RICE UNIVERSITY

Programming Models and Runtimes for Heterogeneous Systems

by

Max Grossman

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

Alan Cox
Associate Professor of Computer Science
Associate Professor of Electrical and
Computer Engineering

John Mellor-Crummey
Professor of Computer Science
Professor of Electrical and Computer
Engineering

Houston, Texas

May, 2013

ABSTRACT

Programming Models and Runtimes for Heterogeneous Systems

by

Max Grossman

With the plateauing of processor frequencies and increase in energy consumption in computing, application developers are seeking new sources of performance acceleration. Heterogeneous platforms with multiple processor architectures offer one possible avenue to address these challenges. However, modern heterogeneous programming models tend to be either so low-level as to severely hinder programmer productivity, or so high-level as to limit optimization opportunities. The novel systems presented in this thesis strike a better balance between abstraction and transparency, enabling programmers to be productive and produce high-performance applications on heterogeneous platforms.

This thesis starts by summarizing the strengths, weaknesses, and features of existing heterogeneous programming models. It then introduces and evaluates four novel heterogeneous programming models and runtime systems: JCUDA, CnC-CUDA, DyGR, and HadoopCL. We'll conclude by positioning the key contributions of each piece in this thesis relative to the state-of-the-art, and outline possible directions for future work.

Acknowledgments

I would first like to thank my advisor, Vivek Sarkar. His mentorship, guidance, and enthusiasm for research has influenced my development as a student more than anything else. Asking to work with him five years ago was one of the best and most fortuitous decisions I have ever made. I owe so much of who I am as a person to him and the Habanero group.

I would also like to thank my thesis committee members, John Mellor-Crummey and Alan Cox, for the time put into their valuable comments and feedback. Their critiques have produced a much better thesis than it ever would have been without them.

I also need to thank all of the co-authors on work presented in this thesis: Mauricio Breternitz, Zoran Budimlic, Sanjay Chatterjee, Alina Sbirlea, and Yonghong Yan. Their collaboration was crucial to the projects presented here, and it was a privilege to be able to work with and learn from each of them.

Finally, I'd like to thank all members of the Habanero research group for being such a significant and positive part of my life for 5 years. My time at Rice would not have been as entertaining, educational, or challenging without having the chance to work alongside the good people in the Habanero group.

Contents

Illustrations

Tables

Chapter 1

Introduction

Recent trends in processor development, cluster composition, and programming model design attest to the increasing recognition that heterogeneous computing is receiving for its value in performance-critical applications across multiple industries. Heterogeneous computing refers to the use of multiple processor architectures by a single application. By executing kernels whose computational characteristics fit the characteristics of different architectures, applications can see improvements in in performance and energy efficiency. Types of processors which are used in existing heterogeneous platforms include multi-core CPUs, many-core GPUs, FPGAs, or other application-specific hardware. Heterogeneous computing may also refer to using processors with the same or similar architectures and instruction sets, but differing quality[1].

As of November 2012, the first, seventh, and eighth highest-performing supercomputers in the world include nodes with co-processors [2]. This is not only evidence of the performance augmentation heterogeneous platforms can provide, but also the improvement in FLOPs/Watt large-scale heterogeneous systems demonstrate relative to those based on homogeneous architectures. The Center for Domain Specific Computing (CDSC) [3] and its work with many architectures in a single platform is evidence of the ongoing change in system composition at a smaller scale.

However, programming models for these systems are a subject of active research and development.

1.1 Benefits of Heterogeneous Programming

Numerous studies have shown the performance and energy benefits of heterogeneous processors [4][5][6][7][8]. While heterogeneous systems can include many architectures, this work focuses on the use of multi-core CPUs and many-core GPUs in a single application. This is arguably the most popular heterogeneous platform configuration today. Many of the ideas and techniques described here are extensible to systems with greater degrees of heterogeneity.

1.1.1 Performance

The most often cited advantage of GPUs is their performance for suitable applications. For example, performance evaluations of JCUDA [9] show that a single GPU can execute data-parallel computation up to 86x faster than 12 Java threads. The primary causes of this are the highly parallel architecture of GPUs and their high bandwidth to and from graphics memory.

In general GPUs contain 10-20 SIMD cores, each of which can execute the same instruction on multiple elements of data at once. As a result, GPUs can execute hundreds to thousands of parallel threads simultaneously.

Crucial to keeping the SIMD units of a GPU utilized is the memory bandwidth to GPU global memory, which is generally higher than the bandwidth between CPUs and system memory. Insufficient bandwidth or inefficient utilization of bandwidth causes threads to stall and prevents full utilization of the compute units in a GPU.

GPUs also provide several hardware features which are useful for optimization. For instance, local (or shared) memory acts as scratchpad memory (or a user-managed cache) which is physically close to the compute units in a GPU and demonstrates much lower latency than GPU global memory [10]. For algorithms that reuse data, making use of local memory to store intermediate values can dramatically improve performance. GPUs also support special types of memory buffers, such as constant or texture memory, which can improve latency and bandwidth for certain access patterns or read-only data.

In general, GPUs perform well on applications similar to their originally intended purpose in graphics processing: those that are streaming, highly-parallel, compute-intensive, and SIMD. For applications with highly divergent kernels or random memory access patterns GPUs are slower than more general-purpose processors, like CPUs.

1.1.2 Energy

In addition to performance, GPUs exhibit better energy efficiency than multi-core CPUs (as measured in FLOPs/Watt).

One of the primary factors contributing to energy efficiency is the lower core clock frequencies which GPUs operate at. For instance, consider the CPUs and GPUs installed in the DAVINCI cluster at Rice University. The core frequency of each CPU core is 2.83GHz, while each GPU's core frequency is only 1.15GHz. If we assume a simplified power equation:

$$Power = Capacitance * Voltage^2 * Frequency \tag{1.1}$$

and realize that higher frequencies require higher voltages, the power consumption of a processor is proportional to $Frequency^3$ [11]. As a result, a single CPU core, while exhibiting higher performance, consumes $\tilde{1}$5x more energy than a single GPU SIMD unit.

Huang et al. [7] showed that while the peak energy consumption of GPUs is higher than that of multi-core CPUs, the speed at which they compute contributes to a decrease in overall energy consumption. Benchmarks with the GEM benchmark showed that GPU power consumption peaked at 250 watts. Multi-threaded CPU execution only reached 100 watts. However, the GPU code completed execution 25 times faster than the multi-threaded CPU implementation. This resulted in $\tilde{1}$0x lower overall power consumption for the GPU despite higher peak consumption.

GPUs also dedicate more transistors to performing the actual computation in an application, whereas CPUs dedicate transistors to ancillary functions such as branch prediction, supporting a larger instruction set, or caches. These architectural differences lead to improved energy efficiency for GPUs. On the other hand, it is the lack of these more advanced architectural features that causes GPUs to exhibit lackluster performance on certain applications.

1.2 Challenges for Productive Heterogeneous Programming

While there are clearly defined benefits to using heterogeneous processors, the challenges of harnessing heterogeneity have prevented mainstream adoption of heterogeneous programming. These challenges include the complexities of a new programming model, the performance complexities of heterogeneous processors, and the fact that many accelerators are only well-suited for applications with certain characteristics.

1.2.1 Programmability

Programmability measures the ability of a programmer to express their logic without solving problems unrelated to the actual task being performed. GPU programming models are notoriously low-level, exposing many of the nitty-gritty details of graphics hardware without high-level abstractions. For programmers unaccustomed to GPU programming, this presents a major hurdle to accessing the computational power of GPUs.

For instance, in many of today's GPU programming models programmers explicitly manage the allocation, freeing, and consistency of data in GPU global memory through allocation, deallocation, and transfer API calls. This often requires maintaining two discrete copies of all data in a program, with different versions present in GPU memory and in the host system's memory. Complexity rises when the full data set cannot fit into GPU global memory and the programmer must track the subset of a data set which currently resides on the device. The existence of several different types of GPU memory adds more conceptual overhead for the programmer.

Support for GPU execution often needs to be integrated into an existing code base, which already makes use of several other programming models. For instance, adding support for GPU computation to an existing distributed, multi-threaded application would require partitioning work among MPI processes, OpenMP threads, and CUDA threads (for example). This would also require separate GPU and CPU kernels implementing the same computation. This generally leads to an increase in complexity in the application, more difficulty in maintaining the code, and added management overhead at runtime.

1.2.2 Complexity

When optimizing applications for execution on GPUs or other accelerators, there are many added factors for optimal performance than on more general architectures. One reason for this was mentioned in Section 1.1.2: fewer transistors dedicated to ancillary functions such as branch prediction or caching. As a result, the negative effect of irregular computation is more pronounced on GPUs than CPUs.

Optimal performance in GPUs also often relies on exploiting GPU-specific features or being aware of certain architectural characteristics. For instance, utilizing texture and local memory can have significant performance benefits when programs exhibit certain suboptimal access patterns on GPUs. Also, transforming programs to increase coalesced memory accesses can be very beneficial for performance.

Finally, when using devices with either physically or conceptually separate address spaces the requirement to transfer data between address spaces can introduce significant amounts of overhead in heterogeneous applications. This overhead can reduce or negate the performance benefits of using heterogeneous hardware. Recent work by AMD and Intel on using a shared physical memory between CPUs and GPUs serves as proof of the impact this added overhead has on many applications. The added energy required to transmit data between physically separate memories can also increase power requirements and degrade the energy efficiency of accelerators.

1.2.3 Problem Domain

GPUs are generally considered efficient for problems which exhibit highly regular computation across a large data set. This is partly a result of the abstractions which current GPU programming models present to the user to simplify GPU programming and achieve high performance. In many cases, GPUs are capable of matching CPU performance on irregular applications that common best practices wouldn't recommend for mapping to graphics hardware. Achieving this mapping incurs programming overhead necessary to "hack" around the programming model's abstractions.

As an example, best practices dictate that optimal GPU applications have many independent streams of identical execution. Otherwise, GPU performance suffers due to thread divergence. However, this performance penalty is only incurred for threads executing on the same SIMD unit. Divergence between threads executing in separate SIMD units has no effect. Modifying the data layout and thread grouping to place divergent threads on different SIMD units is a difficult task for most applications. Developing heterogeneous programming models and runtimes which present different abstractions to the user and can automatically schedule divergent threads on separate SIMD units could augment the performance of many applications on GPUs and expand the application domain of GPUs.

1.2.4 Problem at Hand

Clearly there are many challenges to efficient execution on heterogeneous, specialized platforms. The performance and power benefits of heterogeneous platforms make the development of high-level high-performance programming models for heterogeneous systems crucial to optimizing computationally demanding applications.

However, heterogeneous programming models are struggling to keep programmers productive. In general, there are two clusters of heterogeneous programming models, as depicted in Figure 1.1. At one end of the spectrum are very low-level models which expose architectural features to the programmer. These models give programmers the greatest flexibility for optimization, but add a large amount of programming overhead. At the other end of the spectrum are high-level heterogeneous programming models that emphasize programmer productivity but provide a more opaque view of the underlying hardware. This limits optimization opportunities for the programmer. This thesis develops and explores heterogeneous programming models that lie between these two clusters and offer a better compromise between programmability and performance.

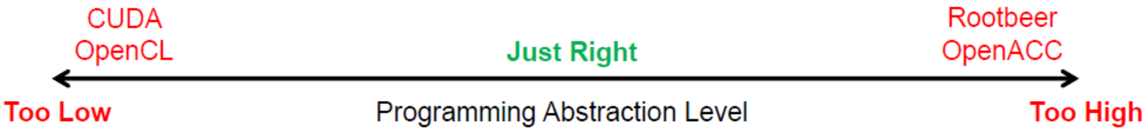


Figure 1.1 : The spectrum of programming models, from low-level models emphasizing performance to high-level models emphasizing programmability

Chapter 2

Background

This chapter provides background on the current tools and APIs used in heterogeneous software development.

2.1 CUDA & OpenCL

This section briefly covers the two most widely used and low-level programming models for software development on heterogeneous platforms: the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) [12]. This includes a comparison of CUDA and OpenCL by studying their platform models (Section 2.1.1), memory models (Section 2.1.3), and execution models (Section 2.1.2)).

At a high level, CUDA is a proprietary tool for execution of general purpose programs on NVIDIA graphics cards. To use it, you must have NVIDIA hardware and NVIDIA's compiler. OpenCL is a more recent and open heterogeneous programming standard supported by the Khronos Compute Working Group. Members of the Khronos Group each have their own OpenCL implementation for different processors, and include companies such as Intel, AMD, NVIDIA, and ARM.

2.1.1 Platform Model

A platform model specifies how the hardware available to a programmer on a certain system is presented, both conceptually and through the API. Both OpenCL and CUDA platform models portray discrete devices which are managed through an API from a host program. These devices have separate address spaces from the host program and use explicit transfers to receive inputs and return outputs to the host program. OpenCL and CUDA provide methods for accessing metadata on each device in a platform (such as memory size, computational units available, etc).

Because OpenCL is intended as a general heterogeneous programming model but CUDA exclusively targets NVIDIA GPUs, most of the differences between the two are additions made to OpenCL's platform model to represent a more general class of devices. At the highest granularity, an OpenCL installation can contain one or more platforms, each of which contains one or more devices. Within each OpenCL device there are multiple compute units. An example of a compute unit would be a single core in a multi-core CPU or a single SIMD unit in a GPU. Each compute unit can also contain one or more processing elements, which could represent the individual ALUs in a vector core.

Access to the platforms and devices in an OpenCL program is more explicit and verbose than in CUDA and requires the creation of contexts and command queues. An OpenCL context is a collection of one or more OpenCL devices. OpenCL command queues are used to issue commands to devices, and each command queue is explicitly associated with a single OpenCL device. Examples of commands include launching computation on the device or transferring data to and from the device.

CUDA is by default less explicit than OpenCL, though it still supports many of the same operations on a CUDA platform. At all times, CUDA has a selected device which CUDA operations are implicitly issued to, though CUDA allows you to explicitly set a currently active device. While there is a model of "streams" of work in CUDA similar to OpenCL's command queues, CUDA streams are not required to be explicitly provided by the programmer for every device operation.

Concept	**CUDA Term**	**OpenCL Term**	**Term Used Here**
Thread of Execution	Thread	Work Item	Thread
Collection of Threads	Thread Block	Work Group	Thread Group
Batched Kernel Launch	Grid of Blocks	ND Range	Kernel Invocation

Table 2.1 : Different terminology for thread organization in CUDA and OpenCL.

2.1.2 Execution Model

The execution model of a heterogeneous programming model describes the conceptual model for the execution of user-written computation. Both OpenCL and CUDA are SIMD programming models. The programmer writes a kernel for the device and explicitly indicates it is for device execution using language keywords. Threads executing these kernels use a unique thread ID to select their inputs. Both OpenCL and CUDA use a batched kernel invocation model where large numbers of kernel instances or threads are launched in a single API call. Both CUDA and OpenCL group individual threads into small collections. Kernel invocations launch multiple thread collections at once. Table 2.1 shows the different terminology used by OpenCL and CUDA for these different thread groupings, as well as the terminology used in this thesis when the particular framework being discussed is unimportant. Both CUDA and OpenCL schedule threads in the same thread group onto processing elements in the same compute unit.

One area in which OpenCL's and CUDA's execution model diverge is preparing a user-written kernel for execution on a device. CUDA compiles kernels for execution at compile-time using NVIDIA's compiler. OpenCL programs and OpenCL kernels are compiled or loaded at runtime. An OpenCL program represents a collection of executable functions. An OpenCL kernel object is associated with a program object and specifies a single entry point to that program. This makes OpenCL both a more flexible and explicit programming model than CUDA when it comes to executing computation on different types of devices. On the other hand, every OpenCL program must set up executable objects before executing them on a device whereas CUDA prepares them for the user implicitly.

2.1.3 Memory Model

Both CUDA and OpenCL use discrete address spaces to represent the memory accessible from a device, even in the cases where an OpenCL host application is executing using the same memory as an OpenCL device (as is often the case when multi-core CPUs are accessed through OpenCL). In order for computation executing on a device to have access to input values from the host program, those values must have been previously and explicitly copied to global buffers associated with that device. For the host application to access output values from device computation, those values must be copied out of global device buffers and into the host program's address space. Both CUDA and OpenCL provide API calls for copy-in and copy-out of global memory, as well as ways to use special purpose memory (such as texture memory) which may improve performance for the right access patterns on certain hardware. The CUDA and OpenCL kernel languages also include special keywords for specifying local, scratchpad memory accessible from a kernel. This content of scratchpad memory has the same lifetime as a thread group on a compute unit and exhibits lower latency.

2.2 Programming Abstractions Built on CUDA & OpenCL

In this section, we'll examine higher level programming models and tools which build on top of CUDA and OpenCL. This includes discussion of the current state of heterogeneous libraries, OpenACC, GPU support in Java, and heterogeneous, distributed programming models. All of the techniques presented in this section are used as evidence of the relevance and novelty of the work presented in later chapters.

2.2.1 Heterogeneous Libraries

One of the most common use cases for heterogeneous hardware is through one or more domain-specific libraries which wrap CUDA or OpenCL API calls in higher-level, commonly used operations. NVIDIA supplies a number of mathematical and physical libraries. These libraries provide expert-tuned code that any domain-expert can take advantage of to accelerate their application with GPUs.

The drawback of these domain-specific heterogeneous libraries is obvious: generality. While they are useful when an application can be decomposed into a pipeline of library calls, once an unsupported operation is required the application developer must revert to low-level, hand-coded CUDA. In addition, these libraries are generally optimized for a specific platform, such as NVIDIA GPUs. Highly-optimized libraries may not be sufficiently adaptable to future architecture changes.

2.2.2 OpenACC

The OpenACC standard [13] defines a number of compiler directives and library functions which enable application development for “high-level host+accelerator programs” by providing programmers with an OpenMP-like API for parallelism on accelerators.

The core directive for OpenACC is *#pragma acc kernel*, which indicates to the compiler that the following statements (such as a for loop) can be converted to execute on heterogeneous architectures. The *kernel* pragma includes clauses such as *copyin*, *copyout*, and *copy* to indicate data motion between the host system and an accelerator.

While OpenACC provides a familiar and high-level API to software developers, it leaves a number of unanswered questions. It is still only accessible from low-level programming languages (such as C or Fortran). Its high level directives don’t allow users to exploit some of the low-level optimizations in OpenCL and CUDA which lead to the dramatic acceleration users expect from heterogeneous execution, and yet OpenACC still does little to hide the very strict SIMD programming model these lower-level programming models provide. As a result, it is questionable how well OpenACC will map to novel architectures which may not fit neatly into the “host+accelerator” model that OpenACC targets.

2.2.3 GPU Support in Java

Recently there has been increased interest in adding support for executing parallel code sections in Java on GPUs. In this section we focus on one of the most cited projects, APARAPI [14].

APARAPI is an open-source tool developed at AMD which facilitates execution of Java programs on OpenCL devices. One of APARAPI’s main features is JIT compilation of Java bytecode to OpenCL kernels at runtime. The APARAPI runtime uses the Java Native Interface (JNI) and OpenCL API calls to handle transfers of primitives and arrays of primitives between the Java address space and the OpenCL address space, invocation of OpenCL kernels, and a number of useful auxiliary functions such as device management. For the APARAPI programmer, porting a Java program to use APARAPI is straightforward. APARAPI provides a *Kernel* class which, when extended, executes its *run()* method in parallel on an OpenCL device. This API is very similar to using Java’s *Runnable* class for multi-threaded Java execution. One advantage of APARAPI is that because it is built on OpenCL, it can be used to execute code in native threads on multi-core CPUs, GPUs, FPGAs, or any architecture which supports or will support OpenCL. However, APARAPI only supports a subset of the Java bytecode specification, most notably only allowing references to primitive types or single-dimensional arrays of primitives in kernel code.

An even more recent development than APARAPI is Project Sumatra [15]. In

Project Sumatra the OpenJDK community aims to allow the Java Virtual Machine's JIT compiler to generate GPU ISA code directly. This would allow the JVM to target code which seems suited to GPU offload. While APARAPI requires explicit annotation of GPU kernels in Java code by placing them in a specific class, Sumatra's scope includes a number of parallel operations available in Java 8, such as filter or map.

Both of these projects demonstrate a recent surge of community interest in accelerating Java and other high-level languages using GPUs. They hide OpenCL execution behind a high-level programming language and use an interface similar to what Java programmers are familiar with. However, APARAPI and projects like it also hide much of the low-level features of heterogeneous architectures, preventing optimal execution.

2.2.4 Heterogeneous, Distributed Programming Models

While significant effort has been put forth on programming models and runtimes for heterogeneous systems in a single-node host+accelerator model, supporting software development in distributed heterogeneous systems has received less attention. The most common approaches combine a distributed programming model such as MPI or UPC, a heterogeneous programming model such as CUDA or OpenCL, and a multi-threaded programming model such as OpenMP to fully utilize the hardware available [5][16][17]. Managing this many resources efficiently while partitioning work at several different granularities leads to complex systems which are difficult to maintain or port to multiple platforms. There is usually a disconnect between the work partitioning that is done between nodes versus within nodes. Techniques that use a metric to augment the load given to accelerated nodes suffers from inaccurate methodologies for accurately predicting the rate of progress of different types of hardware on arbitrary kernels.

Recent work [18] [19] has integrated accelerators into MapReduce frameworks, some of which are distributed. MapReduce is a high-level, two-stage programming abstraction. The input to a MapReduce job is a vector of (key,value) pairs. A single instance of the first stage, Map, converts a single input (key,value) pair to zero or more output (key,value) pairs. A single instance of the second stage, Reduce, takes as input a key paired with a list of values, where that list is composed of all output Map values associated with that same key. Reduce then produces zero or more output (key,value) pairs as the final output of a MapReduce job. The data-parallelism of both stages in MapReduce make it an attractive candidate for GPU execution.

In general, application development on distributed and heterogeneous systems for the kind of complex and irregular applications now entering the realm of heterogeneous processing is an open problem.

Chapter 3

JCUDA: Lightweight GPU Programming from Java

The work presented in this chapter, JCUDA, auto-generates bridge code between Java and CUDA which enables the execution of programmer-written CUDA kernels from a Java application. Using JCUDA requires enough CUDA experience to write CUDA kernels. However, by using programmer-written kernels JCUDA makes a larger collection of CUDA optimizations available to the programmer. JCUDA also removes much of the programming burden of calling CUDA from Java, such as writing CUDA memory management and JNI code. Performance results obtained on three double-precision floating-point Java Grande benchmarks show that using JCUDA can deliver significant performance improvements to Java applications. The results for Size C (the largest data size of the Java Grande benchmarks) show speedups ranging from 1.52× to 86.57× with the use of one GPU, relative to multi-threaded Java execution on a 12-core CPU.

3.1 JCUDA Design and Implementation

The JCUDA model is designed to be a programmer-friendly foreign function interface for invoking CUDA kernels from Java code, particularly for programmers who may be familiar with Java and CUDA but not with JNI.

The example in Figure 3.1 illustrates JCUDA syntax and usage. The interface to external CUDA functions is declared in lines 90-95, which includes a static library definition using the `lib` keyword. The two arguments to a `lib` declaration specify the name and location of the external library using string constants. The library definition contains declarations for two external functions, `foo1` and `foo2`. The `acc` modifier indicates that these external functions are CUDA-accelerated kernel functions. Each function argument can be declared as IN, OUT, or INOUT to indicate if a data transfer should be performed before the kernel call, after the kernel call, or both. These modifiers allow the responsibility of device memory allocation and data transfer to be delegated to the JCUDA compiler. However, they are not flexible enough to support device resident memory. The current JCUDA implementation only supports scalar primitives and rectangular arrays of primitives as arguments to CUDA kernels. The OUT and INOUT modifiers are only permitted on arrays of primitives, not on scalar primitives. If no modifier is specified for an argument, it defaults to INOUT.

Lines 16–17 show a sample invocation of the CUDA kernel function `foo1`. Similar to CUDA's C interface, `<<<<...>>>>`[*] is used to identify a kernel call. The geometries for the CUDA grid and blocks are specified using two three-element integer arrays, `BlocksPerGrid` and `ThreadsPerBlock`. In this example, the kernel executes with $16 \times 16 = 256$ threads per block and with a number of blocks per grid that depends on the input data size (`NUM1` and `NUM2`).

The JCUDA compiler performs source-to-source translation of JCUDA programs like the one shown in Figure 3.1 to Java programs. The implementation is built on Polyglot [20], a compiler front end for the Java programming language. The build process for a JCUDA program is shown in Figure 3.2. The initial inputs are JCUDA files like the example shown in Figure 3.1. Polyglot converts JCUDA files to auto-generated Java and C source code which include any JNI, CUDA, or JCUDA runtime API calls. The generated C files are then transparently compiled into a shared object library. At this point, the programmer builds the generated Java files with the option

[*] Four angle brackets are used instead of three as in CUDA syntax because the "`>>>`" is already used as unsigned right shift operator in Java programming language.

```
double[ ][ ] l_a = new double[NUM1][NUM2];
double[ ][ ][ ] l_aout = new double[NUM1][NUM2][NUM3];
double[ ][ ] l_aex = new double[NUM1][NUM2];

initArray(l_a); initArray(l_aex); //initialize value in array

int [ ] ThreadsPerBlock = {16, 16, 1};
int [ ] BlocksPerGrid = new int[3];
BlocksPerGrid[0] = (NUM1 + ThreadsPerBlock[0] - 1) /
    ThreadsPerBlock[0];
BlocksPerGrid[1] = (NUM2 + ThreadsPerBlock[1] - 1) /
    ThreadsPerBlock[1];
BlocksPerGrid[2] = 1;

/* invoke device on this block/thread grid */
cudafoo.foo1<<<<BlocksPerGrid, ThreadsPerBlock>>>>(
    l_a, l_aout, l_aex);
printArray(l_a); printArray(l_aout); printArray(l_aex);

...                           ... ...

static lib cudafoo ("cfoo", "/opt/cudafoo/lib") {
  acc void foo1 (IN double[ ][ ]a, OUT int[ ][ ][ ] aout,
      INOUT float[ ][ ] aex);
  acc void foo2 (IN short[ ][ ]a, INOUT double[ ][ ][ ] aex,
      IN int total);
}
```

Figure 3.1 : JCUDA example

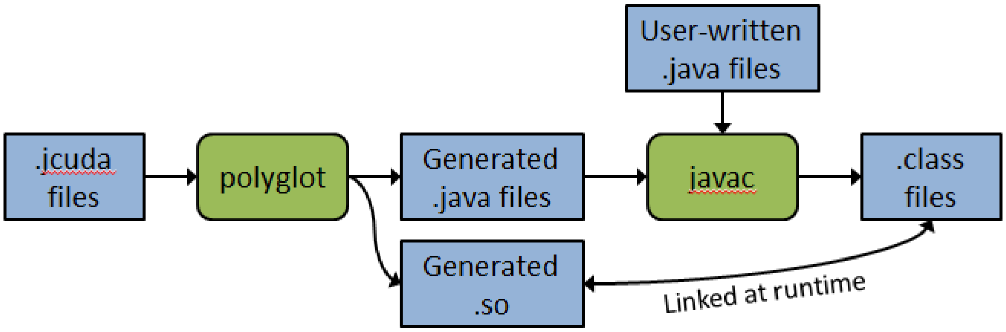


Figure 3.2 : Build process for a JCUDA program

of including user-written Java files in the build process. User-written Java can call JCUDA methods. The final outputs of the JCUDA build process are a collection of Java CLASS files which are linked with the generated library at runtime.

Figures 3.3 and 3.4 show the Java static class declaration and the C glue code generated from the `lib` declaration in Figure 3.1. The Java static class introduces declarations with mangled names for native functions corresponding to JCUDA functions `foo1` and `foo2` respectively, as well as a static class initializer to load the stub library. In addition, three parameters are added to each call — *dimGrid*, *dimBlock*, and *sizeShared* — corresponding to the CUDA grid geometry, block geometry, and shared memory size. Figure 3.4 shows the generated C code with host-device transfers in accordance with the IN, OUT and INOUT modifiers in Figure 3.1. Note the use of JCUDA library functions `copyArrayJVMToDevice`, `copyArrayDeviceToJVM`, `allocArrayOnDevice`, and `freeDeviceMem`. These functions wrap `cudaMemcpy`, `cudaMalloc`, and `cudaFree`. `copyArrayJVMToDevice` and `copyArrayDeviceToJVM` are useful because they take multi-dimensional Java arrays as input rather than C pointers, and can infer the dimensionality and size of the input or output array using JNI.

JCUDA uses the `copyArrayJVMToDevice` and `copyArrayDeviceToJVM` functions

```
private static class cudafoo {

  native static void HelloL_00024cudafoo_foo1(double[ ][ ] a,
      int[ ][ ][ ] aout, float[ ][ ] aex, int[ ] dimGrid,
      int[ ] dimBlock, int sizeShared);

  static void foo1(double[ ][ ] a, int[ ][ ][ ] aout,
      float[ ][ ] aex, int[ ] dimGrid, int[ ] dimBlock,
      int sizeShared) {
    HelloL_00024cudafoo_foo1(a, aout, aex, dimGrid,
      dimBlock, sizeShared);
  }

  native static void HelloL_00024cudafoo_foo2(short[ ][ ] a,
      double[ ][ ][ ] aex, int total, int[ ] dimGrid,
      int[ ] dimBlock, int sizeShared);

  static void foo2(short[ ][ ] a, double[ ][ ][ ] aex, int total,
      int[ ] dimGrid, int[ ] dimBlock, int sizeShared) {
    HelloL_00024cudafoo_foo2(a, aex, total, dimGrid, dimBlock,
  sizeShared);
  }

  static{java.lang.System.loadLibrary("HelloL_00024cudafoo_stub");}
}
```

Figure 3.3 : Java static class declaration generated from `lib` definition in Figure 3.1

```
extern __global__ void foo1(double * d_a, signed int * d_aout,
    float * d_aex);

JNIEXPORT void JNICALL
Java_HelloL_00024cudafoo_HelloL_100024cudafoo_1foo1(JNIEnv *env,
    jclass cls, jobjectArray a, jobjectArray aout, jobjectArray aex,
jintArray dimGrid, jintArray dimBlock, int sizeShared) {
  /* copy array a to the device */
  int dim_a[3] = {2};
  double * d_a = (double*) copyArrayJVMToDevice(env, a, dim_a,
      sizeof(double));

  /* Allocate array aout on the device */
  int dim_aout[4] = {3};
  signed int * d_aout = (signed int*) allocArrayOnDevice(env, aout,
      dim_aout, sizeof(signed int));

  /* copy array aex to the device */
  int dim_aex[3] = {2};
  float * d_aex = (float*) copyArrayJVMToDevice(env, aex, dim_aex,
      sizeof(float));

  /* Initialize the dimension of grid and block in CUDA call */
  dim3 d_dimGrid; getCUDADim3(env, dimGrid, &d_dimGrid);
  dim3 d_dimBlock; getCUDADim3(env, dimBlock, &d_dimBlock);

  foo1<<<d_dimGrid, d_dimBlock, sizeShared>>> ((double *)d_a,
      (signed int *)d_aout, (float *)d_aex);

  freeDeviceMem(d_a);

  /* copy array d_aout-> aout from device to JVM */
  copyArrayDeviceToJVM(env, d_aout, aout, dim_aout,
      sizeof(signed int));
  freeDeviceMem(d_aout);

  /* copy array d_aex-> aex from device to JVM */
  copyArrayDeviceToJVM(env, d_aex, aex, dim_aex, sizeof(float));
  freeDeviceMem(d_aex);
}
```

Figure 3.4 : C glue code generated for the `foo1` function defined in Figure 3.1

to support multidimensional Java arrays. A multidimensional array in Java is represented as an array of arrays. For example, a two-dimensional float array is represented as a one-dimensional array of objects, each of which references a one-dimensional float array. This representation supports general nested and ragged arrays, as well as the ability to pass subarrays as parameters while still preserving pointer safety. However, it has been observed that this generality comes with a large overhead for the common case of multidimensional rectangular arrays [21].

This work focuses on the special case of dense rectangular two and three dimensional arrays of primitive types as in C and Fortran. These arrays are allocated as nested arrays in Java and as contiguous arrays in CUDA. The JCUDA runtime performs the necessary gather and scatter operations when copying array data between the JVM and the GPU device by iterating through the nested Java arrays using JNI and transferring them each to an offset in a pre-allocated device buffer. For example, to copy a Java array of `double[20][40][80]` from the JVM to the GPU device, the JCUDA runtime makes $20 \times 40 = 800$ calls to the CUDA *cudaMemcpy* memory copy function with 80 double-words transferred in each call.

3.2 JCUDA Performance Evaluation

3.2.1 Experimental Setup

Three benchmarks from the Java Grande Forum (JGF) Benchmarks [22, 23] were used to evaluate the JCUDA programming interface and compiler — Fourier coefficient analysis (Series), Sparse matrix multiplication (Sparse), and IDEA encryption (Crypt). Each of these benchmarks has three problem sizes for evaluation — A, B and C — with Size A being the smallest and Size C the largest. For each of these benchmarks, the compute-intensive portions were rewritten in CUDA while the rest of the code was retained in its original Java form, except for the JCUDA extensions used for kernel invocation. The rewritten CUDA codes are parallelized in the same way as the original Java multi-threaded code.

The GPU used in these performance evaluations was a NVIDIA Tesla M2050, containing a GPU with 448 thread processors, a 1.15 GHz clock speed, and 2687MB of global memory. All benchmarks were evaluated with double-precision arithmetic, as in the original Java versions. The host of this GPU consists of a pair of 6-core Intel CPUs with a 2.80GHz clock speed and 48GB of RAM. The software used include a Sun Java HotSpot 64-bit virtual machine included in version 1.6.0_25 of the Java SE Development Kit (JDK), version 4.4.6 of the GNU Compiler Collection (gcc), version 304.54 of the NVIDIA CUDA driver, and version 5.0 of the NVIDIA CUDA Toolkit.

3.2.2 Evaluation and Analysis

Table 3.1 shows the execution time of the multi-threaded Java and JCUDA versions for all three data sizes of each of the three benchmarks. The Java execution times were obtained using 12 threads on a 12 core CPU. The JCUDA execution times were obtained by using a single Java thread on the CPU combined with multiple threads on the GPU. The JCUDA and Java kernels were repeated 5 times, and the median time for each benchmark and test size is listed in the table.

The final column shows the speedup of the JCUDA version relative to the 12-threaded Java version. The results for Size C (the largest data size) show speedups ranging from 1.52× to 86.57×, whereas smaller speedups or slowdowns were obtained for smaller data sizes. The benchmark that showed the smallest speedup was Crypt. This is primarily due to poor memory access coherency in the device kernel and high CUDA initialization and cleanup overheads. In contrast, the Series examples show the largest speedup because it is an embarrassingly parallel application with a high ratio of computation to communication.

Benchmark	Java Kernel Time (s)	JCUDA Kernel Time (s)	Speedup
Sparse-A	0.015	0.004	3.75×
Sparse-B	0.017	0.006	2.83×
Sparse-C	0.071	0.023	3.09×
Crypt-A	0.017	0.010	1.70×
Crypt-B	0.097	0.064	1.52×
Crypt-C	0.715	0.467	1.53×
Series-A	0.599	0.017	35.24×
Series-B	6.018	0.147	40.94×
Series-C	124.751	1.441	86.57×

Table 3.1 : Execution times in seconds, and Speedup of JCUDA relative to 12-thread Java executions

Figure 3.5 shows the percentage breakdown of JCUDA kernel execution times into *computation*, *setup*, and *cleanup* components. The computation component is the time spent in kernel execution on the GPU. The setup and cleanup components represent the time spent in data transfer before and after kernel execution. The figure shows that Series achieved the highest speedups due to large amounts of computation and little communications. Crypt and Sparse both demonstrated more moderate speedups from larger overheads due to data transfer.

3.3 Discussion

JCUDA can be used by Java programmers to invoke CUDA kernels by auto-generating JNI and CUDA bridge code at compile-time for programmer-written kernels operating on flat and multidimensional arrays. The JCUDA runtime handles data transfer to and from Java arrays. The performance results obtained on three Java Grande benchmarks show significant performance improvements of the JCUDA-accelerated versions over their original versions. In summary, JCUDA makes accelerating existing Java programs with many-core GPUs simple for programmers already familiar with writing CUDA kernels, while allowing many of the low-level optimizations required

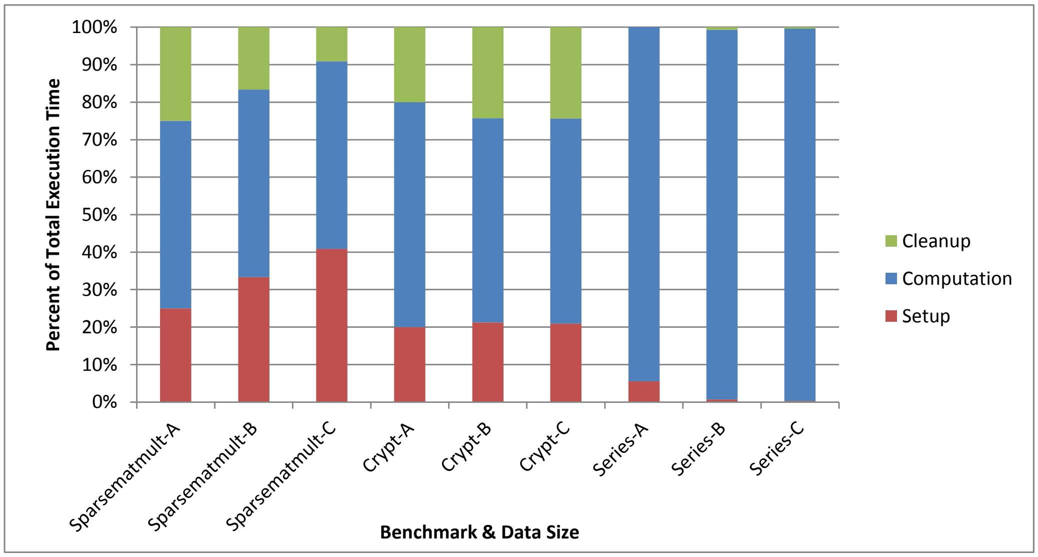


Figure 3.5 : Percentage breakdown of JCUDA execution between CUDA setup, computation, and cleanup

for efficient execution in CUDA that other high-level programming models hide.

Chapter 4

CnC-CUDA: Integrating CPU and GPU Computation Into a Dataflow Language

While work on JCUDA made executing CUDA kernels on graphics hardware from Java applications a far simpler and natural process than was previously possible, it did little to add abstractions to the CUDA programming model for developers with little experience in CUDA.

CnC-CUDA extends Intel's Concurrent Collections (CnC) programming model to address the challenge of efficiently executing workloads on a multi-architecture (hybrid) platform by scheduling execution on CPUs and GPUs. CnC is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps which are related by data and control dependence edges represented by a CnC graph. CnC-CUDA extends previous work on CnC-HJ, a multi-threaded CPU-only CnC implementation built on top of the parallel Habanero-Java (HJ) [24] runtime. The CnC-CUDA extensions to CnC-HJ include the definition of multi-threaded steps for execution on GPUs, as well as the automatic generation of data and control flow between CPU steps and GPU steps. By building support for heterogeneous processors into CnC-CUDA, higher-level abstractions can be provided to the user of heterogeneous platforms than in JCUDA or other existing programming models. In addition, because of the dataflow nature of CnC and hints it is able to provide at runtime, CnC-CUDA can take advantage of a wider range of CUDA optimizations. Experimental results show that this approach can yield significant performance benefits with both

GPU execution and hybrid CPU/GPU execution.

4.1 Background on the CnC Model

This section provides a brief summary of the CnC model, as described in [25]. A CnC program is a graph of sequential kernels, communicating with one another. The three main constructs in CnC are *step collections*, *data item collections*, and *control tag collections*. Statically, each of these constructs is a *collection* representing a set of dynamic *instances*. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization and communication. Control tag instances are the unit of control. Figure 4.1 shows the relationships between these collections.

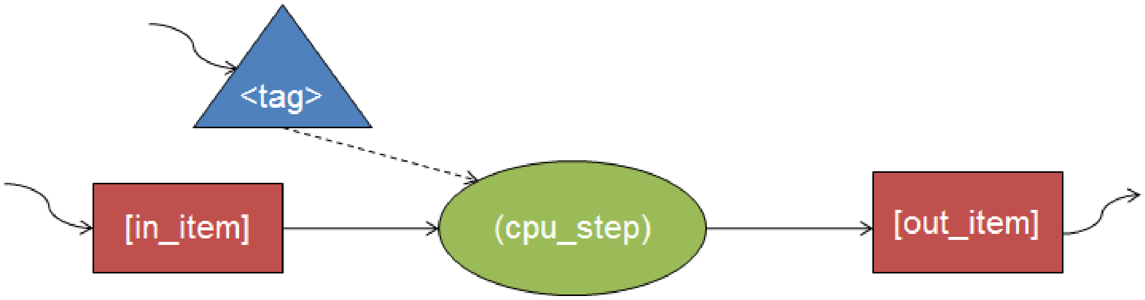


Figure 4.1 : Relationship between CnC tags, items, and computation steps

Control, data, and step instances are all identified by a unique *tag* within each collection. In CnC, tags are arbitrary values that support an equality test and hash function. Each type of collection uses tags as follows:

- Putting a tag into a control collection causes the corresponding steps (in the prescribed step collections) to eventually execute.
- Each step instance is a computation that takes a single tag (originating from the prescribing control collection) as an argument.

- A data collection is an associative container indexed by tags. The entry for a tag i, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection is necessary for determinism.

A CnC program is represented as a graph, stored in a textual graph file. Within the graph file, the CnC graph is represented using `()` to denote computation steps, `[]` to denote data items, and `<>` to denote control tags. The edges in the graph specify the partial ordering constraints required by the semantics. One type of ordering constraint arises from a *data dependence*. This relationship occurs when an instance of a step, produces an instance of an item which is later consumed by an instance of another step. Clearly the producing step instance must occur before the consuming step instance. Another type of ordering constraint arises from a *control dependence*, where one computation step instance determines if another computation step instance will execute. In that case, the controller step puts a control tag in a tag collection, which in turn *prescribes* the controllee step. The execution order of step instances is constrained only by their dynamic data and control dependences.

As an example, consider the CnC graph file in Figure 4.2. This graph file describes a cryptographic benchmark (Crypt) from the JGF benchmark suite which first encrypts and then decrypts a string of characters. Included in the graph file are four item collections, two computation steps, and two tag collections. `[plain]` is the original, unencrypted input. `[z]` and `[dk]` are the keys used for encryption and decryption, respectively. `crypt` is the encrypted string. `plain2` is the final decrypted output, which is identical to the original input in `[plain]`. The computation steps for encrypting and decrypting data are represented by `(encrypt)` and `(decrypt)`. The creation of instances of these computation steps is controlled by puts into the `<crypt_tag>` and `<decrypt_tag>` tag collections. The control and data dependencies of Crypt are expressed on lines 14 and 15 in Figure 4.2. Also note the special

```
# Crypt CnC Graph File

env -> [plain];
env -> [z];
env -> [dk];

[plain2] -> env;

env -> <crypt_tag>;

<crypt_tag>::(encrypt);
<decrypt_tag>::(decrypt);

[plain],[z] -> (encrypt) -> [crypt],<decrypt_tag>;
[crypt],[dk] -> (decrypt) -> [plain2];
```

Figure 4.2 : An example CnC graph file for the Crypt benchmark

`env` variable. `env` is used to indicate values that are either produced or consumed by the host application. Since all items and tags can be passed as generic Objects in CnC-HJ, the CnC graph file in Figure 4.2 contains no type information. Type information is necessary for CnC-CUDA extensions because of static, compile-time typing in CUDA.

In addition to the CnC graph file, a CnC programmer must also define an entry point to the application and the implementation for each computational kernel. In CnC-HJ, the entry point is provided by a Java class that defines `public static void main`, within which the initial Puts and final Gets are done on item and tag collections (as described by the use of `env` in the graph file). The computation step implementations are also provided by Java classes. Each computation step definition must implement a `compute` function which defines the logic performed on a single CnC tag and its associated items. Abridged examples of a CnC entry point and computation step for Crypt are shown in Figures 4.3 and 4.4. Note the use of Put on item and tag collections in lines 14–16 and 21, and the use of Get on the `plain2` item collection in line 26. In Figure 4.3, the notation `[i]` declares an HJ range. An HJ range can represent a scalar or multidimensional range of values. `[i]` represents a range including only one value, the value of `i`. `[0:N-1]` would represent a one-dimensional range of values including all integers from 0 to N-1, inclusive.

4.2 CnC-CUDA Design and Implementation

CnC-CUDA includes several extensions to the CnC programming model to support CUDA steps.

4.2.1 Graph File

Some of the features added require new syntax in the graph file. First, a new syntax is introduced for CUDA steps. CUDA steps are declared with braces, {}, instead of the parentheses, (), used for CPU steps.

Second, support for type specification in the CnC graph file is added for item and tag collections. This is necessary for auto-generation of CUDA and JNI API calls at compile-time.

Third, support was added for the definition of constants in the graph file using the following notation:

|*const_name const_value*|;

where *const_value* is an integer. This definition generates constant values that can be referenced from both HJ and CUDA. These constants are used for specifying the exact size of item and tags that are passed to CUDA, ensuring the copying of the correct amount of data from Java arrays onto the CUDA device. For example, if each CUDA thread executing a CnC step takes 1000 integers, this item collection would be declared as:

```
public class CryptMain {
  public static void main(String[] args) {
    EncryptStep encrypt = new EncryptStep();
    DecryptStep decrypt = new DecryptStep();
    cryptGraph graph = cryptGraph.Factory(encrypt,
        decrypt);

    byte[] plain = ...;
    int[] z = ...;
     int[] dk = ...;
    byte[] plain2 = new byte[N];

    for(int i = 0; i < N; i++) {
      graph.plain.Put([i], plain);
      graph.z.Put([i], z);
      graph.dk.Put([i], dk);
    }

    finish {
      for(int i = 0; i < N; i++) {
        graph.crypt_tag.Put([i]);
      }
    }

    for(int i = 0; i < N; i++) {
      plain2[i] = graph.plain2.get([i]);
    }
    validate(plain, plain2);
  }
}
```

Figure 4.3 : Example of a CnC entry point for the Crypt application

```
public class EncryptStep {
  CnCReturnValue compute(point tag, InputCollection plain,
      InputCollection z, OutputCollection crypt) {
    byte[] plain = plain.Get(tag);
    int[] z = z.Get(tag);
    ...
    crypt.Put(tag, finalOutput);
    return CnCReturnValue.Success;
  }
}
```

Figure 4.4 : Example of a computation step definition in the Crypt application

$[int\ items[1000]]$;, or $|size\ 1000|$; $[int\ items[size]]$;

These extensions to the CnC graph file are illustrated in a modified and more verbose Crypt graph file in Figure 4.5. Note the declaration of type and size for the item and tag collections in lines 2–8, and the use of {} on lines 23 and 24 to indicate that encrypt and decrypt are now CUDA computation steps.

4.2.2 Item Collections

To enforce strict typing of items and tags, HJ definitions for all item collections are automatically generated by the CnC Parser at compile time. Item collections retain Put and Get methods for adding and retrieving individual items. In the CnC-CUDA implementation, each item is put into a ConcurrentHashMap. Once the number of buffered tags reaches a threshold, the items corresponding to those tags are collected from the ConcurrentHashMap, converted to a native format (i.e., java.lang.Integer→int) and passed to CUDA. While this approach is correct, the individual accumulation and conversion of potentially large numbers of tags and items is inefficient.

In CnC-CUDA, PutRegion/GetRegion primitives allow the programmer to put

```
|crypt_chunk 2000|;
<int crypt_tag>;
<int decrypt_tag>;
[byte plain[crypt_chunk]];
[byte plain2[crypt_chunk]];
[byte crypt[crypt_chunk]];
[int dk[52]];
[int z[52]];

env -> [plain];
env -> [dk];
env -> [array_rows];
env -> [z];
env -> [ele_per_thread];

[plain2] -> env;

env -> <crypt_tag>;

<crypt_tag>::{encrypt};
<decrypt_tag>::{decrypt};

[plain],[z] -> {encrypt} -> [crypt],<decrypt_tag>;
[crypt],[dk] -> {decrypt} -> [plain2];
```

Figure 4.5 : An example CnC-CUDA graph file for the Crypt benchmark

a potentially multidimensional array of items in a single API call. This approach eliminates putting and then extracting individual items, and the array can be directly passed to the kernel. Currently only arrays of primitive types are supported (int[], float[], etc.).

4.2.3 Tag Collections

Tag collections are automatically generated using type definitions in the graph file. The preliminary implementation only supports integer tags. This can be easily extended to any hashable type, as in traditional CnC.

As described in Section 4.1, tag collections control the execution and synchronization of computation steps. New techniques must be used to support synchronization between CUDA computation steps using tag collections. First, a pthread mutex is used to indicate that the device is currently in use by a computation step and inaccessible by any new computation steps for GPU architectures which do not support concurrent kernel execution. Second, the number of CUDA computation steps that can be prescribed by another CUDA computation step is limited to 1, simplifying the complexity of synchronizing CUDA computation steps. If a CUDA step prescribes another CUDA step (as determined by analyzing the CnC graph file), the second step is invoked immediately following the first without returning to HJ. No limitation is placed on a CUDA step prescribing multiple HJ steps.

Like item collections, tag collections also implement the PutRegion operation. PutRegion immediately launches a CUDA kernel for all tags in the range once the required items are available. On individual tag Puts, the tag collection waits for a threshold number of tags to be put, and then launches a CUDA kernel with those tags and their associated items. This threshold has been empirically set to 8192 tags. Once all tags have been put into a GPU tag collection, the programmer has to issue a call to that tag collection's *Wait()* function to be certain that all CUDA threads have completed and their results have been returned to host memory.

The conceptual changes to the CnC graph dependencies caused by these changes to tag and item collections are illustrated in Figure 4.6. An example of the modifications required for the the CnC entry point from Figure 4.3 is shown in Figure 4.7.

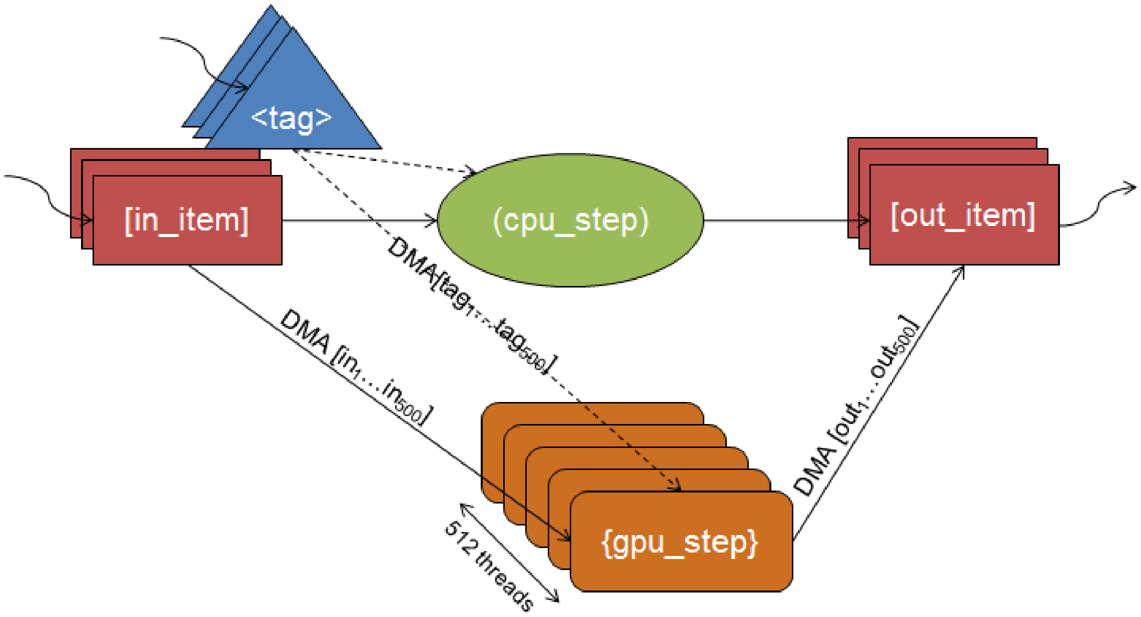

<tag>
[in_item]
(cpu_step)
[out_item]
DMA[tag_1...tag_{500}]
DMA [in_1...in_{500}]
DMA [out_1...out_{500}]
{gpu_step}
512 threads

Figure 4.6 : Relationship between CnC-CUDA tags, items, and computation steps with the added PutRegion/GetRegion primitives

For more advanced CUDA programmers the option of defining a two dimensional tag is provided:

```
<int tag:two_region>
```

This offers the opportunity of placing a tag with 2 regions on the graph. Those two regions are interpreted as number of threads per thread group and number of groups per kernel invocation. In addition, the desired number of threads per thread group can be specified by compiling the graph file with the flag: *-t <number of threads>*.

```
public class CryptMain {
  public static void main(String[] args) {
    EncryptStep encrypt = new EncryptStep();
    DecryptStep decrypt = new DecryptStep();
    cryptGraph graph = cryptGraph.Factory();

    byte[] plain = ...;
    int[] z = ...;
    int[] dk = ...;

    graph.plain.PutRegion([0:N-1], plain);
    graph.z.PutRegion([0:N-1], z);
    graph.dk.PutRegion([0:N-1, dk);

    finish {
      graph.crypt_tag.PutRegion([0:N-1]);
    }

    byte[] plain2 = graph.plain2.GetRegion([0:N-1])
    validate(plain, plain2);
  }
}
```

Figure 4.7 : Example of a CnC-CUDA entry point for the Crypt application

A novel item collection property One-For-All (OFA) is also supported, which passes the same data to each thread on a device, where each thread is prescribed by a different tag. This property follows the format:

```
[type item:ofa]
```

where type can be any supported data type. This can result in both considerable saving in device and host memory as well as better performance from less data transfer overhead.

4.2.4 CUDA Kernel

A CnC-CUDA programmer is responsible for providing CUDA kernels which implement the functionality of CUDA computation steps. The CnC-CUDA compiler is responsible for generating stub codes that allocate memory and copy data to the device before a step is executed. The auto-generated stub codes also free device memory after computation steps complete. The actual programmer-written step code is defined as a CUDA __device__ function by the CnC programmer. The step function is called from an auto-generated CUDA kernel which serves as the CUDA entry point and protects against threads entering the programmer-defined kernel which do not have input. Threads which lack input are created when the number of CnC tags put is not evenly divisible by the selected thread group size. CUDA computation steps are limited to putting a single item on each output item collection. This limitation is a result of CUDA lacking dynamic memory allocation. An abridged example of a programmer-written CUDA computation step for the Crypt benchmark is shown in Figure 4.8.

```
__device__ void encryptKernel(int tag, plainCollection plainC,
    zCollection keyC, cryptCollection cryptC,
    decrypt_tagCollection decrypt_tagC) {
  Byte *plain = Get(plainC);
  int *key = Get(keyC);
  Byte crypt[CRYPT_SIZE];

  ...

  Put(cryptC, crypt);
  Put(decrypt_tagC,tag);
}
```

Figure 4.8 : A code snippet from a CUDA computation step in the Crypt benchmark

4.2.5 Implementation Details

The CnC-CUDA Polyglot-based compiler builds a library of runtime and programmer-written CUDA and C procedures which communicate data and execute kernels. In CnC-CUDA, the complexity of inter-language function calls and device memory management is hidden from the CnC programmer and auto-generated at compile time, allowing them to focus on developing and implementing the algorithms in their application.

There are two CUDA files generated from compile-time analysis of the CnC-CUDA graph file: `cudaLaunchers.cu` and `kernelCallers.cu`. `cudaLaunchers.cu` provides the interface between CnC-HJ JVM execution and CUDA computation steps. Pseudocode for the code auto-generated in `cudaLaunchers.cu` by the CnC-CUDA compiler for an arbitrary computation step is shown in Algorithm 1. All of the information needed to generate `cudaLaunchers.cu` is contained in the CnC-CUDA graph file. Page-locked CPU memory is used to permit asynchronous memory copies. Note that the functionality for transferring data between JVM¡-¿page-locked memory and page-locked memory¡-¿CUDA is supported by runtime methods similar to the

```
__global__ void encryptKernelCaller(int *crypt_tag,
  int num_crypt_tag, plainCollection *plain,
  zCollection *z, cryptCollection *crypt,
  decrypt_tagCollection *decrypt_tag) {
 int tid = blockDim.x * blockIdx.x + threadIdx.x;

 if(tid < num_crypt_tag) {
  encryptKernel(crypt_tag[tid], *plain, *z, *crypt,
    *decrypt_tag);
 }
}
```

Figure 4.9 : An example computation step `__global__` function in `kernelCallers.cu`

`copyArrayJVMToDevice` and `copyArrayDeviceToJVM` functions used in CnC-CUDA.

`kernelCallers.cu` contains a CUDA `__global__` function for each computation step. These `__global__` functions pass control from the kernel launch in `cudaLaunchers.cu` to the programmer-written CUDA kernels. The only purpose of `kernelCallers.cu` is to ensure CUDA threads with no input are blocked from entering the computation step. An example `kernelCallers.cu` is shown in Figure 4.9.

4.3 CnC-CUDA Performance Evaluation

To compare the performance of CnC-CUDA to CnC-HJ and other programming models and languages three benchmarks were used: Fourier coefficient analysis (Series), IDEA encryption (Crypt), and Heart Wall Tracking. Series and Crypt come from the the Java Grande Forum (JGF) benchmark suite [26]. Heart Wall Tracking comes from the Rodinia benchmark suite [27]. Each benchmark was run on varying data sizes using some combination of CnC-CUDA, CnC-HJ, hand-coded CUDA, and either single-threaded Java (from JGF) or multi-threaded OpenMP (from Rodinia). Additionally, the Crypt benchmark was run in CnC-CUDA using CPU and GPU

```
totalMemReq = $CalcMemoryFootprintForAllCollections()$
pageLockedHostBuffer = $PageLockedAlloc(totalMemReq)$
// Preparation
for collection in input items/tags do
    Copy collection from JVM heap to offset in pageLockedHostBuffer
    deviceBuffer$_i$ = $DeviceAlloc(sizeof(collection))$
end
for collection in output items/tags do
    Preallocate device memory for outputs
end
// Host → Device
for collection in input items/tags do
    Asynchronously copy from pageLockedHostBuffer → deviceBuffer$_i$
end
// Execute
Launch Asynchronous Kernel
// Device → Host
for collection in output items/tags do
    Asynchronously copy from deviceBuffer$_i$ → pageLockedHostBuffer
end
Wait For Queued Work to Finish
// Host → JVM
for collection in output items/tags do
    Copy collection from pageLockedHostBuffer into JVM
end
// Cleanup
for collection in all items/tags do
    if Not Used In Future Computation Steps then
        $Free(collection)$
    end
end
```

Algorithm 1: Pseudocode for the code generated for an arbitrary kernel in `cudaLaunchers.cu`

Crypt	GPU Performance (s)		CPU Performance (s)		
Data Size (bytes)	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Java	Speedup
50,000,000 (JGF Size C)	0.886	0.161	2.067	2.92	2.33
75,000,000	1.208	0.253	3.239	4.387	2.68
100,000,000	1.488	0.341	4.460	5.818	3.00
150,000,000	2.311	0.550	6.903	8.716	2.99

Table 4.1 : Execution times in seconds of JGF Crypt benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ.

Series	GPU Performance (s)		CPU Performance (s)		
Data Size	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Java	Speedup
10,000	0.332	0.0095	3.587	6.777	10.80
100,000	0.441	0.116	36.588	69.074	60.78
1,000,000	1.411	1.279	572.86	N/A	406.00

Table 4.2 : Execution times in seconds of JGF Series benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ.

computation steps in parallel. The timing of each benchmark was started just before the first set of tag puts were performed to launch the CnC graph or the function call to launch the actual computation for non-CnC benchmarks. Timing was stopped when all CnC steps completed or the core function completed. For GPU execution, these timings included the overhead of copying data to and from device memory.

For these evaluations, an NVIDIA GTX 480 GPU was used. The GTX 480 has 480 processing cores and 1.6 GB of memory. The CPU host of this GPU is 4 AMD Phenom 9850 Quad-Core Processors with a 1.25 GHz clock, 512 KB cache, and 8 GB of memory. The software used includes a Java HotSpot 64-bit virtual machine from version 1.6.0_20 of the Java Development Kit (JDK), a GNU C compiler v. 4.1.2, and version 3.1 of the NVIDIA CUDA Toolkit.

Heart Wall	GPU Performance (s)		CPU Performance (s)		
Data Size (# of frames)	CnC-CUDA	CUDA	CnC-HJ (16 cores)	OpenMP (16 cores)	Speedup
1	0.427	0.985	0.246	0.005	0.57
104	4.4842	3.6133	11.058	13.863	2.47

Table 4.3 : Execution times in seconds on GPU - CnC-CUDA and hand-coded (Rodinia) CUDA versions - and CPU - Serial, OpenMP on 16 cores and CnC-HJ on 16 cores - of the Heart Wall Tracking benchmark.

Crypt	Hybrid Performance (s)			
Percent of Load on GPU	Average	Slowest	Fastest	Speedup (Relative to CnC-CUDA)
10%	3.042	3.806	2.493	0.76
20%	3.066	3.765	2.727	0.75
30%	2.720	3.048	2.223	0.85
40%	2.289	2.750	1.878	1.01
50%	2.139	2.397	1.973	1.08
60%	2.035	2.242	1.538	1.14
70%	2.076	2.799	1.755	1.11
80%	2.189	2.511	1.883	1.06
90%	2.143	2.344	1.968	1.08

Table 4.4 : Execution times in seconds and Speedup of a hybrid CnC-CUDA/HJ version of Crypt against only CnC-CUDA.

Tables 4.1, 4.2, and 4.3 show the average execution time across ten runs of the respective benchmarks in different programming models or languages. In the CnC versions (HJ or CUDA), the CnC Parser was used to auto-generate all glue code. The programmer only provided the CnC step code in CUDA or HJ and the code for launching the CnC graph in a host HJ application. The CnC-CUDA measurements on the GPU were compared to CnC-HJ runs of the same benchmarks on the CPU and the results listed in the Speedup column of each table. Each CUDA kernel launch was performed with a constant 256 threads per thread group. The thread groups per kernel invocation were determined by the iteration size. Each CnC-HJ execution was performed using 16 worker threads. (Over-provisioning the number of worker threads per CPU degraded performance for the benchmarks and hardware studied in this paper.)

No special CUDA memory (e.g., texture, shared, constant) was used in the execution of these benchmarks.

These results demonstrate that the performance of many-core GPUs can be made accessible to non-expert programmers through CnC-CUDA. Without any knowledge of CUDA's memory or threading model, a CnC-CUDA programmer can go from working with CnC-HJ to exploiting the computational power of a GPU using CnC-CUDA quickly and easily, achieving an order of magnitude improved performance relative to a quad-core CPU.

The speedup of CUDA over HJ increases as the size of the data set increases, with the maximum average speedup (406.00×) observed for the embarrassingly parallel Series benchmark at its largest data size. While not observed in these results, it is of course possible for a GPU version of an application to run slower than a CPU version, when the relative overheads of host-device data transfers, CUDA initialization, control flow divergence, or other GPU-specific overheads lead to performance degradation on the GPU.

The results in Table 4.4 shows the potential for performance improvement using hybrid CPU-GPU execution in the CnC model. For consistency, the hybrid CUDA/HJ tests were performed using 256 threads per block for GPU execution. These results were obtained by evaluating different load distributions between the CPU and GPU for the Crypt benchmark with its largest size, for which the average speedup of the GPU over the CPU in Table 4.1 was 1.82×. In Table 4.4, there is an additional 1.14× speedup can be obtained over the pure CnC-CUDA version by a hybrid execution in which 60% of the load is placed on the GPU and 40% of the load on the CPU. An interesting topic for future research is to extend the CnC runtime to perform this load distribution adaptively and automatically, allowing for a single CnC-CUDA graph to dynamically and efficiently handle a wide range of data set sizes.

Finally, Table 4.3 shows the execution times in seconds for the CnC-CUDA and hand-coded CUDA versions of the Heart Wall Tracking benchmark (the hand-coded version was obtained from the Rodinia benchmark set [27]). Both versions have comparable performance when processing a single frame, but the CnC-CUDA version is 1.25× slower than the hand-coded version for 104 frames. This reflects the extra coordination overhead in CnC involved in synchronizing the computation across frames.

4.4 CnC-CUDA Discussion

CnC-CUDA extends past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge. The CnC-CUDA extensions in this paper include the definition of SIMD computation steps for execution on GPUs and automatic generation of data and control flow between CPU steps and GPU steps. The details of creating and managing parallel tasks and data transfers are handled by the CnC-CUDA framework. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU-GPU execution.

Compared to the current state-of-the-art in heterogeneous programming models, CnC-CUDA provides an entirely novel, high-level, dataflow programming model. Most if not all existing heterogeneous programming models still present procedural or SIMD programming models to the user. CnC-CUDA differentiates itself by instead requiring the user write computational kernels for a single input CnC tag and automatically mapping those kernels to GPUs. The CnC-CUDA runtime also automatically takes advantage of CUDA optimizations such as asynchronous transfers and kernels, while allowing the programmer to optimize their own kernels with CUDA-specific features like scratchpad memory.

Chapter 5

Enabling Dynamic Task Parallelism on GPUs

A majority of the previous work in heterogeneous runtimes and programming models has viewed accelerators as atomic entities to be managed from the host. The work discussed in this chapter branches out from that, and instead utilizes the GPU as a collection of SIMD cores to achieve higher performance on a wider range of applications.

While JCUDA and CnC-CUDA both 1) provide higher-level abstractions for GPU programming by integrating GPU computation into existing programming models and 2) eliminate many of the nitty gritty details of software development on graphics hardware, neither made any significant modifications to how the application was scheduled on hardware. This work focuses on the design and implementation of a Dynamic task-parallel, load-balancing GPU runtime (DyGR) which provides a number of useful features for GPU kernels. This runtime enables the dynamic creation of new tasks on the GPU, balances those tasks across SMs, reduces data transfer overhead by overlapping it with kernel execution, and manages multiple GPUs. In addition, the infrastructure built on the GPU provides for a simpler API supporting asynchronous task creation and data-driven execution.

5.1 Design of DyGR

Both dynamic task creation and parallel task synchronization can be addressed by the *finish-async* style of programming [28]. In this model, an *async* creates or spawns a task that can potentially execute in parallel with the continuation of the spawning task. The *finish* provides a scope for all spawned tasks, both direct and nested, to complete before execution of the continuation of a finish. The *finish* and *async* constructs provide a natural way for programmers to express task parallelism using dynamic task creation and synchronization.

DyGR is a runtime system on the GPU that provides a task-based backend for abstractions such as finish and async. The goal of the *finish-async* model on the GPU is to be faster than the current divergent execution model for irregular applications without sacrificing the performance of regular computations. This runtime provides users with a simpler programming model for task parallelism, while handling the problem of load balancing inherent to all systems supporting dynamic task creation.

5.2 Implementation of the GPU Load Balancing System

The goals of DyGR are to balance work across the threads of a GPU with lower overhead, improved performance, and less programmer effort than a hand written application. To achieve this goal, a hybrid task distribution model is used which uses both work-stealing and work-sharing queues to provide load balancing between SMs on a CUDA device, and across different devices.

The runtime starts by launching N groups of CUDA threads on each CUDA device available, where N is the number of SMs on that device. Conceptually, these thread groups are treated as worker groups, analogous to worker threads in CPU work-stealing runtimes. Each of these worker groups executes the runtime kernel and maintains its own work-stealing deque, as shown in Figure 5.1. A worker can also steal from other workers' deques that reside on the same device. A separate FIFO shared queue is maintained per device to place tasks from the host onto the device. Only the host can push tasks onto this queue while the workers on the device compete to pop these tasks. In the current implementation, tasks are distributed uniformly

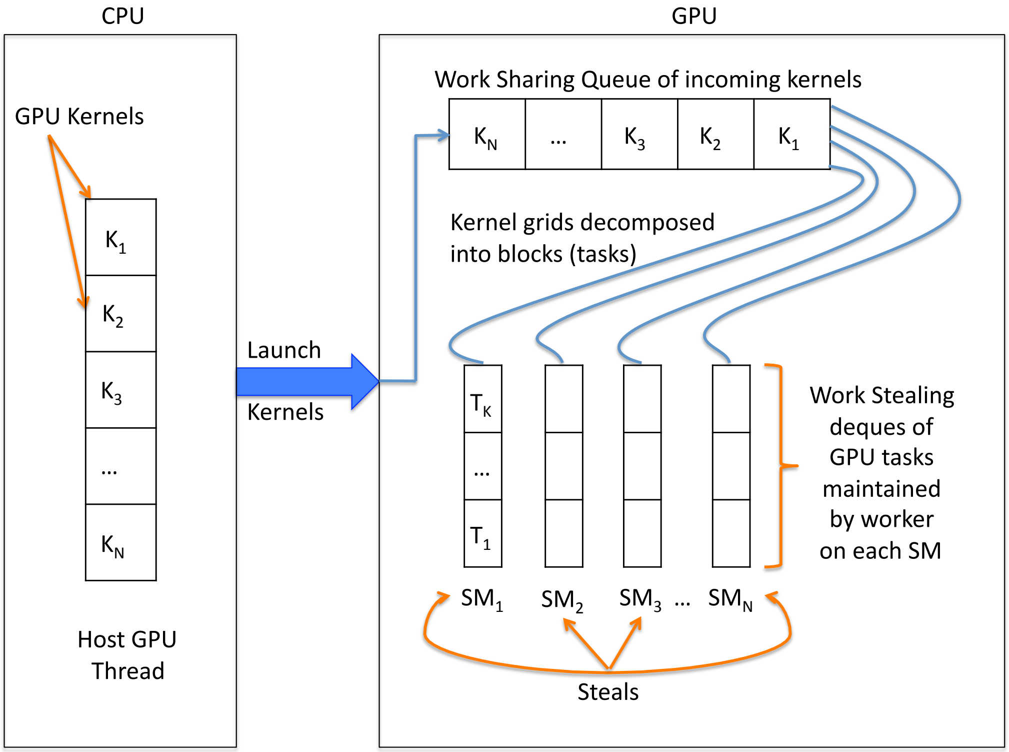

CPU
GPU Kernels
K1
K2
K3
...
KN
Host GPU Thread
Launch
Kernels
GPU
Work Sharing Queue of incoming kernels
KN
...
K3
K2
K1
Kernel grids decomposed into blocks (tasks)
TK
...
T1
Work Stealing deques of GPU tasks maintained by worker on each SM
SM1
SM2
SM3
...
SMN
Steals

Figure 5.1 : GPU runtime for dynamic task parallelism

and naively among devices from the host.

5.2.1 Task Representation

As described earlier, CUDA has two levels of parallelism: SIMD threads within a thread group on the same SM, and thread groups executing on different SMs. The tasks described in this work are scheduled on a group of threads rather than individual threads, though tasks can be created by any thread in a group. Scheduling tasks at the fine grain level of individual GPU threads would be prohibitively expensive, in most cases leading to major divergence and serialization of execution.

Tasks designated for execution on the device are represented by a GPU-specific

```
typedef struct {
    void *ptr; // address of the data
    size_t size; // number of bytes in this parameter
    volatile int *done_flag; // Indicates readiness of data
    unsigned char type; // Input to or output from device
} param;

typedef struct task_ {
    int type; // what code to run for this task
    param *p; // list of parameters to this task
    int *ready_flag; // indicates if this task has completed
    int num_params; // number of parameters
    ...
} task;
```

Figure 5.2 : Definition of structures defining tasks and their parameters on the GPU

task structure. This task structure contains an array of *param* structures representing inputs and outputs of a device task. The work sharing and stealing deques contain pointers to these task structures. The definition of the task and param structures is shown in Figures 5.2.

The *type* field of a task structure is an integer ID identifying the programmer-written kernel to be executed for this task. The *param* pointer points to a list of parameters to the task. Tagging a parameter as being input or output using the *type* field in the param struct only has significance in the context of transfers to and from the device, and does not necessarily describe its relation to a task. The task structure's *ready_flag* field is used to test when a continuation task is ready to be executed in our *finish-async* model for on-device dynamic task creation. Every time a task is created, it increments the *ready_flag* field in its continuation (if one exists). Once a task completes, the same *ready_flag* is decremented. When *ready_flag* reaches zero for a particular continuation task, it is ready to be scheduled by either being pushed to a work-stealing deque or immediately executed on the current worker group. Dynamic task creation is discussed in the next section. The *done_flag* in the param struct is used by the host to detect when a buffer is ready to be retrieved. The contents of the memory location pointed to be *done_flag* are cleared before starting every task, and set upon completion. Because the *done_flag* points to page-locked host memory that is mapped into the device address space, the GPU is able to signal the host that operations on a certain chunk of data is complete without any CUDA API calls or host intervention. Only pointer dereferences on the device and host are necessary to write or read the value of *done_flag*.

Information on the kernels and parameters of all task types in a DyGR application is passed to a DyGr initialization routine on the host. The task information provided includes the number and size of inputs and outputs for each task type. Assuming DyGR has exclusive access to the GPUs in a system, all device memory can be used to preallocate task structures, each with the memory necessary for storing task inputs and outputs. When a new task is created from this task pool or a task popped from a work queue, the pre-allocated device memory associated with that task becomes exclusively accessible to the current worker group. This approach to pseudo-dynamic memory allocation is extremely efficient, though not as flexible as the dynamic global memory allocation supported by CUDA.

5.2.2 Task Creation

A task is created by filling a task structure with the appropriate parameters and type. It is only eligible for execution after being placed in the work sharing queue if launched from the host, or a worker group's work stealing deque if launched from the device. If a task is launched from the host, the runtime also manages asynchronously allocating and copying buffers for the input and output data as part of the task creation process.

Once a task is placed in a work queue it can then be fetched by a worker group. The worker group that retrieved a task calls a user-written kernel based on the *type* field of the task and passes the task with its parameters to that kernel. All concurrent queues on the device are implemented using CUDA's atomic CAS instruction to manage head and tail indices into a fixed-size task array.

The worker groups are notified when the host has placed all available work on the device by a special token in the work sharing queue. Upon finding this value, each worker block can be sure that there will be no new incoming tasks from the host. This implies that all remaining work is already in the work-stealing deques. DyGR's termination detection mechanism then uses two global counters to determine when all device work has completed: one counting tasks created and the other counting tasks run. By comparing these counters, DyGR can implement efficient and accurate termination detection. As long as work remains on the GPU, worker groups continue to attempt to pop from their own and others' work deques.

5.2.3 Runtime Kernel

The DyGR CUDA kernel is responsible for finding work and handing off control to programmer-written kernels with the appropriate inputs. The kernel code includes policies on which queues to check for work, efficient and correct termination detection, and managing pre-allocated task structures. Pseudocode for the kernel algorithms is shown in Algorithm 2.

5.2.4 Communication Between Device and Host

One of the largest burdens placed on CUDA programmers trying to achieve optimal execution on the GPU is device memory management. DyGR hides all memory allocation and communication from the user.

```
// Success is shared by a thread group
// Only set by master thread
// Indicates whether a task was successfully found
__shared__ Success = False
TID = Thread Index Within Group
Terminate = False
HostSignalledDone = False
StealTarget = (GroupID + 1) % NumGroups
while Terminate == False do
    // TID == 0 is the master thread for a worker group
    // This block searches different queues for work
    if TID == 0 then
        Success = PopNewestFromPrivateQueue()
        // Try to pop from shared queue
        if Success == False and HostSignalledDone == False then
            Success = PopNewestFromSharedQueue()
            // If popped value is a special value indicating that
            // the host has no new work, set a flag for use in
            // termination detection
            if poppedTask == DoneToken then
                HostSignalledDone = True
                Success = False
            end
        end
        // Try to steal
        if Success == False then
            Success = StealOldestFromQueue(StealTarget)
            StealTarget = (StealTarget + 1) % NumGroups
        end
    end
    __syncthreads()
    // Either handle a new task, or check for termination
    if Success then
        HandleTask()
    else
        // If host signalled completion and the # of tasks
        // run equals # of tasks created
        if HostSignalledDone and CheckForTermination() then
            Terminate = True
        end
    end
end
```

Algorithm 2: Pseudocode for the DyGR kernel

CUDA streams are used to overlap communication and computation. Each time a new task is placed onto the device from the host, the associated input is copied with it. Using asynchronous memory copies in independent CUDA streams allows this communication to occur in parallel with the runtime kernel and any programmer-written code already executing on the device.

DyGR maintains a mapping from host to device addresses that allows the runtime on the host to keep track of what host memory locations have already been copied to the device, what device location they were copied to, and how many bytes were transferred.

5.3 Programming Model

5.3.1 DyGR Host & Device API

DyGR is a standalone tool which can be integrated with a frontend to provide a productive heterogeneous programming interface.

The DyGR API accessible from the host includes the following four functions.

1. **insert_task**: Places tasks constructed on the host into a device work sharing queue on one of the CUDA devices.

2. **get_data**: Retrieve data from a device to the host once the task responsible for that data completes computing its values. This is a blocking call.

3. **init_runtime**: Initializes the state of the GPU runtime, including creating work queues and preallocating task structures. The arguments to this kernel specify the number and size of every parameter to each type of task.

4. **launch_kernel**: Closely related to init_runtime, launch_kernel starts the runtime kernel on a specified device.

5. **finish_device**: Signal the device that there is no remaining work from the host and wait for it to complete all remaining tasks on the GPU.

On the other hand, the device API accessible from inside of user-written CUDA kernels includes:

1. **alloc_task**: Request a task structure to populate with inputs for a new dynamically generated task.
2. **push_task**: Place a populated task onto the work stealing deque of the current worker group.

5.3.2 DyGR as a Backend for High-Level Programming Models

The above DyGR API functions provide basic building blocks on which higher level programming models could be built to take advantage of the performance benefits and generality of DyGR. Let us use the CnC-CUDA programming model presented in Chapter 4 as an example. CnC-CUDA makes writing hybrid CPU-GPU applications much simpler for programmers with little experience in multi-core CPU or many-core GPU development. It is possible to integrate CnC-CUDA's frontend with DyGR's runtime system to make the CnC-CUDA programming model more computationally efficient and applicable to more general applications. As evidence, consider the mapping from CnC-CUDA operations to their implementation using the DyGR API in Table 5.1.

It would also be trivial to map JCUDA operations to this runtime. The added abstractions and generality of the runtime API and infrastructure provided by DyGR make extending many existing dynamic task parallel programming models to support GPU execution a far simpler task.

CnC-CUDA Operation	**Runtime Implementation**
	Host API
item.PutRegion	Reserve a task structure on the host containing the necessary fields for the items and tags for the step associated with the specified item collection. Save the allocated task structure for future item and tag puts.
tag.PutRegion	Place the prescribed region of tags into the reserved task structure, and use *insert_task* to copy all inputs to the device and launch the appropriate user-written kernel.
item.GetRegion	Use *get_data* to wait for the computation associated with the specified region in the specified item collection to complete.
	Device API
item.PutRegion	Use *alloc_task* to request a new task on the device to place newly created items into, and copy the items put to the allocated structure.
tag.PutRegion	Place the prescribed tags into the preallocated task structure and use *push_task* to make that work eligible for execution.
item.GetRegion	Inputs are automatically passed to the user-written kernels in DyGR, making this CnC-CUDA operation unnecessary.

Table 5.1 : Mapping from CnC-CUDA operations to their implementation using the DyGR API.

5.3.3 Building Applications with DyGR's API

While Section 5.3.2 proposed DyGR as a backend to higher-level programming models DyGR's API can also be accessed directly from application code, as was done in evaluating its performance for this thesis and past publications. We'll use the Unbalanced Tree Search (UTS) benchmark [29] to illustrate DyGR's usage for building applications.

UTS is a benchmark "designed to evaluate the performance and ease of programming for parallel applications requiring dynamic load balancing" [29] by counting "the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size, and imbalance." An implicitly constructed tree is able to generate all information on the children of a node based on the information contained within that node. As an irregular application, UTS would not normally mapped to GPUs due to 1) poor performance from thread divergence or underutilization, and 2) a lack of dynamic parallelism and other necessary programming constructs. However, we can use DyGR to efficiently execute UTS on GPUs.

Figure A.1 in Appendix A contains a code snippet of the main entry point of a DyGR UTS implementation. The high-level steps in executing a basic application using DyGR are shown to be:

1. Set up data structures on the GPUs using **init_runtime**
2. Launch the runtime kernel on all GPUs using **launch_kernel**
3. Seed the GPU with a few initial tasks using application-specific code and **insert_task**
4. Wait for device execution to complete using **finish_device**

Figure A.2 in Appendix A contains the programmer-written UTS kernel that is executed by DyGR on the GPU. Most of the code shown is specific to UTS. We can observe that the user-written kernel must take a number of arguments, most of which are passed on as arguments to **push_task** or **alloc_task** for use by the DyGR runtime. Also note that the parallelism in the UTS kernel is across children of the current node. This is a pattern very common in graph and tree algorithms which is not easily expressable in current GPU programming models, but is trivially implemented using DyGR.

5.4 Performance Evaluation

The performance of DyGR was evaluated using examples of applications that are challenging to implement efficiently on graphics hardware as well as data parallel applications that are already well suited for CUDA. The benchmarks used are:

1. **NQueens** from BOTS[30]. The BOTS implementation of NQueens on a CUDA device is difficult because it can result in unbalanced computation trees and requires dynamic task creation and load balancing.

2. **Quicksort**, based on [31].

3. **Crypt** from the Java Grande Forum Benchmark Suite[26]. Crypt is an application with regular parallelism, and recognized to be a good candidate for GPU execution.

4. Shortest path computation based on the implementation of **Dijkstra's** algorithm in [32]. This benchmark starts with a single task and must then spread the load across all SMs as evenly as possible.

5. **Unbalanced Tree Search** (UTS) [29]. UTS is described in Section 5.3.3. UTS tests the load balancing capabilities of the runtime.

6. **Series** from the Java Grande Forum Benchmark Suite[26]. Series is another data parallel benchmark which demonstrates the low overhead of our runtime

system.

Runs with different numbers of devices and different data sizes are compared to study the impact on execution time. Additionally, diagnostic data from the runtime is used to measure how effectively an application's workload is balanced across SMs.

Benchmark tests were performed with 1 or 2 NVIDIA Tesla C2050 GPUs. Each Tesla C2050 has 14 multiprocessors, 2.8 GB of global memory, 1.15 GHz clock cycle and are using CUDA Driver and Runtime version 3.20. The host machine consists of 3 Quad Core AMD CPUs with a clock of 2.5 GHz.

5.4.1 NQueens

For the NQueens benchmark the BOTS implementation of NQueens was ported to the GPU using the DyGR API. The BOTS benchmark suite is designed to test the effect that irregular parallelism has on a multicore system. Irregular parallelism results in less predictable numbers and distributions of tasks. Because of this, NQueens is a challenge to effectively port to the CUDA programming model. The DyGR runtime and API facilitate irregular parallelism on graphics hardware. Figure 5.3 shows that the NQueens benchmark scales well across multiple devices. From experience in developing it, building the NQueens benchmark for the GPU was much simpler with the runtime than it would have been without.

5.4.2 Crypt

The Crypt benchmark from the Java Grande Forum Benchmark Suite performs the IDEA cryptographic algorithm on a sequence of bytes, both encryption and decryption. Evaluation of JCUDA and CnC-CUDA performance in Chapters 4 and 3 show that Crypt is well-suited for GPU execution. The encryption and decryption of every 8 bytes can be run independent of the rest of the data set. Crypt is included in these

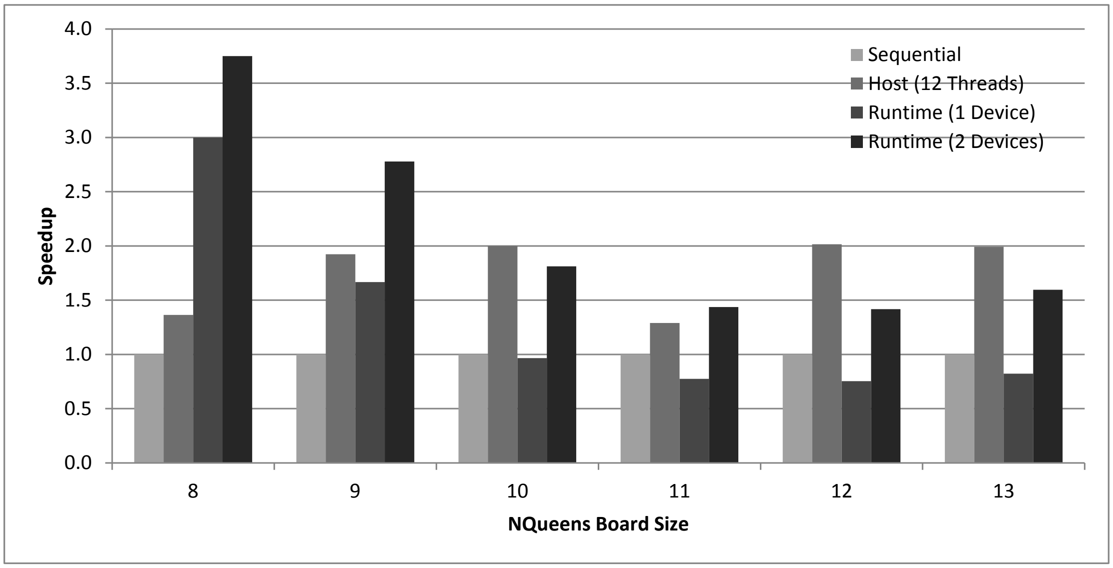


Figure 5.3 : Speedup normalized to single-threaded execution of the NQueens benchmark using DyGR on 1 or 2 devices and 12 threads on the host.

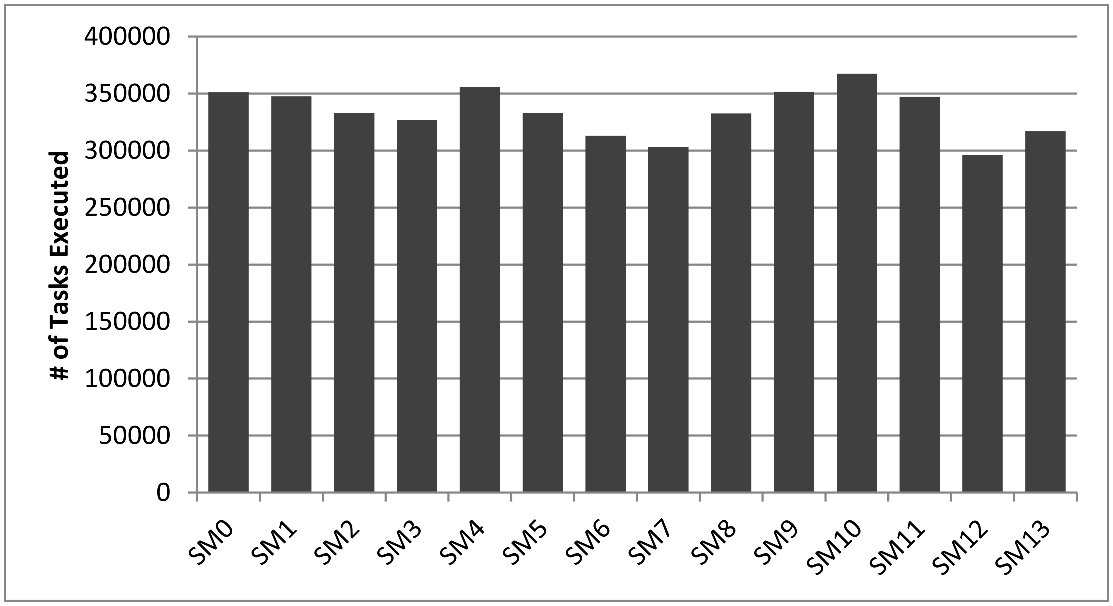


Figure 5.4 : Tasks executed by each SM on a single device running the NQueens benchmark with board size=13×13.

experiments to demonstrate that using this runtime to run an application which is already well suited for CUDA does not result in significant degradation of performance. For that reason, no comparison was done against CPU implementations.

The hand coded CUDA version of Crypt tested does not try to take advantage of any overlapping of communication and computation, as it was predicted that doing so would not benefit performance but would add significantly to the complexity of the implementation. Later tests showed that better performance was achievable with hand coded CUDA code, but required considerable more experience and effort on the part of the CUDA programmer.

Though not shown in Figure 5.5, an unexpected benefit of DyGR's automatic multi-device management was the ability to handle larger data sets due to the expanded storage available from multiple devices.

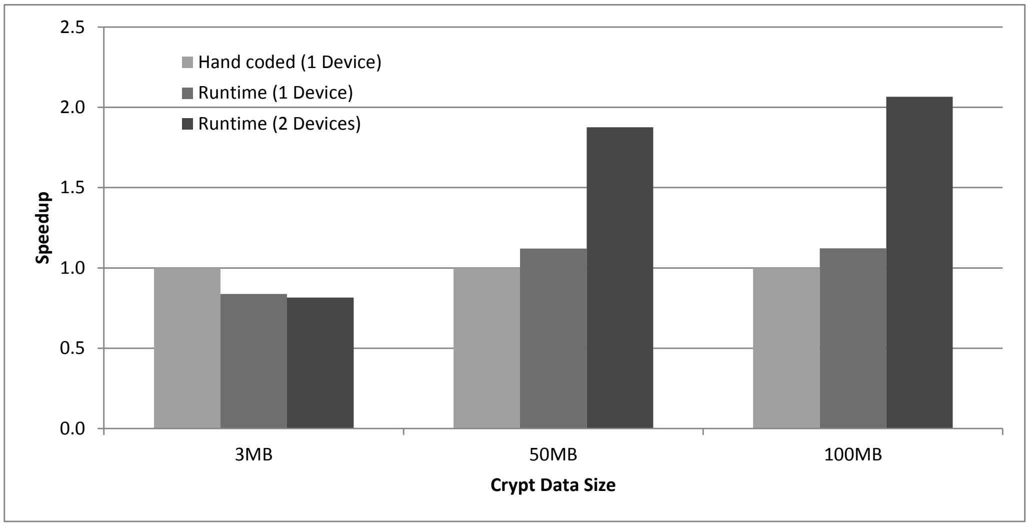


Figure 5.5 : Speedup of the Crypt benchmark using DyGR on 1 or 2 devices and hand coded CUDA on a single device. Speedup is normalized to single device execution.

5.4.3 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm was implemented using the DyGR's API based on the algorithm used in "Dynamic Work Scheduling for GPU Systems" [32] because it effectively tests the load balancing abilities of the runtime. Figure 5.6 shows how many tasks each worker group executes while finding the distance from each node to a destination node in a 10,000 node bidirectional weighted graph. Initially a single task is placed on a single worker. From there, the DyGR runtime is able to evenly distribute tasks to all SMs on the device.

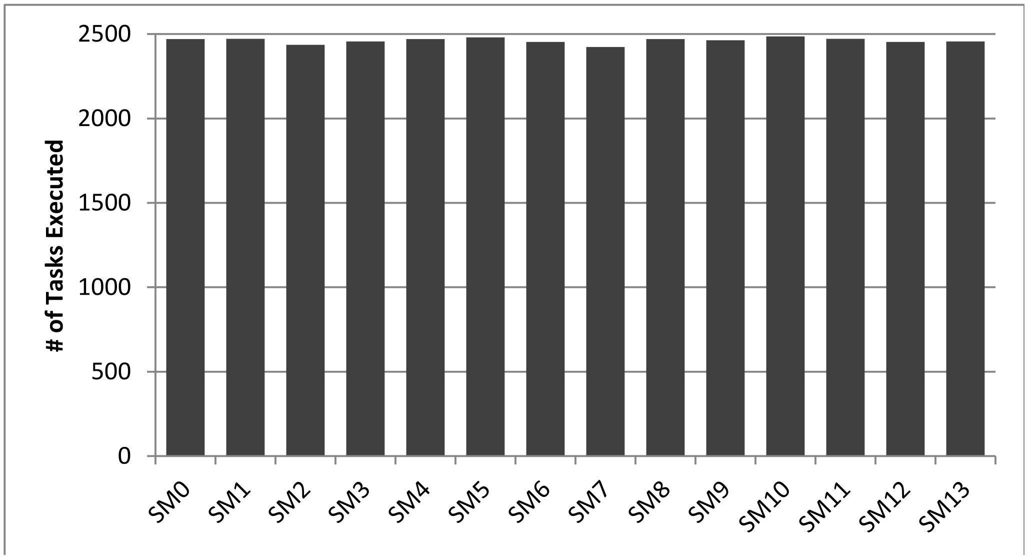


Figure 5.6 : Tasks executed by each SM on a single device running the Dijkstra benchmark.

5.4.4 Unbalanced Tree Search

As discussed earlier, UTS explores an unbalanced and implicitly defined tree. UTS was implemented using the DyGR API based on a multi-threaded OpenMP implementation from OSU. Tests were run on a geometrically generated tree using b_0 (the branching factor) set to 50. Figure 5.7 shows that, in general, the GPU implementation was able to maintain performance parity with a 12 core host system.

5.4.5 Series

The Series benchmark from the Java Grande Forum Benchmark Suite is extremely data parallel, and well suited to execution on the GPU. One interesting result of this benchmark is that the runtime using one device severely underperforms compared to

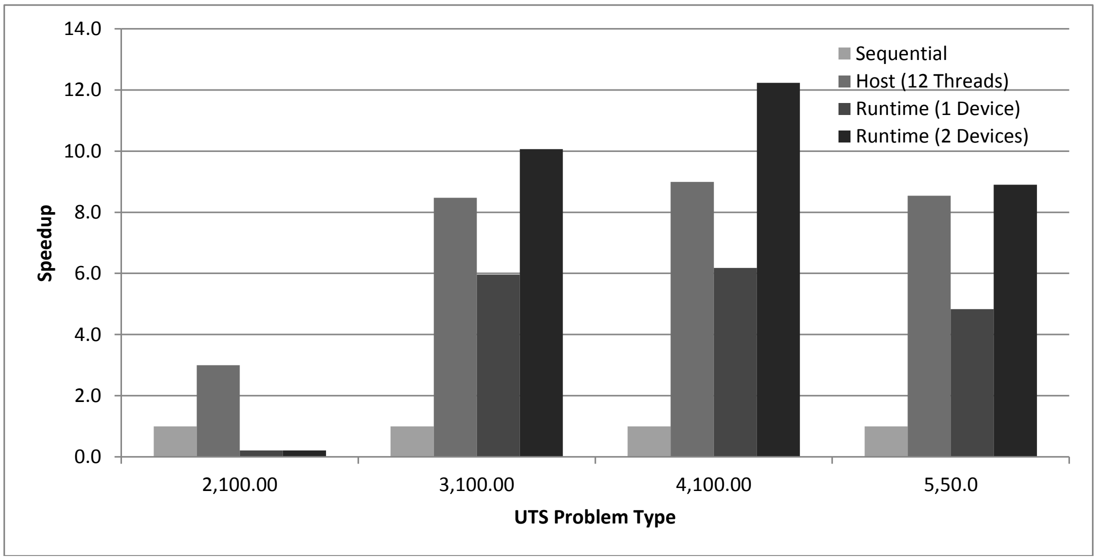


Figure 5.7 : Speedup of the UTS benchmark using DyGR on 1 or 2 devices, using 12 threads on a 12 core host system, and running in single threaded mode. Speedup is normalized to the single threaded implementation.

the hand coded implementation. The cause of this is the small input data sizes to Series. Because there is very little input to the Series benchmark, the small copies necessary to launch tasks on the device add enough overhead to degrade performance. This suggests a small subset of applications which the runtime may not perform well on. However, using 2 devices outstrips single device hand-coded CUDA without any additional programmer effort.

5.4.6 Multi-GPU Performance

For many of the above benchmarks, lower than expected acceleration is seen from using multiple devices, a counterintuitive result which requires some explanation. Further investigation found that the primary cause of this is redundant memory copies. As an example, consider the Crypt benchmark. In Crypt, there are encryption and decryption keys which must be accessible from each device. Increasing the number of devices therefore also increases the amount of necessary communication. Two

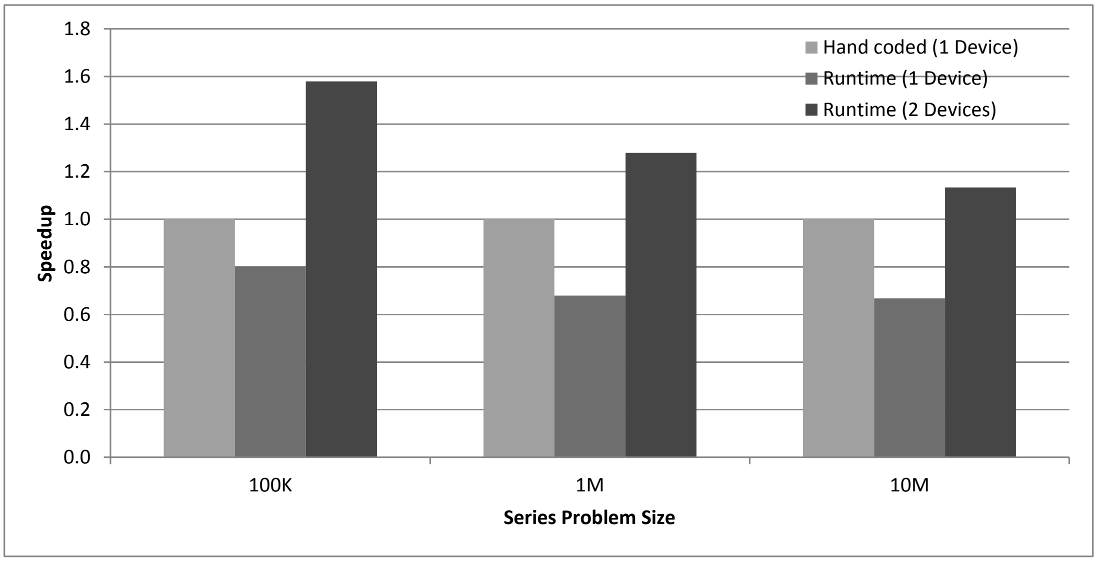


Figure 5.8 : Speedup of the Series benchmark using DyGR on 1 or 2 devices, compared against hand-coded CUDA on 1 device. Speedup is normalized to the single device implementation.

potential solutions to this problem could be: 1) more intelligent task placement based on data locality, or 2) using CUDA's unified virtual device address space to access a single copy from any device. In the case of the Crypt benchmark and others like it with globally shared data structures, the usefulness of more intelligent task placement would be minimal as it would mean even when 3 devices are available not all would necessarily be used. Additionally, while early experience with the unified virtual address space shows that it is useful in easing porting, its use can result in performance degradation.

5.5 Discussion

This chapter described a GPU work stealing runtime that supports dynamic task parallelism at thread group granularity. The effectiveness of combining work stealing and work sharing queues in distributing tasks across a device for irregular applications was demonstrated using UTS, NQueens, and Dijkstra. Each of these benchmarks starts with a small pool of initial tasks and requires low overhead task distribution to achieve good performance and workload balance. The low overhead of DyGR was also demonstrated using Crypt and Series, both of which are well suited for CUDA and demonstrated performance comparable to hand-coded CUDA. An overview was given of the simplified host-side and device-side API currently available for DyGR, and a hypothetical extension of CnC-CUDA using this API was proposed.

After the publication of this work, NVIDIA added Dynamic Parallelism to the CUDA feature set, allowing CUDA programmers to dynamically create new thread groups on the GPU without relying on the CPU as the only source of work. Their implementation closely mirrors the dynamic parallelism enabled by this work. However, many of the other features demonstrated here remain beyond the scope of CUDA. These include dynamic memory management, advanced load balancing through work stealing, and a tighter coupling between host and accelerator in a more cooperative and less "host+accelerator" relationship where all devices can dynamically generate new work for each other.

Chapter 6

HadoopCL

The previous chapters focus exclusively on single-node heterogeneous systems. However, it is very common to find heterogeneous hardware in distributed systems, for instance with multiple GPUs per node in a cluster. While the work presented previously could be utilized in such a system to accelerate the computation performed within a single node, having an integrated distributed and heterogeneous programming platform would potentially yield improved performance and programmability on heterogeneous distributed platforms.

Distributed, heterogeneous systems are the focus of the work discussed in this chapter: HadoopCL. HadoopCL is a runtime and compile-time framework for executing the computational sections of a distributed Hadoop MapReduce job on heterogeneous hardware through JIT compilation of Java bytecode to OpenCL kernels. The main contributions of this work include:

1. Extension of Hadoop Mapper and Reducer classes to support execution of user-written Java kernels on heterogeneous devices, with a focus on minimizing modifications to legacy code.

2. The use of dedicated communication threads and asynchronous communication to maximize utilization of available bandwidth and limit blocking on communication.

3. Automatic translation of Java bytecode to OpenCL kernels using APARAPI[14], and extensions to APARAPI's existing features.

4. Evaluation of HadoopCL's performance in a heterogeneous cluster containing multi-core CPUs and discrete GPUs.

6.1 Approach

This section covers the techniques and algorithms used in executing HadoopCL Mappers and Reducers on heterogeneous hardware.

6.1.1 Heterogeneous Mapper and Reducer

As described in Section 2.2.4, the computation in MapReduce jobs is encapsulated in user-implemented Mappers and Reducers. Mappers transform an input (key,value) pair to zero or more output (key,value) pairs. Reducers take as input the outputs of the Mappers as (key,value-list) pairs, where the value list is composed of all values associated with the same key in the output of the Mappers. Reducers then generate an output of 0 or more (key,value) pairs. Mappers and Reducers are therefore inherently data parallel, and exhibit high degrees of parallelism for data intensive applications.

Apache Hadoop is a distributed, open-source, Java MapReduce implementation which provides a MapReduce programming model, distributed filesystem, and reliability guarantees. While Hadoop is popular for its simplicity, it also demonstrates inefficiencies which decrease its usefulness for certain problems due to sub-par computational performance and network utilization. Mapper and Reducer tasks run inside of potentially short-lived Java virtual machines. Creating, managing, and executing inside these JVMs incurs processor and memory overhead and may reduce the effectiveness of JIT compilation. However, using separate JVMs provides a significant reliability advantage by isolating the Hadoop system from Mapper and Reducer failures. Evaluations also show that HDFS I/O operations consume a significant percentage of total execution time for most applications, but HDFS provides data reliability via replication as well as a number of other desirable features.

The main contributions of this work are heterogeneous Mappers and Reducers which, when extended, automatically:

1. Execute user-written Hadoop Map and Reduce computation natively on all available devices in a platform.
2. Use multiple input and output buffers, dedicated communication threads, and a pipeline of Java threads with different responsibilities to maximize utilization of disk and inter-device bandwidth.

Figure 6.1 is a high-level system diagram of the HadoopCL system contained in a single node. The TaskRunner JVM manages the spawning of child JVMs within a node. These child JVMs execute the Mapper or Reducer computation on provided inputs, isolated from the Hadoop system. In HadoopCL, TaskRunner is responsible for tracking device usage and assigning a device to each JVM as it is spawned based on a load balancing device management algorithm. The device management algorithm used in this work selects the device to execute based on a weight given to each device. A larger weight makes it more likely for a child JVM to be assigned that device. In this work, larger weights were given to GPU devices as they were generally underutilized by a single JVM. Device usage is tracked as JVMs start and complete.

Each Child JVM is assigned a single HDFS chunk of keys and values for processing. These Child JVMs can execute either Map or Reduce computation, and much of the HadoopCL runtime implementation is shared between heterogeneous Mappers and Reducers. Each Child JVM contains a pipeline of 3 Java threads with different responsibilities. The main Java thread is responsible for

1. Reading and buffering input keys and values from the input HDFS chunk.
2. Feeding the buffered inputs to the compute thread.

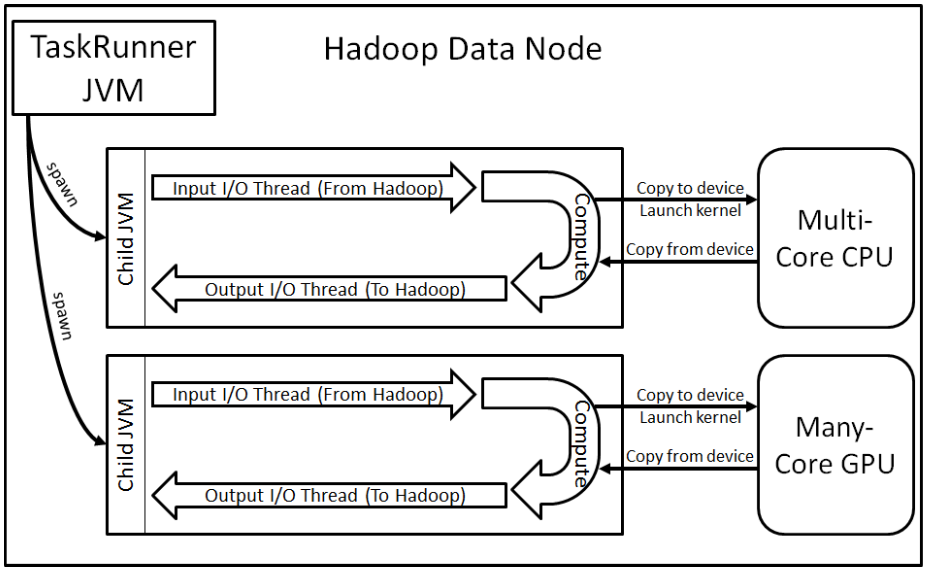


Figure 6.1 : System diagram for HadoopCL.

The child compute thread is responsible for:

1. Launching OpenCL kernels on the buffered inputs received from the main thread. This involves 1) copying buffered data to the OpenCL device, 2) launching a Mapper or Reducer kernel on that input, and 3) retrieving the output of the kernel from the device.

2. Feeding the generated outputs to the communication thread.

The child communication thread is responsible for receiving output data from the compute thread and writing it back into the HDFS system. These pipelined threads are connected via concurrent queues containing buffers of input or output data. Each thread (main, compute, and communicate) uses Java `wait()` and `notify()` calls on the concurrent queues to minimize CPU occupancy, allowing stalled threads to give up cycles to threads with work to do.

Using dedicated compute and communication threads maximizes the overlap of computation and communication, keeps the network and device bandwidth utilized, and prevents all threads from blocking on most operations.

6.1.2 Heterogeneous Execution of JIT Compiled OpenCL Kernels

HadoopCL relies on APARAPI [14], an open-source, independently developed tool, to translate the bytecode of user-written Java kernels to OpenCL kernels. OpenCL kernel code is generated for both the user-written Map and Reduce functions, as well as for HadoopCL glue code which passes keys and values into the user-written functions. The loop chunking and stride in HadoopCL glue code is written so as to encourage improved memory access patterns depending on the current device. This optimization is currently supported for GPUs and multi-core CPUs.

Figure 6.2 shows the process by which APARAPI translates Java bytecode to OpenCL kernels and executes OpenCL kernels on devices. This includes inspecting the bytecode instructions for translation to the OpenCL kernel language, collecting information in Java on the primitive arrays being passed as input or output, using that information in native code to transfer from the JVM heap to OpenCL devices and back using JNI and the OpenCL API, and executing OpenCL kernels using the created OpenCL buffers.

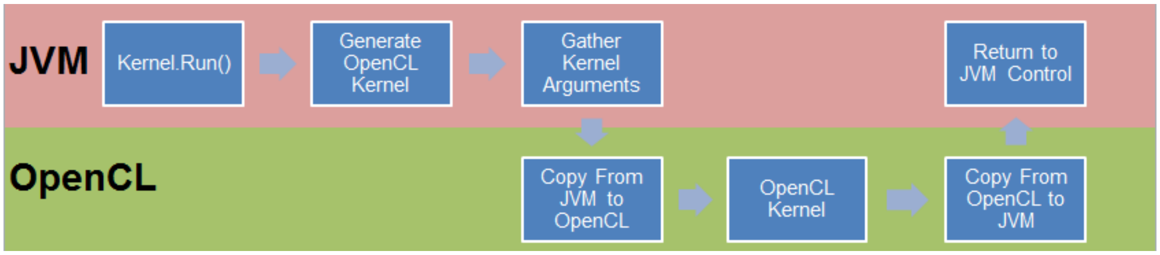


Figure 6.2 : Process by which APARAPI translates Java bytecode to OpenCL kernels and executes those kernels on OpenCL devices

As part of HadoopCL, APARAPI was extended to detect the direction of a buffer (input, output, or both) based on naming conventions. Any variable name starting with `input_` is assumed to be input and any variable name starting with `output_` is assumed to be output. If no naming convention is detected, the variable is assumed to be both input and output. This HadoopCL-specific optimization enables removal of unnecessary transfers. All checks are performed at runtime by the modified APARAPI runtime.

6.1.3 Programming Framework

One of the main benefits of Hadoop is its programmability. It derives this programmability from the high-level abstractions of MapReduce and the use of Java as a high-level language. While the primary goal of HadoopCL is improved performance, it is important to consider the impact these modifications have on the programming model's flexibility for developers.

Because ARARAPI does not support the full Java language, all Java Mappers and Reducers originally written for Hadoop require some modifications to be usable in the HadoopCL framework. The resulting code is still written in (a subset of) standard Java.

For copying arrays of primitives to and from the device, APARAPI uses a number of type-specific methods. For example, `Kernel.put(int[])` and `Kernel.get(int[])` are used to copy integer arrays. This conflicts with Hadoop's extensive use of generics, and means that the types of input keys and values must be known at compile time for Mappers and Reducers that are to be executed as OpenCL kernels. As a result the generic Hadoop Mapper and Reducer classes are replaced by a number of auto-generated, type-specific HadoopCL Mapper and Reducer classes which can be automatically replaced by OpenCL. For instance, extending `Mapper<IntWritable, FloatWritable,IntWritable,FloatWritable>` in Hadoop would be translated to extending `IntFloatIntFloatHadoopCLMapperKernel` in HadoopCL. Because these type-specific OpenCL Mapper and Reducer classes are formulaic, auto-generating them for arbitrary combinations of key and value types is trivial (while keeping in mind APARAPI only supports a subset of Java types). For this task, a simple Python script was written that fills in the type-specific methods for HadoopCL Mappers and Reducers. Examples of these methods include methods for accumulating input data into buffers for OpenCL processing, for writing output keys and values back to Hadoop, and for calling the type-specific, user-written Map or Reduce kernel.

HadoopCL adds support for commonly used object types, such as pairs or triples of primitives, and generates the glue code for converting these objects to primitive arrays which APARAPI can then transfer to OpenCL and generate OpenCL kernels for. For instance, if the key for the Mapper stage is a HadoopCL.Pair object containing two integer values, HadoopCL would serialize all HadoopCL.Pair objects to two integer arrays before transferring them to OpenCL. The HadoopCL OpenCL glue code would then pass the input key by-value as two integers to the user-written Map kernel, hiding serialization and deserialization of the HadoopCL.Pair objects from the programmer.

OpenCL does not support dynamic memory allocation. As a result, HadoopCL must preallocate all memory for Mapper/Reducer output keys and values before launching kernels. The developer must provide the framework with a limit on the number of output (key,value) pairs that can be generated from any input pair passed to Mappers and Reducers. This value is then used to calculate the required size of preallocated memory. Future work would expand on this approach to make it more flexible for applications which may generate unbalanced numbers of output (key,value) pairs from different inputs to prevent over-allocation. In our experience, this requirement does not hinder application development.

As a case study, let us compare the Hadoop and HadoopCL implementations of the Mapper computation from a Pi benchmark which approximates the value of pi using

```
class PiJavaMapper extends
    Mapper<DoubleWritable, DoubleWritable,
           IntWritable, IntWritable> {

  public void map(DoubleWritable key,
      DoubleWritable value,
   Context context)
      throws IOException,
   InterruptedException {
    double x = key.get() - 0.5;
    double y = value.get() - 0.5;

    if(x * x + y * y > 0.25) {
      context.write(new IntWritable(0),
        new IntWritable(1));
    } else {
      context.write(new IntWritable(1),
        new IntWritable(1));
    }
  }
}
```

Figure 6.3 : Java implementation of Pi Mapper computation extending Hadoop's Mapper class

randomly generated numbers. The original implementation in Java using Hadoop's Mapper class is in Figure 6.3. The HadoopCL-compatible Java implementation is in Figure 6.4.

As described in the text above, the main differences are:

1. Extension of `DoubleDoubleIntIntHadoopCLMapperKernel` instead of `Mapper< DoubleWritable,DoubleWritable, IntWritable,IntWritable>`

2. A different map signature: `map(double, double)` instead of `map(DoubleWritable, DoubleWritable, Context)`

3. A replacement `write` method which writes to the preallocated output array, replacing `context.write()`.

```
class PiCLMapper extends
    DoubleDoubleIntIntHadoopCLMapperKernel {

  protected void map(double key,
      double val) {
    double x = key - 0.5;
    double y = val - 0.5;

    if(x * x + y * y > 0.25) {
      write(0, 1);
    } else {
      write(1, 1);
    }
  }

  public int getOutputPairsPerInput() {
    return 1;
  }
}
```

Figure 6.4 : HadoopCL-compatible implementation of the Pi Mapper computation

4. The addition of `getOutputPairsPerInput()`, which the application developer uses to indicate to HadoopCL the maximum number of outputs that can be emitted per input. This is used for output array preallocation.

6.2 Experimental Setup

To evaluate HadoopCL's performance, 4 benchmarks are used: Pi, KMeans, Sort, and Black-Scholes. Though small in size, these benchmarks are representative of high-performance data-intensive computing. This section describes the function and implementation of each benchmark as well as the methodology used in gathering results.

	Map In	Map Out/Reduce In	Reduce Out
Pi	(double,double)	(int,int)	(int,int)
Black-Scholes	(int,float)	(int,float)	(int,float)
KMeans	(double,double)	(int,pair)	(double,double)
Sort	(long,long)	(int,long)	(int,long)

Table 6.1 : Input and output types for each benchmark

6.2.1 Pi

The Pi implementation used in this work is based on the example provided with Hadoop. Pi is a common Hadoop benchmarking workload which estimates the value of pi by randomly placing 2D points and classifying them as inside or outside the unit circle.

The Map computation takes as input a pair of Doubles, representing the 2D coordinates of a point. It classifies this point as either inside or outside the unit circle, and outputs two integers. The first integer indicates whether the point is inside the circle (0 or 1). The second integer is always 1, indicating that this is the data for a single 2D point.

The Reduce computation counts the number of values associated with each key (in Pi there are only two Reduce input keys, 0 and 1) and outputs an accumulated sum. Once the Hadoop job has completed, it is trivial to check the results against the true value of pi.

400,000,000 2D points were used as input to all Pi jobs in the results.

6.2.2 KMeans

KMeans is an iterative clustering algorithm and also a common Hadoop benchmarking workload. In our tests, we applied KMeans to 2D points whose coordinates were double floating point values.

The Map stage of KMeans takes a pair of Doubles, representing a 2D point.

The Map function iterates through the current known centroids and finds the closest centroid to the input point. Finally, it outputs an integer indicating the closest cluster/centroid as the output key, and a pair containing the coordinates of the point as the output value.

The Reduce stage of KMeans consumes all of the points which have been classified as belonging to a certain cluster, and recalculates the centroid of the cluster by finding the point which is closest to the average of all points' positions. The new centroid is then output as a pair of doubles.

The implementation of KMeans required adding the ability to retrieve global data from arbitrary devices. This was accomplished by allowing the creation of specially named system properties whose values are automatically interpreted as floating point and made accessible from OpenCL kernels.

In our tests, a single iteration of the KMeans algorithm is tested on 20,000,000 points and 10,000 clusters.

6.2.3 Black-Scholes

Black-Scholes is a common data-parallel financial application, but is not generally executed in Hadoop because it has essentially no required reduction.

The Map stage takes as input a unique integer ID and floating point value. The Black-Scholes financial algorithm is executed on the input value. Then two (key,value) pairs are output: the integer ID paired with a put value, and the integer ID paired with the call value.

The Reduce stage is essentially a no-op, simply passing the outputs of the Map stage as the final output of the job.

In the Black-Scholes tests, 401920000 input values were used.

6.2.4 Sort

Our sort benchmark is a distributed radix sort of 64-bit long integers, which maps well to MapReduce.

The Map stage takes as input a pair of longs, both of which are values to be sorted. It outputs a radix for each paired with the input value. In our implementation we use the top 32 bits of each 64-bit long as the radix.

The Reduce stage then performs a local sort on all values which share a radix and outputs the sorted results.

In these distributed sort tests 400,000,000 values are sorted.

6.2.5 Methodology

Results were gathered on the DAVINCI cluster at Rice University. Each node in DAVINCI contains two six-core Intel X5660 CPUs running at 2.80GHz and two discrete NVIDIA Tesla M2050 GPUs. Each GPU has access to 2.5GB of global memory. There is 48GB of system memory accessible from the CPUs. All tests were run using two nodes in the cluster: 1 NameNode and 1 DataNode.

Each test on the DAVINCI cluster at Rice University was run 5 times. The minimum time for each system (Hadoop and HadoopCL) is reported in our results. OpenCL v1.2 and Java Hotspot v1.6.0_25 were used. For OpenCL, 196 threads per thread group were used on the multi-core CPU and 256 threads per thread group on the GPU. For tests with Hadoop, an HDFS chunk size of 64MB was used. For tests with HadoopCL, an HDFS chunk size of 256MB was used. The larger chunk size for HadoopCL reflects the fact that individual Child JVMs in Hadoop are generally only responsible for keeping a single CPU core occupied, while HadoopCL Child JVMs feed an entire device. These values were arrived at after evaluating several different configurations for Hadoop and HadoopCL. Tests of Hadoop with larger HDFS chunk sizes show no improvement in performance, eliminating that as a factor in these results.

As results were collected, we were able to quickly draw the conclusion that I/O was a major bottleneck for some of these applications. The I/O bottleneck became more significant with OpenCL-accelerated computation. Hadoop provides built-in support for compressing the initial inputs to a MapReduce job, the intermediate values produced by the Mapper stage and consumed by the Reducer stage, and the final outputs of a MapReduce job. All Hadoop and HadoopCL tests on DAVINCI were run with compressed inputs and intermediate values. These tests were run with all of the same parameters and configurations as uncompressed tests. While many different compression codecs are supported in Hadoop, exhaustive tests with the Sort benchmark on the BZip2, GZip, Default, Snappy, and LZO codecs indicate that the best combination is to use the Default(ZLib) compression codec for the initial inputs, and LZO compression for the Mapper outputs. Though further investigation may reveal different optimal combinations for the other benchmarks, time constraints meant that all compression tests were run with these codecs.

6.3 Results

In this section we present performance results of the benchmarks described in Section 6.2 and perform more detailed analysis to explain the reasons for performance improvement or loss.

6.3.1 Overall Performance

The overall performance of all applications tested on the DAVINCI cluster running in Hadoop and HadoopCL with and without compression is shown in Figure 6.5. Table 6.2 shows the actual speedups achieved on DAVINCI using HadoopCL with

Benchmark	Speedup w/o compression	Speedup w/ compression
Pi	1.97x	3.15x
Black-Scholes	3.76x	4.49x
KMeans	6.73x	8.69x
Sort	1.13x	1.34x

Table 6.2 : Overall speedup of the HadoopCL implementation of the benchmarks on the DAVINCI cluster with and without compression, relative to Hadoop without compression.

and without compression. These measurements include overhead from all network and inter-device I/O.

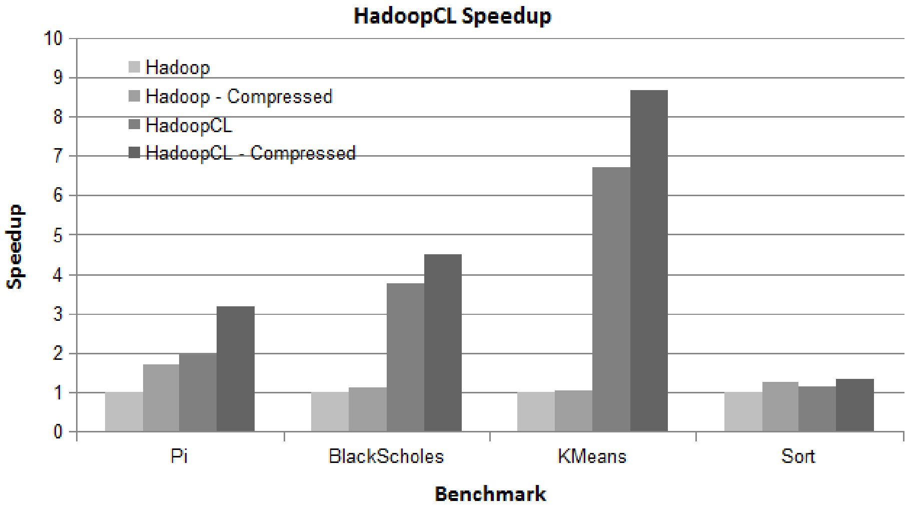


Figure 6.5 : Overall speedup of all benchmarks on DAVINCI with and without compression, normalized to Hadoop without compression.

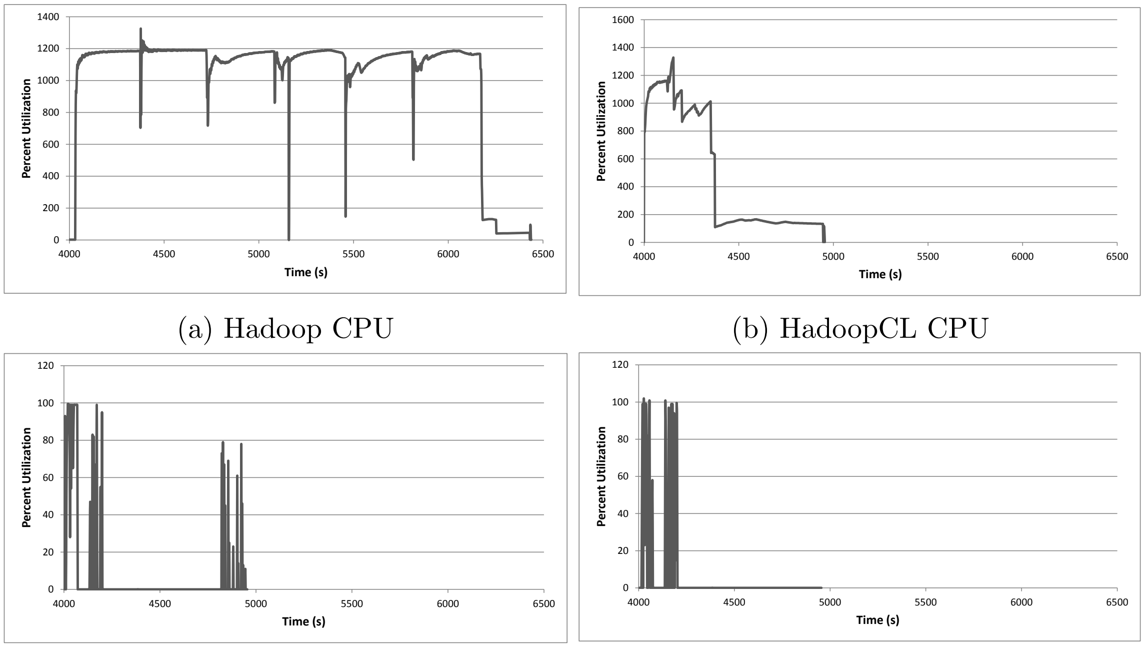
More detailed metrics were gathered on DAVINCI for the applications with consistently best and worst performance relative to Hadoop. All of the below tests were gathered without compression enabled.

HadoopCL demonstrated the worst relative performance on the distributed sort benchmark. We observed that removing all computation from the Sort benchmark but producing the same communication patterns led to near-identical performance in HadoopCL, and only a drop in execution time of 10-20% on Hadoop. In addition we can observe that adding compression to Hadoop Sort caused execution time to nearly match HadoopCL performance. From this, we are able to conclude that 1) the Sort benchmark is I/O bound and the benefits of improved computation are therefore negligible, and 2) as expected the overheads in HadoopCL have increased relative to Hadoop though the I/O optimization strategies seem to be effective and the cause of any Sort speedup. The increased overhead in HadoopCL is most likely caused by the added communication between discrete devices and overhead incurred from APARAPI's translation mechanism.

KMeans's performance was also investigated. Kmeans demonstrated a speedup of 6.73x without compression and 8.69x with compression on DAVINCI. The command line tools 'ps' and 'nvidia-smi' were used to collect statistics on CPU and GPU utilization in the child node for a larger input size. The collected data can be seen in Figure 6.6. Note that the time scales start at 4000 seconds. Prior to that, no activity was seen in the node. We infer this to mean that time was entirely spent performing I/O. Figure 6.6a shows that Hadoop is able to keep all 12 cores utilized, but we can see that it takes much longer for it to complete the work due to inefficiencies in virtual machine execution. The HadoopCL results show that the CPU and GPUs are fully utilized for much briefer periods. Executing in native threads on both devices clearly leads to the input data being processed in a much shorter time span. We can also observe 2 trailing threads in Figure 6.6b, most likely dedicated I/O threads completing writes to HDFS.

6.3.2 Mapper and Reducer Performance

More detailed metrics on Mapper and Reducer performance in Hadoop and HadoopCL were also collected, including information on how long the processing of an input



(a) Hadoop CPU

(b) HadoopCL CPU

(c) HadoopCL GPU 0

(d) HadoopCL GPU 1

Figure 6.6 : Processor utilization for Hadoop and HadoopCL collected during a KMeans execution on 400,000,000 points. Note that the time scales on all graphs are kept the same for easier comparison.

chunk took. Table 6.3 shows the improvement in average processing bandwidth for all benchmarks of HadoopCL Mappers and Reducers as a factor of Hadoop performance, based on keys processed per second. Before discussing each benchmark individually, it is important to mention that analysis of logs shows that nearly all computation in HadoopCL was scheduled on the GPUs in the platform due to their weighting in the device scheduling algorithm. Only the Reducer stage of Pi shows tasks being executed on the CPU.

In Pi, Mapper performance in HadoopCL and Hadoop was on par. This is expected: the Pi Mapper is extremely simple and performs a small amount of computation relative to the input size for each Mapper instance. A reasonable explanation would be that the added communication overhead to the discrete OpenCL devices neutralized any computational benefit from those devices, though more detailed pro-

Benchmark	Mapper Improvement	Reducer Improvement
Pi	1.02x	3.59x
Black-Scholes	7.87x	0.32x
KMeans	16.19x	0.90x
Sort	0.80x	0.32x

Table 6.3 : Throughput improvement of HadoopCL Mappers and Reducers as a factor of Hadoop performance w/ compression.

filing would be necessary to verify this claim. As mentioned earlier, the Pi Reducer makes use of the CPU. The 3.59x performance improvement for the Pi Reducer is most likely due to 1) native execution, and 2) large amounts of computation relative to the input size.

Black-Scholes demonstrates the second most Mapper speedup, as expected for a computation bound kernel. However, the Reduce stage experiences $\tilde{3}$x slowdown. This slowdown is caused by the same reason for Pi's poor Mapper performance: minimal computation for each (key,value) pair led to significant communication overhead. In the case of Black-Scholes, the Mapper stage dominates execution time which limits the impact of the Reducer slowdown on overall application performance.

KMeans also has a computationally heavy Mapper and shows even better Mapper speedup than Black-Scholes. The KMeans Reduce stage is not as negligible as Black-Scholes's and so the same slowdown is not observed, but it does introduce divergence on the GPU which limits the speedup possible. As a result, KMeans Reducer performance on HadoopCL is similar to Hadoop.

Sort showed the worst overall performance on HadoopCL, demonstrating negligible improvement over Hadoop. Sort's Map stage is computationally light, and communication overhead is probably the cause of the slight slowdown shown. The cause of poor Reducer performance in Sort on HadoopCL is significant divergence in the Reduce kernel on the GPU. While the computational sections of Sort both experience a slowdown, the dedicated communication threads still enable overall performance improvement when running in HadoopCL.

Our analysis of the Mapper and Reducer kernel performance for these four applications shows that HadoopCL could greatly benefit from intelligent scheduling of Mappers and Reducers on optimal devices. Naively scheduling kernels which exhibit large amounts of divergence or small computation per input (key,value) pair on GPUs led to significant performance degradation. Future work includes characterization of Mappers and Reducers during APARAPI's translation pass over the bytecode.

6.4 Discussion

As distributed systems become larger, more pervasive, and more heterogeneous, the experience and knowledge required to efficiently execute complex and critical applications to them has risen higher than what most application developers and domain experts are capable of. This change in high performance computing has led to a higher demand for high level distributed and heterogeneous programming models which hide hardware complexity from the user and allow tuning experts to manipulate platform configurations to optimize performance, energy efficiency, and reliability.

The strengths of Hadoop and OpenCL naturally complement each other. Hadoop provides a robust and proven distributed system with a MapReduce execution model and distributed file system. However, our tests show that its computational performance is lacking. OpenCL enables execution of Hadoop computation in native threads on heterogeneous high-performance, low-power architectures. APARAPI allows the seamless integration of these two prevalent programming models to provide a high performance distributed system with usability on par with Hadoop. While the experimental results for HadoopCL show significant overall performance improvement for computation-bound applications, some applications also show little or no improvement due to poor mapping of kernels to architectural features and no consideration for added latency to discrete devices when scheduling tasks. Future work could improve on both of these areas by bytecode analysis for kernel features such as memory access patterns, thread divergence, or computation-communication ratios.

Chapter 7

Related Work

Some related work was covered in previous discussions of CUDA, OpenCL, APARAPI, and OpenACC. This section focuses on more recent research.

GPUSs [33] is an extension of the StarSs programming model to support GPU execution. GPUSs, similar to OpenACC, is pragma based and has an API similar to OpenMP. It uses two pragmas in particular to denote functions which may be executed as tasks on CUDA devices. `#pragma css task` indicates that the annotated function can be executed as a parallel task, and `#pragma css target device(cuda)` indicates that the annotated task should be executed on a CUDA GPU. The GPUSs runtime uses three “actors” to execute an application. The master actor (or thread) executes user code and generates tasks where they have been identified by the previously described `#pragmas`. A helper thread consumes tasks when the GPUs in a system are idle, mapping those tasks to the optimal CUDA device based on data locality. A set of worker threads (one per GPU) receive tasks from the helper thread, perform the necessary transfers to the managed GPU, execute the CUDA kernel, and copy any results back. GPUSs is another example of a high-level, pragma based GPU programming model which provides even less transparency than OpenACC. It's extensive use of dedicated device management threads in the host would interfere with effective utilization of all hardware but ensures that all operations are performed asynchronously and all GPUs are fully utilized given sufficient work.

The work presented in EXOCHI [34] is divided into two parts. The Exoskeleton Sequencer (EXO) includes hardware support for a number of useful features in a shared-memory heterogeneous platform, including a MISP exoskeleton which allows interaction between a IA32 processor and non-IA32 accelerator using user-level interrupts, an Address Translation Remapping mechanism which has the IA32 processor in a heterogeneous processor handle page faults by proxy for an accelerator to support a shared virtual address space, and Collaborative Exception Handling which again uses an IA32 processor as a proxy for handling hardware exceptions on a non-IA32 accelerator. C for Heterogeneous Integration (CHI) provides a programming environment for EXO-like architectures by adding inline accelerator-specific computation, fork-join or producer-consumer style parallelism for sections of inline accelerator code, and a way to specify input/output/resident memory regions for accelerator code segments. CHI extends OpenMP to provide a familiar programming environment for heterogeneous, shared-memory processors. In the paper, performance improvements of up to 10.97× were demonstrated for highly-optimized media applications. This work is similar to OpenACC in its OpenMP-like interface, and supports multiple ISAs in a single binary. Due to the hardware support added by EXO, CHI is a high-performance and productive programming environment for the shared-memory heterogeneous accelerators it targets. It is unclear how applicable this type of programming environment would be for processors with discrete memory, as the latency of inter-processor communication would lead to unexpected overheads in such a tightly coupled programming model.

Merge [35] describes a programming model, compiler, and runtime built on EXOCHI that uses programmer annotations of functions to understand the target architecture as well as necessary conditions for correct execution. These architecture-specific implementations of common functions are often at different granularities, reflecting the architecture-specific optimal granularity for different processors. By filling a work queue with tasks that include metadata about supported architectures Merge can schedule available work on all processors in a heterogeneous platform and do so efficiently thanks to EXO's hardware support. Merge decomposes a MapReduce application into side-effect free C++ tasks for scheduling on the Merge runtime. Merge was evaluated on the same heterogeneous processors as EXOCHI as well as a 32-way Unisys SMP system and demonstrated up to 8.5× and 22× speedup on those platforms, respectively. Merge's main contributions are a higher-level MapReduce framework on top of EXOCHI and the dynamic selection from multiple architecture-specific implementations of the same function to achieve high utilization and well-performing mappings of tasks to architectures. This work shares many similarities with HadoopCL's MapReduce framework. One of the main differences is that it requires programmer implementation of architecture-specific functions while HadoopCL uses OpenCL's multi-architecture portability to automatically generate binaries for different architectures.

Rootbeer [36] is similar to APARAPI. It uses compile-time analysis to build a NVIDIA binary from Java bytecode. It also supports a wider range of Java language features than APARAPI, including exceptions and object types. Because it builds on CUDA, Rootbeer is specific to NVIDIA hardware. It shares many of the same strengths with APARAPI by providing a programming interface similar to Java Runnables and handling all memory management and kernel invocation for the developer.

X10 provides support for GPU execution [37]. GPUs are exposed as X10 places where X10 activities can be executed. The programmer writes 2 parallel-for loops to represent the thread groups in a single kernel invocation, exposing the chunked parallelism of CUDA through a familiar programming construct. X10 programmers must explicitly manage device memory through X10 APIs which wrap matching CUDA functions but also integrate with existing X10 features. For instance, the programmer must wrap asynchronous transfers in a finish scope to ensure completion. X10-CUDA also allows programmers to utilize GPU shared memory, barriers, texture memory, and constant memory. These are features which are hidden in most other heterogeneous programming models at a higher abstraction than CUDA and OpenCL, offering a better balance of performance and programmability than many other heterogeneous programming models.

Charm++ is a distributed, object-oriented, parallel programming model which was extended to support GPU execution[38]. Inter-object messages spawn new work on local or remote processors, associated with the receiving object. Because the programmer has no information on the current node or processor, Charm++ objects can be migrated across cores or nodes completely transparently by the Charm++ runtime. The work in [38] proposes a scheduling algorithm for heterogeneous and distributed systems using message priorities to schedule work on different processors. Messages whose processing is on the critical path of an application are given high priority and are greedily handled on the CPU. Lower priority messages are accumulated in a FIFO queue and executed in batches on GPUs. By building this scheduling algorithm on Charm++, the work in [38] supports a unified heterogeneous and distributed programming model. By efficiently aggregating messages received at Charm++ objects on a node, Charm++ can hide all knowledge of the executing hardware from the programmer and efficiently map computation to GPUs and CPUs using the hints provided by message priority.

Additionally, other research has been done on utilizing GPUs to accelerate MapReduce jobs.

Catanzaro et al. [18] did work on a single-node MapReduce Framework. While not built on Hadoop, it executes user-written Mapper and Reducer CUDA kernels on a GPU by auto-generating CUDA code at compile time based on user-specifications of Mapper and Reducer kernels. Intermediate results of the Map stage are stored in

GPU shared memory for partial reduction before flushing results to global memory, and expose some CUDA-specific configurable parameters for expert tuning. Future work includes auto-tuning of these parameters, such as thread group size. While this work is not distributed, some of the ideas could lead to future optimization of HadoopCL.

GPMR [19] is a high performance, standalone distributed MapReduce implementation in C++ and CUDA which uses the GPUs to execute Mappers, Reducers, and a number of intermediate stages added to the MapReduce pipeline in this work. These intermediate stages take advantage of the GPU's parallelism to minimize inter-node communication by performing preprocessing of outputs. Similar to HadoopCL, much of the work presented on GPMR is on optimizing or hiding communication. GPMR supports data sets that do not fit into a single GPU's memory. This work tries to expose many features of CUDA (such as local synchronization) to the application developer. GPMR also supports multiple GPUs per node with dedicated processes for each GPU. However, no application computation is executed on the CPU in GPMR which may lead to underutilized hardware.

Chapter 8

Discussion & Conclusions

This thesis presents four heterogeneous programming frameworks, each with the high-level goal of improving programmability of heterogeneous platforms while either maintaining or improving performance. This goal is accomplished by balancing architectural transparency with programming abstractions.

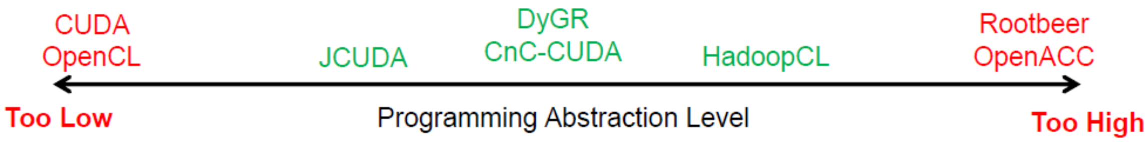
JCUDA is a lightweight Java wrapper for CUDA. JCUDA's intended audience is CUDA developers who are unfamiliar with or unwilling to work with JNI. It auto-generates bridge code between the JVM and CUDA devices that would otherwise add significant programming overhead for any Java developer using GPUs to accelerate computationally heavy sections of code. This bridge code handles the interface between Java and CUDA, as well as CUDA memory allocation, copies, and kernel invocation. JCUDA allows the application developer to write their own CUDA kernels and take advantage of many low-level CUDA optimizations necessary for optimal performance on GPUs. JCUDA supports the copying of multi-dimensional Java arrays, something which is still lacking in modern hybrid Java+CUDA programming frameworks. However, JCUDA does nothing to hide the CUDA programming model and presents essentially the same model to the user, limiting portability across future architectures.

CnC-CUDA builds on the auto-generation of JCUDA, but adds 1) a more general and useful programming abstraction, and 2) an optimized device management runtime which uses asynchronous operations to maximize utilization of all devices and bandwidth in a single node system. In CnC-CUDA, the user expresses a control and data flow graph of computational steps. Multiple parallel instances of these computational steps can exist at one time. In addition, a computational step can be designated as sequential or containing SIMD parallelism. Using these hints from the application developer, CnC-CUDA can execute different types of computational steps on optimal architectures without the application developer's involvement.

Rather than continuing the standard "host+accelerator" model which most if not all heterogeneous programming models make use of, DyGR uses GPUs as coprocessors of the CPU by allowing both devices to generate work for the other. Enabling dynamic task parallelism on the GPU through a hybrid work-sharing/work-stealing runtime system also improved the flexibility and generality of the GPU on irregular applications. While DyGR provides a simpler and more familiar task-based API to application developers, its true strength would be as a backend for higher-level programming models like JCUDA and CnC-CUDA.

Finally, building upon experiences in JCUDA, CnC-CUDA, and DyGR we come to HadoopCL. HadoopCL incorporates features of each to construct a heterogeneous, distributed programming model. By building on top of Hadoop MapReduce, HadoopCL automatically gains reliability guarantees, a distributed filesystem, and a programming model which balances flexibility for domain experts with optimization opportunities for tuning experts (similar to CnC-CUDA). HadoopCL uses APARAPI to automatically pull data-parallel Map and Reduce computation out of the JVM and into native, efficient, and high-performance OpenCL threads on every OpenCL device available in a system (similar to JCUDA's auto-generation). All OpenCL devices are viewed as equal coprocessors in the HadoopCL system (similar to the GPU runtime), but with specific optimizations applied to computation running on each so as to adapt to features of each architecture. An inter-device load-balancing algorithm is used to feed larger amounts of data-parallel computation to devices which can maintain higher computational throughput.

Each of the programming models or runtime systems presented positions itself at a different point between emphasizing programmability and performance. Arguably all of them lie somewhere between the low-level heterogeneous programming models (like CUDA and OpenCL) and higher-level models (like OpenACC or CUDA libraries). By striking a better balance between abstraction and transparency, these programming models enable programmers to be productive and produce high-performance applications on heterogeneous platforms.



However, despite research effort in heterogeneous programming models as part of this thesis, in other projects at Rice University, and at other institutions around the world, it is easy to argue that the problem of efficient development of realistically complex applications on real-world, distributed, heterogeneous systems is largely unsolved. Most production application development in industry, laboratories, and other major users of heterogeneous systems is still done through the low-level APIs provided by CUDA, OpenCL, Verilog, or a domain-specific library which wraps these APIs. The primary driver for this is the raw performance which these programming models provide. However, the overhead added in development and maintenance serves to significantly counteract the benefits. In addition, the generality and forward-compatibility of these programming models is questionable as we move towards greater heterogeneity. For instance, consider that simple mobile devices can now contain more than 10 different types of compute units in a single package. If HPC moves even somewhat in a similar direction, simple SIMD programming models will be insufficient to efficiently utilize all of the architectures available.

A more dataflow programming model similar to CnC may be the best candidate for future programming models. By expressing the main computational kernels of an application and the data dependencies between them, optimizations by the compiler, runtime, and even tuning experts are enabled so long as they do not violate the constraints of those dependencies. This also has a significant advantage in being future-proof. Expressing an application as its most atomic elements of computation and communication allows that application to be executed reasonably well on new architectures as they are developed. One problem with heterogeneous systems that was only touched on briefly in the discussion of HadoopCL was the worsening of I/O bottlenecks in heterogeneous systems as the same network infrastructure must now support drastically higher compute throughput. Dataflow programming models have the advantage of defining both computation and communication, allowing for optimization of both.

Heterogeneous systems are important to meeting future performance requirements while remaining power efficient. While there are many benefits to heterogeneous systems, the problem of efficiently building applications for them is still open. This thesis explores some opportunities for improvement in heterogeneous programming models and runtimes. We can conclude that while much progress has been made towards bringing heterogeneous systems to bear on challenging and computationally demanding applications, the techniques and technologies for developing those applications can still be improved. New dataflow, message-driven, or high-level frameworks (like MapReduce) are necessary to allow both flexible application development and efficient execution on future heterogeneous platforms.

Appendix A

DyGR

The following pages include code samples from DyGR applications.

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

    int num_types_of_tasks = 1;
    task dummy_tasks[num_types_of_tasks];
    param *dummy_params[num_types_of_tasks];
    dummy_params[0] = (param *)malloc(sizeof(param) * 1);
    dummy_params[0][0].size = sizeof(Node);
    dummy_tasks[0].p = dummy_params[0];
    dummy_tasks[0].num_params = 1;
    // Pre-allocate device memory and set up queues on device
    init_runtime(size, private_queue_length, num_types_of_tasks,
        number_of_tasks, dummy_tasks);

    // Launch runtime kernel on each device
    for(DEVICE = 0; DEVICE < num_valid_devices; DEVICE++) {
        SET_DEVICE(DEVICE)
        launch_kernel(c, d_out[DEVICE], size, n_insert,
            private_queue_length, THREADS_PER_BLOCK);
    }

    // Initialize root and children of root for UTS tree
    Node root; uts_initRoot(&root);
    Node *children = constructInitialUTSNodes(root, &numChildren);

    // For each child node, insert on device
    for(i = 0 ; i < numChildren; i++) {
        task t; param p[1]; t.params = p;
        t.type = 0; t.num_params = 1;
        p[0].size = sizeof(Node);
        p[0].ptr = children+i;
        p[0].type = IN;
        insert_task(&t);
    }

    // Wait for processing to complete
    finish_device();
}
```

Figure A.1 : Code snippets from the main entry point for a UTS implementation on top of DyGR

```
__device__ void uts_kernel(task *t, volatile task **heads,
    int *head_locks, int *task_counter, int tid, int blockSize) {
  \BlankLine
  // Get the UTS node for which we are generating
  // children from the executing task
  volatile Node *parent = (volatile Node *)((t->p)[0].ptr);
  \BlankLine
  int parentHeight = parent->height;
  \BlankLine
  int numChildren, childType;
  \BlankLine
  numChildren = d_uts_numChildren(parent);
  childType   = d_uts_childType(parent);
  \BlankLine
  // record number of children in parent
  parent->numChildren = numChildren;
  \BlankLine
  if (numChildren > 0) {
    int i, j;
    Node child;
    child.type = childType; child.height = parentHeight + 1;
    \BlankLine
    // Parallelize creation of child nodes across thread group
    for( i = tid; i < numChildren; i += blockSize) {
      for (j = 0; j < computeGranularity; j++) {
        // UTS-specific code
        d_rng_spawn(parent->state.state, child.state.state, i);
      }
      \BlankLine
      // Create a new task and set its
      // input to be the initialized child node
      task *tmp = alloc_task(0, heads, head_locks);
      tmp->type = 0;
      tmp->continuation = NULL;
      memcpy((tmp->p)[0].ptr, &child, sizeof(Node));
      \BlankLine
      // Make the new task eligible for execution
      push_task(tmp, task_counter);
    }
  }
}
```

Figure A.2 : The programmer-written GPU kernel for a DyGR UTS implementation

Bibliography

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ Home: an Experiment in Public-Resource Computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[2] Top500, "Top500." `http://top500.org/lists/2012/11/`, November 2012.

[3] UCLA, Rice, OSU, and UCSB, "Center for Domain-Specific Computing (CDSC)." `http://cdsc.ucla.edu`.

[4] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 73–82, ACM, 2008.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, (Washington, DC, USA), p. 47, IEEE Computer Society, 2004.

[6] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[7] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *Parallel & Distributed Processing, 2009.*

IPDPS 2009. IEEE International Symposium on, pp. 1–8, IEEE, 2009.

[8] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-aware high performance computing with graphic processing units," in *Workshop on power aware computing and system*, 2008.

[9] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, (Berlin, Heidelberg), pp. 887–899, Springer-Verlag, 2009.

[10] "Benchmarks: Measuring GP (GPU/APU) Cache and Memory Latencies." `http://www.sisoftware.net/?d=qa&f=gpu_mem_latency`.

[11] Vivek Sarkar, "Rice University COMP 322 Lecture 1," p. 11, Spring 2013.

[12] Khronos OpenCL Working Group, "The OpenCL Specification Version 1.1 Document Revision 44," 6/1/2011.

[13] OpenACC Standards Group, "OpenACC Application Program Interface." `http://www.openacc-standard.org/`.

[14] Gary Frost, "APARAPI in AMD Developer Website." `http://developer.amd.com/tools/heterogeneous-computing/aparapi/`.

[15] "OpenJDK Project Sumatra." `http://openjdk.java.net/projects/sumatra/`.

[16] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, "GPU clusters for high-performance computing," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–8, IEEE, 2009.

[17] C.-T. Yang, C.-L. Huang, and C.-F. Lin, “Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters,” *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, 2011.

[18] B. Catanzaro, N. Sundaram, and K. K., “A Map Reduce Framework for Programming Graphics Processors,” in *Workshop on Software Tools for MultiCore Systems*, 2008.

[19] J. Stuart and J. Owens, “Multi-GPU MapReduce on GPU Clusters,” in *Parallel & Distributed Processing Symposium*, 2011.

[20] N. Nystrom, M. R. Clarkson, and A. C. Myers, “Polyglot: An Extensible Compiler Framework for Java,” in *Compiler Construction*, vol. 2662 of *Lecture Notes in Computer Science*, pp. 138–152, Springer Berlin / Heidelberg, 2003.

[21] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, “Java Programming for High-Performance Numerical Computing,” *IBM Systems Journal*, vol. 39, no. 1, pp. 21–56, 2000.

[22] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, “Benchmarking Java Against C and Fortran for Scientific Applications,” in *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pp. 97–105, ACM, 2001.

[23] L. A. Smith, J. M. Bull, and J. Obdrzálek, “A Parallel Java Grande Benchmark Suite,” in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pp. 8–8, ACM, 2001.

[24] R. Barik, Z. Budimlić, V. Cavé, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, “The Habanero multicore software research project,” in *Proceeding of the 24th ACM SIGPLAN conference*

companion on Object oriented programming systems languages and applications, OOPSLA '09, (New York, NY, USA), pp. 735–736, ACM, 2009.

[25] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[26] "The Java Grande Forum benchmark suite." http://www.epcc.ed.ac.uk/javagrande.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[28] P. Charles *et al.*, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA*, (NY, USA), pp. 519–538, 2005.

[29] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*, pp. 235–250, Springer, 2007.

[30] A. Duran *et al.*, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *ICPP'09*, pp. 124–131, 2009.

[31] D. Cederman and P. Tsigas, "GPU-Quicksort: A practical Quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, January 2010.

[32] M. Lastras-Montano, M. Michael, and J. Bivens, "Dynamic work scheduling for

gpu systems," in *International Workshop of GPUs and Scientific Applications, GPUScA*, 2010.

[33] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par 2009 Parallel Processing*, pp. 851–862, Springer, 2009.

[34] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, 2007.

[35] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 287–296, ACM, 2008.

[36] P. Pratt-Szeliga, J. Fawcett, and R. Welch, "Rootbeer: Seamlessly Using GPUs from Java," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 375–380, 2012.

[37] R. B. Cunningham, Dave and V. Saraswat, "GPU Programming in a High Level Language," in *Proceedings of the ACM SIGPLAN X10'11 Workshop*, 2011.

[38] J. Lifflander, G. Evans, A. Arya, and L. Kale, "Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 2404–2413, 2012.