU version. The output of the accelerated FMT application has been utilized to produce output for verifying experimental data in a number of scenarios[39]. By accelerating the 3D CBCT application, we have achieved speedups of 197x and 315x on computing platforms running 8 local GPUs and 16 distributed GPUs respectively. Porting the FMT application from the CPU-based version to the accelerated version required no more than 9 SLOC change in addition to the OpenCL kernel code. These results successfully demonstrate that the heterogeneous tasks and conduits framework can be used to

achieve significant speedups of applications on diverse computing architectures with minimal developer effort.

5.1 Future Work

From the current state of the tasks and conduits framework, there are two obvious directions to take future work. One is to concentrate on the task and conduit support of additional heterogeneous architectures and features to make the framework more comprehensive. The other is to build up higher-level support to make use of the framework easier for developers.

To address the first point, the tasks and conduits framework contains integrated support for OpenMPI, CUDA, OpenCL, and standard C/C++ programming models, but has not yet tested all of the possible configurations of those models. While the framework is capable of keeping track of information such as memory copy latencies and whether memory buffers have waiting data or not, it does not yet make any complex decisions about work distribution, load balancing, or run time re-mapping of tasks. The research to date has focused on the framework's functionality and correctness on different platforms. Researching and implementing appropriate load balancing and the ability to re-map tasks to processing elements at run time would enable the framework to achieve much greater performance on very complex and distributed systems. The run time re-mapping of tasks in itself would be novel, and allow the framework to account for failures and imbalances or bottlenecks in work

loads.

The second area of future work deals with developer usability. Currently, the developer constructs, initializes and runs their application in the *main* function. The important pieces of information for the framework is the mapping of processing elements and functions to tasks, and the connection of conduits among them. With this information, it is possible to construct the code for each task and approximately half of the code in the *main* function automatically. For many cases, it would be trivial to generate the conduit connections between tasks from the input and output arguments specified by each task's function(s). This means that instead of writing the *main* function, a more user-friendly interface could be developed for the user to only specify relevant information and not write any code other than the necessary core functions. This interface could be taken in a number of directions, either as a simple comma separated value input file or a GUI.

Appendix A

Appendixes

A.1 Heterogeneous PVTOL Objects

A.1.1 Heterogeneous Map

```
1  /**
2   *  File: HeterogeneousMap.h
3   *
4   *  \file    HeterogeneousMap.h
5   *  \author  $LastChangedBy: $
6   *  \date    $LastChangedDate: $
7   *  \version $LastChangedRevision: $
8   *  \brief   Definition of the HeterogeneousMap class.
9   *
10  *  Author: James Brock
11  *
12  *  $Id: $
13  */
14 #ifndef PVTOL_HETEROGENEOUSMAP_H_
15 #define PVTOL_HETEROGENEOUSMAP_H_
16
17 namespace ipvtol {
18
19   /// \brief class HeterogeneousMap
20   /// The HeterogeneousMap class is used to provide mapping information for
21   /// Heterogeneous Tasks.  It inherits from the abstract base Map class.
22   class HeterogeneousMap {
23
24   public:
25     /// \brief Default Constructor
26     HeterogeneousMap();
27
28     /// \brief Constructor
29     HeterogeneousMap(const HTaskInfo * ranks, const int size);
30
31     /// \brief Copy contructor
32     HeterogeneousMap(const HeterogeneousMap & other);
33
34     /// \brief Destructor
35     ~HeterogeneousMap();
```

```
36
37      /// \brief setMap function
38      void setMap(vector<HTaskInfo> map);
39
40      /// \brief getMap function
41      vector<HTaskInfo> * getMap();
42
43      /// \brief getSize function
44      int getSize();
45
46      /// \brief getTaskInfo function
47      HTaskInfo * getTaskInfo(int idx = 0);
48
49      vector<HTaskInfo> hMap;
50
51    }; // class HeterogeneousMap
52
53    //------------------
54    // INLINE FUNCTIONS
55    //------------------
56
57    // \brief Destructor
58    inline
59    HeterogeneousMap::~HeterogeneousMap() { hMap.clear(); }
60
61    // \brief Default constructor
62    inline
63    HeterogeneousMap::HeterogeneousMap() {
64     static HTaskInfo tmp_info = { LOC_CPU, 0, 0, PLAT_C};
65     hMap.push_back(tmp_info);
66    }
67
68    // \brief Copy contructor
69    inline
70    HeterogeneousMap::HeterogeneousMap(const HeterogeneousMap & other) {
71        hMap = other.hMap;
72    }
73
74    /// \brief Constructor
75    inline
76    HeterogeneousMap::HeterogeneousMap(const HTaskInfo * ranks, const int size) {
77     for (int idx = 0; idx < size; idx++) { hMap.push_back(ranks[idx]); }
78    }
79
80    /// \brief getMap function
81    inline
82    void HeterogeneousMap::setMap(vector<HTaskInfo> map) { hMap.clear(); hMap = map; }
83
84    /// \brief getMap function
85    inline
86    vector<HTaskInfo> * HeterogeneousMap::getMap() { return &hMap; }
87
88    /// \brief getSize
89    inline
90    int HeterogeneousMap::getSize() { return hMap.size(); }
91
92    /// \brief getTaskInfo function
93    inline
94    HTaskInfo * HeterogeneousMap::getTaskInfo(int idx) { return &(hMap[idx]); }
95
96  } // namespace ipvtol
97
98
```

```
99 #endif // PVTOL_HETEROGENEOUSMAP_H_ not defined
```

Listing A.1: Heterogeneous Map object class listing

A.2 Heterogeneous Utility Functions

A.2.1 Heterogeneous Utility Functions API

```
1  /**
2   *
3   *  \file    hUtil.h
4   *  \author  $LastChangedBy: $
5   *  \date    $LastChangedDate: $
6   *  \version $LastChangedRevision: $
7   *  \brief   Function declarations and wrappers for the CUDA Utility functions.
8   *           This file is used to link the CUDA Utility functions to the PVTOL
9   *           Tasks and Conduits framework software and can not contain any
10  *           CUDA specific functions or types.
11  *
12  *  $Id: $
13  *
14  *  Author: James Brock
15  */
16 #ifndef HUTIL_H
17 #define HUTIL_H
18
19 #include <PvtolBasics.h>
20 #include "cpuUtil.h"
21 #include "cudaUtil.h"
22 #include "oclUtil.h"
23
24 #define NO_HSYS_FLAG    0x0
25 #define CUDA_SYS_FLAG   0x10
26 #define OCL_SYS_FLAG    0x20
27
28 ////////////////////// FUNCTION DECLARATIONS ////////////////////////////
29
30 // System and device functions
31 extern "C" void initSystem(  int flags );
32 extern "C" void initDevice(  HTaskInfo * info );
33 extern "C" void closeSystem( int flags );
34
35 // Memory operation functions
36 extern "C" void * initMem(  int * dims, int typeSize, int * stride,
37                             const char * name, HTaskInfo * info, int mapHostFlag );
38 extern "C" void   freeMem(  void * ptr, int * dims, int typeSize, const char * name,
39                             HTaskInfo * info, int mapHostFlag );
40 extern "C" void   clearMem( int * dims, int typeSize, int stride, void * ptr,
41                             HTaskInfo * info);
42 extern "C" void   moveData( void * dst, int dstStride, HTaskInfo * dstInfo,
43                             void * src, int srcStride, HTaskInfo * srcInfo,
44                             int * dims, int typeSize, const char * name);
45 // Kernel build and launch functions
46 extern "C" void build( HTaskInfo * info, char * srcFile);
47 extern "C" void safeBuild( HTaskInfo * info, char * srcFile);
48 extern "C" void launchKernel( const char * krn, int * dims, int nParams,
49                               int * paramSizes, void ** params, int * gDim,
50                               int * bDim, int locMem, HTaskInfo * info);
```

```
51 // Helper functions
52 extern "C" hTaskLoc findLocation(HTaskInfo * infos, int ni);
53
54 ////////////////////// FUNCTION IMPLEMENTATIONS ///////////////////////////
55
56 /* findLocation function */
57 extern "C" inline
58 hTaskLoc findLocation(HTaskInfo * infos, int ni) {
59    int idx = 0;
60    hTaskLoc loc = LOC_INVALID;
61    for (idx = 0; idx < ni; idx++) {
62       if (infos[idx].location > loc) { loc = infos[idx].location; }
63    }
64    return loc;
65 }
66
67 /* initSystem function */
68 extern "C" inline
69 void initSystem(int flags) {
70    if (flags & CUDA_SYS_FLAG) { initCUDA(flags); }
71    if (flags & OCL_SYS_FLAG) { initOpenCL(); }
72    return;
73 }
74
75 /* closeSystem function */
76 extern "C" inline
77 void closeSystem(int flags) {
78    if (flags & CUDA_SYS_FLAG) { closeCUDA(); }
79    if (flags & OCL_SYS_FLAG) { closeOpenCL(); }
80    return;
81 }
82
83 /* initDevice function */
84 extern "C" inline
85 void initDevice(HTaskInfo * info) {
86    hTaskLoc loc = findLocation(info, 1);
87    if (loc == LOC_CUDA) {
88       cudaInitDevice(info);
89    } else if (loc == LOC_OCL) {
90       oclInitDevice(info);
91    } else if (loc == LOC_CPU) {
92       // Do nothing
93    } else {
94       printf("HUTIL ERROR: Invalid heterogeneous device location %s:%d\n",
95              __FILE__,__LINE__);
96    }
97    return;
98 }
99
100 /* initMem function */
101 extern "C" inline
102 void * initMem(int * dims, int typeSize, int * stride, const char * name,
103                HTaskInfo * info, int mapHostFlag) {
104    void * ptr = NULL;
105    hTaskLoc loc = findLocation(info, 1);
106    if (loc == LOC_CUDA) {
107       ptr = cudaInitMem(dims, typeSize, stride, name, info, mapHostFlag);
108    } else if (loc == LOC_OCL) {
109       ptr = oclInitMem(dims, typeSize, stride, name, info, mapHostFlag);
110    } else if (loc == LOC_CPU) {
111       ptr = cpuInitMem(dims, typeSize, stride, name, info, mapHostFlag);
112    } else {
113       printf("HUTIL ERROR: Invalid heterogeneous device location %s:%d\n",
```

```
114             __FILE__,__LINE__);
115    }
116    return ptr;
117 }
118
119 /* freeMem function */
120 extern "C" inline
121 void freeMem( void * ptr, int * dims, int typeSize, const char * name,
122              HTaskInfo * info, int mapHostFlag ) {
123    hTaskLoc loc = findLocation(info, 1);
124    if (loc == LOC_CUDA) {
125       cudaFreeMem(ptr, dims, typeSize, name, info, mapHostFlag);
126    } else if (loc == LOC_OCL) {
127       oclFreeMem(ptr, dims, typeSize, name, info, mapHostFlag);
128    } else if (loc == LOC_CPU) {
129       cpuFreeMem(ptr, dims, typeSize, name, info, mapHostFlag);
130    } else {
131       printf("HUTIL ERROR: Invalid heterogeneous device location %s:%d\n",
132             __FILE__,__LINE__);
133    }
134    return;
135 }
136
137 /* clearMem function */
138 extern "C" inline
139 void clearMem(int * dims, int typeSize, int stride, void * ptr, HTaskInfo * info) {
140    hTaskLoc loc = findLocation(info, 1);
141    if (loc == LOC_CUDA) {
142       cudaClearMem(dims, typeSize, stride, ptr, info);
143    } else if (loc == LOC_OCL) {
144       oclClearMem(dims, typeSize, stride, ptr, info);
145    } else if (loc == LOC_CPU) {
146       cpuClearMem(dims, typeSize, stride, ptr, info);
147    } else {
148       printf("HUTIL ERROR: Invalid heterogeneous device location %s:%d\n",
149             __FILE__,__LINE__);
150    }
151    return;
152 }
153
154 /* moveData function */
155 extern "C" inline
156 void moveData(void * dst, int dstStride, HTaskInfo * dstInfo, void * src,
157              int srcStride, HTaskInfo * srcInfo, int * dims, int typeSize,
158              const char * name) {
159    HTaskInfo infos[2];
160    infos[0] = *srcInfo; infos[1] = *dstInfo;
161    hTaskLoc loc = findLocation(infos, 2);
162
163    if (loc == LOC_CUDA) {
164       cudaMoveData(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,
165                    typeSize, name);
166    } else if (loc == LOC_OCL) {
167       oclMoveData(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,
168                   typeSize, name);
169    } else if (loc == LOC_CPU) {
170       cpuMoveData(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,
171                   typeSize, name);
172    } else {
173       printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
174             loc,__FILE__,__LINE__);
175    }
176    return;
```

```
177 }
178
179 /* launchKernel function */
180 extern "C" inline
181 void launchKernel(const char * krn, int * dims, int nParams, int * paramSizes,
182                   void ** params, int * gDim, int * bDim, int locMem,
183                   HTaskInfo * info) {
184    hTaskLoc loc = findLocation(info, 1);
185    if (loc == LOC_CUDA) {
186       cudaLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
187                        locMem, info);
188    } else if (loc == LOC_OCL) {
189       oclLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
190                       locMem, info);
191    } else if (loc == LOC_CPU) {
192       cpuLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
193                       locMem, info);
194    } else {
195       printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
196              loc,__FILE__,__LINE__);
197    }
198    return;
199 }
200
201 extern "C" inline
202 void build(HTaskInfo * info, char * srcFile) {
203    hTaskLoc loc = info->location;
204    if (loc == LOC_OCL) {
205       oclBuild(info, srcFile);
206    } else {
207       printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
208              loc,__FILE__,__LINE__);
209    }
210 }
211
212 extern "C" inline
213 void safeBuild(HTaskInfo * info, char * srcFile) {
214    hTaskLoc loc = info->location;
215    if (loc == LOC_OCL) {
216       oclSafeBuild(info, srcFile);
217    } else {
218       printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
219              loc,__FILE__,__LINE__);
220    }
221    return;
222 }
223
224 #endif
225 /* END UTILITY FUNCTIONS */
```

Listing A.2: Heterogeneous utility function API

A.2.2 CUDA Utility Functions

```
1 /**
2  *
3  *  \author  $LastChangedBy$
4  *  \date    $LastChangedDate$
5  *  \version $LastChangedRevision$
6  *  \brief
```

```
 7  *
 8  *  $Id$
 9  *
10  *  Author: James Brock
11  */
12
13 #include "cudaUtil.h"
14 #define CUDA
15 #include "kernels.h"
16 #undef CUDA
17
18 // Global variables
19 cudaDeviceProp ** devProp;  // CUDA Device propertie structures
20 int nDev;                   // Number of CUDA devices available
21 int cudaSysInit = 0;        // System initialization flag
22 int sysFlags;               // System flags
23
24 /**
25  * cudaCheckErr function
26  *
27  * \brief This function checks to see if any errors have occured in using CUDA, and
28  *        prints out the relevent error information. This function only checks the
29  *        last error to occur.
30  *
31  * \param err The error code to check
32  * \param line The line at which the error code was generated
33  * \param file The file in which the error code was generated
34  * \return None
35  *
36  */
37 extern "C" void cudaCheckErr(cudaError_t err, int line, char * file) {
38    if (err != cudaSuccess) {
39       const char * err_str = cudaGetErrorString(err);
40       printf("CUDA ERROR %d: %s on line %d in file %s\n", err, err_str, line, file);
41       #ifdef KILLONERR
42       exit(err);
43       #endif
44    }
45    return;
46 }
47
48 /**
49  * initCUDA function
50  *
51  * \brief This function will initialize the CUDA system, finding any devices
52  *        available, and initializing any 3rd party libraries requested
53  * \return None
54  *
55  */
56 extern "C" void initCUDA(int flags) {
57    sysFlags = flags;
58    if (cudaSysInit == 0) {
59       cudaCheckErr(cudaGetDeviceCount(&nDev),__LINE__,__FILE__);
60       devProp = (cudaDeviceProp**)malloc(sizeof(cudaDeviceProp*)*nDev);
61       for (int idx = 0; idx < nDev; idx++) {
62          devProp[idx] = new cudaDeviceProp;
63       }
64       if (sysFlags & CUBLAS_FLAG) { /* Init CUBLAS interface */ }
65       if (sysFlags & CUFFT_FLAG) { /* Init CUFFT interface */ }
66       if (sysFlags & CURAND_FLAG) { /* Init CURAND interface */ }
67    }
68    return;
69 }
```

```
70
71 /**
72  * closeCUDA function
73  *
74  * \brief  This function will clean up, releasing and deleting all of the CUDA
75  *         objects that were allocated or initialized during program execution
76  * \return None
77  *
78  */
79 extern "C" void closeCUDA() {
80    if (sysFlags & CUFFT_FLAG) { /* Close CUFFT interface */ }
81    if (sysFlags & CUBLAS_FLAG) { /* Close CUBLAS interface */ }
82    if (sysFlags & CURAND_FLAG) { /* Close CURAND interface */ }
83    // Delete and close main CUDA structures last
84    for (int idx = 0; idx < nDev; idx++) { delete devProp[idx]; }
85    free(devProp); nDev = 0;
86    return;
87 }
88
89 /**
90  * cudaInitDevice function
91  *
92  * \brief  This function selects the coprocessor device and mapping and performs
93  *         any initialization tasks that need to occur.
94  * \param  info Heterogeneous task information structure of the device to be
95  *         initialized
96  * \return None
97  *
98  */
99 extern "C" void cudaInitDevice(HTaskInfo * info) {
100    int idx = 0;
101    cudaError_t err = cudaSuccess;
102    int dev = info->device;
103    int proc = info->process;
104    hTaskLoc loc = info->location;
105
106    // Device Management and initialization
107    if (dev != -1) {
108       err = cudaSetDevice(dev);
109       if (err != cudaSuccess) {
110          printf("CUDAUTIL WARNING: Could not assign to previously initialized \
111                 device %d\n", dev);
112          idx = 0;
113          while ((cudaSetDevice(idx) != cudaSuccess) && (idx < nDev)) { idx++; }
114          if (idx == nDev) {
115             printf("CUDAUTIL ERROR: Could not assign a CUDA device!\n"); exit(-333);
116          } else {
117             cudaCheckErr(cudaGetDevice(&dev),__LINE__,__FILE__);
118             info->device = dev;
119             printf("CUDAUTIL WARNING: Re-assigning to CUDA device %d\n", dev);
120          }
121       } else {
122          printf("Using CUDA Device %d\n", dev);
123       }
124    } else {
125       idx = 0;
126       while ((cudaSetDevice(idx) != cudaSuccess) && (idx < nDev)) { idx++; }
127       if (idx == nDev) {
128          printf("CUDAUTIL ERROR: Could not assign a CUDA device!\n"); exit(-334);
129       } else {
130          cudaCheckErr(cudaGetDevice(&dev),__LINE__,__FILE__);
131          info->device = dev;
132          printf("Assigning to CUDA device %d\n", dev);
```

```
133       }
134    }
135
136    // Process management and initialization
137    if ((proc != -1) && (proc != 0)) {
138       cudaCheckErr(cudaStreamCreate((cudaStream_t*)&proc),__LINE__,__FILE__);
139       info->process = proc;
140       printf("Assigning to CUDA Stream %d\n", proc);
141    } else {
142       info->process = proc;
143    }
144    return;
145 }
146
147 /**
148  * cudaInitMem function
149  *
150  * \brief  This function intelligently allocates memory on the host or device
151  *         as specified and returns pointer to the memory as well as the
152  *         stride for the data, which is used for multi-dimensional data
153  *         blocks.
154  * \param  dims The dimensions of the memory to be allocated
155  * \param  stride The stride (width in bytes) of the data. This only matters
156  *         for multi-dimensional data
157  * \param  loc The location of the memory to be allocated (host or device)
158  * \param  ptr A pointer to where the memory pointer is to be stored
159  * \return None
160  *
161  */
162 extern "C" void * cudaInitMem(int * dims, int typeSize, int * stride,
163                               const char * name, HTaskInfo * info,
164                               int mapHostFlag) {
165    void * locPtr = NULL;
166    size_t free, total = 0;
167    int datSize = 1;
168    int dev = info->device;
169    hTaskLoc loc = info->location;
170
171    // Set to proper device
172    cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__);
173
174    // Get full data size
175    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
176    datSize *= typeSize;
177
178    if (!strcmp(name,"")) { // Data is not a symbol
179       // CHECK 1: Memory requested available in global memory
180       cudaCheckErr(cudaMemGetInfo(&free, &total),__LINE__,__FILE__);
181       if (free > datSize) { // There is adequate available memory
182          // Allocate memory for an object
183          cudaExtent extent = make_cudaExtent(dims[HLENGTH]*typeSize,
184                                              dims[HWIDTH],
185                                              dims[HDEPTH]);
186          cudaPitchedPtr pitchedPtr;
187          cudaCheckErr(cudaMalloc3D(&pitchedPtr, extent),__LINE__,__FILE__);
188          cudaCheckErr(cudaMemset3D(pitchedPtr, 0, extent),__LINE__,__FILE__);
189          locPtr = pitchedPtr.ptr;
190          *stride = pitchedPtr.pitch;
191          cudaCheckErr(cudaMemGetInfo(&free, &total),__LINE__,__FILE__);
192          printf("Allocated %dB of memory on device. %dB remaining of %dB total\n",
193                 datSize, free, total);
194          if (*stride != (dims[HLENGTH]*typeSize)) {
195             printf("CUDAUTIL WARNING: Stride is not the same size as \
```

```
196                     dims[HLENGTH]\n");
197          }
198       } else {
199          printf("CUDAUTIL ERROR: Attempting to allocate %dB of memory when only \
200                  %dB are available out of a total of %dB\n", datSize, free, total);
201          exit(-336);
202       }
203    } else { // Data is a CUDA symbol
204       cudaCheckErr(cudaGetSymbolAddress(&locPtr, name),__LINE__,__FILE__);
205       cudaCheckErr(cudaGetSymbolSize((size_t*)(&dims[HLENGTH]), name),
206                     __LINE__,__FILE__);
207    }
208    cudaThreadSynchronize();
209    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
210    return locPtr;
211 }
212
213 /**
214  * cudaFreeMem function
215  *
216  * \brief  This function frees allocated memory on the host or device
217  *         as specified.
218  * \param  ptr Pointer to the data to be freed
219  * \param  dims The dimensions of the data to be freed
220  * \param  loc The location of the data to be freed
221  * \return None
222  *
223  */
224 extern "C" void cudaFreeMem( void * ptr, int * dims, int typeSize,
225                              const char * name, HTaskInfo * info,
226                              int mapHostFlag ) {
227    hTaskLoc loc = info->location;
228    int dev = info->device;
229
230    if (mapHostFlag) {
231       cudaCheckErr(cudaFreeHost(ptr),__LINE__,__FILE__);
232    } else {
233       cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__);
234       if (!strcmp(name,"")) { cudaCheckErr(cudaFree(ptr),__LINE__,__FILE__); }
235    }
236    cudaThreadSynchronize();
237    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
238    return;
239 }
240
241 /**
242  * cudaClearMem function
243  *
244  * \brief
245  * \param  dims The dimensions of the memory to be allocated
246  * \param  stride The stride (width in bytes) of the data. This only matters
247  *         for multi-dimensional data
248  * \param  loc The location of the memory to be allocated (host or device)
249  * \param  ptr A pointer to where the memory pointer is to be stored
250  * \return None
251  *
252  */
253 extern "C" void cudaClearMem(int * dims, int typeSize, int stride, void * ptr,
254                              HTaskInfo * info) {
255    int datSize = 0;
256    int dev = info->device;
257    hTaskLoc loc = info->location;
258
```

```
259     // Set to proper device
260     cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__);
261
262     // Get full data size
263     for (int idx = 0; idx < HNDIMS; idx++) { datSize += dims[idx]; }
264     datSize *= typeSize;
265
266     // Clear memory on the host for device use
267     cudaExtent extent = make_cudaExtent(dims[HLENGTH]*typeSize,
268                                         dims[HWIDTH],
269                                         dims[HDEPTH]);
270     cudaPitchedPtr pitchedPtr;
271     pitchedPtr.pitch = stride;
272     pitchedPtr.ptr = ptr;
273     pitchedPtr.xsize = dims[HLENGTH];
274     pitchedPtr.ysize = dims[HWIDTH];
275     cudaCheckErr(cudaMemset3D(pitchedPtr, 0, extent),__LINE__,__FILE__);
276     cudaThreadSynchronize();
277     cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
278     return;
279 }
280
281 /**
282  * cudaMoveData function
283  *
284  * \brief  This function intelligently copies data from the source location
285  *         with the specified stride to the destination location with the
286  *         specified stride. The dimensions parameter indicates the
287  *         dimensions of the data and must be the same for both source and
288  *         destination.
289  * \param  dest Pointer to the destination memory location
290  * \param  destStride Stride (width in bytes) of the destination
291  *         memory location
292  * \param  destLoc The location of the destination memory
293  * \param  src Pointer to the source memory location
294  * \param  srcStride Stride (width in bytes) of the source memory
295  *         location
296  * \param  srcLoc The location of the destination memory
297  * \param  dims The dimensions of the data to be copied
298  * \return None
299  *
300  */
301 extern "C" void cudaMoveData( void * dst, int dstStride, HTaskInfo * dstInfo,
302                               void * src, int srcStride, HTaskInfo * srcInfo,
303                               int * dims, int typeSize, const char * name) {
304     hTaskLoc srcLoc = srcInfo->location;
305     hTaskLoc dstLoc = dstInfo->location;
306     int srcDev = srcInfo->device;
307     int dstDev = dstInfo->device;
308     int dev = -1;
309     int proc = -1;
310     cudaMemcpy3DParms cpyParms = {0};
311     cudaStream_t stream;
312     cudaMemcpyKind kind;
313
314     // Set to proper device
315     if (srcLoc == LOC_CUDA) { dev = srcDev; }
316     if (dstLoc == LOC_CUDA) { dev = dstDev; }
317     if (dev != -1) { cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__); }
318
319     if (!strcmp(name,"")) { // conduit is not for a symbol
320        // Determine stream value
321        if ((srcLoc == LOC_CUDA) && (dstLoc == LOC_CPU)) {
```

```
322          proc = srcInfo->process;
323          kind = cudaMemcpyDeviceToHost;
324       } else if ((dstLoc == LOC_CUDA) && (srcLoc == LOC_CPU)) {
325          proc = dstInfo->process;
326          kind = cudaMemcpyHostToDevice;
327       } else if ((dstLoc == LOC_CUDA) && (srcLoc == LOC_CUDA)) {
328          proc = srcInfo->process;
329          kind = cudaMemcpyDeviceToDevice;
330       } else if ((dstLoc == LOC_CPU) && (srcLoc == LOC_CPU)) {
331          proc = srcInfo->process;
332          kind = cudaMemcpyHostToHost;
333       }
334       // Both src and dst are CPUs
335       int stride = (srcStride > dstStride) ? srcStride : dstStride;
336       cpyParms.extent = make_cudaExtent(dims[HLENGTH]*typeSize,
337                                         dims[HWIDTH],
338                                         dims[HDEPTH]);
339       cpyParms.srcPtr.pitch = stride;
340       cpyParms.srcPtr.ptr = src;
341       cpyParms.srcPtr.xsize = dims[HLENGTH]*typeSize;
342       cpyParms.srcPtr.ysize = dims[HWIDTH];
343       cpyParms.dstPtr.pitch = stride;
344       cpyParms.dstPtr.ptr = dst;
345       cpyParms.dstPtr.xsize = dims[HLENGTH]*typeSize;
346       cpyParms.dstPtr.ysize = dims[HWIDTH];
347       cpyParms.kind = kind;
348       cudaCheckErr(cudaMemcpy3DAsync(&cpyParms, (cudaStream_t)(proc)),
349                    __LINE__,__FILE__);
350       cudaStreamSynchronize((cudaStream_t)(proc));
351    } else { // this is a symbol
352       size_t datSize = dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH]*typeSize;
353       if (dstLoc == LOC_CUDA) {
354          cudaMemcpyToSymbol(name, src, datSize, 0, cudaMemcpyHostToDevice);
355       } else if (srcLoc == LOC_CUDA) {
356          cudaMemcpyFromSymbol(dst, name, datSize, 0, cudaMemcpyDeviceToHost);
357       } else {
358          printf("CUDAUTIL ERROR: Invalid device location for symbol\n"); exit(-340);
359       }
360    }
361    cudaThreadSynchronize();
362    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
363    return;
364 }
365
366 /**
367  * cudaLaunchKernel function
368  *
369  * \brief  This function uses the specified parameters to execute the
370  *         coprocessor function specified by kernel. Any parameters to
371  *         the kernel should be included in params.
372  *
373  * \param  kernel A string naming the kernel to execute
374  * \param  dims The dimensions of the data to execute the kernel on
375  * \param  nParams The number of parameters to be passed to the kernel
376  * \param  params Array of parameters to be passed to the kernel
377  * \param  gDim The grid dimensions of the kernel
378  * \param  bDim The block dimensions of the kernel
379  * \param  locMem The amount of local or shared memory to allocate for kernel
380  *         execution
381  * \param  stream The stream index to associate this kernel's execution and
382  *         data with
383  * \return None
384  *
```

```
385  */
386 extern "C" void cudaLaunchKernel(const char * krn, int * dims, int nParams,
387                                   int * paramSizes, void ** params, int * gDim,
388                                   int * bDim, int locMem, HTaskInfo * info) {
389    hTaskLoc loc = info->location;
390    int dev = info->device;
391    int proc = info->process;
392    size_t offset = 0;
393    dim3 gridDims = dim3(gDim[HLENGTH],gDim[HWIDTH],gDim[HDEPTH]);
394    dim3 blockDims = dim3(bDim[HLENGTH],bDim[HWIDTH],bDim[HDEPTH]);
395
396    // Set the device and configure the kernel call to a stream
397    cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__);
398    cudaCheckErr(cudaConfigureCall(gridDims, blockDims, locMem, (cudaStream_t)proc),
399                 __LINE__,__FILE__);
400
401    // For each kernel parameter passed in, push the kernel argument onto the stack
402    //   turn into internal function
403    for (int i = 0; i < nParams; i++) {
404       cudaCheckErr(cudaSetupArgument(params[i], paramSizes[i], offset),
405                    __LINE__,__FILE__);
406       offset = offset + paramSizes[i];
407    }
408
409    printf("Launching %s on dev:%d\n", krn, dev);
410    if (!strcmp(krn,"gpu_mc_stage1")) {
411       cudaCheckErr(cudaLaunch(gpu_mc_stage1),__LINE__,__FILE__);
412    }
413    if (!strcmp(krn,"gpu_mc_stage2")) {
414       cudaCheckErr(cudaLaunch(gpu_mc_stage2),__LINE__,__FILE__);
415    }
416    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
417
418    if ((proc != 0) && (proc != -1)) { cudaStreamSynchronize((cudaStream_t)proc); }
419    cudaThreadSynchronize();
420    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
421    return;
422 }
```

Listing A.3: CUDA utility functions code

A.2.3 OpenCL Utility Functions

```
1  /**
2   *
3   *  \author  $LastChangedBy$
4   *  \date    $LastChangedDate$
5   *  \version $LastChangedRevision$
6   *  \brief
7   *
8   *  $Id$
9   *
10  *  Author: James Brock
11  */
12 #include "oclUtil.h"
13
14 clState glblState = {0};
15 int oclSysInit = 0;
16 char * buildOpts = "-DOCLBUILD -I./platforms/app -I./platforms/util";
17
```

```
18 /**
19  * initOpenCL function
20  *
21  * \brief This function will initialize the OpenCL system, finding any
22  *        platforms and devices available, and printing out a short list of them.
23  *
24  */
25 extern "C" void initOpenCL() {
26    int idx, jdx = 0;
27    size_t psize;
28    char * pstr;
29    cl_int err = CL_SUCCESS;
30    cl_device_type tmptype;
31
32    if (oclSysInit == 0) {
33       // Query and list platforms
34       clCheckErr(clGetPlatformIDs(0, NULL, &(glblState.nplat)),__LINE__,__FILE__);
35       cl_platform_id * plats = (cl_platform_id*)malloc(sizeof(cl_platform_id)*
             glblState.nplat);
36       glblState.plats = (oclPlatInfo*)malloc(sizeof(oclPlatInfo)*glblState.nplat);
37       memset(glblState.plats, 0, sizeof(oclPlatInfo)*glblState.nplat);
38       clCheckErr(clGetPlatformIDs(glblState.nplat,plats,NULL),__LINE__,__FILE__);
39
40       // For each platform, find and list the devices
41       for (idx = 0; idx < glblState.nplat; idx++) {
42          oclPlatInfo * plt = &(glblState.plats[idx]);
43          plt->pid = plats[idx];
44          printf("Found platform %d: 0x%X ==> ", idx, plt->pid);
45          clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, NULL,
46                                       NULL, &psize),
47                     __LINE__,__FILE__);
48          pstr = (char*)malloc(psize);
49          clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, psize,
50                                       pstr, NULL),
51                     __LINE__,__FILE__);
52          printf("%s ", pstr); free(pstr);
53          clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_NAME, NULL,
54                                       NULL, &psize),
55                     __LINE__,__FILE__);
56          pstr = (char*)malloc(psize);
57          clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_NAME, psize,
58                                       pstr, NULL),
59                     __LINE__,__FILE__);
60          printf("%s\n", pstr); free(pstr);
61
62          clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, NULL,
63                                    NULL, &(plt->ndevs)),
64                     __LINE__,__FILE__);
65          cl_device_id * devs = (cl_device_id*)malloc(sizeof(cl_device_id)*plt->ndevs
                );
66          plt->devs = (oclDeviceInfo*)malloc(sizeof(oclDeviceInfo)*plt->ndevs);
67          memset(plt->devs, 0, sizeof(oclDeviceInfo)*plt->ndevs);
68          clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, plt->ndevs,
69                                    devs, NULL),
70                     __LINE__,__FILE__);
71          cl_context_properties cprops[3] = {CL_CONTEXT_PLATFORM,
72                                             (cl_context_properties)(plt->pid), 0};
73          plt->ctx = clCreateContext(cprops, plt->ndevs, devs, NULL, NULL, &err);
74          clRetainContext(plt->ctx);
75          clCheckErr(err,__LINE__,__FILE__);
76
77          // For each device in the platform, print some basic information
78          for (jdx = 0; jdx < plt->ndevs; jdx++) {
```

```
79             oclDeviceInfo * dev = &(plt->devs[jdx]);
80             for (kdx = 0; kdx < MAX_PROCESSES; kdx++) { dev->queues[kdx] = 0); }
81             dev->did = devs[jdx];
82             printf("\tDevice %d (ID:0x%X):", jdx, dev->did );
83             clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, NULL,
84                                        NULL, &psize),
85                        __LINE__,__FILE__);
86             pstr = (char*)malloc(psize);
87             clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, psize,
88                                        pstr, NULL),
89                        __LINE__,__FILE__);
90             printf(" %s", pstr);
91             free(pstr);
92             clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_TYPE,
93                                        sizeof(cl_device_type),
94                                        &tmptype, NULL),
95                        __LINE__,__FILE__);
96             pstr = (char*)malloc(sizeof(char)*STRLEN);
97             clDecodeDeviceType(tmptype, pstr);
98             printf(" (%s)", pstr);
99             free(pstr);
100            clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_VERSION,
101                                       NULL, NULL, &psize),
102                       __LINE__,__FILE__);
103            pstr = (char*)malloc(psize);
104            clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_VERSION,
105                                       psize, pstr, NULL),
106                       __LINE__,__FILE__);
107            printf(" ver. %s\n", pstr);
108            free(pstr);
109         }
110         free(devs);
111      }
112      free(plats);
113   } else {
114      printf("OCLUTIL WARNING: OpenCL system already initialized\n");
115   }
116   return;
117 }
118
119 /**
120  * oclInitDevice function
121  *
122  * \brief  This function selects the coprocessor device and mapping and performs
123  *         any initialization tasks that need to occur.
124  *
125  */
126 extern "C" void oclInitDevice(HTaskInfo * info) {
127    cl_int err = CL_SUCCESS;
128    int didx = info->device;
129    int qidx = info->process;
130    hTaskPlat platName = info->platform;
131    int pidx = clFindPlatform(platName);
132    oclPlatInfo * plat = &(glblState.plats[pidx]);
133    oclDeviceInfo * dev = &(plat->devs[didx]);
134
135    if (dev->queues[qidx] == 0) {
136       dev->queues[qidx] = clCreateCommandQueue(plat->ctx, dev->did, 0, &err);
137       clRetainCommandQueue(dev->queues[qidx]);
138       clCheckErr(err,__LINE__,__FILE__);
139    } else {
140       printf("OCLUTIL WARNING: Device has already been initialized pidx=%d, \
141               didx=%d, %s:%d\n", pidx, didx, __FILE__,__LINE__);
```

```
142    }
143    return;
144 }
145 
146 /**
147  * closeOpenCL function
148  *
149  * \brief  This function will clean up, releasing and deleting all of the OpenCL
150  *         objects that were allocated or initialized during program execution
151  *
152  */
153 extern "C" void closeOpenCL() {
154    int idx, jdx, kdx = 0;
155    int nplat = glblState.nplat;
156 
157    for (idx = 0; idx < nplat; idx++) {
158       oclPlatInfo * plat = &(glblState.plats[idx]);
159       for (jdx = 0; jdx < plat->ndev; jdx++) {
160          oclDeviceInfo * dev = plat->devs[jdx];
161          for (kdx = 0; kdx < dev->nkrn; kdx++) {
162             clCheckErr(clReleaseKernel(dev->kerns[kdx]),__LINE__,__FILE__);
163          }
164          for (kdx = 0; kdx < MAX_PROCESSES; kdx++) {
165             clCheckErr(clReleaseCommandQueue(dev->queues[kdx]),__LINE__,__FILE__);
166          }
167          clCheckErr(clReleaseDevice(dev->did),__LINE__,__FILE__);
168       }
169       clCheckErr(clReleaseProgram(plat->prg),__LINE__,__FILE__);
170       clCheckErr(clReleaseContext(plat->ctx),__LINE__,__FILE__);
171       free(plat->devs);
172    }
173    free(glblState.plats);
174    glblState.nplat = 0;
175    return;
176 }
177 
178 /**
179  * oclInitMem function
180  *
181  * \brief  This function intelligently allocates memory on the host or device
182  *         as specified and returns pointer to the memory as well as the
183  *         stride for the data, which is used for multi-dimensional data
184  *         blocks.
185  *
186  */
187 extern "C" void * oclInitMem(int * dims, int typeSize, int * stride,
188                              const char * name, HTaskInfo * info, int mapHostFlag) {
189    int size = 0;
190    hTaskLoc loc = info->location;
191    int dev = info->device;
192    int proc = info->process;
193    cl_int err = CL_SUCCESS;
194    void * ptr = NULL;
195    int pidx = clFindPlatform(info->platform);
196    cl_context ctx = glblState.plats[pidx].ctx;
197    cl_command_queue queue = glblState.plats[pidx].devs[dev].queues[proc];
198    int fill = 0;
199    size = typeSize*dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH];
200 
201    // If the memory is to be allocated on the host
202    if (loc == LOC_CPU) {
203       if (mapHostFlag) {
204          ptr = clCreateBuffer(ctx, (CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR),
```

```
205                                 size, NULL, &err);
206 #ifdef OPENCL_1.2
207           clCheckErr(clEnqueueFillBuffer(queue, ptr, &fill, sizeof(int), 0, size, 0,
208                                          NULL, NULL),
209                      __LINE__,__FILE__);
210           clFlush(queue);
211 #else
212           memset(ptr, 0, size);
213 #endif
214       } else {
215          ptr = malloc(size);
216          memset(ptr, 0, size);
217       }
218    } else if (loc == LOC_OCL) {
219       if (!strcmp(name,"")) {
220          ptr = clCreateBuffer(ctx, CL_MEM_READ_WRITE, size, NULL, &err);
221       } else {
222          ptr = clCreateBuffer(ctx, CL_MEM_READ_ONLY, size, NULL, &err);
223       }
224       clCheckErr(err,__LINE__,__FILE__);
225    } else {
226       printf("OCLUTIL ERROR: Invalid device location: line:%d file:%s\n",
227              __LINE__,__FILE__);
228       exit(-447);
229    }
230    return ptr;
231 }
232
233 /**
234  * oclFreeMem function
235  *
236  * \brief  This function frees allocated memory on the host or device
237  *         as specified.
238  *
239  */
240 extern "C" void oclFreeMem(void * ptr, int * dims, int typeSize, const char * name,
241                            HTaskInfo * info, int mapHostFlag) {
242    hTaskLoc loc = info->location; //getLocation();
243    if (loc == LOC_CPU) {
244       if (mapHostFlag) {
245          clCheckErr(clReleaseMemObject((cl_mem)ptr),__LINE__,__FILE__);
246       } else {
247          free(ptr);
248       }
249    } else if (loc == LOC_OCL) {
250       clCheckErr(clReleaseMemObject((cl_mem)ptr),__LINE__,__FILE__);
251    } else {
252       printf("OCLUTIL ERROR: Invalid device location: line:%d file:%s\n",
253              __LINE__,__FILE__);
254       exit(-448);
255    }
256    return;
257 }
258
259 /**
260  * oclClearMem function
261  *
262  * \brief  This function will zero out the specified memory buffer
263  *
264  */
265 extern "C" void oclClearMem(int * dims, int typeSize, int stride, void * ptr,
266                             HTaskInfo * info) {
267    int size = 0;
```

```
268    hTaskLoc loc = info->location; //getLocation();
269    int dev = info->device;
270    int proc = info->process;
271    int pidx = clFindPlatform(info->platform); //getPlatform());
272    cl_command_queue queue = glblState.plats[pidx].devs[dev].queues[proc];
273    int fill = 0;
274    size = typeSize*dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH];
275 #ifdef OPENCL_1.2
276    clCheckErr(clEnqueueFillBuffer(queue, ptr, &fill, sizeof(int), 0, size, 0,
277                                   NULL, NULL),
278               __LINE__,__FILE__);
279 #endif
280    clCheckErr(clEnqueueBarrier(queue),__LINE__,__FILE__);
281    clFlush(queue);
282    return;
283 }
284
285 /**
286  * oclMoveData function
287  *
288  * \brief  This function intelligently copies data from the source location
289  *         with the specified stride to the destination location with the
290  *         specified stride. The dimensions parameter indicates the
291  *         dimensions of the data and must be the same for both source and
292  *         destination.
293  *
294  */
295 extern "C" void oclMoveData(  void * dst, int dstStride, HTaskInfo * dstInfo,
296                               void * src, int srcStride, HTaskInfo * srcInfo,
297                               int * dims, int typeSize, const char * name) {
298    hTaskLoc srcLoc = srcInfo->location;
299    hTaskLoc dstLoc = dstInfo->location;
300    int size = typeSize*dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH];
301    int dev, proc, pidx;
302    cl_command_queue queue;
303
304    if ((srcLoc == LOC_CPU) && (dstLoc == LOC_OCL)) {
305       dev = dstInfo->device;
306       proc = dstInfo->process;
307       pidx = clFindPlatform(dstInfo->platform);
308       queue = glblState.plats[pidx].devs[dev].queues[proc];
309       clCheckErr(clEnqueueWriteBuffer(queue, (cl_mem)dst, CL_TRUE, 0, size, src, 0,
310                                       NULL, NULL),
311                  __LINE__,__FILE__);
312    } else if ((srcLoc == LOC_OCL) && (dstLoc == LOC_CPU)) {
313       dev = srcInfo->device;
314       proc = srcInfo->process;
315       pidx = clFindPlatform(srcInfo->platform);
316       queue = glblState.plats[pidx].devs[dev].queues[proc];
317       clCheckErr(clEnqueueReadBuffer(queue, (cl_mem)src, CL_TRUE, 0, size, dst, 0,
318                                      NULL, NULL),
319                  __LINE__,__FILE__);
320    } else if ((srcLoc == LOC_OCL) && (dstLoc == LOC_OCL)) {
321       dev = dstInfo->device;
322       proc = dstInfo->process;
323       pidx = clFindPlatform(dstInfo->platform);
324       queue = glblState.plats[pidx].devs[dev].queues[proc];
325       clCheckErr(clEnqueueCopyBuffer(queue, (cl_mem)src, (cl_mem)dst, 0, 0, size, 0,
326                                      NULL, NULL),
327                  __LINE__,__FILE__);
328    } else {
329       printf("OCLUTIL ERROR: Invalid device locations: line:%d file:%s\n",
330              __LINE__,__FILE__);
```

```
331       exit(-449);
332    }
333
334    clCheckErr(clEnqueueBarrier(queue),__LINE__,__FILE__);
335    clCheckErr(clFinish(queue),__LINE__,__FILE__);
336    return;
337 }
338
339 /**
340  * clSafeBuild function
341  *
342  * \brief  This function will build the kernels in the specified source file for
343  *         the given heterogeneous device, printing build errors and debug
344  *         information out at each compilation.
345  *
346  */
347 extern "C" void oclSafeBuild(HTaskInfo * info, char * srcFile) {
348    cl_int err = CL_SUCCESS;
349    size_t psize;
350    char * pstr;
351    char * src = NULL;
352    FILE * fid = NULL;
353    size_t srcSz = 0;
354    bool built = false;
355    bool first = true;
356    int pidx = clFindPlatform(info->platform); //getPlatform());
357    oclPlatInfo * plat = &(glblState.plats[pidx]);
358    cl_kernel * krns = (cl_kernel*)malloc(sizeof(cl_kernel)*MAX_KERNS);
359    cl_device_id * devs = (cl_device_id*)malloc(sizeof(cl_device_id)*plat->ndevs);
360    for (int idx = 0; idx < plat->ndevs; idx++) { devs[idx] = plat->devs[idx].did; }
361
362    while(!built) {
363       fid = fopen(srcFile,"rb");
364       if (fid == NULL) {
365          printf("OCLUTIL ERROR: Unable to open file %s for building program!\n",
366                 srcFile);
367          exit(-551);
368       }
369       fseek(fid, 0, SEEK_END);
370       srcSz = ftell(fid);
371       rewind(fid);
372       src = (char*)malloc(sizeof(char)*srcSz);
373       srcSz = fread(src, sizeof(char), srcSz, fid);
374       fclose(fid);
375       if (!first) { clReleaseProgram(plat->prg); first = false; }
376       plat->prg = clCreateProgramWithSource(plat->ctx, 1,
377                                             (const char**)&src, &srcSz, &err);
378       clCheckErr(err,__LINE__,__FILE__);
379       err = clBuildProgram(plat->prg, plat->ndevs, devs, buildOpts, NULL, NULL);
380
381       if (err != CL_SUCCESS) {
382          cl_build_status bst;
383          char * pstr = (char*)malloc(sizeof(char)*STRLEN);
384          clDecodeErr(err, pstr);
385          printf("Encountered program build error %d: %s\n", err, pstr);
386          free(pstr);
387          for (int idx = 0; idx < plat->ndevs; idx++) {
388             oclDeviceInfo * dev = &(plat->devs[idx]);
389             clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, NULL,NULL, &psize),
390                        __LINE__,__FILE__);
391             pstr = (char*)malloc(psize);
392             clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, psize, pstr, NULL),
393                        __LINE__,__FILE__);
```

```
394             printf("\tProgram build for device: %s", pstr);
395             free(pstr);
396             clCheckErr(clGetProgramBuildInfo(plat->prg, dev->did,
397                                              CL_PROGRAM_BUILD_STATUS,
398                                              sizeof(cl_build_status), &bst, NULL),
399                        __LINE__,__FILE__);
400             if (bst != CL_BUILD_SUCCESS) {
401                printf(" has FAILED! See information below ...\n");
402                clProgramBuildDump(plat->prg, dev->did);
403                bool input = false;
404                char inval = 'x';
405                while (!input) {
406                   printf("\nWould you like to try to re-build the program [Y/N]? ");
407                   std::cin >> inval;
408                   input = ((inval=='Y')||(inval=='N')) ? true : false;
409                }
410                if (inval=='N') { built = true; break; }
411             } else {  printf(" has SUCCEEDED!\n"); }
412          }
413       } else {
414          built = true; printf("Built program successfully!\n");
415          for (int idx = 0; idx < plat->ndevs; idx++) {
416             oclDeviceInfo * dev = &(plat->devs[idx]);
417             clCheckErr(clCreateKernelsInProgram(plat->prg, MAX_KERNS, krns,
418                                                 &(dev->nkrn)),
419                        __LINE__,__FILE__);
420             for (int jdx = 0; jdx < dev->nkrn; jdx++) {
421                dev->kerns[jdx] = krns[jdx];
422             }
423          }
424       }
425       free(src);
426    }
427    free(krns);
428    free(devs);
429    return;
430 }
431
432 /**
433  * clBuild function
434  *
435  * \brief  This function will build the kernels in the specified source file for the
436  *         given heterogeneous device
437  *
438  */
439 extern "C" void oclBuild(HTaskInfo * info, char * srcFile) {
440    cl_int err = CL_SUCCESS;
441    char * src = NULL;
442    FILE * fid = NULL;
443    size_t srcSz = 0;
444    int pidx = clFindPlatform(info->platform); //getPlatform());
445    oclPlatInfo * plat = &(glblState.plats[pidx]);
446    cl_kernel * krns = (cl_kernel*)malloc(sizeof(cl_kernel)*MAX_KERNS);
447
448    fid = fopen(srcFile,"rb");
449    if (fid == NULL) {
450       printf("OCLUTIL ERROR: Unable to open file %s for building program!\n",
451              srcFile);
452       exit(-552);
453    }
454    fseek(fid, 0, SEEK_END);
455    srcSz = ftell(fid);
456    rewind(fid);
```

```
457    src = (char*)malloc(sizeof(char)*srcSz);
458    srcSz = fread(src, sizeof(char), srcSz, fid);
459    fclose(fid);
460    plat->prg = clCreateProgramWithSource(plat->ctx, 1, (const char**)&src,
461                                          &srcSz, &err);
462    clCheckErr(err,__LINE__,__FILE__);
463
464    for (int idx = 0; idx < plat->ndevs; idx++) {
465       oclDeviceInfo * dev = &(plat->devs[idx]);
466       clCheckErr(clBuildProgram(plat->prg, 1, &(dev->did), "", NULL, NULL),
467                  __LINE__,__FILE__);
468       clCheckErr(clCreateKernelsInProgram(plat->prg, MAX_KERNS, krns, &(dev->nkrn)),
469                  __LINE__,__FILE__);
470       for (int jdx = 0; jdx < dev->nkrn; jdx++) {
471          dev->kerns[jdx] = krns[jdx];
472       }
473    }
474    free(src);
475    free(krns);
476    return;
477 }
478
479 /**
480  * oclLaunchKernel function
481  *
482  * \brief  This function uses the specified parameters to execute the
483  *         coprocessor function specified by kernel. Any parameters to
484  *         the kernel should be included in params.
485  *
486  */
487 extern "C" void oclLaunchKernel(const char * krn, int * dims, int nParams,
488                                 int * paramSizes, void ** params,
489                                 int * gDim, int * bDim, int locMem,
490                                 HTaskInfo * info) {
491    int work_dim = 1;
492    hTaskLoc loc = info->location;
493    int dev = info->device;
494    int proc = info->process;
495    int pidx = clFindPlatform(info->platform);
496    oclPlatInfo * plat = &(glblState.plats[pidx]);
497    cl_command_queue queue = plat->devs[dev].queues[proc];
498    cl_kernel *kern = clFindKernel(plat->devs[dev].kerns, plat->devs[dev].nkrn, krn);
499
500    size_t gSize[3] = {gDim[0]*bDim[0], gDim[1]*bDim[1], gDim[2]*bDim[2]};
501    size_t bSize[3] = {bDim[0], bDim[1], bDim[2]};
502    for (int idx = 1; idx < 3; idx++) {
503       if (gSize[idx] > 1) {
504          work_dim++;
505       }
506    }
507    for (int idx = 0; idx < nParams; idx++) {
508       clCheckErr(clSetKernelArg(*kern, idx, paramSizes[idx], params[idx]),
509                  __LINE__,__FILE__);
510    }
511    clCheckErr(clEnqueueNDRangeKernel(queue, *kern, work_dim, NULL, gSize, bSize, 0,
512                                      NULL, NULL),
513               __LINE__,__FILE__);
514    clCheckErr(clEnqueueBarrier(queue),__LINE__,__FILE__);
515    clCheckErr(clFinish(queue),__LINE__,__FILE__);
516    return;
517 }
```

Listing A.4: OpenCL utility function code

A.2.4 OpenCL Helper Functions

```
/**
 *
 *  \file    oclHelpers.h
 *  \author  $LastChangedBy: $
 *  \date    $LastChangedDate: $
 *  \version $LastChangedRevision: $
 *  \brief   Function declarations and wrappers for the OpenCL helper functions.
 *
 *  $Id: $
 *
 *  Author: James Brock
 */
#include "oclHelpers.h"

// OpenCL Helper variables
cl_int clErr;
size_t psize;
char * pstr;
cl_uint puint;
cl_bool pbool;
cl_ulong pulong;

//////////////////////////////////////////////////////////////
//               OPENCL HELPER FUNCTIONS                    //
//////////////////////////////////////////////////////////////
/**
 * clDecodeErr function
 *
 * \brief This function checks, analyzes, and diagnoses OpenCL error codes
 *
 * \param err the error code to check
 * \param str the string to write error messages to
 *
 * \return None
 *
 */
extern "C" void clDecodeErr(cl_int err, char * str) {
   str[0] = '\0';
   switch(err) {
      case CL_SUCCESS:
         strcat(str, "CL_SUCCESS"); break;
      case CL_DEVICE_NOT_FOUND:
         strcat(str, "CL_DEVICE_NOT_FOUND"); break;
      case CL_DEVICE_NOT_AVAILABLE:
         strcat(str, "CL_DEVICE_NOT_AVAILABLE"); break;
      case CL_COMPILER_NOT_AVAILABLE:
         strcat(str, "CL_COMPILER_NOT_AVAILABLE"); break;
      case CL_MEM_OBJECT_ALLOCATION_FAILURE:
         strcat(str, "CL_MEM_OBJECT_ALLOCATION_FAILURE"); break;
      case CL_OUT_OF_RESOURCES:
         strcat(str, "CL_OUT_OF_RESOURCES"); break;
      case CL_OUT_OF_HOST_MEMORY:
         strcat(str, "CL_OUT_OF_HOST_MEMORY"); break;
      case CL_PROFILING_INFO_NOT_AVAILABLE:
         strcat(str, "CL_PROFILING_INFO_NOT_AVAILABLE"); break;
      case CL_MEM_COPY_OVERLAP:
         strcat(str, "CL_MEM_COPY_OVERLAP"); break;
      case CL_IMAGE_FORMAT_MISMATCH:
         strcat(str, "CL_IMAGE_FORMAT_MISMATCH"); break;
```

```
60        case CL_IMAGE_FORMAT_NOT_SUPPORTED:
61           strcat(str, "CL_IMAGE_FORMAT_NOT_SUPPORTED"); break;
62        case CL_BUILD_PROGRAM_FAILURE:
63           strcat(str, "CL_BUILD_PROGRAM_FAILURE"); break;
64        case CL_MAP_FAILURE:
65           strcat(str, "CL_MAP_FAILURE"); break;
66        case CL_MISALIGNED_SUB_BUFFER_OFFSET:
67           strcat(str, "CL_MISALIGNED_SUB_BUFFER_OFFSET"); break;
68        case CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST:
69           strcat(str, "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST"); break;
70        // Insert cases 15-30 here
71        case CL_INVALID_VALUE:
72           strcat(str, "CL_INVALID_VALUE"); break;
73        case CL_INVALID_DEVICE_TYPE:
74           strcat(str, "CL_INVLIAD_DEVICE_TYPE"); break;
75        case CL_INVALID_PLATFORM:
76           strcat(str, "CL_INVALID_PLATFORM"); break;
77        case CL_INVALID_DEVICE:
78           strcat(str, "CL_INVALID_DEVICE"); break;
79        case CL_INVALID_CONTEXT:
80           strcat(str, "CL_INVALID_CONTEXT"); break;
81        case CL_INVALID_QUEUE_PROPERTIES:
82           strcat(str, "CL_INVALID_QUEUE_PROPERTIES"); break;
83        case CL_INVALID_COMMAND_QUEUE:
84           strcat(str, "CL_INVALID_COMMAND_QUEUE"); break;
85        case CL_INVALID_HOST_PTR:
86           strcat(str, "CL_INVALID_HOST_PTR"); break;
87        case CL_INVALID_MEM_OBJECT:
88           strcat(str, "CL_INVALID_MEM_OBJECT"); break;
89        case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
90           strcat(str, "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR"); break;
91        case CL_INVALID_IMAGE_SIZE:
92           strcat(str, "CL_INVALID_IMAGE_SIZE"); break;
93        case CL_INVALID_SAMPLER:
94           strcat(str, "CL_INVALID_SAMPLER"); break;
95        case CL_INVALID_BINARY:
96           strcat(str, "CL_INVALID_BINARY"); break;
97        case CL_INVALID_BUILD_OPTIONS:
98           strcat(str, "CL_INVALID_BUILD_OPTIONS"); break;
99        case CL_INVALID_PROGRAM:
100          strcat(str, "CL_INVALID_PROGRAM"); break;
101       case CL_INVALID_PROGRAM_EXECUTABLE:
102          strcat(str, "CL_INVALID_PROGRAM_EXECUTABLE"); break;
103       case CL_INVALID_KERNEL_NAME:
104          strcat(str, "CL_INVALID_KERNEL_NAME"); break;
105       case CL_INVALID_KERNEL_DEFINITION:
106          strcat(str, "CL_INVALID_KERNEL_DEFINITION"); break;
107       case CL_INVALID_KERNEL:
108          strcat(str, "CL_INVALID_KERNEL"); break;
109       case CL_INVALID_ARG_INDEX:
110          strcat(str, "CL_INVALID_ARG_INDEX"); break;
111       case CL_INVALID_ARG_VALUE:
112          strcat(str, "CL_INVALID_ARG_VALUE"); break;
113       case CL_INVALID_ARG_SIZE:
114          strcat(str, "CL_INVALID_ARG_SIZE"); break;
115       case CL_INVALID_KERNEL_ARGS:
116          strcat(str, "CL_INVALID_KERNEL_ARGS"); break;
117       case CL_INVALID_WORK_DIMENSION:
118          strcat(str, "CL_INVALID_WORK_DIMENSION"); break;
119       case CL_INVALID_WORK_GROUP_SIZE:
120          strcat(str, "CL_INVALID_WORK_GROUP_SIZE"); break;
121       case CL_INVALID_WORK_ITEM_SIZE:
122          strcat(str, "CL_INVALID_WORK_ITEM_SIZE"); break;
```

```
123       case CL_INVALID_GLOBAL_OFFSET:
124          strcat(str, "CL_INVALID_GLOBAL_OFFSET"); break;
125       case CL_INVALID_EVENT_WAIT_LIST:
126          strcat(str, "CL_INVALID_EVENT_WAIT_LIST"); break;
127       case CL_INVALID_EVENT:
128          strcat(str, "CL_INVALID_EVENT"); break;
129       case CL_INVALID_OPERATION:
130          strcat(str, "CL_INVALID_OPERATION"); break;
131       case CL_INVALID_GL_OBJECT:
132          strcat(str, "CL_INVALID_GL_OBJECT"); break;
133       case CL_INVALID_BUFFER_SIZE:
134          strcat(str, "CL_INVALID_BUFFER_SIZE"); break;
135       case CL_INVALID_MIP_LEVEL:
136          strcat(str, "CL_INVALID_MIP_LEVEL"); break;
137       case CL_INVALID_GLOBAL_WORK_SIZE:
138          strcat(str, "CL_INVALID_GLOBAL_WORK_SIZE"); break;
139       //case CL_INVALID_PROPERTY:
140          //strcat(str, "CL_INVALID_PROPERTY"); break;
141       // 64+ go here
142       default: strcat(str, "UNKNOWN ERROR CODE");
143    }
144    return;
145 }
146 
147 /**
148  * clCheckErr function
149  *
150  * \brief This function checks, analyzes, and diagnoses OpenCL error codes
151  *
152  * \param err the error code to check
153  * \param line the line number where the error code is being checked
154  * \param file the name of the file in which the error code was generated
155  *
156  * \return None
157  *
158  */
159 extern "C" void clCheckErr(cl_int err, int line, char * file) {
160    char errStr[STRLEN];
161    if (err != CL_SUCCESS) {
162       clDecodeErr(err, errStr);
163       printf("OpenCL Error %d: %s. In file %s, line %d\n", err, errStr, file, line);
164 #ifdef KILLONERR
165       exit(err);
166 #endif
167    }
168    return;
169 }
170 
171 /**
172  * clDecodeFPConfig function
173  *
174  * \brief This function decodes the OpenCL floating point configuration type
175  *
176  * \param cfg Device FP configuration code
177  * \param str  Output string to write to
178  * \return None
179  */
180 extern "C" void clDecodeFPConfig(cl_device_fp_config cfg, char * str) {
181    int on = 0;
182    str[0] = '\0';
183    if (cfg & CL_FP_DENORM) { strcat(str, "DENORM"); on = 1; }
184    if (cfg & CL_FP_INF_NAN) {
185       if(on){strcat(str,", ");}
```

```
186       strcat(str, "INF_NAN"); on = 1;
187    }
188    if (cfg & CL_FP_ROUND_TO_NEAREST) {
189       if(on){strcat(str,", ");}
190       strcat(str, "ROUND_TO_NEAREST"); on = 1;
191    }
192    if (cfg & CL_FP_ROUND_TO_ZERO) {
193       if(on){strcat(str,", ");}
194       strcat(str, "ROUND_TO_ZERO"); on = 1;
195    }
196    if (cfg & CL_FP_ROUND_TO_INF) {
197       if(on){strcat(str,", ");}
198       strcat(str, "ROUND TO INF"); on = 1;
199    }
200    if (cfg & CL_FP_FMA) {
201       if(on){strcat(str,", ");}
202       strcat(str, "FMA"); on = 1;
203    }
204    if (cfg & CL_FP_SOFT_FLOAT) {
205       if(on){strcat(str,", ");}
206       strcat(str, "SOFT FLOAT");
207    }
208    return;
209 }
210
211 /**
212  * clDecodeExecCapabilities function
213  *
214  * \brief This function decodes the OpenCL device execution capabilities type
215  *
216  * \param dcap  Device execution capabilities code
217  * \param str  Output string to write to
218  * \return None
219  */
220 extern "C" void clDecodeExecCapabilities(cl_device_exec_capabilities dcap,
221                                          char * str) {
222    int on = 0;
223    str[0] = '\0';
224    if (dcap & CL_EXEC_KERNEL) { strcat(str, "KERNEL"); on = 1; }
225    if (dcap & CL_EXEC_NATIVE_KERNEL) {
226       if(on){strcat(str,", ");}
227       strcat(str, "NATIVE_KERNEL");
228    }
229    return;
230 }
231
232 /**
233  * clDecodeDevMemCacheType function
234  *
235  * \brief This function decodes the OpenCL device memory cache type
236  *
237  * \param mctype  Device memory cache type code
238  * \param str  Output string to write to
239  * \return None
240  */
241 extern "C" void clDecodeDevMemCacheType(cl_device_mem_cache_type mctype,
242                                         char * str) {
243    str[0] = '\0';
244    switch(mctype) {
245       case CL_NONE: strcat(str, "NONE"); break;
246       case CL_READ_ONLY_CACHE: strcat(str, "READ_ONLY_CACHE"); break;
247       case CL_READ_WRITE_CACHE: strcat(str, "READ_WRITE_CACHE"); break;
248       default: strcat(str, "ERROR: Invalid device memory cache type");
```

```
249    }
250    return;
251 }
252
253 /**
254  * clDecodeLocalMemType function
255  *
256  * \brief This function decodes the OpenCL device local memory type
257  *
258  * \param lmtype  Device local memory type code
259  * \param str  Output string to write to
260  * \return None
261  */
262 extern "C" void clDecodeLocalMemType(cl_device_local_mem_type lmtype, char * str) {
263    str[0] = '\0';
264    switch(lmtype) {
265       case CL_LOCAL: strcat(str, "LOCAL"); break;
266       case CL_GLOBAL: strcat(str, "GLOBAL"); break;
267       default: strcat(str, "ERROR: Invalid device local memory type");
268    }
269    return;
270 }
271
272 /**
273  * clDecodeQueueProperties function
274  *
275  * \brief This function decodes the OpenCL command queue properties type
276  *
277  * \param qprop  Command queue properties code
278  * \param str  Output string to write to
279  * \return None
280  */
281 extern "C" void clDecodeQueueProperties(cl_command_queue_properties qprop,
282                                         char * str) {
283    int on = 0;
284    str[0] = '\0';
285    if (qprop & CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE) {
286       strcat(str, "OUT_OF_ORDER_EXEC_ENABLE"); on = 1;
287    }
288    if (qprop & CL_QUEUE_PROFILING_ENABLE) {
289       if(on){strcat(str,", ");}
290       strcat(str, "PROFILING_ENABLE");
291    }
292    return;
293 }
294
295 /**
296  * clDecodeDeviceType function
297  *
298  * \brief This function decodes the OpenCL device type
299  *
300  * \param dtype Device type code
301  * \param str  Output string to write to
302  * \return None
303  *
304  */
305 extern "C" void clDecodeDeviceType(cl_device_type dtype, char * str) {
306    int on = 0;
307    str[0] = '\0';
308    if (dtype & CL_DEVICE_TYPE_DEFAULT) {
309       strcat(str, "DEFAULT"); on = 1;
310    }
311    if (dtype & CL_DEVICE_TYPE_CPU) {
```

```
312       if(on){strcat(str,", ");}
313       strcat(str, "CPU"); on = 1;
314    }
315    if (dtype & CL_DEVICE_TYPE_GPU) {
316       if(on){strcat(str,", ");}
317       strcat(str, "GPU");  on = 1;
318    }
319    if (dtype & CL_DEVICE_TYPE_ACCELERATOR) {
320       if(on){strcat(str,", ");}
321       strcat(str, "ACCELERATOR");  on = 1;
322    }
323    if (dtype & CL_DEVICE_TYPE_ALL) {
324       if(on){strcat(str,", ");}
325       strcat(str, "ALL");
326    }
327    return;
328 }
329
330 /**
331  * clDecodeMemObjType function
332  *
333  * \brief This function decodes the OpenCL memory object type
334  *
335  * \param mtype Memory object type code
336  * \param str  Output string to write to
337  * \return None
338  *
339  */
340 extern "C" void clDecodeMemObjType(cl_mem_object_type mtype, char * str) {
341    str[0] = '\0';
342    switch(mtype) {
343       case CL_MEM_OBJECT_BUFFER: strcat(str, "CL_MEM_OBJECT_BUFFER"); break;
344       case CL_MEM_OBJECT_IMAGE2D: strcat(str, "CL_MEM_OBJECT_IMAGE2D"); break;
345       case CL_MEM_OBJECT_IMAGE3D: strcat(str, "CL_MEM_OBJECT_IMAGE3D"); break;
346       default: strcat(str, "ERROR: Invalid Memory Object Type");
347    }
348    return;
349 }
350
351 /**
352  * clDecodeMemObjFlags function
353  *
354  * \brief This function decodes the OpenCL memory flag type
355  *
356  * \param mflags Memory object flags code
357  * \param str  Output string to write to
358  * \return None
359  *
360  */
361 extern "C" void clDecodeMemObjFlags(cl_mem_flags mflags, char * str) {
362    int on = 0;
363    str[0] = '\0';
364    if (mflags & CL_MEM_READ_WRITE) {
365       strcat(str, "CL_MEM_READ_WRITE "); on = 1;
366    }
367    if (mflags & CL_MEM_WRITE_ONLY) {
368       if(on){strcat(str,", ");}
369       strcat(str, "CL_MEM_WRITE_ONLY "); on = 1;
370    }
371    if (mflags & CL_MEM_READ_ONLY) {
372       if(on){strcat(str,", ");}
373       strcat(str, "CL_MEM_READ_ONLY "); on = 1;
374    }
```

```
375    if (mflags & CL_MEM_USE_HOST_PTR) {
376       if(on){strcat(str,", ");}
377       strcat(str, "CL_MEM_USE_HOST_PTR "); on = 1;
378    }
379    if (mflags & CL_MEM_ALLOC_HOST_PTR) {
380       if(on){strcat(str,", ");}
381       strcat(str, "CL_MEM_ALLOC_HOST_PTR "); on = 1;
382    }
383    if (mflags & CL_MEM_COPY_HOST_PTR) {
384       if(on){strcat(str,", ");}
385       strcat(str, "CL_MEM_COPY_HOST_PTR ");
386    }
387    return;
388 }
389
390 /**
391  * clDecodeImageFormat function
392  *
393  * \brief This function decodes the OpenCL image format type
394  *
395  * \param imgfmt Image format code
396  * \param str  Output string to write to
397  * \return None
398  *
399  */
400 extern "C" void clDecodeImageFormat(cl_image_format imgfmt, char * str) {
401    str[0] = '\0';
402    strcat(str, "\n   Channel Order              ");
403    switch (imgfmt.image_channel_order) {
404       case CL_R : strcat(str, "CL_R"); break;
405       case CL_A : strcat(str, "CL_A"); break;
406       case CL_RG : strcat(str, "CL_RG"); break;
407       case CL_RA : strcat(str, "CL_RA"); break;
408       case CL_RGB : strcat(str, "CL_RGB"); break;
409       case CL_RGBA : strcat(str, "CL_RGBA"); break;
410       case CL_BGRA : strcat(str, "CL_BGRA"); break;
411       case CL_ARGB : strcat(str, "CL_ARGB"); break;
412       case CL_INTENSITY : strcat(str, "CL_INTENSITY"); break;
413       case CL_LUMINANCE : strcat(str, "CL_LUMINANCE"); break;
414       default: strcat(str, "ERROR: Invalid Color Order");
415    }
416    strcat(str, "\n   Channel Data Type          ");
417    switch (imgfmt.image_channel_data_type) {
418       case CL_SNORM_INT8 : strcat(str, "CL_SNORM_INT8"); break;
419       case CL_SNORM_INT16 : strcat(str, "CL_SNORM_INT16"); break;
420       case CL_UNORM_INT8 : strcat(str, "CL_UNORM_INT8"); break;
421       case CL_UNORM_INT16 : strcat(str, "CL_UNORM_INT16"); break;
422       case CL_UNORM_SHORT_565 : strcat(str, "CL_UNORM_SHORT_565"); break;
423       case CL_UNORM_SHORT_555 : strcat(str, "CL_UNORM_SHORT_555"); break;
424       case CL_UNORM_INT_101010 : strcat(str, "CL_UNORM_INT_101010"); break;
425       case CL_SIGNED_INT8 : strcat(str, "CL_SIGNED_INT8"); break;
426       case CL_SIGNED_INT16 : strcat(str, "CL_SIGNED_INT16"); break;
427       case CL_SIGNED_INT32 : strcat(str, "CL_SIGNED_INT32"); break;
428       case CL_UNSIGNED_INT8 : strcat(str, "CL_UNSIGNED_INT8"); break;
429       case CL_UNSIGNED_INT16 : strcat(str, "CL_UNSIGNED_INT16"); break;
430       case CL_UNSIGNED_INT32 : strcat(str, "CL_UNSIGNED_INT32"); break;
431       case CL_HALF_FLOAT : strcat(str, "CL_HALF_FLOAT"); break;
432       case CL_FLOAT : strcat(str, "CL_FLOAT"); break;
433       default: strcat(str, "ERROR: Invalid Data Type");
434    }
435    return;
436 }
437
```

```
438 /**
439  * clDecodeAddressingMode function
440  *
441  * \brief This function decodes the OpenCL addressing mode type
442  *
443  * \param amd Addressing mode code
444  * \param str  Output string to write to
445  * \return None
446  *
447  */
448 extern "C" void clDecodeAddressingMode(cl_addressing_mode amd, char * str) {
449    str[0] = '\0';
450    switch (amd) {
451       case CL_ADDRESS_NONE :
452          strcat(str, "CL_ADDRESS_NONE"); break;
453       case CL_ADDRESS_CLAMP_TO_EDGE :
454          strcat(str, "CL_ADDRESS_CLAMP_TO_EDGE"); break;
455       case CL_ADDRESS_CLAMP :
456          strcat(str, "CL_ADDRESS_CLAMP"); break;
457       case CL_ADDRESS_REPEAT :
458          strcat(str, "CL_ADDRESS_REPEAT"); break;
459       default:
460          strcat(str, "ERROR: Invalid Addressing Mode");
461    }
462    return;
463 }
464
465 /**
466  * clDecodeFilterMode function
467  *
468  * \brief This function decodes the OpenCL filter mode type
469  *
470  * \param fmd Filter mode code
471  * \param str  Output string to write to
472  * \return None
473  *
474  */
475 extern "C" void clDecodeFilterMode(cl_filter_mode fmd, char * str) {
476    str[0] = '\0';
477    switch (fmd) {
478       case CL_FILTER_NEAREST : strcat(str, "CL_FILTER_NEAREST"); break;
479       case CL_FILTER_LINEAR : strcat(str, "CL_FILTER_LINEAR"); break;
480       default: strcat(str, "ERROR: Invalid Filter Mode");
481    }
482    return;
483 }
484
485 /**
486  * clDecodeBuildStatus function
487  *
488  * \brief This function decodes the OpenCL build status type
489  *
490  * \param bst Build status code
491  * \param str  Output string to write to
492  * \return None
493  *
494  */
495 extern "C" void clDecodeBuildStatus(cl_build_status bst, char * str) {
496    str[0] = '\0';
497    switch (bst) {
498       case CL_BUILD_SUCCESS : strcat(str, "CL_BUILD_SUCCESS"); break;
499       case CL_BUILD_NONE : strcat(str, "CL_BUILD_NONE"); break;
500       case CL_BUILD_ERROR : strcat(str, "CL_BUILD_ERROR"); break;
```

```
501       case CL_BUILD_IN_PROGRESS : strcat(str, "CL_BUILD_IN_PROGRESS");
502    }
503    return;
504 }
505
506 /**
507  * clPlatformDump function
508  *
509  * \brief This function prints out all information about the given platform ID
510  *
511  * \param pid Platform id pointer
512  * \return None
513  *
514  */
515 extern "C" void clPlatformDump(cl_platform_id pid) {
516    printf("Printing information for platform ID:0x%X\n", pid);
517
518    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_PROFILE, NULL, NULL, &psize),
519               __LINE__,__FILE__);
520    pstr = (char*)malloc(psize);
521    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_PROFILE, psize, pstr, NULL),
522               __LINE__,__FILE__);
523    printf("\tProfile:        %s\n", pstr);
524    free(pstr);
525
526    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VERSION, NULL, NULL, &psize),
527               __LINE__,__FILE__);
528    pstr = (char*)malloc(psize);
529    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VERSION, psize, pstr, NULL),
530               __LINE__,__FILE__);
531    printf("\tVersion:        %s\n", pstr);
532    free(pstr);
533
534    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_NAME, NULL, NULL, &psize),
535               __LINE__,__FILE__);
536    pstr = (char*)malloc(psize);
537    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_NAME, psize, pstr, NULL),
538               __LINE__,__FILE__);
539    printf("\tName:           %s\n", pstr);
540    free(pstr);
541
542    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, NULL, NULL, &psize),
543               __LINE__,__FILE__);
544    pstr = (char*)malloc(psize);
545    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, psize, pstr, NULL),
546               __LINE__,__FILE__);
547    printf("\tVendor:         %s\n", pstr);
548    free(pstr);
549
550    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_EXTENSIONS, NULL, NULL, &psize),
551               __LINE__,__FILE__);
552    pstr = (char*)malloc(psize);
553    clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_EXTENSIONS, psize, pstr, NULL),
554               __LINE__,__FILE__);
555    printf("\tExtensions:     %s\n", pstr);
556    free(pstr);
557
558    printf("\n");
559    return;
560 }
561
562 /**
563  * clDeviceDump function
```

```
564  *
565  * \brief This function prints out all information about the given device ID
566  *
567  * \param did Device id
568  * \return None
569  *
570  */
571 extern "C" void clDeviceDump(cl_device_id did) {
572    size_t * pdims;
573    int idx = 0;
574    cl_device_fp_config fpcfg;
575    cl_device_type dtype;
576    cl_device_exec_capabilities dcap;
577    cl_device_mem_cache_type mctype;
578    cl_device_local_mem_type lmtype;
579    cl_platform_id dplat;
580    cl_command_queue_properties dqprop;
581
582    printf("\nPrinting information for device ID:0x%X\n", did);
583    printf("-----------------------------------------------------------------\n");
584    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR_ID, sizeof(cl_uint),
585                               &puint, NULL),
586               __LINE__,__FILE__);
587    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR, NULL, NULL, &psize),
588               __LINE__,__FILE__);
589    pstr = (char*)malloc(psize);
590    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR, psize, pstr, NULL),
591               __LINE__,__FILE__);
592    printf("\tVendor:     %s (ID: 0x%X)\n", pstr, puint);
593    free(pstr);
594    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NAME, NULL, NULL, &psize),
595               __LINE__,__FILE__);
596    pstr = (char*)malloc(psize);
597    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NAME, psize, pstr, NULL),
598               __LINE__,__FILE__);
599    printf("\tName:       %s", pstr);
600    free(pstr);
601    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_TYPE, sizeof(cl_device_type),
602                               &dtype, NULL),
603               __LINE__,__FILE__);
604    pstr = (char*)malloc(sizeof(char)*STRLEN);
605    clDecodeDeviceType(dtype, pstr);
606    printf(" (%s) ", pstr);
607    free(pstr);
608    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VERSION, NULL, NULL, &psize),
609               __LINE__,__FILE__);
610    pstr = (char*)malloc(psize);
611    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VERSION, psize, pstr, NULL),
612               __LINE__,__FILE__);
613    printf("ver. %s\n", pstr);
614    free(pstr);
615    clCheckErr(clGetDeviceInfo(did, CL_DRIVER_VERSION, NULL, NULL, &psize),
616               __LINE__,__FILE__);
617    pstr = (char*)malloc(psize);
618    clCheckErr(clGetDeviceInfo(did, CL_DRIVER_VERSION, psize, pstr, NULL),
619               __LINE__,__FILE__);
620    printf("\tDriver:     ver. %s\n", pstr);
621    free(pstr);
622    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PLATFORM, sizeof(cl_platform_id),
623                               &dplat, NULL),
624               __LINE__,__FILE__);
625    printf("\tPlatform:   0x%X\n", dplat);
626    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_OPENCL_C_VERSION, NULL, NULL, &psize),
```

```
627             __LINE__,__FILE__);
628     pstr = (char*)malloc(psize);
629     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_OPENCL_C_VERSION, NULL, NULL, &psize),
630             __LINE__,__FILE__);
631     printf("\tOpenCL C:   ver. %s\n", pstr);
632     free(pstr);
633     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_AVAILABLE, sizeof(cl_bool), &pbool,
            NULL),
634             __LINE__,__FILE__);
635     printf("\tAvailable:  %s\n", (pbool) ? "Yes" : "No");
636     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_COMPILER_AVAILABLE, sizeof(cl_bool),
637                             &pbool, NULL),
638             __LINE__,__FILE__);
639     printf("\tCompiler:   %s\n", (pbool) ? "Yes" : "No");
640     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_EXECUTION_CAPABILITIES,
641                             sizeof(cl_device_exec_capabilities), &dcap, NULL),
642             __LINE__,__FILE__);
643     pstr = (char*)malloc(sizeof(char)*STRLEN);
644     clDecodeExecCapabilities(dcap,pstr);
645     printf("\tExecution:  %s\n", pstr);
646     free(pstr);
647     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_EXTENSIONS, NULL, NULL, &psize),
648             __LINE__,__FILE__);
649     pstr = (char*)malloc(psize);
650     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_EXTENSIONS, psize, pstr, NULL),
651             __LINE__,__FILE__);
652     printf("\tExtensions: %s\n", pstr);
653     free(pstr);
654
655     printf("\nDevice compute capabilities\n");
656     printf("-------------------------------------------------------------\n");
657     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PROFILE, NULL, NULL, &psize),
658             __LINE__,__FILE__);
659     pstr = (char*)malloc(psize);
660     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PROFILE, psize, pstr, NULL),
661             __LINE__,__FILE__);
662     printf("\tDevice Profile:    %s\n", pstr);
663     free(pstr);
664     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PROFILING_TIMER_RESOLUTION,
665                             sizeof(size_t), &psize, NULL),
666             __LINE__,__FILE__);
667     printf("\tTimer Resolution:  %dns\n", psize);
668     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_QUEUE_PROPERTIES,
669                             sizeof(cl_command_queue_properties), &dqprop, NULL),
670             __LINE__,__FILE__);
671     pstr = (char*)malloc(sizeof(char)*STRLEN);
672     clDecodeQueueProperties(dqprop, pstr);
673     printf("\tCommand Queue:     %s\n", pstr);
674     free(pstr);
675     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CLOCK_FREQUENCY,
676                             sizeof(cl_uint), &puint, NULL),
677             __LINE__,__FILE__);
678     printf("\tClock Frequency:   %dMHz (max)\n", puint);
679     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_COMPUTE_UNITS,
680                             sizeof(cl_uint), &puint, NULL),
681             __LINE__,__FILE__);
682     printf("\tCompute Units:     %d (max)\n", puint);
683     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_GROUP_SIZE,
684                             sizeof(size_t), &psize, NULL),
685             __LINE__,__FILE__);
686     printf("\tWork Group Size:   %d (max)\n", psize);
687     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
688                             sizeof(cl_uint), &puint, NULL),
```

```
689             __LINE__,__FILE__);
690     pdims = (size_t*)malloc(sizeof(size_t)*puint);
691     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_ITEM_SIZES,
692                                sizeof(size_t)*puint, pdims, NULL),
693             __LINE__,__FILE__);
694     printf("\tWork Item Sizes:   [%d", pdims[0]);
695     for (idx = 1; idx < puint; idx++) { printf(", %d",pdims[idx]); }
696     printf("] (max)\n");
697     free(pdims);
698
699     printf("\nDevice Memory Information\n");
700     printf("-----------------------------------------------------------\n");
701     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ADDRESS_BITS , sizeof(cl_uint),
702                                &puint, NULL),
703             __LINE__,__FILE__);
704     printf("\tAddress Bits:                 %d\n", puint);
705     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ENDIAN_LITTLE , sizeof(cl_bool),
706                                &pbool, NULL),
707             __LINE__,__FILE__);
708     printf("\tLittle Endian:                %s\n", (pbool) ? "Yes" : "No");
709     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MEM_BASE_ADDR_ALIGN, sizeof(cl_uint),
710                                &puint, NULL),
711             __LINE__,__FILE__);
712     printf("\tMemory Base Addr Align:       %d bits\n", puint);
713     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE,
714                                sizeof(cl_uint),&puint, NULL),
715             __LINE__,__FILE__);
716     printf("\tError Correction:             %s\n", (pbool) ? "Yes" : "No");
717     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_HOST_UNIFIED_MEMORY,
718                                sizeof(cl_bool), &pbool, NULL),
719             __LINE__,__FILE__);
720     printf("\tDev/Host Unified Memory:      %s\n", (pbool) ? "Yes" : "No");
721     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_SIZE,
722                                sizeof(cl_ulong), &pulong, NULL),
723             __LINE__,__FILE__);
724     printf("\tGlobal Memory Size:           %luB\n", pulong);
725     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,
726                                sizeof(cl_ulong), &pulong, NULL),
727             __LINE__,__FILE__);
728     printf("\tGlobal Memory Cache:          %luB\n", pulong);
729     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHE_TYPE,
730                                sizeof(cl_device_mem_cache_type), &mctype, NULL),
731             __LINE__,__FILE__);
732     pstr = (char*)malloc(sizeof(char)*STRLEN);
733     clDecodeDevMemCacheType(mctype,pstr);
734     printf("\tGlobal Memory Cache Type:     %s\n", pstr);
735     free(pstr);
736     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,
737                                sizeof(cl_uint), &puint, NULL),
738             __LINE__,__FILE__);
739     printf("\tGlobal Memory Cacheline Size: %dB\n", puint);
740     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_LOCAL_MEM_SIZE,
741                                sizeof(cl_ulong), &pulong, NULL),
742             __LINE__,__FILE__);
743     printf("\tLocal Memory Size:            %dB\n", pulong);
744     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_LOCAL_MEM_TYPE,
745                                sizeof(cl_device_local_mem_type), &lmtype, NULL),
746             __LINE__,__FILE__);
747     pstr = (char*)malloc(sizeof(char)*STRLEN);
748     clDecodeLocalMemType(lmtype, pstr);
749     printf("\tLocal Memory Type:            %s\n", pstr);
750     free(pstr);
751     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE_SUPPORT,
```

```
752                             sizeof(cl_bool), &pbool, NULL),
753                 __LINE__,__FILE__);
754     printf("\tImage Support:                %s\n", (pbool) ? "Yes" : "No");
755     if (pbool) {
756        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_READ_IMAGE_ARGS,
757                             sizeof(cl_uint), &puint, NULL),
758                 __LINE__,__FILE__);
759        printf("\t\tRead Image Args:     %d (max)\n", puint);
760        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WRITE_IMAGE_ARGS,
761                             sizeof(cl_uint), &puint, NULL),
762                 __LINE__,__FILE__);
763        printf("\t\tWrite Image Args:    %d (max)\n", puint);
764        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_SAMPLERS,
765                             sizeof(cl_uint), &puint, NULL),
766                 __LINE__,__FILE__);
767        printf("\t\tSamplers:            %d (max)\b", puint);
768        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE2D_MAX_HEIGHT,
769                             sizeof(size_t), &psize, NULL),
770                 __LINE__,__FILE__);
771        printf("\t\tImage 2D Height:     %d (max)\n", psize);
772        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE2D_MAX_WIDTH,
773                             sizeof(size_t), &psize, NULL),
774                 __LINE__,__FILE__);
775        printf("\t\tImage 2D Width:      %d (max)\n", psize);
776        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_DEPTH,
777                             sizeof(size_t), &psize, NULL),
778                 __LINE__,__FILE__);
779        printf("\t\tImage 3D Depth:      %d (max)\n", psize);
780        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_HEIGHT,
781                             sizeof(size_t), &psize, NULL),
782                 __LINE__,__FILE__);
783        printf("\t\tImage 3D Height:     %d (max)\n", psize);
784        clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_WIDTH,
785                             sizeof(size_t), &psize, NULL),
786                 __LINE__,__FILE__);
787        printf("\t\tImage 3D Width:      %d (max)\n", psize);
788     }
789     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CONSTANT_ARGS,
790                             sizeof(cl_uint), &puint, NULL),
791                 __LINE__,__FILE__);
792     printf("Constant Args:                 %d (max)\n", puint);
793     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
794                             sizeof(cl_ulong), &pulong, NULL),
795                 __LINE__,__FILE__);
796     printf("Constant Buffer Size           %dB (max)\n", pulong);
797     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_MEM_ALLOC_SIZE,
798                             sizeof(cl_ulong), &pulong, NULL),
799                 __LINE__,__FILE__);
800     printf("Memory Allocation Size:        %dB (max)\n", pulong);
801     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_PARAMETER_SIZE,
802                             sizeof(size_t), &psize, NULL),
803                 __LINE__,__FILE__);
804     printf("Parameter Size:                %dB (max)\n", psize);
805
806     printf("\nDevice data formatting\n");
807     printf("-------------------------------------------------------------------\n");
808 #ifdef OPENCL_1.2
809     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_DOUBLE_FP_CONFIG,
810                             sizeof(cl_device_fp_config), &fpcfg, NULL),
811                 __LINE__,__FILE__);
812     pstr = (char*)malloc(sizeof(char)*STRLEN);
813     clDecodeFPConfig(fpcfg,pstr);
814     printf("\tDouble FP Config:      %s\n", pstr);
```

```
815     free(pstr);
816 #endif
817     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_SINGLE_FP_CONFIG,
818                                sizeof(cl_device_fp_config), &fpcfg, NULL),
819                __LINE__,__FILE__);
820     pstr = (char*)malloc(sizeof(char)*STRLEN);
821     clDecodeFPConfig(fpcfg,pstr);
822     printf("\tSingle FP Config:      %s\n", pstr);
823     free(pstr);
824 #ifdef OPENCL_1.2
825     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_HALF_FP_CONFIG,
826                                sizeof(cl_device_fp_config), &fpcfg, NULL),
827                __LINE__,__FILE__);
828     pstr = (char*)malloc(sizeof(char)*STRLEN);
829     clDecodeFPConfig(fpcfg,pstr);
830     printf("\tHalf FP Config:        %s\n", pstr);
831     free(pstr);
832 #endif
833     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ERROR_CORRECTION_SUPPORT,
834                                sizeof(cl_bool), &pbool, NULL),
835                __LINE__,__FILE__);
836     printf("\tDevice Error Correction Support (ECS): %s\n", (pbool) ? "Yes" : "No");
837     printf("\tDevice Native Vector Widths\n");
838     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR,
839                                sizeof(cl_uint), &puint, NULL),
840                __LINE__,__FILE__);
841     printf("\t\tCHAR:      %d\n", puint);
842     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT,
843                                sizeof(cl_uint), &puint, NULL),
844                __LINE__,__FILE__);
845     printf("\t\tSHORT:     %d\n", puint);
846     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_INT,
847                                sizeof(cl_uint), &puint, NULL),
848                __LINE__,__FILE__);
849     printf("\t\tINT:       %d\n", puint);
850     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG,
851                                sizeof(cl_uint), &puint, NULL),
852                __LINE__,__FILE__);
853     printf("\t\tLONG:      %d\n", puint);
854     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT,
855                                sizeof(cl_uint), &puint, NULL),
856                __LINE__,__FILE__);
857     printf("\t\tFLOAT:     %d\n", puint);
858     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE,
859                                sizeof(cl_uint), &puint, NULL),
860                __LINE__,__FILE__);
861     printf("\t\tDOUBLE:    %d\n", puint);
862     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF,
863                                sizeof(cl_uint), &puint, NULL),
864                __LINE__,__FILE__);
865     printf("\t\tHALF:      %d\n", puint);
866     printf("\tDevice Preferred Vector Widths\n");
867     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
868                                sizeof(cl_uint), &puint, NULL),
869                __LINE__,__FILE__);
870     printf("\t\tCHAR:      %d\n", puint);
871     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT,
872                                sizeof(cl_uint), &puint, NULL),
873                __LINE__,__FILE__);
874     printf("\t\tSHORT:     %d\n", puint);
875     clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT,
876                                sizeof(cl_uint), &puint, NULL),
877                __LINE__,__FILE__);
```

```
878    printf("\t\tINT:       %d\n", puint);
879    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG,
880                               sizeof(cl_uint), &puint, NULL),
881               __LINE__,__FILE__);
882    printf("\t\tLONG:      %d\n", puint);
883    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT,
884                               sizeof(cl_uint), &puint, NULL),
885               __LINE__,__FILE__);
886    printf("\t\tFLOAT:     %d\n", puint);
887    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE,
888                               sizeof(cl_uint), &puint, NULL),
889               __LINE__,__FILE__);
890    printf("\t\tDOUBLE:    %d\n", puint);
891    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF,
892                               sizeof(cl_uint), &puint, NULL),
893               __LINE__,__FILE__);
894    printf("\t\tHALF:      %d\n", puint);
895    printf("\n");
896    return;
897 }
898
899 /**
900  * clContextDump function
901  *
902  * \brief This function prints out all of the information about the given context
903  *
904  * \param cid Context id
905  * \return None
906  *
907  */
908 extern "C" void clContextDump(cl_context cid) {
909    int idx = 0;
910    cl_device_id * cdevs;
911    cl_context_properties cprops[3] = {0, 0, 0};
912
913    printf("Printing information for context ID:0x%X\n", cid);
914    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_REFERENCE_COUNT,
915                               sizeof(cl_uint), &puint, NULL),
916               __LINE__,__FILE__);
917    printf("\tReference Count:   %d\n", puint);
918    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_NUM_DEVICES,
919                               sizeof(cl_uint), &puint, NULL),
920               __LINE__,__FILE__);
921    printf("\tDevice Count:      %d\n", puint);
922    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_DEVICES, NULL, NULL, &psize),
923               __LINE__,__FILE__);
924    cdevs = (cl_device_id*)malloc(psize);
925    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_DEVICES, psize, cdevs, NULL),
926               __LINE__,__FILE__);
927    printf("\tDevices:           ");
928    for (idx = 0; idx < puint; idx++) {
929       if (idx > 0) { printf(", "); }
930       printf("0x%X", cdevs[idx]);
931    }
932    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_PROPERTIES, NULL, NULL, &psize),
933               __LINE__,__FILE__);
934    clCheckErr(clGetContextInfo(cid, CL_CONTEXT_PROPERTIES, psize, cprops, NULL),
935               __LINE__,__FILE__);
936    pstr = (char*)malloc(sizeof(char)*STRLEN);
937    clDecodeQueueProperties(*cprops, pstr);
938    printf("\tProperties:        %s\n", pstr);
939    free(pstr);
940 #ifdef OPENCL_1.2
```

```
941     clCheckErr(clGetContextInfo(cid, CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR,
942                                 sizeof(cl_bool), &pbool, NULL),
943                __LINE__,__FILE__);
944 #endif
945     pbool = false;
946     printf("\tD3D10 Support:     %s\n", (pbool) ? "Yes" : "No");
947     printf("\n");
948     return;
949 }
950
951 /**
952  * clMemObjDump function
953  *
954  * \brief This function prints out all of the information about the given
955  *        memory object
956  *
957  * \param mobj Memory object
958  * \return None
959  *
960  */
961 extern "C" void clMemObjDump(cl_mem mobj) {
962     cl_mem_object_type   mtype;
963     cl_mem_flags mflags;
964     cl_context  ctx;
965     void * vptr;
966
967     printf("Printing information for memory object ID:0x%X\n", mobj);
968     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_TYPE, sizeof(cl_mem_object_type),
969                                   &mtype, NULL),
970                __LINE__,__FILE__);
971     pstr = (char*)malloc(sizeof(char)*STRLEN);
972     clDecodeMemObjType(mtype, pstr);
973     printf("Type              %s\n", pstr);
974     free(pstr);
975     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_FLAGS, sizeof(cl_mem_flags),
976                                   &mflags, NULL),
977                __LINE__,__FILE__);
978     pstr = (char*)malloc(sizeof(char)*STRLEN);
979     clDecodeMemObjFlags(mflags, pstr);
980     printf("Flags             %s\n", pstr);
981     free(pstr);
982     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_SIZE, sizeof(size_t),
983                                   &psize, NULL),
984                __LINE__,__FILE__);
985     printf("Size              %sB\n", psize);
986     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_HOST_PTR, sizeof(void*),
987                                   &vptr, NULL),
988                __LINE__,__FILE__);
989     printf("Host Ptr          0x%X\n", vptr);
990     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_MAP_COUNT, sizeof(cl_uint),
991                                   &puint, NULL),
992                __LINE__,__FILE__);
993     printf("Map Count         %d\n", puint);
994     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_REFERENCE_COUNT, sizeof(cl_uint),
995                                   &puint, NULL),
996                __LINE__,__FILE__);
997     printf("Reference Count   %d\n", puint);
998     clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_CONTEXT, sizeof(cl_context),
999                                   &ctx, NULL),
1000                __LINE__,__FILE__);
1001    printf("Context           0x%X\n", &ctx);
1002    clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_ASSOCIATED_MEMOBJECT, sizeof(cl_mem),
1003                                  &mobj, NULL),
```

```
1004              __LINE__,__FILE__);
1005    printf("Assoc. Mem Obj    0x%X\n", &mobj);
1006    clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_OFFSET, sizeof(size_t),
1007                                  &psize, NULL),
1008              __LINE__,__FILE__);
1009    printf("Offset            %d\n", psize);
1010    printf("\n");
1011    return;
1012 }
1013
1014 /**
1015  * clImgObjDump function
1016  *
1017  * \brief This function prints all of the information about the given
1018  *        image object
1019  *
1020  * \param img Image object
1021  * \return None
1022  *
1023  */
1024 extern "C" void clImgObjDump(cl_mem img) {
1025    cl_image_format fmt;
1026
1027    printf("Printing information for image object ID:0x%X\n", img);
1028    clCheckErr(clGetImageInfo(img, CL_IMAGE_FORMAT, sizeof(cl_image_format),
1029                              &fmt, NULL),
1030              __LINE__,__FILE__);
1031    pstr = (char*)malloc(sizeof(char)*STRLEN);
1032    clDecodeImageFormat(fmt, pstr);
1033    printf("%s\n",pstr);
1034    free(pstr);
1035    clCheckErr(clGetImageInfo(img, CL_IMAGE_ELEMENT_SIZE, sizeof(size_t),
1036                              &psize, NULL),
1037              __LINE__,__FILE__);
1038    printf("Element Size                  %d\n", psize);
1039    clCheckErr(clGetImageInfo(img, CL_IMAGE_ROW_PITCH, sizeof(size_t),
1040                              &psize, NULL),
1041              __LINE__,__FILE__);
1042    printf("Row Pitch                     %d\n", psize);
1043    clCheckErr(clGetImageInfo(img, CL_IMAGE_SLICE_PITCH, sizeof(size_t),
1044                              &psize, NULL),
1045              __LINE__,__FILE__);
1046    printf("Slice Pitch                   %d\n", psize);
1047    clCheckErr(clGetImageInfo(img, CL_IMAGE_WIDTH, sizeof(size_t),
1048                              &psize, NULL),
1049              __LINE__,__FILE__);
1050    printf("Width                         %d\n", psize);
1051    clCheckErr(clGetImageInfo(img, CL_IMAGE_HEIGHT, sizeof(size_t),
1052                              &psize, NULL),
1053              __LINE__,__FILE__);
1054    printf("Height                        %d\n", psize);
1055    clCheckErr(clGetImageInfo(img, CL_IMAGE_DEPTH, sizeof(size_t),
1056                              &psize, NULL),
1057              __LINE__,__FILE__);
1058    printf("Depth                         %d\n", psize);
1059    printf("\n");
1060    return;
1061 }
1062
1063 /**
1064  * clSamplerDump function
1065  *
1066  * \brief This function prints out all of the information about the sampler object
```

```
1067  *
1068  * \param smp Sampler object
1069  * \return None
1070  *
1071  */
1072 extern "C" void clSamplerDump(cl_sampler smp) {
1073    cl_context ctx;
1074    cl_addressing_mode amd;
1075    cl_filter_mode fmd;
1076
1077    printf("Printing information for sampler ID:0x%X\n", smp);
1078    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_REFERENCE_COUNT,
1079                                sizeof(cl_uint), &puint, NULL),
1080               __LINE__,__FILE__);
1081    printf("Reference Count             %d\n", puint);
1082    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_CONTEXT,
1083                                sizeof(cl_context), &ctx, NULL),
1084               __LINE__,__FILE__);
1085    printf("Context                     0x%X\n", ctx);
1086    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_ADDRESSING_MODE,
1087                                sizeof(cl_addressing_mode), &amd, NULL),
1088               __LINE__,__FILE__);
1089    pstr = (char*)malloc(sizeof(char)*STRLEN);
1090    clDecodeAddressingMode(amd, pstr);
1091    printf("Addressing Mode             %s\n", pstr);
1092    free(pstr);
1093    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_FILTER_MODE,
1094                                sizeof(cl_filter_mode), &fmd, NULL),
1095               __LINE__,__FILE__);
1096    pstr = (char*)malloc(sizeof(char)*STRLEN);
1097    clDecodeFilterMode(fmd, pstr);
1098    printf("Filter Mode                 %s\n", pstr);
1099    free(pstr);
1100    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_NORMALIZED_COORDS,
1101                                sizeof(cl_bool), &pbool, NULL),
1102               __LINE__,__FILE__);
1103    printf("Normalized Coordinates      %s\n", (pbool) ? "Yes" : "No");
1104    printf("\n");
1105    return;
1106 }
1107
1108 /**
1109  * clProgramDump function
1110  *
1111  * \brief This function prints out all of the information about the
1112  *        given program object
1113  *
1114  * \param pid Program id
1115  * \return None
1116  *
1117  */
1118 extern "C" void clProgamDump(cl_program pid) {
1119    int jdx, kdx = 0;
1120    cl_context ctx;
1121    cl_device_type dtp;
1122    size_t * psizes = NULL;
1123    unsigned char ** bins = NULL;
1124    unsigned char * bin = NULL;
1125    cl_device_id * dptr = NULL;
1126
1127    printf("Printing information for program ID:0x%X\n", pid);
1128    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_REFERENCE_COUNT,
1129                                sizeof(cl_uint), &puint, NULL),
```

```
1130              __LINE__,__FILE__);
1131    printf("Reference Count         %d\n", puint);
1132    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_CONTEXT,
1133                                sizeof(cl_context), &ctx, NULL),
1134              __LINE__,__FILE__);
1135    printf("Context                 0x%d\n", ctx);
1136    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_SOURCE, NULL, NULL, &psize),
1137              __LINE__,__FILE__);
1138    pstr = (char*)malloc(psize);
1139    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_SOURCE, psize, pstr, NULL),
1140              __LINE__,__FILE__);
1141    printf("Program Source\n");
1142    printf("%s\n", pstr);
1143    free(pstr);
1144    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_NUM_DEVICES,
1145                                sizeof(cl_uint), &puint, NULL),
1146              __LINE__,__FILE__);
1147    printf("Number of Devices               %d\n", puint);
1148    // Program devices
1149    psize = sizeof(cl_device_id)*puint;
1150    dptr = (cl_device_id*)malloc(psize);
1151    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_DEVICES, psize, dptr, NULL),
1152              __LINE__,__FILE__);
1153    // Program binary sizes
1154    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_BINARY_SIZES, NULL, NULL, &psize),
1155              __LINE__,__FILE__);
1156    psizes = (size_t*)malloc(psize);
1157    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_BINARY_SIZES, psize, psizes, NULL),
1158              __LINE__,__FILE__);
1159    psize = sizeof(unsigned char*) * puint;
1160    bins = (unsigned char**)malloc(psize);
1161    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_BINARIES, psize, bins, NULL),
1162              __LINE__,__FILE__);
1163    printf("Devices\n");
1164    for (jdx = 0; jdx < puint; jdx++) {
1165       printf("\t%02d) ", jdx);
1166       // Device name
1167       clCheckErr(clGetDeviceInfo(dptr[jdx], CL_DEVICE_NAME, NULL, NULL, &psize),
1168                 __LINE__,__FILE__);
1169       pstr = (char*)malloc(psize);
1170       clCheckErr(clGetDeviceInfo(dptr[jdx], CL_DEVICE_NAME, psize, pstr, NULL),
1171                 __LINE__,__FILE__);
1172       printf("%s ", pstr);
1173       free(pstr);
1174       // Device type
1175       clCheckErr(clGetDeviceInfo(dptr[jdx], CL_DEVICE_TYPE,
1176                                  sizeof(cl_device_type), &dtp, NULL),
1177                 __LINE__,__FILE__);
1178       pstr = (char*)malloc(sizeof(char)*STRLEN);
1179       clDecodeDeviceType(dtp, pstr);
1180       printf("(%s)\n", pstr);
1181       free(pstr);
1182       // Device Binary
1183       bin = bins[jdx];
1184       for (kdx = 0; kdx < psizes[jdx]; kdx++) {
1185          printf("%02X", bin[kdx]);
1186          if ((kdx % 4) == 3) { printf(" "); }
1187          if ((kdx % 16) == 15) { printf("\n"); }
1188       }
1189       printf("\n");
1190    }
1191    free(bins);
1192    free(psizes);
```

```
1193    free(dptr);
1194    return;
1195 }
1196
1197 /**
1198  * clProgramBuildDump function
1199  *
1200  * \brief This function prints out all of the information about the given program
1201  *        object's build for the specified device
1202  *
1203  * \param prg Program id
1204  * \param dev Device id
1205  * \return None
1206  *
1207  */
1208 extern "C" void clProgramBuildDump(cl_program prg, cl_device_id dev) {
1209    cl_build_status bst;
1210    cl_device_type dtp;
1211
1212    printf("Printing information for Program ID:0x%X Device ID:0x%X\n", prg, dev);
1213    // Device name
1214    clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_NAME, NULL, NULL, &psize),
1215               __LINE__,__FILE__);
1216    pstr = (char*)malloc(psize);
1217    clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_NAME, psize, pstr, NULL),
1218               __LINE__,__FILE__);
1219    printf("%s ", pstr);
1220    free(pstr);
1221    // Device type
1222    clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_TYPE, sizeof(cl_device_type),
1223                               &dtp, NULL),
1224               __LINE__,__FILE__);
1225    pstr = (char*)malloc(sizeof(char)*STRLEN);
1226    clDecodeDeviceType(dtp, pstr);
1227    printf("(%s)\n", pstr);
1228    free(pstr);
1229    // Program build status
1230    clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_STATUS,
1231                                     sizeof(cl_build_status), &bst, NULL),
1232               __LINE__,__FILE__);
1233    pstr = (char*)malloc(sizeof(char)*STRLEN);
1234    clDecodeBuildStatus(bst, pstr);
1235    printf("Build Status            %s\n", pstr);
1236    free(pstr);
1237    // Program build options
1238    clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_OPTIONS,
1239                                     NULL, NULL, &psize),
1240               __LINE__,__FILE__);
1241    pstr = (char*)malloc(psize);
1242    clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_OPTIONS,
1243                                     psize, pstr, NULL),
1244               __LINE__,__FILE__);
1245    printf("Build Options           %s\n", pstr);
1246    free(pstr);
1247    clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_LOG,
1248                                     NULL, NULL, &psize),
1249               __LINE__,__FILE__);
1250    pstr = (char*)malloc(psize);
1251    clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_LOG,
1252                                     psize, pstr, NULL),
1253               __LINE__,__FILE__);
1254    printf("Build Log\n%s", pstr);
1255    return;
```

```
1256 }
1257
1258 /**
1259  * clKernelDump function
1260  *
1261  * \brief This function prints out all of the information for the given
1262  *        kernel object
1263  *
1264  * \param krn Kernel object
1265  * \return None
1266  *
1267  */
1268 extern "C" void clKernelDump(cl_kernel krn) {
1269    printf("Printing information for Kernel ID:0x%X\n", krn);
1270    // Function name
1271    clCheckErr(clGetKernelInfo(krn, CL_KERNEL_FUNCTION_NAME,
1272                               NULL, NULL, &psize),
1273               __LINE__,__FILE__);
1274    pstr = (char*)malloc(psize);
1275    clCheckErr(clGetKernelInfo(krn, CL_KERNEL_FUNCTION_NAME,
1276                               psize, pstr, NULL),
1277               __LINE__,__FILE__);
1278    printf("Function Name: %s\n", pstr);
1279    free(pstr);
1280    // Kernel Number of Arguments
1281    clCheckErr(clGetKernelInfo(krn, CL_KERNEL_NUM_ARGS,
1282                               sizeof(cl_uint), &puint, NULL),
1283               __LINE__,__FILE__);
1284    printf("Num Args: %d\n", puint);
1285    // Kernel reference count
1286    clCheckErr(clGetKernelInfo(krn, CL_KERNEL_REFERENCE_COUNT,
1287                               sizeof(cl_uint), &puint, NULL),
1288               __LINE__,__FILE__);
1289    printf("Ref Cnt: %d\n", puint);
1290    return;
1291 }
1292
1293 /**
1294  * clPrintPlats function
1295  *
1296  * \brief This function prints out a very short list of the platforms
1297  *        found on the system
1298  *
1299  * \return None
1300  *
1301  */
1302 extern "C" void clPrintPlats() {
1303    cl_uint idx, np = 0;
1304    cl_platform_id * ps;
1305    cl_platform_id curplat;
1306
1307    // Query and list platforms
1308    clCheckErr(clGetPlatformIDs(0, NULL, &np),__LINE__,__FILE__);
1309    ps = (cl_platform_id*)malloc(sizeof(cl_platform_id)*(np));
1310    clCheckErr(clGetPlatformIDs(np,ps,NULL),__LINE__,__FILE__);
1311    for (idx = 0; idx < np; idx++) {
1312       curplat = ps[idx];
1313       printf("Platform %d: 0x%X ==> ", idx, curplat);
1314       clCheckErr(clGetPlatformInfo(curplat, CL_PLATFORM_VENDOR,
1315                                    NULL, NULL, &psize),
1316                  __LINE__,__FILE__);
1317       pstr = (char*)malloc(psize);
1318       clCheckErr(clGetPlatformInfo(curplat, CL_PLATFORM_VENDOR,
```

```
1319                                    psize, pstr, NULL),
1320              __LINE__,__FILE__);
1321       printf("%s ", pstr);
1322       free(pstr);
1323       clCheckErr(clGetPlatformInfo(curplat, CL_PLATFORM_NAME,
1324                                    NULL, NULL, &psize),
1325              __LINE__,__FILE__);
1326       pstr = (char*)malloc(psize);
1327       clCheckErr(clGetPlatformInfo(curplat, CL_PLATFORM_NAME,
1328                                    psize, pstr, NULL),
1329              __LINE__,__FILE__);
1330       printf("%s\n", pstr);
1331       free(pstr);
1332    }
1333    free(ps);
1334    return;
1335 }
1336
1337 /**
1338  * clPrintDevs function
1339  *
1340  * \brief This function prints out a very short list of the devices found
1341  *        on the specifed platform
1342  *
1343  * \param pid Platform ID
1344  * \return None
1345  *
1346  */
1347 extern "C" void clPrintDevs(cl_platform_id pid) {
1348    cl_uint idx, nd = 0;
1349    cl_device_id * ds;
1350    cl_device_id currdev;
1351    cl_device_type tmptype;
1352
1353    // Query and list devices
1354    clCheckErr(clGetDeviceIDs(pid, CL_DEVICE_TYPE_ALL, NULL, NULL, &nd),
1355               __LINE__,__FILE__);
1356    ds = (cl_device_id*)malloc(sizeof(cl_device_id)*(nd));
1357    clCheckErr(clGetDeviceIDs(pid, CL_DEVICE_TYPE_ALL, nd, ds, NULL),
1358               __LINE__,__FILE__);
1359
1360    for (idx = 0; idx < nd; idx++) {
1361       currdev = ds[idx];
1362       printf("\tDevice %d (ID:0x%X):", idx, currdev );
1363       clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_NAME, NULL, NULL, &psize),
1364              __LINE__,__FILE__);
1365       pstr = (char*)malloc(psize);
1366       clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_NAME, psize, pstr, NULL),
1367              __LINE__,__FILE__);
1368       printf(" %s", pstr);
1369       free(pstr);
1370       clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_TYPE,
1371                                  sizeof(cl_device_type), &tmptype, NULL),
1372              __LINE__,__FILE__);
1373       pstr = (char*)malloc(sizeof(char)*STRLEN);
1374       clDecodeDeviceType(tmptype, pstr);
1375       printf(" (%s)", pstr);
1376       free(pstr);
1377       clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_VERSION, NULL, NULL, &psize),
1378              __LINE__,__FILE__);
1379       pstr = (char*)malloc(psize);
1380       clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_VERSION, psize, pstr, NULL),
1381              __LINE__,__FILE__);
```

```
1382        printf(" ver. %s\n", pstr);
1383        free(pstr);
1384     }
1385     free(ds);
1386     return;
1387  }
1388
1389  /**
1390   * clDecodeTaskPlat function
1391   *
1392   * \brief This function concatenates a platform description string from the
1393   *        given platform index
1394   *
1395   * \param plat Platform type
1396   * \param cmpStr Output string
1397   * \return None
1398   *
1399   */
1400  extern "C" void clDecodeTaskPlat(int plat, char * cmpStr) {
1401     cmpStr[0] = '\0';
1402     switch(plat) {
1403        case 0: strcat(cmpStr, "NVIDIA"); break;
1404        case 1:    strcat(cmpStr, "AMD"); break;
1405        case 2:  strcat(cmpStr, "Intel"); break;
1406        default:
1407           printf("OCLUTIL ERROR: Invalid task platform\n");
1408           #ifdef KILLONERR
1409           exit(-559);
1410           #endif
1411     }
1412     return;
1413  }
1414
1415  /**
1416   * clFindPlatform function
1417   *
1418   * \brief This function returns the system platform index from the given
1419   *        platform type input
1420   *
1421   * \param plat Platform type
1422   * \return int Platform index
1423   *
1424   */
1425  extern "C" int clFindPlatform(int plat) {
1426     cl_uint nplat = 0;
1427     clCheckErr(clGetPlatformIDs(0, NULL, &nplat),__LINE__,__FILE__);
1428     cl_platform_id * plats = (cl_platform_id*)malloc(sizeof(cl_platform_id)*nplat);
1429     clCheckErr(clGetPlatformIDs(nplat,plats,NULL),__LINE__,__FILE__);
1430     char cmpStr[STRLEN];
1431     clDecodeTaskPlat(plat, cmpStr);
1432
1433     // For each platform, find and list the devices
1434     for (int idx = 0; idx < nplat; idx++) {
1435        cl_platform_id pid = plats[idx];
1436        clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, NULL, NULL, &psize),
1437                __LINE__,__FILE__);
1438        pstr = (char*)malloc(psize);
1439        clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, psize, pstr, NULL),
1440                __LINE__,__FILE__);
1441        if (strncmp(cmpStr,pstr,6)) { return idx; }
1442        free(pstr);
1443     }
1444     return -1;
```

```
1445 }
1446
1447 /**
1448  * clFindKernel function
1449  *
1450  * \brief This function returns the OpenCL kernel object specified by the
1451  *        kernel name in the specified list of kernels
1452  *
1453  * \param krns List of kernels
1454  * \param nrkn Number of kernels in list, krns
1455  * \param nrkName String of the kernel name to find
1456  * \return cl_kernel Found OpenCL kernel object
1457  *
1458  */
1459 extern "C" cl_kernel * clFindKernel(cl_kernel * krns, int nkrn,
1460                                     const char * krnName) {
1461    size_t nameSz = 0;
1462    char * name = NULL;
1463    for (int idx = 0; idx < nkrn; idx++) {
1464       clCheckErr(clGetKernelInfo(krns[idx], CL_KERNEL_FUNCTION_NAME,
1465                                  NULL, NULL, &nameSz),
1466                __LINE__,__FILE__);
1467       name = (char*)malloc(nameSz);
1468       clCheckErr(clGetKernelInfo(krns[idx], CL_KERNEL_FUNCTION_NAME,
1469                                  nameSz, name, NULL),
1470                __LINE__,__FILE__);
1471       if (!strcmp(name,krnName)) {
1472          free(name);
1473          return &(krns[idx]);
1474       }
1475       free(name);
1476    }
1477    printf("OCLUTIL ERROR: No valid kernel found in file:%s at line:%d\n",
1478                __FILE__,__LINE__);
1479    exit(-560);
1480    return NULL;
1481 }
```

Listing A.5: OpenCL helper function code

A.2.5 CPU Utility Functions

```
1  /**
2   *  Copyright (c) 2009, Massachusetts Institute of Technology
3   *  All rights reserved.
4   *
5   *  \author  $LastChangedBy$
6   *  \date    $LastChangedDate$
7   *  \version $LastChangedRevision$
8   *  \brief
9   *
10  *  $Id$
11  *
12  *  Author: James Brock
13  */
14 #include "cpuUtil.h"
15 #define CPU
16 #include "kernels.h"
17 #undef CPU
18
```

```
19 /**
20  * cpuInitMem function
21  *
22  * \brief  This function intelligently allocates memory on the host or device
23  *         as specified and returns pointer to the memory as well as the
24  *         stride for the data, which is used for multi-dimensional data
25  *         blocks.
26  * \param  dims The dimensions of the memory to be allocated
27  * \param  stride The stride (width in bytes) of the data. This only matters
28  *         for multi-dimensional data
29  * \param  loc The location of the memory to be allocated (host or device)
30  * \param  ptr A pointer to where the memory pointer is to be stored
31  * \return None
32  *
33  */
34 extern "C" void * cpuInitMem(int * dims, int typeSize, int * stride,
35                              const char * name, HTaskInfo * info,
36                              int mapHostFlag) {
37    void * locPtr = NULL;
38    int datSize = 1;
39
40    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
41    datSize *= typeSize;
42    locPtr = malloc(datSize);
43    memset(locPtr, 0, datSize);
44    return locPtr;
45 }
46
47 /**
48  * cpuFreeMem function
49  *
50  * \brief  This function frees allocated memory on the host or device
51  *         as specified.
52  * \param  ptr Pointer to the data to be freed
53  * \param  dims The dimensions of the data to be freed
54  * \param  loc The location of the data to be freed
55  * \return None
56  *
57  */
58 extern "C" void cpuFreeMem(void * ptr, int * dims, int typeSize, const char * name,
59                            HTaskInfo * info, int mapHostFlag ) {
60    free(ptr);
61    return;
62 }
63
64 /**
65  * cpuClearMem function
66  *
67  * \brief
68  * \param  dims The dimensions of the memory to be allocated
69  * \param  stride The stride (width in bytes) of the data. This only matters
70  *         for multi-dimensional data
71  * \param  loc The location of the memory to be allocated (host or device)
72  * \param  ptr A pointer to where the memory pointer is to be stored
73  * \return None
74  *
75  */
76 extern "C" void cpuClearMem(int * dims, int typeSize, int stride, void * ptr,
77                             HTaskInfo * info) {
78    int datSize = 1;
79    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
80    datSize *= typeSize;
81    memset(ptr, 0, datSize);
```

```
82    return;
83 }
84
85 /**
86  * cpuMoveData function
87  *
88  * \brief  This function intelligently copies data from the source location
89  *         with the specified stride to the destination location with the
90  *         specified stride. The dimensions parameter indicates the
91  *         dimensions of the data and must be the same for both source and
92  *         destination.
93  * \param  dest Pointer to the destination memory location
94  * \param  destStride Stride (width in bytes) of the destination
95  *         memory location
96  * \param  destLoc The location of the destination memory
97  * \param  src Pointer to the source memory location
98  * \param  srcStride Stride (width in bytes) of the source memory
99  *         location
100 * \param  srcLoc The location of the destination memory
101 * \param  dims The dimensions of the data to be copied
102 * \return None
103 *
104 */
105 extern "C" void cpuMoveData(  void * dst, int dstStride, HTaskInfo * dstInfo,
106                               void * src, int srcStride, HTaskInfo * srcInfo,
107                               int * dims, int typeSize, const char * name ) {
108    int datSize = 1;
109    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
110    datSize *= typeSize;
111    memcpy(dst, src, datSize);
112    return;
113 }
114
115 /**
116  * cpuLaunchKernel function
117  *
118  * \brief  This function uses the specified parameters to execute the
119  *         coprocessor function specified by kernel. Any parameters to
120  *         the kernel should be included in params.
121  *
122  * \param  kernel A string naming the kernel to execute
123  * \param  dims The dimensions of the data to execute the kernel on
124  * \param  nParams The number of parameters to be passed to the kernel
125  * \param  params Array of parameters to be passed to the kernel
126  * \param  gDim The grid dimensions of the kernel
127  * \param  bDim The block dimensions of the kernel
128  * \param  locMem The amount of local or shared memory to allocate for kernel
129  *         execution
130  * \param  stream The stream index to associate this kernel's execution and
131  *         data with
132  * \return None
133  *
134  */
135 extern "C" void cpuLaunchKernel( const char * krn, int * dims, int nParams,
136                                  int * paramSizes, void ** params,
137                                  int * gDim, int * bDim, int locMem,
138                                  HTaskInfo * info) {
139    printf("Launching %s on cpu\n", krn);
140    if (!strcmp(krn,"cpu_mc_stage1")) {
141       cpu_mc_stage1(*((randState**)params[0]), *((phDetPkg**)params[1]),
142                     *((uint**)params[2]), *((float*)params[3]), *((int*)params[4]),
143                     *((InputStruct**)params[5]));
144    }
```

```
145     if (!strcmp(krn,"cpu_mc_stage2")) {
146        cpu_mc_stage2(*((phDetPkg**)params[0]), *((positionGrid**)params[1]),
147                      *((uint**)params[2]), *((float*)params[3]));
148     }
149     return;
150 }
```

Listing A.6: CPU utility function code

Bibliography

[1] V. Aggarwal, R. Garcia, G. Stitt, A. George, and H. Lam. Scf: a device- and language-independent task coordination framework for reconfigurable, heterogeneous systems. In *HPRCTA '09: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 19–28, New York, NY, USA, 2009. ACM.

[2] V. Aggarwal, A. George, K. Yalamanchili, C. Yoon, H. Lam, and G. Stitt. Bridging parallel and reconfigurable computing with multilevel PGAS and SHMEM+. In *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 47–54, Portland, Oregon, 2009. ACM.

[3] G. R. Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[4] M. S. Aninash C. Kak. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, 2001.

[5] J. Breitbart. CuPP - a framework for easy CUDA integration. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.

[6] J. Brock, M. Leeser, and M. Niedre. Adding support for GPUs to PVTOL: The Parallel Vector Tile Optimizing Library. In *14th Annual High Performance Embedded Computing Workshop*, 2010.

[7] J. Brock, M. Leeser, and M. Niedre. Heterogeneous Tasks and Conduits Framework for Rapid Application Portability and Deployment. In *Innovative Parallel Computing 2012*, 2012.

[8] I. Buck. The Brook+ Project. http://sourceforge.net/projects/brookplus/, 2010.

[9] R. Chamberlain, M. Franklin, E. Tyson, J. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. Shands, and N. Singla. Auto-Pipe: streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, 2010.

[10] S.-L. Chu and C.-C. Hsiao. OpenCL: make ubiquitous supercomputing possible. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 556 –561, Sept. 2010.

[11] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369 –1386, aug. 2012.

[12] Q. Fang and D. A. Boas. Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units. *Optics Express*, 17(22):20178–20190, Oct. 2009.

[13] D. Fay, L. Shang, and D. Grunwald. A platform for developing adaptable multicore applications. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 157–166, New York, NY, USA, 2009. ACM.

[14] L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America A*, 1(6):612–619, June 1984.

[15] J. Fessler. Image Reconstruction Toolbox Software. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, June 2012.

[16] A. H. Hielscher. Optical tomographic imaging of small animals. *Current Opinion in Biotechnology*, 16(1):79–88, Feb. 2005.

[17] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. Pfunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 43:1–43:11, New York, NY, USA, 2009. ACM.

[18] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. *arXiv:1005.2581*, May 2010.

[19] Khronos Group. The OpenCL Specification v1.2. http://www.khronos.org/registry/cl/, June 2012.

[20] H. Kim, E. Rutledge, S. Sacco, S. Mohindra, M. Marzilli, J. Kepner, R. Haney, J. Daly, and N. Bliss. PVTOL: providing productivity, performance and portability to DoD signal processing applications on multicore processors. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 327–333, 2008.

[21] Y. Liu, E. Zhang, and X. Shen. A cross-input adaptive framework for GPU program optimizations. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[22] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[23] W. mei Hwu. *GPU Computing Gems: Emerald Edition*. Morgan Kaufmann Publishers, 2011.

[24] Mercury Federal Systems, Inc. Open component portability infrastructure (opencpi), 2010.

[25] S. Mohindra, J. Daly, R. Haney, and G. Schrader. Task and conduit framework for multi-core systems. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 506–513, 2008.

[26] MPI Working Group. Message Passing Interface Specification. http://www.mpi-forum.org/, Sept. 2012.

[27] S. Mukherjee, N. Moore, J. Brock, and M. Leeser. CUDA and OpenCL Implementations of 3D CT Reconstruction for Biomedical Imaging. In *High Performance Extreme Computing Conference*, 2012.

[28] M. J. Niedre, R. H. de Kleine, E. Aikawa, D. G. Kirsch, R. Weissleder, and V. Ntziachristos. Early photon tomography allows fluorescence detection of lung carcinomas and disease progression in mice in vivo. *Proceedings of the National Academy of Sciences of the United States of America*, 105(49):19126–19131, Dec. 2008. PMID: 19015534.

[29] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[30] P. B. Nol, A. M. Walczak, J. Xu, J. J. Corso, K. R. Hoffmann, and S. Schafer. GPU-based cone beam computed tomography. *Computer Methods and Programs in Biomedicine*, 98(3):271–277, June 2010.

[31] NVIDIA. NVIDIA Compute Unified Device Architecture (CUDA) 4.2. http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation, Mar. 2012.

[32] Y. Okitsu, F. Ino, and K. Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Computing*, 36(2-3):129–141, Feb. 2010.

[33] Open Source Community. Boost C++ Libraries. http://www.boost.org/, Mar. 2012.

[34] OpenMPI Group. The OpenMPI Project. http://www.open-mpi.org/, June 2012.

[35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of General-Purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80113, 2007.

[36] M. Sato. OpenMP: Parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *System Synthesis, 2002. 15th International Symposium on*, pages 109 –111, Oct. 2002.

[37] S. Singh. Computing without processors. *Commun. ACM*, 54(8):46–54, Aug. 2011.

[38] J. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[39] N. Valim, J. Brock, and M. Niedre. Experimental measurement of time-dependant photon scatter for diffuse optical tomography. *Journal of Biomedical Optics*, 15(6):065006, 2010.

[40] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11, Nov. 2008.

[41] L. V. Wang, H. Wu, and B. R. Masters. Biomedical optics, principles and imaging. *Journal of Biomedical Optics*, 13(4):049902, 2008.

[42] M. Wolfe. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, page 4350, New York, NY, USA, 2010. ACM. ACM ID: 1735697.