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 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

1 Introduction

1.1 Research Contributions ........................................ 3

2 Background ......................................................... 7

2.1 Heterogeneous and Parallel Programming ....................... 8

2.1.1 Graphics Processing Unit (GPU) ............................ 8

2.1.2 NVIDIA Compute Unified Device Architecture (CUDA) ... 10

2.1.3 Open Computing Language (OpenCL) ....................... 13

2.1.4 Other Parallel Programming Models ........................... 16

2.2 PVTOL Tasks and Conduits Framework ......................... 18

2.2.1 PVTOL Tasks ................................................. 20

2.2.2 PVTOL Conduits ............................................. 22

2.2.3 PVTOL Applications and Architectures ..................... 24
2.3 Fluorescence Mediated Tomography (FMT) .................................. 26
2.4 3D Cone Beam Computed Tomography (3D CBCT) ..................... 28
2.5 Related Work ......................................................... 31
   2.5.1 Programming Languages ........................................ 31
   2.5.2 Compilers and Interpreters .................................... 33
   2.5.3 Software Frameworks .......................................... 35
   2.5.4 Summary of Related Work ..................................... 38

3 Methodology and Design .................................................. 39
   3.1 Heterogeneous Tasks and Conduits ................................ 39
      3.1.1 Heterogeneous Utility Functions ............................... 42
       Heterogeneous Map Structure .................................... 46
       CUDA Support ...................................................... 47
       OpenCL Support .................................................. 48
       C/C++ Support .................................................. 49
      3.1.2 Heterogeneous Task ......................................... 50
      3.1.3 Heterogeneous Conduit ..................................... 52
   3.2 Heterogeneous Applications ........................................ 54

4 Experimental Setup and Results ........................................ 60
   4.1 Accelerated FMT Application ...................................... 60
   4.2 Accelerated 3D CBCT Application ................................. 64
4.3 Experimental Computing Architectures .......................... 66
  4.3.1 NVIDIA 9800GX2 Workstation .............................. 66
  4.3.2 NVIDIA GTX560 Ti Workstation .............................. 67
  4.3.3 NEU Medusa NVIDIA S1070 Cluster Node .................. 67
  4.3.4 NEU Medusa AMD Cypress 5870 Server ..................... 68
  4.3.5 Harvard NNIN Cluster (NVIDIA Tesla C1060) ............. 68

4.4 Results .......................................................... 69
  4.4.1 FMT Application .............................................. 69
    Portability and Performance .................................... 71
  4.4.2 3D CBCT Application ......................................... 77
    Portability and Performance .................................... 78
  4.4.3 Framework Overhead .......................................... 82

5 Conclusions ......................................................... 86
  5.1 Future Work .................................................... 88

A Appendixes .......................................................... 90
  A.1 Heterogeneous PVTOL Objects ................................. 90
    A.1.1 Heterogeneous Map ......................................... 90
  A.2 Heterogeneous Utility Functions ............................... 92
    A.2.1 Heterogeneous Utility Functions API ...................... 92
    A.2.2 CUDA Utility Functions .................................... 95
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 capa-
cable of executing on many modern heterogeneous computing platforms with minimal
effort.

1.1 Research Contributions

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 maintaining a platform independent framework, a barrier between the different programming model APIs and the tasks and conduits is needed. We have developed the heterogeneous utility functions which create an interface between the higher level functionality of the tasks and conduits framework and the various platform-specific programming models they will need to utilize. The tasks and conduits code can now interface to the appropriate programming model based on the run time application mapping. This also has the 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.

![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
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.
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.

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

Figure 2.4: OpenCL memory and computational organization[19]
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\(^\text{[22]}\). 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\(^\text{[11]}\). 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\(^\text{[3]}\). The model is very flexible but low level, and is usually associated with shared memory and operating systems\(^\text{[11]}\). This model has locks and condition variables for managing access to shared memory locations and uses the fork/join parallel programming pattern\(^\text{[22]}\). 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 runtime 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)\cite{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\cite{26}.

One of the most popular implementations of the message passing interface is OpenMPI\cite{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\cite{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\cite{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 (\texttt{pthreads}), MPI\[26\], and \texttt{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.

![PVTOL TASK](image)

**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](image)

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
CHAPTER 2. BACKGROUND

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.

```c
int main(int argc, char *argv[]) {
    // Initialize PVTOL and each task map
    PvtolProgram prog(argc, argv);
    rank.push_back(0);
    RankList ranks(rank);
    TaskMap taskMap(ranks);

    // Declare each task and conduit
    Task<DI T> dit("DI T", taskMap);
    Task<FTT> fft("FTT", taskMap);
    Task<FTL R> filt("FILTER", taskMap);
    Task<IFFT> ifft("IFFT", taskMap);
    Task<DOT> dot("DOT", taskMap);
    HeterogeneousConduit< float> cdt0("DI T to FTT");
    HeterogeneousConduit< float> cdt1("FTT to FILTER");
    HeterogeneousConduit< float> cdt2("FILTER to IFFT");
    HeterogeneousConduit< float> cdt3("IFFT to DOT");

    // Initialize the tasks with their conduits
    dit.init(cdt0.getWriter());
    fft.init(cdt0.getReader(), cdt1.getWriter());
    filt.init(cdt1.getReader(), cdt2.getWriter());
    ifft.init(cdt2.getReader(), cdt3.getWriter());
    dot.init(cdt3.getReader());

    // Run the tasks
    dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();

    // Wait until tasks complete
    dit.waitTillDone(); fft.waitTillDone();
    filt.waitTillDone(); ifft.waitTillDone();
    dot.waitTillDone();
}
```

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).](image)

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 adapt-
able 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].

$$F(x, y, z) = \frac{1}{2\pi t} \sum_{i=1}^{t} W_2(x, y, i)Q_1(u(x, y, i), v(x, y, z, i))$$  \hspace{1cm} (2.1)
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.

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

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

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

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 or-
der 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 program-
ing by enabling code to be adapted to different architectures for better data par-
allelization 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.\cite{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\cite{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 \texttt{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 \texttt{X} language is a development environment for pipelined applications ex-
executing 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 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.

```c
int main(int argc, char *argv[]) {
    // Declare variables
    cufftComplex *hostIn, *hostOut;
    cufftComplex *devIn, *devOut;
    cufftHandle planFwd;
    int N = 1024, batchSize = 10;
    int idx, nBytes;

    // Allocate host-side memory
    nBytes = sizeof(cufftComplex)*N*batchSize;
    hostIn = (float*)malloc(nBytes);
    hostOut = (float*)malloc(nBytes);

    // Initialize input data
    for (idx = 0; idx < N*batchSize; idx++) {
        hostIn[idx].x = sinf(idx); // Data Input Function
        hostIn[idx].y = cosf(idx); // inFunc
    }

    // Allocate device-side memory
    cudaMalloc((void**)&devIn, nBytes);
    cudaMalloc((void**)&devOut, nBytes);

    // Copy input data from host to device
    cudaMemcpy(devIn, hostIn, nBytes, cudaMemcpyHostToDevice);

    // Execute kernels to process data
    cufftPlan1d(&planFwd, N, CUFFT_C2C);
    for (idx = 0; idx < batchSize; idx++) {
        cufftExecC2C(planFwd,
                    devIn[idx*N], // Forward FFT
                    devOut[idx*N]);
    }

    // Copy output data from device to host
    cudaMemcpy(hostOut, devOut, nBytes, cudaMemcpyDeviceToHost);

    // Process output data (compare, write to disk, etc.)
    double maxErr = 0.0;
    double tmp = 0.0;
    for (idx = 0; idx < N*batchSize; idx++) {
        tmp = hostOut[idx].x/N - sinf(idx); // Data Output Function
        maxErr = max(fabs(tmp), maxErr); // outFunc
        tmp = hostOut[idx].y/N - cosf(idx);
        maxErr = max(fabs(tmp), maxErr);
    }
    printf("Max FFT error = %g\n", maxErr);
```

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.

```c
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.

ability 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, GUFFTW, 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 straightforward implementations using the standard C memory functions \texttt{malloc}, \texttt{memset}, and \texttt{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 (\texttt{init} and \texttt{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 \texttt{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.

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

Figure 3.2: Example processing element mappings for a heterogeneous task
can be mapped to one or more CUDA streams or OpenCL command queues. This allows a single 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-
Table 3.2: All supported non-distributed memory heterogeneous conduit configurations

<table>
<thead>
<tr>
<th>Source Endpoint</th>
<th>Intermediate Interface</th>
<th>Destination Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>CPU</td>
<td>CUDA Device</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>OpenCL Device</td>
<td></td>
</tr>
<tr>
<td>CUDA Device</td>
<td>CPU</td>
<td>CUDA Device B</td>
</tr>
<tr>
<td>OpenCL Device</td>
<td>CPU</td>
<td>OpenCL Device B</td>
</tr>
<tr>
<td>CUDA Device A</td>
<td>CPU</td>
<td>OpenCL Device</td>
</tr>
<tr>
<td>OpenCL Device A</td>
<td>CPU</td>
<td>CUDA Device A</td>
</tr>
<tr>
<td>CUDA Device</td>
<td>shared</td>
<td>CUDA Device B</td>
</tr>
<tr>
<td>OpenCL Device A</td>
<td>shared</td>
<td>OpenCL Device B</td>
</tr>
</tbody>
</table>

Intermediate 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.

```c
int main(int argc, char *argv[]) {
    // Declare variables
    cufftHandle plan;
    int N = 1024, batchSize = 10;
    int idx, nBytes;

    // Initialize system
    PvtolProgram prog(argc, argv);
    initSystem(CUDA_SYS_FLAG);
    cufftPlan1d(&plan, N, CUFFT_C2C);

    // Create task maps
    rank.push_back(0);
    RankList ranks(rank);
```
CHAPTER 3. METHODOLOGY AND DESIGN

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

```c
int main(int argc, char *argv[]) {
    // Initialize PVTOL and each task map
    PvtolProgram prog(argc, argv);
    rank.push_back(0);
    RankList ranks(rank);
```
CHAPTER 3. METHODOLOGY AND DESIGN

```cpp
// Declare each task and conduit
Task<DIR> dit("DIR", taskMap);
Task<FFT> fft("FFT", taskMap, gpuMap);
Task<FILTER> filt("FILTER", taskMap, gpuMap);
Task<IFFT> ifft("IFFT", taskMap, gpuMap);
Task<DOT> dot("DOT", taskMap);
HeterogeneousConduit<float> cd0("DIR to FFT");
HeterogeneousConduit<float> cd1("FFT to FILTER");
HeterogeneousConduit<float> cd2("FILTER to IFFT");
HeterogeneousConduit<float> cd3("IFFT to DOT");

// Initialize the tasks with their conduits
dit.init(cd0.getWriter());
fft.init(cd0.getReader(), cd1.getWriter());
filt.init(cd1.getReader(), cd2.getWriter());
ifft.init(cd2.getReader(), cd3.getWriter());
dot.init(cd3.getReader());
// Run the tasks
dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();
// Wait until tasks complete
dit.waitTillDone(); fft.waitTillDone();
filt.waitTillDone(); ifft.waitTillDone();
dot.waitTillDone();
```

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.
CHAPTER 3. METHODOLOGY AND DESIGN

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

```c
int main(int argc, char *argv[]) {
    // Initialize PVTOL and each task map
    PvtolProgram prog(argc, argv);
    rank.push_back(0);
    RankList ranks(rank);
    TaskMap taskMap(ranks);
    HTaskInfo hInfoGPU1 = {LOC_CUDA, 1, 0, PLAT_NVIDIA};
    HTaskInfo hInfoGPU2 = {LOC_CUDA, 2, 0, PLAT_NVIDIA};
    HTaskInfo *hInfos[2] = {hInfoGPU1, hInfoGPU2};
    HeterogeneousMap gpuMap(hInfoGPU1, 2);

    // Declare each task and conduit
    Task<DIT> dit("DIT", taskMap);
    Task<FFT> fft("FFT", taskMap, gpuMap);
    Task<FILTER> filt("FILTER", taskMap, gpuMap);
    Task<IFFT> ifft("IFFT", taskMap, gpuMap);
    Task<DOT> dot("DOT", taskMap);
    HeterogeneousConduit<float> cdt0("DIT to FFT");
    HeterogeneousConduit<float> cdt1("FFT to FILTER");
    HeterogeneousConduit<float> cdt2("FILTER to IFFT");
    HeterogeneousConduit<float> cdt3("IFFT to DOT");

    // Initialize the tasks with their conduits
    dit.init(cdt0.getWriter());
    fft.init(cdt0.getReader(), cdt1.getWriter());
    filt.init(cdt1.getReader(), cdt2.getWriter());
    ifft.init(cdt2.getReader(), cdt3.getWriter());
    dot.init(cdt3.getReader());

    // Run the tasks
    dit.run(); fft.run(); filt.run(); ifft.run(); dot.run();

    // Wait until tasks complete
    dit.waitTillDone(); fft.waitTillDone();
    filt.waitTillDone(); ifft.waitTillDone();
    dot.waitTillDone();
}
```

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 PV-TOL 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\cite{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\cite{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.

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

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

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

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.
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](image)

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

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.

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

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 Core2 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, \( \mu-a \) is the absorption coefficient, and \( \mu-s \) 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.

<table>
<thead>
<tr>
<th>Optical Properties (n,g,μ-s,μ-a)</th>
<th>Geometric Dimensions (x,y,z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.40, 0.85, 113.333cm⁻¹, 0.150cm⁻¹)</td>
<td>(4cm, 4cm, 2cm)</td>
</tr>
<tr>
<td>(1.40, 0.85, 87.000cm⁻¹, 0.200cm⁻¹)</td>
<td>(8cm, 8cm, 4cm)</td>
</tr>
<tr>
<td>(1.40, 0.90, 100.000cm⁻¹, 0.180cm⁻¹)</td>
<td>(6cm, 6cm, 6cm)</td>
</tr>
<tr>
<td>(1.40, 0.90, 95.000cm⁻¹, 0.200cm⁻¹)</td>
<td>(4cm, 4cm, 8cm)</td>
</tr>
</tbody>
</table>

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.

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
is allowed for them to propagate. The intensity images show a 2D slice of the 3D
CHAPTER 4. EXPERIMENTAL SETUP AND RESULTS

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 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.

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

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.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Application Configuration</th>
<th>Execution Time (HH:MM:SS)</th>
<th>Speedup</th>
<th>SLOC change</th>
</tr>
</thead>
<tbody>
<tr>
<td>9800GX2 Workstation</td>
<td>1 CPU (1 Thread)</td>
<td>85:56:20</td>
<td>1x</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>1 CPU (4 threads)</td>
<td>09:57:24</td>
<td>8.63x</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.7: Photon propagation output of the FMT application on CPU

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

Table 4.8: Photon propagation output of the PVTOL GPU FMT application
As can be seen from Table 4.8, the CPU-based version of the two stage FMT tasks and conduits application performs much better than the original single-threaded version. The multi-threaded version of the 2-stage application produces an almost 9x speedup over the single threaded 1-stage version on the CPU, demonstrating the benefits of using the tasks and conduits framework regardless of GPU acceleration. The step of breaking up the FMT code into a two stage algorithm and running it on the CPU and GPU architectures yielded even better speedups, as can be seen from Table 4.8. 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. All speedup results presented in Table 4.8 are relative to the multi-threaded CPU version of the application. 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, and the same GPU kernels are used in all of the GPU implementations with no change.

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 coalesced, is to move memory operations to a third stage where the global memory write operations are re-organized and coalesced 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.

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 phan-
toms, the floating point error of the reconstructed images is $< 0.01\%$ and well within
reasonable bounds to consider the reconstruction successful.

**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
Figure 4.8: Cross sections of the image reconstructions of 3D cone beam phantoms divided amongst the multiple parallel instances. Table 4.9 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
### Table 4.9: Photon propagation output of the PVTOL GPU 3D CBCT application

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

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.

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 coalesced 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 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.
CHAPTER 4. EXPERIMENTAL SETUP AND RESULTS

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. This indicates that for as the data set size increases and kernel run time increases, the framework overhead will approach 0%.

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 allows the developer to easily explore a very large application design space and achieve large speedups with
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.

minimal effort.
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.

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

```cpp
/**
 * File: HeterogeneousMap.h
 *
 * 
 * \file HeterogeneousMap.h
 * \author $LastChangedBy: $
 * \date $LastChangedDate: $
 * \version $LastChangedRevision: $
 * \brief Definition of the HeterogeneousMap class.
 *
 * Author: James Brock
 *
 * \Id: $
 */

#ifndef PVTOL_HETEROGENEOUSMAP_H_
#define PVTOL_HETEROGENEOUSMAP_H_

namespace ipvtol {

/// \brief class HeterogeneousMap
/// The HeterogeneousMap class is used to provide mapping information for
/// Heterogeneous Tasks. It inherits from the abstract base Map class.
class HeterogeneousMap {

public:
    /// \brief Default Constructor
    HeterogeneousMap();

    /// \brief Constructor
    HeterogeneousMap(const HTaskInfo * ranks, const int size);

    /// \brief Copy Constructor
    HeterogeneousMap(const HeterogeneousMap & other);

    /// \brief Destructor
    ~HeterogeneousMap();

```
class HeterogeneousMap {

  // class HeterogeneousMap
  // ------------------
  // INLINE FUNCTIONS
  // ------------------
  // brief Destructor
  inline HeterogeneousMap::~HeterogeneousMap() { hMap.clear(); }
  // brief Default constructor
  inline HeterogeneousMap::HeterogeneousMap() { 
    static HTaskInfo tmp_info = { LOC_CPU, 0, 0, PLAT_C }; 
    hMap.push_back(tmp_info); 
  }
  // brief Copy constructor
  inline HeterogeneousMap::HeterogeneousMap(const HeterogeneousMap & other) {
    hMap = other.hMap; 
  }
  // brief Constructor
  inline HeterogeneousMap::HeterogeneousMap(const HTaskInfo * ranks, const int size) {
    for (int idx = 0; idx < size; idx++) { hMap.push_back(ranks[idx]); } 
  }
  // brief getMap function
  inline void HeterogeneousMap::setMap(vector<HTaskInfo> map) { hMap.clear(); hMap = map; }
  // brief getMap function
  inline vector<HTaskInfo> * HeterogeneousMap::getMap() { return &hMap; }
  // brief getSize function
  inline int HeterogeneousMap::getSize() { return hMap.size(); }
  // brief getTaskInfo function
  inline HTaskInfo * HeterogeneousMap::getTaskInfo(int idx) { return &(hMap[idx]); }
};
} // namespace ipvtol
A.2 Heterogeneous Utility Functions

A.2.1 Heterogeneous Utility Functions API

```c
/**
 * 
 * \file hUtil.h
 * \author $LastChangedBy:
 * \date $LastChangedDate:
 * \version $LastChangedRevision:
 * \brief Function declarations and wrappers for the CUDA Utility functions.
 * This file is used to link the CUDA Utility functions to the PVtol Tasks and Conduits framework software and can not contain any CUDA specific functions or types.
 * 
 * $Id:
 * Author: James Brock
 */

#ifndef HUTIL_H
#define HUTIL_H

#include <PvtolBasics.h>
#include "cpuUtil.h"
#include "cudaUtil.h"
#include "oclUtil.h"

#define NO_HSYS_FLAG 0x0
#define CUDA_SYS_FLAG 0x10
#define OCL_SYS_FLAG 0x20

#define HUTIL_H
#endif

// Function declarations

extern "C" void initSystem( int flags );
extern "C" void initDevice( HTaskInfo * info );
extern "C" void closeSystem( int flags );

extern "C" void * initMem( int * dims , int typeSize , int * stride ,
                           const char * name , HTaskInfo * info , int mapHostFlag );
extern "C" void freeMem( void * ptr , int * dims , int typeSize , const char * name ,
                        HTaskInfo * info , int mapHostFlag );
extern "C" void clearMem( int * dims , int typeSize , int stride , void * ptr ,
                         HTaskInfo * info );
extern "C" void moveData( void * dst , int dstStride , HTaskInfo * dstInfo ,
                         void * src , int srcStride , HTaskInfo * srcInfo ,
                         int * dims , int typeSize , const char * name );

extern "C" void build( HTaskInfo * info , char * srcFile );
extern "C" void safeBuild( HTaskInfo * info , char * srcFile );
extern "C" void launchKernel( const char * krm , int * dims , int nParams ,
                             int * paramSizes , void ** params , int * gDim ,
                             int * bDim , int locMem , HTaskInfo * info );
```
// Helper functions
extern "C" hTaskLoc findLocation(HTaskInfo * infos, int ni);

.Designer Functions

/* findLocation function */
extern "C" inline
hTaskLoc findLocation(HTaskInfo * infos, int ni) {
    int idx = 0;
    hTaskLoc loc = LOC_INVALID;
    for (idx = 0; idx < ni; idx++) {
        if (infos[idx].location > loc) { loc = infos[idx].location; }
    }
    return loc;
}

/* initSystem function */
extern "C" inline
void initSystem (int flags) {
    if (flags & CUDA_SYS_FLAG) { initCUDA(flags); }
    if (flags & OCL_SYS_FLAG) { initOpenCL(); }
    return;
}

/* closeSystem function */
extern "C" inline
void closeSystem (int flags) {
    if (flags & CUDA_SYS_FLAG) { closeCUDA(); }
    if (flags & OCL_SYS_FLAG) { closeOpenCL(); }
    return;
}

/* initDevice function */
extern "C" inline
void initDevice (HTaskInfo * info) {
    hTaskLoc loc = findLocation(info, 1);
    if (loc == LOC_CUDA) {
        cudaInitDevice(info);
    } else if (loc == LOC_OCL) {
        oclInitDevice(info);
    } else if (loc == LOC_CPU) {
        // Do nothing
    } else {
        printf("HUTIL ERROR: Invalid heterogeneous device location \%s:\%d\n", __FILE__, __LINE__);
    }
    return;
}

/* initMem function */
extern "C" inline
void * initMem (int * dims, int typeSize, int * stride, const char * name,
               HTaskInfo * info, int mapHostFlag) {
    void * ptr = NULL;
    hTaskLoc loc = findLocation(info, 1);
    if (loc == LOC_CUDA) {
        ptr = cudaInitMem(dims, typeSize, stride, name, info, mapHostFlag);
    } else if (loc == LOC_OCL) {
        ptr = oclInitMem(dims, typeSize, stride, name, info, mapHostFlag);
    } else if (loc == LOC_CPU) {
        ptr = cpuInitMem(dims, typeSize, stride, name, info, mapHostFlag);
    } else {
        printf("HUTIL ERROR: Invalid heterogeneous device location \%s:\%d\n", __FILE__, __LINE__);
    }
    return ptr;
}
APPENDIX A. APPENDIXES

114             __FILE__, __LINE__);  
115         return ptr;  
116     }  
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         if ( loc == LOC_CUDA ) {  
163             cudaMemcpy(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,  
164                    typeSize, name);  
165         } else if ( loc == LOC_OCL ) {  
166             cudaMemcpy(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,  
167                    typeSize, name);  
168         } else if ( loc == LOC_CPU ) {  
169             cudaMemcpy(dst, dstStride, dstInfo, src, srcStride, srcInfo, dims,  
170                    typeSize, name);  
171         } else {  
172             printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",  
173                    loc, __FILE__, __LINE__);  
174         }  
175     return;
/* launchKernel function */

extern "C" inline

void launchKernel(const char * krn, int * dims, int nParams, int * paramSizes,
                  void ** params, int * gDim, int * bDim, int locMem,
                  HTaskInfo * info) {
    hTaskLoc loc = findLocation(info, 1);
    if (loc == LOC_CUDA) {
        cudaLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
                         locMem, info);
    } else if (loc == LOC_OCL) {
        oclLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
                        locMem, info);
    } else if (loc == LOC_CPU) {
        cpuLaunchKernel(krn, dims, nParams, paramSizes, params, gDim, bDim,
                        locMem, info);
    } else {
        printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
               loc, __FILE__, __LINE__);
    }
    return;
}

extern "C" inline

void build(HTaskInfo * info, char * srcFile) {
    hTaskLoc loc = info->location;
    if (loc == LOC_OCL) {
        oclBuild(info, srcFile);
    } else {
        printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
               loc, __FILE__, __LINE__);
    }
}

extern "C" inline

void safeBuild(HTaskInfo * info, char * srcFile) {
    hTaskLoc loc = info->location;
    if (loc == LOC_OCL) {
        oclSafeBuild(info, srcFile);
    } else {
        printf("HUTIL ERROR: Invalid heterogeneous device location loc:%d %s:%d\n",
               loc, __FILE__, __LINE__);
    }
    return;
}

#endif

/* END UTILITY FUNCTIONS */

Listing A.2: Heterogeneous utility function API

A.2.2 CUDA Utility Functions
APPENDIX A. APPENDIXES

96

* $Id$
*
* Author: James Brock
*/

#include "cudaUtil.h"
define CUDA
#include "kernels.h"
#endif CUDA

// Global variables
cudaDeviceProp ** devProp; // CUDA Device properties structures
int nDev; // Number of CUDA devices available
int cudaSysInit = 0; // System initialization flag
int sysFlags; // System flags

/**
 * cudaCheckErr function
 *
 * brief This function checks to see if any errors have occurred in using CUDA, and
 * prints out the relevant error information. This function only checks the
 * last error to occur.
 *
 * param err The error code to check
 * param line The line at which the error code was generated
 * param file The file in which the error code was generated
 * return None
 *
*/
extern "C" void cudaCheckErr(cudaError_t err, int line, char * file) {
    if (err != cudaSuccess) {
        const char * err_str = cudaGetErrorString(err);
        printf("CUDA ERROR %d: %s on line %d in file %s\n", err, err_str, line, file);
        #ifdef KILLONERR
        exit(err);
        #endif
    }
    return;
}

/**
 * initCUDA function
 *
 * brief This function will initialize the CUDA system, finding any devices
 * available, and initializing any 3rd party libraries requested
 * return None
 *
*/
extern "C" void initCUDA(int flags) {
    sysFlags = flags;
    if (cudaSysInit == 0) {
        cudaCheckErr(cudaGetDeviceCount(&nDev),__LINE__,__FILE__);
        devProp = (cudaDeviceProp**)malloc(sizeof(cudaDeviceProp*)*nDev);
        for (int idx = 0; idx < nDev; idx++) {
            devProp[idx] = new cudaDeviceProp;
        }
        if (sysFlags & CUBLAS_FLAG) { /* Init CUBLAS interface */ }
        if (sysFlags & CUFFT_FLAG) { /* Init CUFFT interface */ }
        if (sysFlags & CURAND_FLAG) { /* Init CURAND interface */ }
    }
    return;
}
/**
 * closeCUDA function
 *
 * \brief This function will clean up, releasing and deleting all of the CUDA
 * objects that were allocated or initialized during program execution
 *
 * \return None
 */
extern "C" void closeCUDA () {
   if ( sysFlags & CUFFT_FLAG ) { /* Close CUFFT interface */ }
   if ( sysFlags & CUBLAS_FLAG ) { /* Close CUBLAS interface */ }
   if ( sysFlags & CURAND_FLAG ) { /* Close CURAND interface */ }
   // Delete and close main CUDA structures last
   for ( int idx = 0; idx < nDev ; idx ++ ) { delete devProp [ idx ]; }
   free ( devProp ); nDev = 0;
   return ;
}

/**
 * cudaInitDevice function
 *
 * \brief This function selects the coprocessor device and mapping and performs
 * any initialization tasks that need to occur.
 *
 * \param info Heterogeneous task information structure of the device to be
 * initialized
 *
 * \return None
 */
extern "C" void cudaInitDevice ( HTaskInfo * info ) {
   int idx = 0;
   cudaError_t err = cudaSuccess ;
   int dev = info -> device ;
   int proc = info -> process ;
   hTaskLoc loc = info -> location ;
   // Device Management and initialization
   if ( dev != -1) {
      err = cudaSetDevice( dev );
      if ( err != cudaSuccess ) {
         printf ( " CUDAUTIL WARNING : Could not assign to previously initialized \"
                  "device %d\n", dev );
         idx = 0;
         while ( ((cudaSetDevice(idx) != cudaSuccess) && (idx < nDev)) ) { idx++ ; }
         if ( idx == nDev ) {
            printf ( " CUDAUTIL ERROR : Could not assign a CUDA device!\n" ); exit (-333);
         } else {
            cudaCheckErr( cudaGetDevice (& dev ), __LINE__, __FILE__ );
            info -> device = dev ;
            printf ( " CUDAUTIL WARNING : Re-assigning to CUDA device %d\n", dev );
         }
      } else {
         printf ( " Using CUDA Device %d\n", dev );
      }
   } else {
      idx = 0;
      while ( ((cudaSetDevice(idx) != cudaSuccess) && (idx < nDev)) ) { idx++ ; }
      if ( idx == nDev ) {
         printf ( " CUDAUTIL ERROR : Could not assign a CUDA device!\n" ); exit (-334);
      } else {
         cudaCheckErr( cudaGetDevice(&dev),__LINE__,__FILE__ );
         info -> device = dev ;
         printf ( " Assigning to CUDA device %d\n", dev );
      } else {
         printf ( " cudaInitDevice() \n" );
      }
   }
}
APPENDIX A. APPENDIXES

// Process management and initialization
if ((proc != -1) && (proc != 0)) {
    cudaCheckErr(cudaStreamCreate((cudaStream_t *)&proc),__LINE__, __FILE__);
    info->process = proc;
    printf("Assigning to CUDA Stream %d\n", proc);
} else {
    info->process = proc;
}
return;
}

**
cudaInitMem function
*
\**
\** brief This function intelligently allocates memory on the host or device
\** as specified and returns pointer to the memory as well as the
\** stride for the data, which is used for multi-dimensional data
\** \param dims The dimensions of the memory to be allocated
\** \param stride The stride (width in bytes) of the data. This only matters
\** for multi-dimensional data
\** \param loc The location of the memory to be allocated (host or device)
\** \param ptr A pointer to where the memory pointer is to be stored
\** \return None
*
extern "C" void * cudaInitMem (int * dims , int typeSize , int * stride ,
    const char * name , HTaskInfo * info ,
    int mapHostFlag) {
    void * locPtr = NULL;
    size_t free , total = 0;
    int datSize = 1;
    int dev = info->device;
    hTaskLoc loc = info->location;
    // Set to proper device
    cudaCheckErr(cudaSetDevice(dev),__LINE__, __FILE__);
    // Get full data size
    for (int idx = 0; idx < HNDIMS ; idx++) { datSize *= dims[idx]; }
    datSize *= typeSize;
    if (!strcmp(name,"")) { // Data is not a symbol
        // CHECK 1: Memory requested available in global memory
        cudaCheckErr(cudaMemGetInfo(&free, &total),__LINE__, __FILE__);
        if (fre > datSize) { // There is adequate available memory
            // Allocate memory for an object
            cudaExtent extent = make_cudaExtent(dims[HLENGTH]*typeSize,
                dims[HWIDTH],
                dims[HDEPTH]);
            cudaPitchedPtr pitchedPtr;
            cudaCheckErr(cudaMalloc3D(&pitchedPtr, extent),__LINE__, __FILE__);
            cudaCheckErr(cudaMemset3D(pitchedPtr, 0, extent),__LINE__, __FILE__);
            locPtr = pitchedPtr.ptr;
            *stride = pitchedPtr.pitch;
            cudaCheckErr(cudaMemGetInfo(&free, &total),__LINE__, __FILE__);
            printf("Allocated %dB of memory on device. %dB remaining of %dB total\n",
                datSize , free , total);
            if (*stride != (dims[HLENGTH]*typeSize)) {
                printf("CUDAUTIL WARNING: Stride is not the same size as \
                
            data size, free, total);
```c
... else {
    printf("CUDAUTIL ERROR: Attempting to allocate %dB of memory when only \n%dB are available out of a total of %dB\n", datSize, free, total);
    exit(-336);
} else { // Data is a CUDA symbol
    cudaCheckErr(cudaGetSymbolAddress(&locPtr, name), __LINE__, __FILE__);
    cudaCheckErr(cudaGetSymbolSize((size_t *)&dims[LENGTH]), name),
    __LINE__, __FILE__);
} cudaThreadSynchronize();
    cudaCheckErr(cudaGetLastError(), __LINE__, __FILE__);
    return locPtr;
}

/**
 * cudaFreeMem function
 *
 * This function frees allocated memory on the host or device
 * as specified.
 * param ptr Pointer to the data to be freed
 * param dims The dimensions of the data to be freed
 * param loc The location of the data to be freed
 * return None
 */
extern "C" void cudaFreeMem( void * ptr, int * dims, int typeSize,
    const char * name, HTaskInfo * info,
    int mapHostFlag ) {
    hTaskLoc loc = info->location;
    int dev = info->device;
    if (mapHostFlag) {
        cudaCheckErr(cudaFreeHost(ptr), __LINE__, __FILE__);
    } else {
        cudaCheckErr(cudaSetDevice(dev), __LINE__, __FILE__);
        if (!strcmp(name, "")) { cudaCheckErr(cudaFree(ptr), __LINE__, __FILE__); }
    }
    cudaThreadSynchronize();
    cudaCheckErr(cudaGetLastError(), __LINE__, __FILE__);
    return ;
}

/**
 * cudaClearMem function
 *
 * This function clears allocated memory on the host or device
 * as specified.
 * param dims The dimensions of the memory to be allocated
 * param stride The stride (width in bytes) of the data. This only matters
 * for multi-dimensional data
 * param loc The location of the memory to be allocated (host or device)
 * param ptr A pointer to where the memory pointer is to be stored
 * return None
 */
extern "C" void cudaClearMem(int * dims, int typeSize, int stride, void * ptr,
    HTaskInfo * info) {
    int datSize = 0;
    int dev = info->device;
    hTaskLoc loc = info->location;
```
// Set to proper device
cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__);

// Get full data size
for (int idx = 0; idx < HNDIMS; idx++) { datSize += dims[idx]; }
datSize *= typeSize;

// Clear memory on the host for device use
cudaExtent extent = make_cudaExtent(dims[HLENGTH]*typeSize,
                                      dims[HWIDTH],
                                      dims[HDEPTH]);
cudaPitchedPtr pitchedPtr;
pitchedPtr.pitch = stride;
pitchedPtr.ptr = ptr;
pitchedPtr.xsize = dims[HLENGTH];
pitchedPtr.ysize = dims[HWIDTH];
cudaCheckErr(cudaMemset3D(pitchedPtr, 0, extent),__LINE__,__FILE__);
cudaThreadSynchronize();
cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
return;
}

/**
 * cudaMoveData function
 *
 * \brief This function intelligently copies data from the source location with the specified stride to the destination location with the specified stride. The dimensions parameter indicates the dimensions of the data and must be the same for both source and destination.
 *
 * \param dest Pointer to the destination memory location
 * \param destStride Stride (width in bytes) of the destination memory location
 * \param destLoc The location of the destination memory
 * \param src Pointer to the source memory location
 * \param srcStride Stride (width in bytes) of the source memory location
 * \param srcLoc The location of the destination memory
 * \param dims The dimensions of the data to be copied
 * \return None
 */
extern "C" void cudaMoveData( void * dst , int dstStride , HTaskInfo * dstInfo ,
                           void * src , int srcStride , HTaskInfo * srcInfo ,
                           int * dims , int typeSize , const char * name ) {
    hTaskLoc srcLoc = srcInfo->location;
    hTaskLoc dstLoc = dstInfo->location;
    int srcDev = srcInfo->device;
    int dstDev = dstInfo->device;
    int dev = -1;
    int proc = -1;
cudaMemcpy3DParms cpyParms = {0};
cudaStream_t stream;
cudaMemcpyKind kind;

    // Set to proper device
    if (srcLoc == LOC_CUDA) { dev = srcDev; }
    if (dstLoc == LOC_CUDA) { dev = dstDev; }
    if (dev != -1) { cudaCheckErr(cudaSetDevice(dev),__LINE__,__FILE__); }
    if (!strcmp(name,"")) { // conduit is not for a symbol
        // Determine stream value
        if ((srcLoc == LOC_CUDA) && (dstLoc == LOC_CPU)) {
proc = srcInfo->process;
kind = cudaMemcpyDeviceToHost;
} else if ((dstLoc == LOC_CUDA) && (srcLoc == LOC_CPU)) {
proc = dstInfo->process;
kind = cudaMemcpyHostToDevice;
} else if ((dstLoc == LOC_CUDA) && (srcLoc == LOC_CUDA)) {
proc = srcInfo->process;
kind = cudaMemcpyDeviceToDevice;
} else if ((dstLoc == LOC_CPU) && (srcLoc == LOC_CPU)) {
proc = srcInfo->process;
kind = cudaMemcpyHostToHost;
}
// Both src and dst are CPUs
int stride = (srcStride > dstStride) ? srcStride : dstStride;
cpyParms.extent = make_cudaExtent(dims[LENGTH]*typeSize,
dims[WIDTH],
dims[DEPTH]);
cpyParms.srcPtr.pitch = stride;
cpyParms.srcPtr.ptr = src;
cpyParms.srcPtr.xsize = dims[LENGTH]*typeSize;
cpyParms.srcPtr.ysize = dims[WIDTH];
cpyParms.dstPtr.pitch = stride;
cpyParms.dstPtr.ptr = dst;
cpyParms.dstPtr.xsize = dims[LENGTH]*typeSize;
cpyParms.dstPtr.ysize = dims[WIDTH];
cpyParms.kind = kind;
cudaCheckErr(cudaMemcpy3DAsync(&cpyParms, (cudaStream_t)(proc)),
_LINE__,__FILE__);
cpyParms.stream = (cudaStream_t)(proc);
cudaThreadSynchronize();
cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
return;
}

/**
 * cudaLaunchKernel function
 *
 * brief This function uses the specified parameters to execute the
 * coprocessor function specified by kernel. Any parameters to
 * the kernel should be included in params.
 *
 * param kernel A string naming the kernel to execute
 * param dims The dimensions of the data to execute the kernel on
 * param nParams The number of parameters to be passed to the kernel
 * param params Array of parameters to be passed to the kernel
 * param gDim The grid dimensions of the kernel
 * param bDim The block dimensions of the kernel
 * param locMem The amount of local or shared memory to allocate for kernel
 * execution
 * param stream The stream index to associate this kernel's execution and
 * data with
 * return None
 */

extern "C" void cudaLaunchKernel(const char * krn, int * dims, int nParams,
    int * paramSizes, void ** params, int * gDim,
    int * bDim, int locMem, HTaskInfo * info) {

    hTaskLoc loc = info->location;
    int dev = info->device;
    int proc = info->process;
    size_t offset = 0;
    dim3 gridDims = dim3(gDim[HLENGTH], gDim[HWIDTH], gDim[HDEPTH]);
    dim3 blockDims = dim3(bDim[HLENGTH], bDim[HWIDTH], bDim[HDEPTH]);

    // Set the device and configure the kernel call to a stream
    cudaCheckErr(cudaSetDevice(dev), __LINE__, __FILE__);
    cudaCheckErr(cudaConfigureCall(gridDims, blockDims, locMem, (cudaStream_t) proc),
                   __LINE__, __FILE__);

    // For each kernel parameter passed in, push the kernel argument onto the stack
    // turn into internal function
    for (int i = 0; i < nParams; i++) {
        cudaCheckErr(cudaSetupArgument(params[i], paramSizes[i], offset),
                     __LINE__, __FILE__);
        offset = offset + paramSizes[i];
    }

    printf("Launching %s on dev:%d\n", krn, dev);
    if (!strcmp(krn, "gpu_mc_stage1")) {
        cudaCheckErr(cudaLaunch(gpu_mc_stage1),__LINE__,__FILE__);
    }
    if (!strcmp(krn, "gpu_mc_stage2")) {
        cudaCheckErr(cudaLaunch(gpu_mc_stage2),__LINE__,__FILE__);
    }
    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);

    if ((proc != 0) && (proc != -1)) { cudaStreamSynchronize((cudaStream_t) proc); }
    cudaThreadSynchronize();
    cudaCheckErr(cudaGetLastError(),__LINE__,__FILE__);
    return;
}

Listing A.3: CUDA utility functions code

A.2.3 OpenCL Utility Functions

#include "oclUtil.h"
c1State glblState = {0};
int oclSysInit = 0;
char * buildOpts = "-DOCLBUILD -I./platforms/app -I./platforms/util";
extern "C" void initOpenCL() {
int idx, jdx = 0;
size_t psize;
char * pstr;
cl_int err = CL_SUCCESS;
cl_device_type tmptype;

if (oclSysInit == 0)
{
    // Query and list platforms
    clCheckErr(clGetPlatformIDs(0, NULL, &(glblState.nplat)),__LINE__, __FILE__);
    cl_platform_id * plats = (cl_platform_id*)malloc(sizeof(cl_platform_id)*
        glblState.nplat);
    glblState.plats = (oclPlatInfo*)malloc(sizeof(oclPlatInfo)*glblState.nplat);
    memset(glblState.plats, 0, sizeof(oclPlatInfo)*glblState.nplat);
    clCheckErr(clGetPlatformIDs(glblState.nplat,plats,NULL),__LINE__, __FILE__);

    // For each platform, find and list the devices
    for (idx = 0; idx < glblState.nplat; idx++) {
        oclPlatInfo * plt = &(glblState.plats[idx]);
        cl_checkErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, NULL,
            NULL, &psize),
            __LINE__, __FILE__);
        pstr = (char*)malloc(psize);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, pszie,
            pstr, NULL),
            __LINE__, __FILE__);
        printf("%s ", pstr); free(pstr);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_NAME, NULL,
            NULL, &psize),
            __LINE__, __FILE__);
        pstr = (char*)malloc(psize);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_NAME, pszie,
            pstr, NULL),
            __LINE__, __FILE__);
        printf("%s\n", pstr); free(pstr);
        clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, NULL,
            NULL, &(plt->ndevs)),
            __LINE__, __FILE__);
        cl_device_id * devs = (cl_device_id*)malloc(sizeof(cl_device_id)*plt->ndevs );
        plt->devs = (oclDeviceInfo*)malloc(sizeof(oclDeviceInfo)*plt->ndevs);
        memset(plt->devs, 0, sizeof(oclDeviceInfo)*plt->ndevs);
        clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, plt->ndevs,
            devs, NULL),
            __LINE__, __FILE__);
        cl_context_properties cprops[3] = {CL_CONTEXT_PLATFORM,
            (cl_context_properties)(plt->pid), 0};
        plt->ctx = clCreateContext(&cprops, plt->ndevs, devs, NULL, NULL, &err);
        clRetainContext(plt->ctx);
        clCheckErr(err, __LINE__, __FILE__);
    }
}

// For each device in the platform, print some basic information
for (jdx = 0; jdx < plt->ndevs; jdx++) {
    /*
     * riev This function will initialize the OpenCL system, finding any
     * platforms and devices available, and printing out a short list of them.
     */
    extern "C" void initOpenCL() {
int idx, jdx = 0;
size_t psize;
char * pstr;
cl_int err = CL_SUCCESS;
cl_device_type tmptype;

if (oclSysInit == 0)
{
    // Query and list platforms
    clCheckErr(clGetPlatformIDs(0, NULL, &(glblState.nplat)),__LINE__, __FILE__);
    cl_platform_id * plats = (cl_platform_id*)malloc(sizeof(cl_platform_id)*
        glblState.nplat);
    glblState.plats = (oclPlatInfo*)malloc(sizeof(oclPlatInfo)*glblState.nplat);
    memset(glblState.plats, 0, sizeof(oclPlatInfo)*glblState.nplat);
    clCheckErr(clGetPlatformIDs(glblState.nplat,plats,NULL),__LINE__, __FILE__);

    // For each platform, find and list the devices
    for (idx = 0; idx < glblState.nplat; idx++) {
        oclPlatInfo * plt = &(glblState.plats[idx]);
        cl_checkErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, NULL,
            NULL, &psize),
            __LINE__, __FILE__);
        pstr = (char*)malloc(psize);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, pszie,
            pstr, NULL),
            __LINE__, __FILE__);
        printf("Found platform %d: 0x%X ==\n", idx, plt->pid);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, NULL,
            NULL, &psize),
            __LINE__, __FILE__);
        pstr = (char*)malloc(psize);
        clCheckErr(clGetPlatformInfo(plt->pid, CL_PLATFORM_VENDOR, pszie,
            pstr, NULL),
            __LINE__, __FILE__);
        printf("%s\n", pstr); free(pstr);
        clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, NULL,
            NULL, &(plt->ndevs)),
            __LINE__, __FILE__);
        cl_device_id * devs = (cl_device_id*)malloc(sizeof(cl_device_id)*plt->ndevs );
        plt->devs = (oclDeviceInfo*)malloc(sizeof(oclDeviceInfo)*plt->ndevs);
        memset(plt->devs, 0, sizeof(oclDeviceInfo)*plt->ndevs);
        clCheckErr(clGetDeviceIDs(plt->pid, CL_DEVICE_TYPE_ALL, plt->ndevs,
            devs, NULL),
            __LINE__, __FILE__);
        cl_context_properties cprops[3] = {CL_CONTEXT_PLATFORM,
            (cl_context_properties)(plt->pid), 0};
        plt->ctx = clCreateContext(&cprops, plt->ndevs, devs, NULL, NULL, &err);
        clRetainContext(plt->ctx);
        clCheckErr(err, __LINE__, __FILE__);
    }
}
oclDeviceInfo * dev = &( plt -> devs [ jdx ]);  
for ( kdx = 0; kdx < MAX_PROCESSES; kdx++) { dev->queues[kdx] = 0); }  
dev->did = devs[jdx];  
printf("\tDevice %d ( ID:0x%X):", jdx, dev->did );  
clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, NULL, NULL, &psize),__LINE__ , __FILE__);  
pstr = (char*)malloc(psize);  
clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_NAME, psize, pstr, NULL),__LINE__ , __FILE__);  
printf(" %s", pstr);  
free(pstr);  
clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_TYPE, sizeof(cl_device_type), &tmptype, NULL),__LINE__ , __FILE__);  
pstr = (char*)malloc(sizeof(char)*STRLEN);  
cDecodeDeviceType(tmptype, pstr);  
printf(" (%s)", pstr);  
free(pstr);  
clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_VERSION, NULL, NULL, &psize),__LINE__ , __FILE__);  
pstr = (char*)malloc(psize);  
clCheckErr(clGetDeviceInfo(dev->did, CL_DEVICE_VERSION, psize, pstr, NULL),__LINE__ , __FILE__);  
printf(" ver. %s\n", pstr);  
free(pstr);  
}  
free(devs);  
}
else {  
printf("OCLUTIL WARNING: OpenCL system already initialized\n");  
return;  
}

/**  
* oclInitDevice function  
*   
* \brief This function selects the coprocessor device and mapping and performs any initialization tasks that need to occur.  
*/  
extern "C" void oclInitDevice(HTaskInfo * info) {  
cl_int err = CL_SUCCESS;  
int didx = info->device;  
int qidx = info->process;  
HTaskPlat platName = info->platform;  
int pidx = clFindPlatform(platName);  
oclPlatInfo * plat = &(glblState.plats[pidx]);  
oclDeviceInfo * dev = &(plat->devs[didx]);  
if (dev->queues[qidx] == 0) {  
  dev->queues[qidx] = clCreateCommandQueue(plat->ctx, dev->did, 0, &err);  
  clRetainCommandQueue(dev->queues[qidx]);  
  clCheckErr(err, __LINE__, __FILE__);  
} else {  
  printf("OCLUTIL WARNING: Device has already been initialized pidx=%d, 
  didx=%d, %s:%d\n", pidx, didx, __FILE__, __LINE__);
/* closeOpenCL function */

/* brief This function will clean up, releasing and deleting all of the OpenCL objects that were allocated or initialized during program execution */

extern "C" void closeOpenCL () {
    int idx, jdx, kdx = 0;
    int nplat = glblState.nplat;
    for (idx = 0; idx < nplat; idx++) {
        oclPlatInfo * plat = &(glblState.plats[idx]);
        for (jdx = 0; jdx < plat->ndev; jdx++) {
            oclDeviceInfo * dev = plat->devs[jdx];
            for (kdx = 0; kdx < dev->ndev; kdx++) {
                clCheckErr(clReleaseKernel(dev->kerns[kdx]),__LINE__,__FILE__);
            }
            for (kdx = 0; kdx < MAX_PROCESSES; kdx++) {
                clCheckErr(clReleaseCommandQueue(dev->queues[kdx]),__LINE__,__FILE__);
            }
            clCheckErr(clReleaseDevice(dev->did),__LINE__,__FILE__);
            clCheckErr(clReleaseProgram(plat->prg),__LINE__,__FILE__);
            clCheckErr(clReleaseContext(plat->ctx),__LINE__,__FILE__);
            free(plat->devs);
        }
        free(plat->devs);
    }
    free(plat->devs);
    glblState.nplat = 0;
    return;
}

/* oclInitMem function */

/* brief This function intelligently allocates memory on the host or device as specified and returns pointer to the memory as well as the stride for the data, which is used for multi-dimensional data blocks. */

extern "C" void * oclInitMem (int * dims, int typeSize, int * stride, const char * name, HTaskInfo * info, int mapHostFlag) {
    int size = 0;
    hTaskLoc loc = info->location;
    int dev = info->device;
    int proc = info->process;
    cl_int err = CL_SUCCESS;
    void * ptr = NULL;
    int pidx = clFindPlatform(info->platform);
    cl_context ctx = glblState.plats[pidx].ctx;
    cl_command_queue queue = glblState.plats[pidx].devs[dev].queues[proc];
    int fill = 0;
    size = typeSize*dims[LENGTH]*dims[WIDTH]*dims[DEPTH];

    if (mapHostFlag) {
        ptr = clCreateBuffer(ctx, (CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR),
#ifdef OPENCL_1.2
    clCheckErr(clEnqueueFillBuffer(queue, ptr, &fill, sizeof(int), 0, size, 0, NULL, NULL),
               __LINE__, __FILE__);
    clFlush(queue);
#else
    memset(ptr, 0, size);
#endif
#else
    ptr = malloc(size);
    memset(ptr, 0, size);
#endif
else if (loc == LOC_OCL) {
    if (!strcmp(name, "")) {
        ptr = clCreateBuffer(ctx, CL_MEM_READ_WRITE, size, NULL, &err);
    } else {
        ptr = clCreateBuffer(ctx, CL_MEM_READ_ONLY, size, NULL, &err);
    }
    clCheckErr(err, __LINE__, __FILE__);
} else {
    printf("OCLUTIL ERROR: Invalid device location: line:%d file:%s\n", __LINE__, __FILE__);
    exit(-447);
}
return ptr;

/**
 * oclFreeMem function
 * \brief This function frees allocated memory on the host or device
 * as specified.
 */
extern "C" void oclFreeMem(void * ptr, int * dims, int typeSize, const char * name,
                           HTaskInfo * info, int mapHostFlag) {
    hTaskLoc loc = info->location; //getLocation();
    if (loc == LOC_CPU) {
        if (mapHostFlag) {
            clCheckErr(clReleaseMemObject((cl_mem)ptr), __LINE__, __FILE__);
        } else {
            free(ptr);
        }
    } else if (loc == LOC_OCL) {
        clCheckErr(clReleaseMemObject((cl_mem)ptr), __LINE__, __FILE__);
    } else {
        printf("OCLUTIL ERROR: Invalid device location: line:%d file:%s\n", __LINE__, __FILE__);
        exit(-448);
    }
    return;
}

/**
 * oclClearMem function
 * \brief This function will zero out the specified memory buffer
 */
extern "C" void oclClearMem(int * dims, int typeSize, int stride, void * ptr,
                           HTaskInfo * info) {
    int size = 0;
hTaskLoc loc = info->location; //getLocation();
int dev = info->device;
int proc = info->process;
int pidx = clFindPlatform(info->platform); //getPlatform();
cl_command_queue queue = glblState.plats[pidx].devs[dev].queues[proc];
int fill = 0;
size = typeSize*dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH];
#ifdef OPENCL_1.2
clCheckErr(clEnqueueFillBuffer(queue, ptr, &fill, sizeof(int), 0, size, 0,
_LINE__, _FILE__));
#endif
clCheckErr(clEnqueueBarrier(queue), _LINE__, _FILE__);
clFlush(queue);
return ;
}

/**
 * oclMoveData function
 * \brief This function intelligently copies data from the source location
 * with the specified stride to the destination location with the
 * specified stride. The dimensions parameter indicates the
 * dimensions of the data and must be the same for both source and
 * destination.
 */
extern "C" void oclMoveData( void * dst, int dstStride, HTaskInfo * dstInfo,
void * src, int srcStride, HTaskInfo * srcInfo,
int * dims, int typeSize, const char * name ) {
    hTaskLoc srcLoc = srcInfo->location;
    hTaskLoc dstLoc = dstInfo->location;
    int size = typeSize*dims[HLENGTH]*dims[HWIDTH]*dims[HDEPTH];
    int dev, proc, pidx;
    cl_command_queue queue;
    if ((srcLoc == LOC_CPU) && (dstLoc == LOC_OCL)) {
        dev = dstInfo->device;
        proc = dstInfo->process;
        pidx = clFindPlatform(dstInfo->platform);
        queue = glblState.plats[pidx].devs[dev].queues[proc];
        clCheckErr(clEnqueueWriteBuffer(queue, (cl_mem)dst, CL_TRUE, 0, size, src, 0,
_LINE__, _FILE__));
    } else if ((srcLoc == LOC_OCL) && (dstLoc == LOC_CPU)) {
        dev = srcInfo->device;
        proc = srcInfo->process;
        pidx = clFindPlatform(srcInfo->platform);
        queue = glblState.plats[pidx].devs[dev].queues[proc];
        clCheckErr(clEnqueueReadBuffer(queue, (cl_mem)src, CL_TRUE, 0, size, dst, 0,
_LINE__, _FILE__));
    } else if ((srcLoc == LOC_OCL) && (dstLoc == LOC_OCL)) {
        dev = srcInfo->device;
        proc = srcInfo->process;
        pidx = clFindPlatform(srcInfo->platform);
        queue = glblState.plats[pidx].devs[dev].queues[proc];
        clCheckErr(clEnqueueCopyBuffer(queue, (cl_mem)src, (cl_mem)dst, 0, 0, size, 0,
_LINE__, _FILE__));
    } else {
        printf("OCLUTIL ERROR: Invalid device locations: line:%d file:%s\n",
_LINE__, _FILE__);
APPENDIX A. APPENDIXES

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

```c
printf("Program build for device: %s", pstr);
free(pstr);
clCheckErr(clGetProgramBuildInfo(plat->prg, dev->did, CL_PROGRAM_BUILD_STATUS, sizeof(cl_build_status), &bst, NULL), __LINE__, __FILE__);
if (bst != CL_BUILD_SUCCESS) {
    printf(" has FAILED! See information below ...
");
    clProgramBuildDump(plat->prg, dev->did);
    bool input = false;
    char inval = 'x';
    while (!input) {
        printf("Would you like to try to re-build the program [Y/N]? ");
        std::cin >> inval;
        input = ((inval=='Y')||(inval=='N')) ? true : false;
    }
    if (inval=='N') { built = true; break; }
} else { printf(" has SUCCEEDED!
"); }
}
if (inval=='N') { built = true; break; }
}
else {
built = true; printf("Built program successfully!
");
for (int idx = 0; idx < plat->ndevs; idx++) {
oclDeviceInfo * dev = &(plat->devs[idx]);
clCheckErr(clCreateKernelsInProgram(plat->prg, MAX_KERNS, krns, &(dev->nkrn)), __LINE__, __FILE__);
for (int jdx = 0; jdx < dev->nkrn; jdx++) {
    dev->kerns[jdx] = krns[jdx];
}
}
free(src);
free(krns);
free(devs);
return;
}
/**
 * clBuild function
 *
 * \brief This function will build the kernels in the specified source file for the given heterogeneous device
 */
extern "C" void oclBuild(HTaskInfo * info, char * srcFile) {
cl_int err = CL_SUCCESS;
char * src = NULL;
FILE * fid = NULL;
size_t srcSz = 0;
int pidx = clFindPlatform(info->platform); //getPlatform();
oclPlatInfo * plat = &(glblState.plats[pidx]);
cl_kernel * k = (cl_kernel*)malloc(sizeof(cl_kernel)*MAX_KERNS);

fid = fopen(srcFile, "rb");
if (fid == NULL) {
    printf("OCLUTIL ERROR: Unable to open file %s for building program!
", srcFile);
    exit(-552);
}
seek(fid, 0, SEEK_END);
srcSz = ftell(fid);
rewind(fid);
```
src = (char*)malloc(sizeof(char)*srcSz);
srcSz = fread(src, sizeof(char), srcSz, fid);
fclose(fid);
plat->prg = clCreateProgramWithSource(plat->ctx, 1, (const char**) &src, &srcSz, &err);
clCheckErr(err, __LINE__, __FILE__);

for (int idx = 0; idx < plat->ndevs; idx++) {
    oclDeviceInfo * dev = &(plat->devs[idx]);
    clCheckErr(clBuildProgram(plat->prg, 1, &(dev->did), "", NULL, NULL),
               __LINE__, __FILE__);
    clCheckErr(clCreateKernelsInProgram(plat->prg, MAX_KERNELS, krns, &(dev->nkern)),
               __LINE__, __FILE__);
    for (int jdx = 0; jdx < dev->nkern; jdx++) {
        dev->kerns[jdx] = krns[jdx];
    }
}
free(src);
free(krns);
return;

Listing A.4: OpenCL utility function code
A.2.4 OpenCL Helper Functions

```c
/*
 * 
 * 
 */
#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;
    default:
        strcat(str, "UNKNOWN_ERROR_CODE"); break;
    }
*/
```
case CL_IMAGE_FORMAT_NOT_SUPPORTED:
  strcat(str, "CL_IMAGE_FORMAT_NOT_SUPPORTED"); break;

case CL_BUILD_PROGRAM_FAILURE:
  strcat(str, "CL_BUILD_PROGRAM_FAILURE"); break;

case CL_MAP_FAILURE:
  strcat(str, "CL_MAP_FAILURE"); break;

case CL_MISALIGNED_SUB_BUFFER_OFFSET:
  strcat(str, "CL_MISALIGNED_SUB_BUFFER_OFFSET"); break;

case CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST:
  strcat(str, "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST"); break;

// Insert cases 15-30 here

case CL_INVALID_VALUE:
  strcat(str, "CL_INVALID_VALUE"); break;

case CL_INVALID_DEVICE_TYPE:
  strcat(str, "CL_INVALID_DEVICE_TYPE"); break;

case CL_INVALID_PLATFORM:
  strcat(str, "CL_INVALID_PLATFORM"); break;

case CL_INVALID_DEVICE:
  strcat(str, "CL_INVALID_DEVICE"); break;

case CL_INVALID_CONTEXT:
  strcat(str, "CL_INVALID_CONTEXT"); break;

case CL_INVALID_QUEUE_PROPERTIES:
  strcat(str, "CL_INVALID_QUEUE_PROPERTIES"); break;

case CL_INVALID_COMMAND_QUEUE:
  strcat(str, "CL_INVALID_COMMAND_QUEUE"); break;

case CL_INVALID_HOST_PTR:
  strcat(str, "CL_INVALID_HOST_PTR"); break;

case CL_INVALID_MEM_OBJECT:
  strcat(str, "CL_INVALID_MEM_OBJECT"); break;

case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
  strcat(str, "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR"); break;

case CL_INVALID_IMAGE_SIZE:
  strcat(str, "CL_INVALID_IMAGE_SIZE"); break;

case CL_INVALID_SAMPLER:
  strcat(str, "CL_INVALID_SAMPLER"); break;

case CL_INVALID_BINARY:
  strcat(str, "CL_INVALID_BINARY"); break;

case CL_INVALID_BUILD_OPTIONS:
  strcat(str, "CL_INVALID_BUILD_OPTIONS"); break;

case CL_INVALID_PROGRAM:
  strcat(str, "CL_INVALID_PROGRAM"); break;

case CL_INVALID_PROGRAM_EXECUTABLE:
  strcat(str, "CL_INVALID_PROGRAM_EXECUTABLE"); break;

case CL_INVALID_KERNEL_NAME:
  strcat(str, "CL_INVALID_KERNEL_NAME"); break;

case CL_INVALID_KERNEL_DEFINITION:
  strcat(str, "CL_INVALID_KERNEL_DEFINITION"); break;

case CL_INVALID_KERNEL:
  strcat(str, "CL_INVALID_KERNEL"); break;

case CL_INVALID_ARG_INDEX:
  strcat(str, "CL_INVALID_ARG_INDEX"); break;

case CL_INVALID_ARG_VALUE:
  strcat(str, "CL_INVALID_ARG_VALUE"); break;

case CL_INVALID_ARG_SIZE:
  strcat(str, "CL_INVALID_ARG_SIZE"); break;

case CL_INVALID_KERNEL_ARGS:
  strcat(str, "CL_INVALID_KERNEL_ARGS"); break;

case CL_INVALID_WORK_DIMENSION:
  strcat(str, "CL_INVALID_WORK_DIMENSION"); break;

case CL_INVALID_WORK_GROUP_SIZE:
  strcat(str, "CL_INVALID_WORK_GROUP_SIZE"); break;

case CL_INVALID_WORK_ITEM_SIZE:
  strcat(str, "CL_INVALID_WORK_ITEM_SIZE"); break;
case CL_INVALID_GLOBAL_OFFSET:
    strcat(str, "CL_INVALID_GLOBAL_OFFSET"); break;

case CL_INVALID_EVENT_WAIT_LIST:
    strcat(str, "CL_INVALID_EVENT_WAIT_LIST"); break;

case CL_INVALID_EVENT:
    strcat(str, "CL_INVALID_EVENT"); break;

case CL_INVALID_OPERATION:
    strcat(str, "CL_INVALID_OPERATION"); break;

case CL_INVALID_GL_OBJECT:
    strcat(str, "CL_INVALID_GL_OBJECT"); break;

case CL_INVALID_BUFFER_SIZE:
    strcat(str, "CL_INVALID_BUFFER_SIZE"); break;

case CL_INVALID_MIP_LEVEL:
    strcat(str, "CL_INVALID_MIP_LEVEL"); break;

case CL_INVALID_GLOBAL_WORK_SIZE:
    strcat(str, "CL_INVALID_GLOBAL_WORK_SIZE"); break;

    // case CL_INVALID_PROPERTY:
    //    strcat(str, "CL_INVALID_PROPERTY"); break;
    // 64+ go here

default: strcat(str, "UNKNOWN ERROR CODE");
}

return;
}

/**
 * clCheckErr function
 *
 * \brief This function checks, analyzes, and diagnoses OpenCL error codes
 *
 * \param err the error code to check
 * \param line the line number where the error code is being checked
 * \param file the name of the file in which the error code was generated
 *
 * \return None
 *
 */
extern "C" void clCheckErr(cl_int err, int line, char * file) {
    char errStr[STRELEN];
    if (err != CL_SUCCESS) {
        clDecodeErr(err, errStr);
        printf("OpenCL Error %d: %s. In file %s, line %d\n", err, errStr, file, line);
        #ifdef KILLONERR
            exit(err);
        #endif
        return;
    }
}

/**
 * clDecodeFPConfig function
 *
 * \brief This function decodes the OpenCL floating point configuration type
 *
 * \param cfg Device FP configuration code
 * \param str Output string to write to
 *
 * \return None
 *
 */
extern "C" void clDecodeFPConfig(cl_device_fp_config cfg, char * str) {
    int on = 0;
    str[0] = '\0';
    if (cfg & CL_FP_DENORM) { strcat(str, "DENORM"); on = 1; }
    if (cfg & CL_FP_INF_NAN) {
        if(on){strcat(str,"\n");}
 APPENDIX A. APPENDIXES

strcat(str, "INF_NAN"); on = 1;
}

if (cfg & CL_FP_ROUND_TO_NEAREST) {
  if(on) {strcat(str, ", ");} 
  strcat(str, "ROUND_TO_NEAREST"); on = 1;
}

if (cfg & CL_FP_ROUND_TO_ZERO) {
  if(on) {strcat(str, ", ");} 
  strcat(str, "ROUND_TO_ZERO"); on = 1;
}

if (cfg & CL_FP_ROUND_TO_INF) {
  if(on) {strcat(str, ", ");} 
  strcat(str, "ROUND_TO_INF"); on = 1;
}

if (cfg & CL_FP_FMA) {
  if(on) {strcat(str, ", ");} 
  strcat(str, "FMA"); on = 1;
}

if (cfg & CL_FP_SOFT_FLOAT) {
  if(on) {strcat(str, ", ");} 
  strcat(str, "SOFT_FLOAT");
}

return;
}

/**
 * clDecodeExecCapabilities function
 * \brief This function decodes the OpenCL device execution capabilities type
 * \param dcap Device execution capabilities code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeExecCapabilities(cl_device_exec_capabilities dcap,
  char * str)
{
  int on = 0;
  str[0] = '\0';
  if (dcap & CL_EXEC_KERNEL) { strcat(str, "KERNEL"); on = 1; }
  if (dcap & CL_EXEC_NATIVE_KERNEL) {
    if(on) {strcat(str, ", ");} 
    strcat(str, "NATIVE_KERNEL");
  }
  return;
}

/**
 * clDecodeDevMemCacheType function
 * \brief This function decodes the OpenCL device memory cache type
 * \param mctype Device memory cache type code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeDevMemCacheType(cl_device_mem_cache_type mctype,
  char * str)
{
  str[0] = '\0';
  switch(mctype) {
  case CL_NONE: strcat(str, "NONE"); break;
  case CL_READ_ONLY_CACHE: strcat(str, "READ_ONLY_CACHE"); break;
  case CL_READ_WRITE_CACHE: strcat(str, "READ_WRITE_CACHE"); break;
  case CL_SOFT_FLOAT: strcat(str, "SOFT_FLOAT"); break;
  default: strcat(str, "ERROR: Invalid device memory cache type");
  }
}
APPENDIX A. APPENDIXES

```c
/**
 * clDecodeLocalMemType function
 *
 * \brief This function decodes the OpenCL device local memory type
 *
 * \param lmtype Device local memory type code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeLocalMemType( cl_device_local_mem_type lmtype , char * str ) {
    str[0] = '\0';
    switch(lmtype) {
        case CL_LOCAL : strcat (str , " LOCAL "); break ;
        case CL_GLOBAL : strcat (str , " GLOBAL "); break ;
        default : strcat (str , " ERROR : Invalid device local memory type ");
    }
    return ;
}

/**
 * clDecodeQueueProperties function
 *
 * \brief This function decodes the OpenCL command queue properties type
 *
 * \param qprop Command queue properties code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeQueueProperties( cl_command_queue_properties qprop , char * str ) {
    int on = 0;
    str[0] = '\0';
    if ( qprop & CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE ) {
        strcat (str , " OUT_OF_ORDER_EXEC_ENABLE "); on = 1;
    }
    if ( qprop & CL_QUEUE_PROFILING_ENABLE ) {
        if(on){ strcat (str ,", ");}
        strcat (str , " PROFILING_ENABLE ");
    }
    return ;
}

/**
 * clDecodeDeviceType function
 *
 * \brief This function decodes the OpenCL device type
 *
 * \param dtype Device type code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeDeviceType( cl_device_type dtype , char * str ) {
    int on = 0;
    str[0] = '\0';
    if ( dtype & CL_DEVICE_TYPE_DEFAULT ) {
        strcat (str , " DEFAULT "); on = 1;
    }
    if ( dtype & CL_DEVICE_TYPE_CPU ) {
    }
    return ;
}
```
if (on) { strcat (str, ", "); }
strcat (str, " CPU "); on = 1;
}
if (dtype & CL_DEVICE_TYPE_GPU) {
if (on) { strcat (str, ", "); }
strcat (str, " GPU "); on = 1;
}
if (dtype & CL_DEVICE_TYPE_ACCELERATOR) {
if (on) { strcat (str, ", "); }
strcat (str, " ACCELERATOR "); on = 1;
}
if (dtype & CL_DEVICE_TYPE_ALL) {
if (on) { strcat (str, ", "); }
strcat (str, " ALL ");
}
return ;
}

/**
 * clDecodeMemObjType function
 */
/**
 * rief This function decodes the OpenCL memory object type
 *
 * \param mtype Memory object type code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeMemObjType(cl_mem_object_type mtype, char * str) {
str[0] = '\0';
switch (mtype) {
case CL_MEM_OBJECT_BUFFER: strcat (str, " CL_MEM_OBJECT_BUFFER "); break;
case CL_MEM_OBJECT_IMAGE2D: strcat (str, " CL_MEM_OBJECT_IMAGE2D "); break;
case CL_MEM_OBJECT_IMAGE3D: strcat (str, " CL_MEM_OBJECT_IMAGE3D "); break;
default: strcat (str, " ERROR : Invalid Memory Object Type ");
}
return ;
}

/**
 * clDecodeMemObjFlags function
 */
/**
 * rief This function decodes the OpenCL memory flag type
 *
 * \param mflags Memory object flags code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeMemObjFlags(cl_mem_flags mflags, char * str) {
int on = 0;
str[0] = '\0';
if (mflags & CL_MEM_READ_WRITE) {
strcat (str, " CL_MEM_READ_WRITE "); on = 1;
}
if (mflags & CL_MEM_WRITE_ONLY) {
if (on) { strcat (str, ", "); }
strcat (str, " CL_MEM_WRITE_ONLY "); on = 1;
}
if (mflags & CL_MEM_READ_ONLY) {
if (on) { strcat (str, ", "); }
strcat (str, " CL_MEM_READ_ONLY "); on = 1;
}
if (mflags & CL_MEM_USE_HOST_PTR) {
    if (on) { strcat(str, " ", " );
    strcat(str, "CL_MEM_USE_HOST_PTR "); on = 1;
}
if (mflags & CL_MEM_ALLOC_HOST_PTR) {
    if (on) { strcat(str, " ", " );
    strcat(str, "CL_MEM_ALLOC_HOST_PTR "); on = 1;
}
if (mflags & CL_MEM_COPY_HOST_PTR) {
    if (on) { strcat(str, " ", " );
    strcat(str, "CL_MEM_COPY_HOST_PTR ");
}
return;
}

/**
 * clDecodeImageFormat function
 * \brief This function decodes the OpenCL image format type
 * \param imgfmt Image format code
 * \param str Output string to write to
 * \return None
 */
extern "C" void clDecodeImageFormat(cl_image_format imgfmt, char * str) {
    str[0] = '\0';
    strcat(str, " Channel Order ");
    switch (imgfmt.image_channel_order) {
    case CL_R: strcat(str, "CL_R"); break;
    case CL_A: strcat(str, "CL_A"); break;
    case CL_RG: strcat(str, "CL_RG"); break;
    case CL_RA: strcat(str, "CL_RA"); break;
    case CL_RGB: strcat(str, "CL_RGB"); break;
    case CL_RGBA: strcat(str, "CL_RGBA"); break;
    case CL_BGRA: strcat(str, "CL_BGRA"); break;
    case CL_ARGB: strcat(str, "CL_ARGB"); break;
    case CL_INTENSITY: strcat(str, "CL_INTENSITY"); break;
    case CL_LUMINANCE: strcat(str, "CL_LUMINANCE"); break;
    default: strcat(str, "ERROR: Invalid Color Order");
    }
    strcat(str, " Channel Data Type ");
    switch (imgfmt.image_channel_data_type) {
    case CL_SIGNED_INT8: strcat(str, "CL_SIGNED_INT8"); break;
    case CL_UNSIGNED_INT16: strcat(str, "CL_UNSIGNED_INT16");
    case CL_UNORM_INT8: strcat(str, "CL_UNORM_INT8");
    case CL_HALF_FLOAT: strcat(str, "CL_HALF_FLOAT");
    case CL_FLOAT: strcat(str, "CL_FLOAT");
    default: strcat(str, "ERROR: Invalid Data Type");
    }
return;
}
extern "C" void clDecodeAddressingMode(cl_addressing_mode amd, char * str) {
    str[0] = '\0';
    switch (amd) {
    case CL_ADDRESS_NONE :
        strcat(str, "CL_ADDRESS_NONE"); break;
    case CL_ADDRESS_CLAMP_TO_EDGE :
        strcat(str, "CL_ADDRESS_CLAMP_TO_EDGE"); break;
    case CL_ADDRESS_CLAMP :
        strcat(str, "CL_ADDRESS_CLAMP"); break;
    case CL_ADDRESS_REPEAT :
        strcat(str, "CL_ADDRESS_REPEAT"); break;
    default :
        strcat(str, "ERROR: Invalid Addressing Mode");
    }
    return;
}

extern "C" void clDecodeFilterMode(cl_filter_mode fmd, char * str) {
    str[0] = '\0';
    switch (fmd) {
    case CL_FILTER_NEAREST :
        strcat(str, "CL_FILTER_NEAREST"); break;
    case CL_FILTER_LINEAR :
        strcat(str, "CL_FILTER_LINEAR"); break;
    default :
        strcat(str, "ERROR: Invalid Filter Mode");
    }
    return;
}

extern "C" void clDecodeBuildStatus(cl_build_status bst, char * str) {
    str[0] = '\0';
    switch (bst) {
    case CL_BUILD_SUCCESS :
        strcat(str, "CL_BUILD_SUCCESS"); break;
    case CL_BUILD_NONE :
        strcat(str, "CL_BUILD_NONE"); break;
    case CL_BUILD_ERROR :
        strcat(str, "CL_BUILD_ERROR"); break;
    case CL_BUILD_ERROR_CONFIGURED :
        strcat(str, "CL_BUILD_ERROR_CONFIGURED"); break;
    case CL_BUILD_ERROR_TOO_MUCH_DATA :
        strcat(str, "CL_BUILD_ERROR_TOO_MUCH_DATA"); break;
    case CL_BUILD_ERROR_INVALID_PROGRAM :
        strcat(str, "CL_BUILD_ERROR_INVALID_PROGRAM"); break;
    case CL_BUILD_ERROR_INVALID_PROGRAM_VERSION :
        strcat(str, "CL_BUILD_ERROR_INVALID_PROGRAM_VERSION"); break;
    case CL_BUILD_ERROR_INVALID_KERNEL_INPUTS :
        strcat(str, "CL_BUILD_ERROR_INVALID_KERNEL_INPUTS"); break;
    case CL_BUILD_ERROR_INVALID_KERNEL_CODE :
        strcat(str, "CL_BUILD_ERROR_INVALID_KERNEL_CODE"); break;
    case CL_BUILD_ERROR_INVALID_KERNEL_OUTPUTS :
        strcat(str, "CL_BUILD_ERROR_INVALID_KERNEL_OUTPUTS"); break;
    case CL_BUILD_ERROR_INVALID_SAMPLER_INPUTS :
        strcat(str, "CL_BUILD_ERROR_INVALID_SAMPLER_INPUTS"); break;
    case CL_BUILD_ERROR_INVALID_SAMPLER_CODE :
        strcat(str, "CL_BUILD_ERROR_INVALID_SAMPLER_CODE"); break;
    case CL_BUILD_ERROR_INVALID_SAMPLER_OUTPUTS :
        strcat(str, "CL_BUILD_ERROR_INVALID_SAMPLER_OUTPUTS"); break;
    case CL_BUILD_ERROR_INVALID_KERNEL_KERNELS :
        strcat(str, "CL_BUILD_ERROR_INVALID_KERNEL_KERNELS"); break;
    case CL_BUILD_ERROR_INVALID_KERNEL_FUNCTION _____________________________________________
case CL_BUILD_IN_PROGRESS : strcat(str, "CL_BUILD_IN_PROGRESS");
}
return;
}

/**
 * clPlatformDump function
 *
 * \brief This function prints out all information about the given platform ID
 *
 * \param pid Platform id pointer
 *
 * \return None
 *
 */
extern "C" void clPlatformDump(cl_platform_id pid) {
  printf("Printing information for platform ID:0x%"\n", pid);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_PROFILE, NULL, NULL, &psize), __LINE__, __FILE__);
  pstr = (char*)malloc(psize);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_PROFILE, psize, pstr, NULL), __LINE__, __FILE__);
  printf("\tProfile: %s\n", pstr);
  free(pstr);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VERSION, NULL, NULL, &psize), __LINE__, __FILE__);
  pstr = (char*)malloc(psize);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VERSION, psize, pstr, NULL), __LINE__, __FILE__);
  printf("\tVersion: %s\n", pstr);
  free(pstr);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_NAME, NULL, NULL, &psize), __LINE__, __FILE__);
  pstr = (char*)malloc(psize);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_NAME, psize, pstr, NULL), __LINE__, __FILE__);
  printf("\tName: %s\n", pstr);
  free(pstr);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, NULL, NULL, &psize), __LINE__, __FILE__);
  pstr = (char*)malloc(psize);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, psize, pstr, NULL), __LINE__, __FILE__);
  printf("\tVendor: %s\n", pstr);
  free(pstr);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_EXTENSIONS, NULL, NULL, &psize), __LINE__, __FILE__);
  pstr = (char*)malloc(psize);
  clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_EXTENSIONS, psize, pstr, NULL), __LINE__, __FILE__);
  printf("\tExtensions: %s\n", pstr);
  free(pstr);
  printf("\n");
  return;
}

/**
 * clDeviceDump function

APPENDIX A. APPENDIXES
119
extern "C" void clDeviceDump(cl_device_id did) {
    size_t * pdims;
    int idx = 0;
    cl_device_fp_config fpcfg;
    cl_device_type dtype;
    cl_device_exec_capabilities dcap;
    cl_device_mem_cache_type mctype;
    cl_device_local_mem_type lmtype;
    cl_platform_id dplat;
    cl_command_queue_properties dqprop;
    printf("Printing information for device ID: 0x%X\n", did);
    printf("------------------------------------------- --------------------\n");
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR_ID, sizeof(cl_uint), &puint, NULL),
               __LINE__, __FILE__);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR, NULL, NULL, &psize),
               __LINE__, __FILE__);
    pstr = (char*) malloc(psize);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VENDOR, psize, pstr, NULL),
               __LINE__, __FILE__);
    printf("Vendor: %s (ID: 0x%X)\n", pstr, puint);
    free(pstr);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NAME, NULL, NULL, &psize),
               __LINE__, __FILE__);
    pstr = (char*) malloc(psize);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NAME, psize, pstr, NULL),
               __LINE__, __FILE__);
    printf("Name: %s", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_TYPE, sizeof(cl_device_type),
                &dtype, NULL),
               __LINE__, __FILE__);
    pstr = (char*) malloc(sizeof(char)*STRLEN);
    clDecodeDeviceType(dtype, pstr);
    printf("%s\", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VERSION, NULL, NULL, &psize),
                __LINE__, __FILE__);
    pstr = (char*) malloc(psize);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_VERSION, psize, pstr, NULL),
                __LINE__, __FILE__);
    printf("ver. %s\n", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(did, CL_DRIVER_VERSION, NULL, NULL, &psize),
               __LINE__, __FILE__);
    pstr = (char*) malloc(psize);
    clCheckErr(clGetDeviceInfo(did, CL_DRIVER_VERSION, psize, pstr, NULL),
               __LINE__, __FILE__);
    printf("\Driver: %s\n", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PLATFORM, sizeof(cl_platform_id),
                    &dplat, NULL),
               __LINE__, __FILE__);
    printf("\Platform: 0x%X\n", dplat);
    clCheckErr(clGetDeviceInfo(did, CL_DEVICE_OPENCL_C_VERSION, NULL, NULL, &psize),
               __LINE__, __FILE__);
    printf("\C: %s\n", pstr);
    free(pstr);
pstr = (char*)malloc(psize);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PROFILE, NULL, NULL, &psize), __LINE__, __FILE__);
printf("Device compute capabilities\n");
printf("-----------------------------\n");
free(pstr);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(cl_uint), &puint), __LINE__, __FILE__);
printf("\n\n\n:\tClock Frequency: %dMHz (max)\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(size_t), &psizei), __LINE__, __FILE__);
printf(":\tWork Group Size: %d (max)\n", psizei);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t), &psizei), __LINE__, __FILE__);
printf(":\tWork Group Size: %d (max)\n", psizei);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &puint), __LINE__, __FILE__);
printf(":\tCompute Units: %d (max)\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_ITEM_THREADS, sizeof(size_t), &psizei), __LINE__, __FILE__);
printf(":\tWork Item Threads: %d\n", psizei);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_LOCAL_MEM_SIZE, sizeof(cl_uint), &puint), __LINE__, __FILE__);
printf(":\tLocal Memory Size: %d\n", puint);
APPENDIX A. APPENDIXES

```
689  pdims = (size_t*)malloc(sizeof(size_t)*puint);
690  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WORK_ITEM_SIZES,
691      sizeof(size_t)*puint, pdims, NULL), __LINE__, __FILE__);
692  printf("\tWork Item Sizes: [\%d", pdims[0]);
693  for (idx = 1; idx < puint; idx++) { printf(" , %d", pdims[idx]); }
694  printf("] (max)\n");
695  free(pdims);
696
697  printf("\nDevice Memory Information\n");
698  printf("------------------------------------------- --------------------\n");
699  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ADDRESS_BITS, sizeof(cl_uint),
700      &puint, NULL), __LINE__, __FILE__);
701  printf("\tAddress Bits: %d\n", puint);
702  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ENDIAN_LITTLE, sizeof(cl_bool),
703      &pbool, NULL), __LINE__, __FILE__);
704  printf("\tLittle Endian: %s\n", (pbool) ? "Yes" : "No");
705  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MEM_BASE_ADDR_ALIGN, sizeof(cl_uint),
706      &puint, NULL), __LINE__, __FILE__);
707  printf("\tMemory Base Addr Align: %d bits\n", puint);
708  clCheckErr(clGetDeviceInfo(did, CLDEVICE_MIN_DATA_TYPE_ALIGN_SIZE,
709      sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
710  printf("\tError Correction: %s\n", (pbool) ? "Yes" : "No");
711  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_HOST_UNIFIED_MEMORY,
712      sizeof(cl_bool), &pbool, NULL), __LINE__, __FILE__);
713  printf("\tDev / Host Unified Memory: %s\n", (pbool) ? "Yes" : "No");
714  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_SIZE,
715      sizeof(cl_ulong), &pulong, NULL), __LINE__, __FILE__);
716  printf("\tGlobal Memory Size: %luB\n", pulong);
717  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,
718      sizeof(cl_ulong), &pulong, NULL), __LINE__, __FILE__);
719  printf("\tGlobal Memory Cache: %luB\n", pulong);
720  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHE_TYPE,
721      sizeof(cl_device_mem_cache_type), &mctype, NULL), __LINE__, __FILE__);
722  pstr = (char*)malloc(sizeof(char)*STRLEN);
723  clDecodeDevMemCacheType(mctype, pstr);
724  printf("\tGlobal Memory Cache Type: %s\n", pstr);
725  free(pstr);
726  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,
727      sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
728  printf("\tGlobal Memory CacheLine Size: %dB\n", puint);
729  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_LOCAL_MEM_SIZE,
730      sizeof(cl_ulong), &pulong, NULL), __LINE__, __FILE__);
731  printf("\tLocal Memory Size: %luB\n", pulong);
732  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_LOCAL_MEM_TYPE,
733      sizeof(cl_device_local_mem_type), &lmtype, NULL), __LINE__, __FILE__);
734  pstr = (char*)malloc(sizeof(char)*STRLEN);
735  clDecodeLocalMemType(lmtype, pstr);
736  printf("\tLocal Memory Type: %s\n", pstr);
737  free(pstr);
738  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE_SUPPORT,
```
printf("\t Image Support : %s
", (pbool) ? "Yes" : "Nofolders/n; 
if (pbool) { 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_READ_IMAGE_ARGS, 
sizeof(cl_uint), &puint, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Read Image Args : %d (max)\n", puint); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_WRITE_IMAGE_ARGS, 
sizeof(cl_uint), &puint, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Write Image Args : %d (max)\n", puint); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_SAMPLERS, 
sizeof(cl_uint), &puint, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Samplers : %d (max)\n", puint); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE2D_MAX_HEIGHT, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Image 2D Height : %d (max)\n", psize); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE2D_MAX_WIDTH, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Image 2D Width : %d (max)\n", psize); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_DEPTH, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Image 3D Depth : %d (max)\n", psize); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_HEIGHT, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Image 3D Height : %d (max)\n", psize); 
  clCheckErr(clGetDeviceInfo(did, CL_DEVICE_IMAGE3D_MAX_WIDTH, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
  printf("\t \t Image 3D Width : %d (max)\n", psize); 
} 
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CONSTANT_ARGS, 
sizeof(cl_uint), &puint, NULL), 
  __LINE__, __FILE__); 
printf("\t Constant Args : %d (max)\n", puint); 
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE, 
sizeof(cl_ulong), &pulong, NULL), 
  __LINE__, __FILE__); 
printf("\t Memory Allocation Size : %dB (max)\n", pulong); 
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_MAX_PARAMETER_SIZE, 
sizeof(size_t), &psize, NULL), 
  __LINE__, __FILE__); 
printf("\t Parameter Size : %dB (max)\n", psize); 
printf("Device data formatting:\n"); 
printf("-----------------------------------------------\n");

#define OPENCL_1.2
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_DOUBLE_FP_CONFIG, 
sizeof(cl_device_fp_config), &fpcfg, NULL), 
  __LINE__, __FILE__); 
pstr = (char*)malloc(sizeof(char)*STRLEN); 
clDecodeFPConfig(fpcfg, pstr); 
printf("\t Double FP Config : %s\n", pstr);
free(pstr);
#endif
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_SINGLE_FP_CONFIG,
    sizeof(cl_device_fp_config), &fpcfg, NULL),
    __LINE__, __FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
clDecodeFPConfig(fpcfg, pstr);
printf("\tSingle FP Config : %s\n", pstr);
free(pstr);
#endif
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_HALF_FP_CONFIG,
    sizeof(cl_device_fp_config), &fpcfg, NULL),
    __LINE__, __FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
clDecodeFPConfig(fpcfg, pstr);
printf("\tHalf FP Config : %s\n", pstr);
free(pstr);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_ERROR_CORRECTION_SUPPORT,
    sizeof(cl_bool), &pbool, NULL),
    __LINE__, __FILE__);
printf("\tDevice Error Correction Support (ECS) : %s\n", (pbool) ? "Yes" : "No");
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tCHAR : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tSHORT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_INT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tINT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tLONG : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tFLOAT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tHALF : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tCHAR : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tSHORT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERREDVECTOR_WIDTH_INT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tINT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tLONG : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tFLOAT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF,
    sizeof(cl_uint), &puint, NULL),
    __LINE__, __FILE__);
printf("\tHALF : %d\n", puint);
APPENDIX A. APPENDIXES

```c
printf("\t\t\t\tINT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG, sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
printf("\t\t\tLONG : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERREDVECTOR_WIDTH_FLOAT, sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
printf("\t\t\tFLOAT : %d\n", puint);
clCheckErr(clGetDeviceInfo(did, CL_DEVICE_PREFERREDVECTOR_WIDTH_DOUBLE, sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
printf("\t\t\tDOUBLE : %d\n", puint);
printf("\t\t\tHALF : %d\n", puint);
printf("\n");
return ;
}

/**
 * clContextDump function
 *
 * brief This function prints out all of the information about the given context
 *
 * param cid Context id
 * return None
 *
 */

extern "C" void clContextDump(cl_context cid) {
int idx = 0;
cl_device_id * cdevs;
cl_context_properties cprops[3] = {0, 0, 0};
printf("Printing information for context ID:0x%X\n", cid);
clCheckErr(clGetContextInfo(cid, CL_CONTEXT_REFERENCE_COUNT, sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
printf("\tReference Count : %d\n", puint);
clCheckErr(clGetContextInfo(cid, CL_CONTEXT_NUM_DEVICES, sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
printf("\tDevice Count : %d\n", puint);
cdevs = (cl_device_id*) malloc(sizeof(cl_device_id)*puint);
clCheckErr(clGetContextInfo(cid, CL_CONTEXT_DEVICES, psize, cdevs, NULL), __LINE__, __FILE__);
printf("\tDevices: ");
for (idx = 0; idx < puint; idx++) {
if (idx > 0) { printf(" , "); }
printf("0x%X", cdevs[idx]);
}
cCheckErr(clGetContextInfo(cid, CL_CONTEXT_PROPERTIES, NULL, NULL, &psize), __LINE__, __FILE__);
cCheckErr(clGetContextInfo(cid, CL_CONTEXT_PROPERTIES, psize, cprops, NULL), __LINE__, __FILE__);
pstr = (char*) malloc(sizeof(char)*STRLEN);
cDecodeQueueProperties(*cprops, pstr);
printf("\t\nProperties: %s\n", pstr);
free(pstr);
#endif OPENCL_1.2
```

clCheckErr(clGetContextInfo(cid, CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR,
                       sizeof(cl_bool), &pbool, NULL),
__LINE__, __FILE__);}
#endif
pbool = false;
printf("\tD3D10 Support: \%s\n", (pbool) ? "Yes" : "No");
printf("\n");
return;
}

/**
 * clMemObjDump function
 *
 * \brief This function prints out all of the information about the given
 * memory object
 *
 * \param mobj Memory object
 * \return None
 */
extern "C" void clMemObjDump(cl_mem mobj) {
cl_mem_object_type mtype;
cl_mem_flags mflags;
cl_context ctx;
void * vptr;

printf(" Printing information for memory object ID: 0x%X
", mobj);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_TYPE, sizeof(cl_mem_object_type),
                           &mtype, NULL),
__LINE__, __FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
clDecodeMemObjType(mtype, pstr);
printf(" Type %s
", pstr);
free(pstr);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_FLAGS, sizeof(cl_mem_flags),
                           &mflags, NULL),
__LINE__, __FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
clDecodeMemObjFlags(mflags, pstr);
printf(" Flags %s
", pstr);
free(pstr);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_SIZE, sizeof(size_t),
                           &psize, NULL),
__LINE__, __FILE__);
printf(" Size %sB
", psize);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_HOST_PTR, sizeof(void*),
                           &vptr, NULL),
__LINE__, __FILE__);
printf(" Host Ptr 0x%X\n", vptr);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_MAP_COUNT, sizeof(cl_uint),
                           &puint, NULL),
__LINE__, __FILE__);
printf(" Map Count %d\n", puint);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_REFERENCE_COUNT, sizeof(cl_uint),
                           &puint, NULL),
__LINE__, __FILE__);
printf(" Reference Count %d\n", puint);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_CONTEXT, sizeof(cl_context),
                           &ctx, NULL),
__LINE__, __FILE__);
printf(" Context 0x%X\n", &ctx);
clCheckErr(clGetMemObjectInfo(mobj, CL_MEM_ASSOCIATED_MEMOBJECT, sizeof(cl_mem),
                           &mobj, NULL),
__LINE__, __FILE__);
APPENDIX A. APPENDIXES

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  clCheckErr("Printing information for image object ID:0x%X\n", img);  
1027  clCheckErr(clGetImageInfo(img, CL_IMAGE_FORMAT, sizeof(cl_image_format),  
1028     &fmt, NULL),  
1029     __LINE__, __FILE__);  
1030  pstr = (char*)malloc(sizeof(char)*STRLEN);  
1031  clDecodeImageFormat(fmt, pstr);  
1032  printf("%s\n", pstr);  
1033  free(pstr);  
1034  clCheckErr(clGetImageInfo(img, CL_IMAGE_ELEMENT_SIZE, sizeof(size_t),  
1035     &psize, NULL),  
1036     __LINE__, __FILE__);  
1037  printf("Element Size %d\n", psize);  
1038  clCheckErr(clGetImageInfo(img, CL_IMAGE_ROW_PITCH, sizeof(size_t),  
1039     &psize, NULL),  
1040     __LINE__, __FILE__);  
1041  printf("Row Pitch %d\n", psize);  
1042  clCheckErr(clGetImageInfo(img, CL_IMAGE_SLICE_PITCH, sizeof(size_t),  
1043     &psize, NULL),  
1044     __LINE__, __FILE__);  
1045  printf("Slice Pitch %d\n", psize);  
1046  clCheckErr(clGetImageInfo(img, CL_IMAGE_WIDTH, sizeof(size_t),  
1047     &psize, NULL),  
1048     __LINE__, __FILE__);  
1049  printf("Width %d\n", psize);  
1050  clCheckErr(clGetImageInfo(img, CL_IMAGE_HEIGHT, sizeof(size_t),  
1051     &psize, NULL),  
1052     __LINE__, __FILE__);  
1053  printf("Height %d\n", psize);  
1054  clCheckErr(clGetImageInfo(img, CL_IMAGE_DEPTH, sizeof(size_t),  
1055     &psize, NULL),  
1056     __LINE__, __FILE__);  
1057  printf("Depth %d\n", psize);  
1058  printf("\n");  
1059  return;  
1060 }
1061  
1062  /**
1063  * clSamplerDump function
1064  *
1065  * \brief This function prints out all of the information about the sampler object
extern "C" void clSamplerDump(cl_sampler smp) {
    cl_context ctx;
    cl_addressing_mode amd;
    cl_filter_mode fmd;

    printf("Printing information for sampler ID:0x%X\n", smp);
    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_REFERENCE_COUNT,
        sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
    printf(" Reference Count %d\n", puint);
    free(pstr);
    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_CONTEXT,
        sizeof(cl_context), &ctx, NULL), __LINE__, __FILE__);
    printf(" Context 0x%X\n", ctx);
    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_ADDRESSING_MODE,
        sizeof(cl_addressing_mode), &amd, NULL), __LINE__, __FILE__);
    pstr = (char*) malloc(sizeof(char)*STRLEN);
    clDecodeAddressingMode(amd, pstr);
    printf(" Addressing Mode %s\n", pstr);
    free(pstr);
    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_FILTER_MODE,
        sizeof(cl_filter_mode), &fmd, NULL), __LINE__, __FILE__);
    pstr = (char*) malloc(sizeof(char)*STRLEN);
    clDecodeFilterMode(fmd, pstr);
    printf(" Filter Mode %s\n", pstr);
    free(pstr);
    clCheckErr(clGetSamplerInfo(smp, CL_SAMPLER_NORMALIZED_COORDS,
        sizeof(cl_bool), &pbool, NULL), __LINE__, __FILE__);
    printf(" Normalized Coordinates %s\n", (pbool) ? "Yes" : "No");
    printf("\n");
    return;
}

extern "C" void clProgramDump(cl_program pid) {
    int jdx, kdx = 0;
    cl_context ctx;
    cl_device_type dtpt;
    size_t * psizes = NULL;
    unsigned char ** bins = NULL;
    unsigned char * bin = NULL;
    cl_device_id * dptr = NULL;

    printf("Printing information for program ID:0x%X\n", pid);
    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_REFERENCE_COUNT,
        sizeof(cl_uint), &puint, NULL), __LINE__, __FILE__);
    printf(" Reference Count %d\n", puint);
    free(pstr);
    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_CONTEXT,
        sizeof(cl_context), &ctx, NULL), __LINE__, __FILE__);
    printf(" Context 0x%X\n", ctx);
    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_ADDRESSING_MODE,
        sizeof(cl_addressing_mode), &amd, NULL), __LINE__, __FILE__);
    pstr = (char*) malloc(sizeof(char)*STRLEN);
    clDecodeAddressingMode(amd, pstr);
    printf(" Addressing Mode %s\n", pstr);
    free(pstr);
    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_FILTER_MODE,
        sizeof(cl_filter_mode), &fmd, NULL), __LINE__, __FILE__);
    pstr = (char*) malloc(sizeof(char)*STRLEN);
    clDecodeFilterMode(fmd, pstr);
    printf(" Filter Mode %s\n", pstr);
    free(pstr);
    clCheckErr(clGetProgramInfo(pid, CL_PROGRAM_NORMALIZED_COORDS,
        sizeof(cl_bool), &pbool, NULL), __LINE__, __FILE__);
    printf(" Normalized Coordinates %s\n", (pbool) ? "Yes" : "No");
    printf("\n");
    return;
}
APPENDIX A. APPENDIXES

```c
__Line__ __file__ );
printf(" Reference Count %d\n", puint);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_CONTEXT,
    sizeof(cl_context), &ctx, NULL),
__Line__ __file__ );
printf(" Context 0x%d\n", ctx);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_SOURCE, NULL, NULL, &psize),
__Line__ __file__ );
pstr = (char*)malloc(psize);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_SOURCE, psize, pstr, NULL),
__Line__ __file__ );
printf(" Program Source \n");
printf("%s\n", pstr);
free(pstr);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_NUM_DEVICES,
    sizeof(cl_uint), &puint, NULL),
__Line__ __file__ );
printf(" Number of Devices %d\n", puint);
// Program devices
psize = sizeof(cl_device_id)*puint;
dptr = (cl_device_id*)malloc(psize);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_DEVICES, psize, dptr, NULL),
__Line__ __file__ );
// Program binary sizes
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_BINARY_SIZES, NULL, NULL, &psize),
__Line__ __file__ );
psizes = (size_t*)malloc(psize);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_BINARY_SIZES, psize, psizes, NULL),
__Line__ __file__ );
size = sizeof(unsigned char*) * puint;
bins = (unsigned char**)malloc(psize);
ciCheckErr(ciGetProgramInfo(pid, CL_PROGRAM_BINARY_SIZES, psize, bins, NULL),
__Line__ __file__ );
printf("Devices\n");
for (jdx = 0; jdx < puint; jdx++) {
    printf(" %02d] " , jdx);
    // Device name
ciCheckErr(ciGetDeviceInfo(dptr[jdx], CL_DEVICE_NAME, NULL, NULL, &psize),
__Line__ __file__ );
pstr = (char*)malloc(psize);
ciCheckErr(ciGetDeviceInfo(dptr[jdx], CL_DEVICE_NAME, psize, pstr, NULL),
__Line__ __file__ );
printf("%s " , pstr);
free(pstr);
    // Device type
ciCheckErr(ciGetDeviceInfo(dptr[jdx], CL_DEVICE_TYPE,
    sizeof(cl_device_type), &dtp, NULL),
__Line__ __file__ );
pstr = (char*)malloc(sizeof(char)*STRLEN);
clDecodeDeviceType(dtp, pstr);
free(pstr);
    // Device Binary
    bin = bins[jdx];
    for (kdx = 0; kdx < psizes[jdx]; kdx++) {
        printf("%02X", bin[kdx]);
        if ((kdx % 4) == 3) { printf(" "); }
        if ((kdx % 16) == 15) { printf("\n"); }
    }
    printf("\n");
free(bins);
free(psizes);
```
free (dptr);
return;
}
/**
 * clProgramBuildDump function
 *
 * \brief This function prints out all of the information about the given program
 * object's build for the specified device
 *
 * \param prg Program id
 * \param dev Device id
 * \return None
 */

extern "C" void clProgramBuildDump (cl_program prg, cl_device_id dev) {

cl_build_status bst;
cl_device_type dtp;

printf("Printing information for Program ID:0x%X Device ID:0x%X\n", prg, dev);
// Device name
clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_NAME, NULL, NULL, &psize),
_LINE__FILE__);
pstr = (char*)malloc(psize);
clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_NAME, psize, pstr, NULL),
_LINE__FILE__);
printf("%s ", pstr);
free(pstr);
// Device type
clCheckErr(clGetDeviceInfo(dev, CL_DEVICE_TYPE, sizeof(cl_device_type),
&dtp, NULL),
_LINE__FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
cDecodeDeviceType(dtp, pstr);
printf("(%s)\n", pstr);
free(pstr);
// Program build status
clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_STATUS,
 sizeof(cl_build_status), &bst, NULL),
_LINE__FILE__);
pstr = (char*)malloc(sizeof(char)*STRLEN);
cDecodeBuildStatus(bst, pstr);
printf("Build Status %s\n", pstr);
free(pstr);
// Program build options
clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_OPTIONS,
 NULL, NULL, &psize),
_LINE__FILE__);
pstr = (char*)malloc(psize);
clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_OPTIONS,
 psize, pstr, NULL),
_LINE__FILE__);
printf("Build Options %s\n", pstr);
free(pstr);
clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_LOG,
 NULL, NULL, &psize),
_LINE__FILE__);
pstr = (char*)malloc(psize);
clCheckErr(clGetProgramBuildInfo(prg, dev, CL_PROGRAM_BUILD_LOG,
 psize, pstr, NULL),
_LINE__FILE__);
printf("Build Log\n%s", pstr);
return;
/**
 * clKernelDump function
 * 
 * \brief This function prints out all of the information for the given
 * kernel object
 * 
 * \param krn Kernel object
 * \return None
 *
 */
extern "C" void clKernelDump( cl_kernel krn ) {
    printf("Printing information for Kernel ID: 0x%X\n", krn);
    // Function name
    clCheckErr( clGetKernelInfo( krn, CL_KERNEL_FUNCTION_NAME, NULL, NULL, &psize ),
                __LINE__, __FILE__);
    pstr = (char*) malloc( psize );
    clCheckErr( clGetKernelInfo( krn, CL_KERNEL_FUNCTION_NAME, psize, pstr, NULL),
                __LINE__, __FILE__);
    printf("Function Name: %s\n", pstr);
    free(pstr);
    // Kernel Number of Arguments
    clCheckErr( clGetKernelInfo( krn, CL_KERNEL_NUM_ARGS, sizeof(cl_uint), &puint, NULL),
                __LINE__, __FILE__);
    printf("Num Args: %d\n", puint);
    // Kernel reference count
    clCheckErr( clGetKernelInfo( krn, CL_KERNEL_REFERENCE_COUNT, sizeof(cl_uint), &puint, NULL),
                __LINE__, __FILE__);
    printf("Ref Cnt: %d\n", puint);
    return;
}

/**
 * clPrintPlats function
 * 
 * \brief This function prints out a very short list of the platforms
 * found on the system
 * 
 * \return None
 *
 */
extern "C" void clPrintPlats() {
    cl_uint idx, np = 0;
    cl_platform_id * ps;
    cl_platform_id curplat;
    // Query and list platforms
    clCheckErr( clGetPlatformIDs(0, NULL, &np), __LINE__, __FILE__);
    ps = (cl_platform_id*) malloc( sizeof(cl_platform_id)*np );
    clCheckErr( clGetPlatformIDs( np, ps, NULL ), __LINE__, __FILE__ );
    for (idx = 0; idx < np; idx++) {
        curplat = ps[idx];
        printf("Platform %d: 0x%X => ", idx, curplat);
        clCheckErr( clGetPlatformInfo( curplat, CL_PLATFORM_VENDOR, NULL, NULL, &psize),
                    __LINE__, __FILE__ );
        pstr = (char*) malloc( psize );
        clCheckErr( clGetPlatformInfo( curplat, CL_PLATFORM_VENDOR, pstr, NULL, NULL),
                    __LINE__, __FILE__ );
        printf("Vendor: %s\n", pstr);
        free(pstr);
    }
}

APPENDIX A. APPENDIXES 131
APPENDIX A. APPENDIXES

```c
APPENDIX A. APPENDIXES

# clPrintDevs function
* brief This function prints out a very short list of the devices found
  on the specified platform
* param pid Platform ID
* return None
*/
extern "C" void clPrintDevs (cl_platform_id pid)
{
  cl_uint idx, nd = 0;
  cl_device_id *ds;
  cl_device_id currdev;
  cl_device_type tmptype;

  // Query and list devices
  clCheckErr(clGetDeviceIDs(pid, CL_DEVICE_TYPE_ALL, NULL, NULL, &nd),
    __LINE__, __FILE__);
  ds = (cl_device_id*)malloc(sizeof(cl_device_id)*nd);
  clCheckErr(clGetDeviceIDs(pid, CL_DEVICE_TYPE_ALL, nd, ds, NULL),
    __LINE__, __FILE__);

  for (idx = 0; idx < nd; idx++) {
    currdev = ds[idx];
    printf("Device %d (ID:0x%lx):", idx, currdev);
    clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_NAME, NULL, NULL, &psize),
      __LINE__, __FILE__);
    pstr = (char*)malloc(psize);
    clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_NAME, psize, pstr, NULL),
      __LINE__, __FILE__);
    printf("%s", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_TYPE, NULL, NULL, &tmptype),
      __LINE__, __FILE__);
    pstr = (char*)malloc(sizeof(char)*STRLEN);
    clDecodeDeviceType(tmptype, pstr);
    printf(" (%s)\n", pstr);
    free(pstr);
    clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_VERSION, NULL, NULL, &psize),
      __LINE__, __FILE__);
    pstr = (char*)malloc(psize);
    clCheckErr(clGetDeviceInfo(currdev, CL_DEVICE_VERSION, psize, pstr, NULL),
      __LINE__, __FILE__);
    printf("%s", pstr);
    free(pstr);
  }
  free(ps);
  return;
}
```
APPENDIX A. APPENDIXES

printf(" ver. %s\n", pstr);
free(pstr);
}
free(ds);
return;
}

.printf("OCLUTIL ERROR: Invalid task platform\n");
#ifdef KILLONERR
exit(-559);
#endif
return;
}

extern "C" void clDecodeTaskPlat(int plat, char * cmpStr) {
    cmpStr[0] = '\0';
    switch(plat) {
    case 0: strcat(cmpStr, " NVIDIA "); break;
    case 1: strcat(cmpStr, " AMD "); break;
    case 2: strcat(cmpStr, " Intel "); break;
    default :
        printf("OCLUTIL ERROR: Invalid task platform\n");
        #ifdef KILLENERR
        exit(-559);
        #endif
    }
    return;
}

extern "C" int clFindPlatform(int plat) {
    cl_uint nplat = 0;
    clCheckErr(clGetPlatformIDs(0, NULL, &nplat),__LINE__,__FILE__);
    cl_platform_id * plats = (cl_platform_id*)malloc(sizeof(cl_platform_id)*nplat);
    clCheckErr(clGetPlatformIDs(nplat,plats,NULL),__LINE__,__FILE__);
    char cmpStr[STRLEN];
    clDecodeTaskPlat(plat, cmpStr);

    for (int idx = 0; idx < nplat; idx++) {
        cl_platform_id pid = plats[idx];
        clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, NULL, NULL, &psize),__LINE__,__FILE__);
        pstr = (char*)malloc(psize);
        clCheckErr(clGetPlatformInfo(pid, CL_PLATFORM_VENDOR, psize, pstr, NULL),__LINE__,__FILE__);
        if (strncmp(cmpStr,pstr,6)) return idx;
    }
    free(pstr);
    return -1;
}
Listing A.5: OpenCL helper function code

A.2.5 CPU Utility Functions
/**
 * cpuInitMem function
 * 
 * \brief This function intelligently allocates memory on the host or device
 * as specified and returns pointer to the memory as well as the
 * stride for the data, which is used for multi-dimensional data
 * blocks.
 * \param dims The dimensions of the memory to be allocated
 * \param stride The stride (width in bytes) of the data. This only matters
 * for multi-dimensional data
 * \param loc The location of the memory to be allocated (host or device)
 * \param ptr A pointer to where the memory pointer is to be stored
 * \return None
 */

extern "C" void * cpuInitMem (int * dims, int typeSize, int * stride, const char * name, HTaskInfo * info, int mapHostFlag) {
    void * locPtr = NULL;
    int datSize = 1;
    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
    datSize *= typeSize;
    locPtr = malloc(datSize);
    memset(locPtr, 0, datSize);
    return locPtr;
}

/**
 * cpuFreeMem function
 * 
 * \brief This function frees allocated memory on the host or device
 * as specified.
 * \param ptr Pointer to the data to be freed
 * \param dims The dimensions of the data to be freed
 * \param loc The location of the data to be freed
 * \return None
 */

extern "C" void cpuFreeMem(void * ptr, int * dims, int typeSize, const char * name, HTaskInfo * info, int mapHostFlag) {
    free(ptr);
    return;
}

/**
 * cpuClearMem function
 * 
 * \brief
 * \param dims The dimensions of the memory to be allocated
 * \param stride The stride (width in bytes) of the data. This only matters
 * for multi-dimensional data
 * \param loc The location of the memory to be allocated (host or device)
 * \param ptr A pointer to where the memory pointer is to be stored
 * \return None
 */

extern "C" void cpuClearMem(int * dims, int typeSize, int stride, void * ptr, HTaskInfo * info) {
    int datSize = 1;
    for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
    datSize *= typeSize;
    memset(ptr, 0, datSize);
/**
 * \brief This function intelligently copies data from the source location
 * with the specified stride to the destination location with the
 * specified stride. The dimensions parameter indicates the
 * dimensions of the data and must be the same for both source and
 * destination.
 *
 * \param dest Pointer to the destination memory location
 * \param destStride Stride (width in bytes) of the destination
 * \param destLoc The location of the destination memory
 * \param src Pointer to the source memory location
 * \param srcStride Stride (width in bytes) of the source memory
 * \param srcLoc The location of the destination memory
 * \param src The dimensions of the data to be copied
 * \return None
 */

extern "C" void cpuMoveData(void * dst, int dstStride, HTaskInfo * dstInfo,
                           void * src, int srcStride, HTaskInfo * srcInfo,
                           int * dims, int typeSize, const char * name) {
  int datSize = 1;
  for (int idx = 0; idx < HNDIMS; idx++) { datSize *= dims[idx]; }
  datSize *= typeSize;
  memcpy(dst, src, datSize);
  return;
}

/**
 * \brief This function uses the specified parameters to execute the
 * coprocessor function specified by kernel. Any parameters to
 * the kernel should be included in params.
 *
 * \param kernel A string naming the kernel to execute
 * \param dims The dimensions of the data to execute the kernel on
 * \param nParams The number of parameters to be passed to the kernel
 * \param params Array of parameters to be passed to the kernel
 * \param gDim The grid dimensions of the kernel
 * \param bDim The block dimensions of the kernel
 * \param locMem The amount of local or shared memory to allocate for kernel
 * \param stream The stream index to associate this kernel’s execution and
 * \return None
 */

extern "C" void cpuLaunchKernel(const char * krn, int * dims, int nParams,
                                int * paramSizes, void ** params,
                                int * gDim, int * bDim, int locMem,
                                HTaskInfo * info) {
  printf("Launching %s on cpu
", krn);
  if (!strcmp(krn, "cpu_mc_stage1")) {
    cpu_mc_stage1(((randState**)params[0]), *((phDetPkg**)params[1]),
                  *((uint**)params[2]), *((float**)params[3]), *((int**)params[4]),
                  *((InputStruct**)params[5]));
  }
}
if (!strcmp(krn, "cpu_mcStage2")) {
    cpu_mcStage2(((phDetPkg**) params[0]), *((positionGrid**) params[1]),
    *((uint**) params[2]), *((float*) params[3]));
}
return;
}

Listing A.6: CPU utility function code
Bibliography


