HYDRA: A CUDA based tool supporting error-resilient GPU computation and general purpose heterogeneous programming

by
Gregory Becker

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 1, 2015

Abstract

The large-scale supercomputers popular in high performance computing require support for error-resiliency to protect against soft errors. We present Hydra, a scalable, CUDA-based tool supporting resilience in heterogeneous GPU/CPU applications. Hydra provides a framework to support redundancy for resilience, as well as programming tools that leverage that framework for other heterogeneous programming tasks. Hydra executes CUDA kernels redundantly on both CPU and GPU to leverage the parallelism of the heterogeneous architecture. To support redundancy with minimal additional programmer effort, Hydra provides wrappers for the CUDA functions that transfer data between GPU and CPU. Our results indicate that Hydra redundancy can be an efficient model for error-resilience in heterogeneous architectures, as well as a useful tool in heterogeneous programming.

Acknowledgments

I would like to thank my adviser, Duane Bailey, for his constant help and support with this project. It would not have been possible without him. I would also like to thank my second reader, Todd Gamblin, for his help on this project and for his mentorship as I enter the world of High Performance Computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary contribution of this work is Hydra. Hydra is a CUDA-based tool that supports both redundancy and parallelism in computing environments that contain GPU(s). Hydra is flexible enough to allow the programmer to specify the particular redundancy or parallelism between CPUs and GPUs for individual computations, while maintaining defaults that allow any existing CUDA code to be easily instrumented for redundant GPU computation. In addition to its framework for redundancy, Hydra provides two modes for general heterogeneous parallel programming that can be tailored by the more advanced user to make use of an almost infinite array of resource management options.

1.1 General Purpose GPU Computation and Heterogeneous Architectures

The supercomputer of the distant past was a single, vector-architecture central processing unit (CPU) and the memory to go with it. The supercomputer of the more recent past is a massive machine, containing thousands, or even millions, of processors, interconnected as closely as possible, to bring the maximum computing power possible to a problem. These machines can execute trillions or even quadrillions of floating-point operations per second, leading to the term "petascale computing." These machines draw millions of dollars worth of electricity and can occupy entire buildings, despite the architects' best efforts to make them smaller and more efficient. This makes drastic improvements like the push to exascale computing (quintillions of operations per second) prohibitively expensive.

One attempt to improve the speed and efficiency of modern supercomputers is to add accelerators, or graphics processing units, to each of the nodes. Graphics processing units (GPUs) naturally work on problems which operate independently on each element of an array, similar to their native use operating on arrays of pixels. This type of parallelism is very common in scientific applications, particularly matrix operations. Thus the supercomputer of the present may have hundreds of thousands of nodes, each containing many CPU cores and one or several GPUs. On

these machines tasks with regular data access are done on the GPU, while other tasks, including maintenance of program state, are done on the CPU. Many of the fastest computers in the world now use heterogeneous architectures of this sort.

On heterogeneous architectures the GPU becomes another resource that must be managed. This is a cause of some overhead: GPUs are more difficult to program than CPUs, and they do not share a memory space with the CPUs. This programming difficulty is a barrier to effective general purpose GPU computation.

1.2 Soft Errors in HPC-Scale Computation

One aspect of managing the GPU as a computing resource is ensuring the accuracy of GPU computations. In our day-to-day lives, we assume that our computers are error-free. However, as the number of cores in a computer increases, hardware error rate becomes a significant concern, particularly in long running applications. This is a simple matter of probability; the more components there are in a machine, the more likely it is that one of them will fail. Increasing error rates for extreme scale machines make resiliency a particularly important area of research for high-performance computing (HPC), when applications often take days to run on tens or hundreds of thousands of cores. Currently, memory in supercomputers is protected from bit errors by error correcting codes (ECC), but it is harder to design resiliency for errors in computation.

Computation is less susceptible to error than memory since values persist in memory much longer than in computation. However, error rates in GPGPU computation are much less well studied than in standard computation, and there is some concern that GPU architectures may be particularly susceptible to transient hardware errors, or soft errors. A single GPU consists of many lightweight computation cores, each of which is independently susceptible to errors.

There are three categories of soft errors which we will discuss. (1) *Fatal errors* cause the program to terminate, for example due to a segmentation fault. These are obviously problematic, but easy to identify. (2) Other errors may cause a *hung program*, or enter an infinite loop. From the halting problem we know that these are impossible to identify perfectly, but in practice we can identify them by the running time and output of the application. (3) The third type of errors are *silent errors*. These errors manifest themselves only in incorrect computation results, and cannot be identified by examining the program logic. This makes them very hard to identify without special protocols. Any system of protecting GPGPU and heterogeneous computation should protect against all three types of faults.

Recovering from a fatal error obviously involves some form of redundancy—either we identify and correct the error as it happens, by comparison to a redundant source, or we redo calculation after the fatal error, which is simply temporally displaced redundancy. Similarly for errors that cause a hung program. This suggests redundancy as a solution to soft errors, as the errors themselves require redundant computation if we do nothing.

1.3 The GPU as a Source of Redundancy

GPGPU computations may be a particularly apt application of redundant computation for multiple reasons. First, because the GPU does not share an address space with CPU, the two naturally work on separate problems. It could then be natural to run a 3-way redundant application once on the CPU and twice on the faster GPU, or some similar scheme to take advantage of the natural parallelism between processors on a heterogeneous architecture. While it is possible to write applications such that the CPU and GPU are engaged in computation at the same time, it is very difficult. Many applications, therefore, have only one processing unit active at a time. For these applications, redundant computation between the GPU and the CPU may be possible with minimal effects on performance. Additionally, GPU architectures vary significantly from CPU architectures in that the computation does not necessarily involve the entire GPU. This suggests the possibility of running small redundant GPU computations at the same time on the same chip, which would be "free" redundancy in that it would not affect performance.

1.4 Similarities between Redundancy and Parallelism

In some circumstances, we may decide that the risk of soft errors is negligible or irrelevant. For example, smaller, shorter applications are less likely to encounter an error, and an application from which an approximate result is sufficient may not require the same type of protection against silent errors. For these programs, the apparatus of redundancy may still prove useful. At its core, redundant computation is a special case of parallel computation—some work is done here, other work there, and the results are combined in some way. Since all redundancy is an example of embarrassing parallelism, the framework that supports redundant computation can be generalized to support parallelization of embarrassingly parallel tasks. The data may not be spread to every process equally, and the combination mechanism may not be a voting scheme, but the rest of the framework is the same for concatenation of arrays acted on in parallel as it is for redundancy.

1.5 In this Work

The primary goal of this work is to make heterogeneous computation error-resilient through redundancy. Hydra redundancy should protect against faults and silent bit errors resulting from soft errors in the GPU. A secondary goal is to generalize the framework that supports redundancy to more general heterogeneous programming. For both tasks we have twin objectives of expressibility and performance. A good parallel programming model should both accurately express the parallelism of a problem and extract that parallelism in the form of efficiency.

In Chapter 2 we will discuss previous work related to transient errors, redundancy, CUDA, and heterogeneous programming, without which the Hydra project would have been impossible. Chapter 3 will provide an in depth look at the architecture of Hydra, in particular looking at its

model for expressing redundant computation. Chapter 4 will discuss how Hydra uses the framework of redundant computation to effectively express other types of parallelism. Chapter 5 will evaluate the performance of Hydra-based redundant computation on HPC benchmarks. Chapter 6 will discuss the heterogeneous programming models provided by Hydra and examine their performance. Chapter 7 will discuss possible future work. Chapter 8 will provide our conclusions. Appendix A provides additional information on possible future work involving transactional memory.

Chapter 2

Related Work

2.1 Evaluating the Impact of Soft Errors in Scientific Computing

While there is very little work on error rates in GPGPU computations specifically, there is a wide breadth of research available on the impact of soft errors on scientific applications in general. Much of this work is fully relevant in a GPGPU context.

As error rates first rose to become an area of concern in HPC, some suggested that they would be irrelevant for the iterative methods most common in scientific applications, as they would effect only the convergence rate of those methods and not the accuracy of the final result. However, Shantharam et al. [15] showed that errors propagate quickly enough through iterative methods often to compromise them entirely in the face of silent bit errors. They both proved and demonstrated a relationship between the sparsity of a matrix and the speed with which errors propagate through sparse matrix multiplication. Consider the sparse matrix as a graph in which the non-zero values are the nodes and two nodes are adjacent if they share a row or a column. Then the depth of that graph under breadth first search is the number of iterations until the entire result vector is corrupted by the original error. This makes explicit methods highly vulnerable to soft errors. Shantharam et al. were able to show that implicit methods were slightly more robust, as implicit methods were affected only in run-time as some had hypothesized, in contrast to the erroneous results produced by explicit methods. However, in their empirical testing, the convergence times of implicit iterative methods in the presence of error injection was often as much as 200 times longer than without error injection, a slowdown which is unacceptable in an HPC environment.

Li et al. [12] studied the effects of error injection on scientific applications, with a focus on how the characteristics of an error change its effect on application state. Rather than changing values in software to reflect errors as Shantharam et al. did [15], Li et al. built a low-level tool to inject bit errors, called BIFIT. BIFIT can inject a fault into any memory object in stack, heap, or global memory regions. BIFIT can also be used to inject a fault at any point in the program's execution to instruction granularity. To minimize overhead, BIFIT includes a profiler that collects memory access information for the application so that it doesn't track global variables not touched by the

application. The memory access data is also used to detect when a proposed error injection logically cannot propagate, and reverts to the uninstrumented version of the application after injecting the fault in those cases. The fault injector itself then tracks all relevant memory regions. When a fault injection is specified, it finds the relevant memory region and randomly flips a single bit in that region. Li et al. also used BIFIT to analyze the effects on application correctness of errors to different parts of program state. They then classified the data structures in three common scientific applications by the susceptibility of the overall application to bit errors in data of that type. Particularly noteworthy was the result that bit errors in small variables more often cause the program to abort, and when they do not cause the program to abort cause larger differences in output than errors in larger variables do. Furthermore, applications are unsurprisingly susceptible to the timing of bit error. Counter-intuitively, the later an error occurs the more it is likely to effect the outcome of the computation.

Bronevetsky and de Supinski [1] also studied the effects of errors specifically on iterative methods, although they restricted themselves even more closely to the linear algebra methods that make up the vast majority of scientific computing applications. They injected errors into a selection of iterative linear algebra methods and computed the percentage of soft errors leading to the main problem areas: program aborts, hung computation (which they defined as computation at least ten times longer than error free computation), and silent data corruptions. Using these values they were able to determine that an application running on one thousand cores for one day would have a 2.4% chance of giving corrupted output if each core uses 100MB of RAM on average. Bronevetsky and de Supinski recognize this rate as a serious problem, and propose a variety of software error detection methods. Their methods include some based on residual tracking and some based on encoding program state, and are all designed to be implemented in a checkpoint-restart system for error tolerance. The encoding based methods are generally more robust, but they require a significantly higher overhead. Bronevetsky and de Supinski calculate theoretical overhead and error rates for each of their methods. The lowest predicted silent error rate is 0.0001% with an overhead of 76%; this is from an algorithm based method testing every $n^{\frac{2}{3}}$ where n is the error-free run-time. The lowest overhead for an effective method is 3.4% while reducing the silent error rate to 0.034%; this is from a residual tracking method that triggers a restart if the residual is over m times greater than the previous, for some parameter m. The quoted values come from using a high m value. These methods reduce the chances of an error but do not eliminate them entirely, and introduce extra computation due to false positives.

These studies show that transient errors are a problem in need of solution. However, none provides a perfect solution to the problem, especially in the context of GPGPU computation.

2.2 Error Detection and Correction via Redundancy

At first glance, redundant computation might appear to be a naive approach to resiliency because of its very high implied overhead. However, as we discussed in the introduction, all resiliency schemes

involve some level of redundancy. Additionally, Ferreira et al. [3] suggest that as the number of cores in supercomputers increase, explicit redundancy may be the way of the future. The current de-facto standard in large-scale scientific computing is to use a checkpoint-restart scheme, which protects against errors that cause application aborts. Checkpoint-restart creates a very large I/O bottleneck, as the application is required to store its entire state with some frequency, and most large-scale machines are not provisioned to write out their entire memory quickly; such provisioning would be prohibitively expensive. In addition, as the error rate increases, the percentage of computer resources that are spent on computation that is not rolled back decreases. Ferreira et al. model application efficiency with redundant computation and without, and determine that under current hardware resiliency, computation becomes more efficient with redundant nodes because restarts are much less common. They identify approximately 20,000 nodes as the point at which checkpoint-restart falls below 50% effective resource utilization and redundant computation increases efficiency. In addition, they build rMPI, an interface that implements MPI while maintaining synchronicity between redundant nodes. rMPI has two coherency protocols to monitor that each replica is sending and receiving the same messages (and consequently that all replicas are still working). In the mirror protocol, each replica of the sending rank sends a copy of its message to each replica of the receiving rank. This makes the comparison easy. In the parallel protocol, the replicas of the sending rank communicate to ensure that each is sending the same message, and then each replica of the sending rank sends to exactly one replica of the receiving rank. In the case that one replica has failed, another replica sends a second message to ensure that all replicas of the receiving rank receive the message. Using rMPI, Ferreira et al. are able to determine the software overhead in the efficiency of redundant computation. Using this result and given the range of failure rates likely for hardware in the near future, they conclude that redundancy is very likely to improve efficiency for early exascale machines, and most likely for the majority of cutting edge high performance clusters going forward.

Fiala et al. [4] bring into focus the other benefit of redundancy for detecting and correcting errors: it can also be used to detect and correct silent bit errors which are not detected by checkpoint-restart. They extend rMPI to create redMPI, which compares the messages between replicas so as to detect and correct errors in incoming messages, using three-way or higher replication. In RedMPI's primary operation mode, each replica of the sending rank sends only one full message, and sends a hash of that message to another replica of the receiving rank. The full messages only need to be compared when the hash values differ for one or more of the replicas of the receiving rank. The error is then effectively quarantined in a single process, as any message to another process is corrected before receipt. Under the assumption that all data must eventually be passed as a message, this system requires a restart only when two of three (or equivalently for greater redundancies) processes have errors, and otherwise can correct MPI messages so that all computation results are correct. This has the additional benefit of naturally checking only data that is still "live" in the program. For most programs tested RedMPI adds barely over 3 times overhead for 3 way redundancy, putting it in line with the numbers Ferreira et al. used when analyzing

whether redundancy was a useful way to detect and correct errors [3]. Between the two works it has been shown that redundancy can be an effective method for resiliency in HPC environments.

Hydra is modeled on these studies in multiple ways. First, we know from Ferreira et al. that redundancy is the way of the future for resiliency in supercomputers. More specifically, the argument on which both studies is based is that in multiprocessor applications, all information must be transferred between processors at some point. We adapt that same idea to heterogeneous architectures, in that all GPU computation results must at some point be passed back to the CPU. Thus all GPU computation can be tested for resiliency purposes in the GPU, which we implement in Hydra.

2.3 Executing General Purpose Computation on GPU Hardware

A heterogeneous and redundant programming model for GPU and CPU requires tools to program for both CPU and GPU. Hydra accomplishes this using CUDA code which is designed to run on the GPU but can also be efficiently compiled for the CPU.

GPGPU computation requires a programming model that specifies scientific-application-style tasks in a format that can be executed by a GPU. For this purpose NVIDIA created CUDA in 2007 [13]. To make use of CUDA's functionality, one writes a serial program that can call parallel "kernels", coarse-grained independent tasks that can be transparently scheduled to the GPU architecture by CUDA. For each kernel, the programmer specifies the number of threads, thread blocks, and the grid, which allow CUDA to schedule the threads efficiently. Because of this paradigm CUDA allows the programmer to specify a C or C++ program for one thread and the hierarchy in which that code will be applied to other threads and blocks of threads. The space of threads is divided into thread blocks which cannot be interdependent in any way. Within each block, threads are automatically grouped into warps. Blocks can be explicitly synchronized for consistency, but every thread within a warp must execute conditional operations identically. This allows each warp to map to one hardware unit of the GPU, each of which can execute a single instruction per warp per cycle. CUDA does not allow for direct memory access from the GPU to CPU, so memory must be explicitly allocated and copied to and from GPU for each kernel operation. Since the GPU memory is non-caching, this memory manipulation must be done with some care for maximum efficiency. NVIDIA reports that real world applications, when re-written for the Tesla architecture using CUDA, have seen speedups between 10 and 263 times over CPU computation. The general process for converting CPU code to GPU is to identify parallel loops with uniform access to consecutive data elements. Each loop then becomes a kernel operation, with each thread x operating on the x^{th} element of the data. The code is therefore easy both to read and understand, which is essential for immediate adoption by the scientific community at large [13].

Translating scientific applications is an involved task. To resolve this, many have attempted to create a unified programming model for CPU and GPU computation. Among these attempts is MCUDA, [17] a tool from the University of Illinois at Urbana-Champaign which compiles CUDA

code into an OpenMP[1] application for multi-threaded CPU. Each kernel invocation is translated into a "parallel for" structure in OpenMP to extract the inherent data parallelism. Each thread block is then executed on an individual thread, which makes maximal use of data locality and minimizes synchronization, since thread blocks must be fully independent. Additional work is then done to split loops at every synchronization point. This process, called "loop fission," ensures that every logical thread of the CUDA program executes in OpenMP to the synchronization point before any continues beyond the barrier. In addition to synchronization points, MCUDA uses loop fission at every side entrance or side exit from the thread loop. Each logical thread of the CUDA program also has its own variables. MCUDA buffers these variables in OpenMP so that each thread can act on its variables independently between synchronization points. For efficiency, MCUDA only buffers the variables whose live range extends beyond a single thread loop, since variables defined and de-allocated within a thread loop can be reused on the next iteration without issue. With each thread block translated in this way, block functions can be assigned arbitrarily to OS threads to take full advantage of the block level parallelism specified by CUDA. For every application studied by Stratton et al. (2008), the worst performance was within 30% of the performance of the most optimized available version for CPU and for some applications the performance followed that of the most optimized version closely. However, this is for applications tuned specifically for CPU, so some re-tuning is still necessary to convert CUDA applications to OpenMP.

Translating in the opposite direction, Lee et al. [11] propose a system for converting OpenMP code to run on general purpose GPU (GPGPU). They chose OpenMP as the programming model from which to translate because of the natural way in which it expresses data level parallelism, causing it to map more easily to GPGPU. They implemented their translation in two stages. Since GPGPUs are stream processors, the first stage is to translate the OpenMP code to optimize for operation on data streams. The second stage is to translate the streamed OpenMP code to GPGPU code. Every `#pragma omp parallel` section of the OpenMP code is identified as a potential kernel region and every work sharing construct of the OpenMP code (`#pragma omp parallel for`, `#pragma omp parallel section`) is parallelized among the threads of the GPGPU, where each iteration or section is mapped to its own thread. Potential kernel regions are split at each synchronization point, since there is an implicit synchronization between kernel invocations. In addition, Lee et al. use a variety of compiler optimizations to improve the resulting GPGPU code. The most important are parallel loop swap for regular applications and loop collapsing for irregular applications. For applications with regular data access patterns, loop swapping can improve inter-thread locality by replacing many passes over memory by long strides with one pass by short, ideally monotone, strides. This makes the memory accesses map more efficiently to the fine-threading of GPGPU calculation. For applications with irregular data access patterns, loop collapsing can make it possible to access memory continuously in cases where the original loop does not appear to have continuous memory accesses. This is important because the fine-threading of GPGPU applications requires regular memory accesses. This can allow potential kernel regions to be parallelized that

[1] OpenMP is an environment for easily creating pools of threads to execute natural parallelism.

otherwise may have been difficult to convert for GPGPU code.

NVIDIA created their own cross-compilation tool for CUDA and OpenMP, and recruited some of the people who worked on MCUDA, including Stratton, to create the tool. Stratton et al. (2010) [16] created a tool in the `nvcc` compiler to compile CUDA code for OpenMP applications. This system is different from MCUDA in a variety of important ways. Among the interesting differences is that each kernel region in the `nvcc` compiled code has a single loop, with the loop fission handled by a series of goto statements and restart points. Because this is implemented in a production compiler, there are also a variety of compiler optimizations available for cross-compilation of OpenMP code using NVIDIA's `nvcc` compiler. One of the important optimizations is variance analysis; this is a static analysis which identifies when the thread identifier is irrelevant to the result of the computation. This is useful because CUDA codes often compute some values in parallel for simplicity in data management, while making those computations more than once in OpenMP is inefficient. Another key optimization is adaptive loop nesting. Adaptive loop nesting evaluates permutations of loop fission, loop interchange, and loop invariant removal to find the most efficient computation which avoids redundancy. For most of the applications analyzed, C code compiled from CUDA using the `nvcc` compiler runs in at most twice the time of native C code and sometimes gives a performance benefit. These tests were all done on single threaded programs, but it is reasonable to expect code compiled from CUDA to scale well compared to native code because CUDA naturally expresses data parallelism.

In addition to these attempts to compile between CUDA and OpenMP code, there have been other efforts to convert code between CPU and GPGPU computation. Hong et al. [9] provided a framework for porting MapReduce programs between CPU and GPU. This framework allows programmers to express their computation only once in a platform agnostic manner and to run the resulting code on either CPU or GPU.

Hydra is designed to be agnostic to which of these projects it interfaces with. That means Hydra can be packaged with any cross compilation tool that can compile CUDA code for execution by both CUDA-compatible GPU and multi-core CPU processing.

2.4 Heterogeneous Computing

Heterogeneous computing uses the combined computing power of two distinct architectures, usually GPU and CPU. This introduces distinct challenges to programming, such as the disjoint memory spaces discussed above. CUDA is an example of a tool for heterogeneous computation. In CUDA, every piece of computation is either done in CPU or, through the invocation of a kernel, in GPU. Therefore CUDA is most efficient for applications in which the portions of the application suitable for GPU computation are not tightly pipelined, so that the CPU does not idly wait for the kernels it invokes to return. Some applications are not natural to express in this form, and for those we would desire direct heterogeneous computation. By direct heterogeneous computation, we mean concurrent computation of a single task in both the GPU and the CPU.

Garland et al. [6] provided a unified programming model for heterogeneous systems called Phalanx. Phalanx uses a PGAS-like global address memory space via the GASnet runtime. (GASnet allows memory to be accessed uniformly from anywhere in the machine.) Phalanx also makes use of OpenMP and CUDA to run on CPU and GPU nodes. Programs in Phalanx consist of some number of independent tasks, which can be scheduled by the programmer to any node of the machine. It is then the programmer's responsibility to decide both the type and location of resource on which it is most efficient to execute a given task. Alternatively, tasks can be scheduled on a portion of the machine, with the Phalanx run-time environment determining which specific hardware within that range executes the task. Tasks are scheduled using a `Place` construct. Each `Place` object is a machine tree in which the leaves are computation cores and memory banks. A programmer can access the tree by traversing from the root node or by accessing slices of the tree according to the type of resource desired. Each task in Phalanx is assigned a policy, either "streaming_group" or "parallel_group". Generally streaming_group tasks efficiently map to GPU compute nodes, with fully independent subgroups, while parallel_group tasks generally map to CPU compute nodes using OpenMP, allowing for synchronization between sub-tasks. Tasks resemble CUDA kernels in that each must be fully autonomous, with the exception that dependent tasks can be scheduled to begin as soon as some antecedent task terminates. This allows for coarse-grained communication between sub-tasks as all results from the first task will be stored before any computation of the second task begins. Phalanx code is executed by either the OpenMP back-end, the CUDA back-end, or the GASnet back-end, depending on the type of resource specified or the type of task (if the programmer does not specify). Garland et al. also demonstrate the Phalanx system effectively scales well to at least 512 cores. Efficiency comparisons between Phalanx and other programming models for scientific applications remain a concern.

In addition to concerns about efficiency, Phalanx is unnecessarily general for many heterogeneous computation needs. The ability to schedule computation to specific nodes in the machine tree may be useful for some particularly susceptible problems, it is not generally relevant to the programmer. Phalanx also loses the ability to express parallelism in CUDA's idiom, which Stratton et al. have shown is a particularly efficient way to model parallelism [17].

There are also existing tools that have not been applied to direct heterogeneous computation, but whose programming model may provide hints as to effective methods of direct heterogeneous programming. Such tools include Ocelot, designed at Georgia Tech University [2]. Ocelot is a more CUDA-like tool than Phalanx–computation is done by kernels that can be executed on either GPU or CPU. CUDA programs are compiled to a virtual instruction set called PTX. Ocelot sits between the PTX instructions and the machine and translates PTX instructions for execution on the CPU. PTX instructions that naturally map to an instruction of the targeted architecture can be translated trivially at the granularity of a single CUDA execution warp (set of threads). PTX instructions that do not naturally map to an instruction of the targeted architecture can be emulated in software. This is particularly difficult when targeting SIMD architectures that do not support CUDA's arbitrary control flow. Ocelot manages this using dynamic warp formation as outlined in

Fung [5]. Ocelot optimistically groups threads into warps, but if a control flow condition separates the program counter among threads of a given warp, those threads are split into two or more separate warps. Dynamic warp formation is handled in the runtime software. Other instructions that do not necessarily map naturally to the targeted architecture include those accessing GPU-specific data structures and special registers. Shared GPU memory can be emulated by a designated region of shared memory or using global memory to share across threads, and special registers can be emulated by designated memory locations. Ocelot uses methods adapted from MCUDA [17] to handle GPU threads exceeding the capacity of the targeted architecture. Ocelot is implemented by intercepting CUDA API calls and wrapping the calls with Ocelot specific code when an architecture other than NVIDIA GPU is targeted.

Although translations to multiple architectures at the same time may be possible under Ocelot's model, no such attempt at direct heterogeneous computation has been made. In this Ocelot is more similar to MCUDA than to a direct heterogeneous programming tool such as Phalanx.

2.5 Summary

GPGPU computation is an effective expression of parallelism, and CUDA applications for GPGPU computation and can speedup application performance many times over sequential code. However, as with any form of parallelism, it comes with programming challenges. It also raises the risk of soft errors. Programming tools must be designed to ease the challenge of heterogeneous programming. Furthermore, there is a trade-off between the efficiency of GPGPU computation over CPU computation and the susceptibility of the GPU to soft errors. Without any error-checking scheme, soft errors can render the results of scientific computations inaccurate to the point of uselessness or even prevent computation. Fortunately, redundancy has been shown to be an effective method for mitigating the effects of soft errors with minimal negative effect on application performance.

Chapter 3

The Hydra Architecture

The purpose of Hydra is to achieve error-resilient GPGPU computation. As we have seen in Chapter 2, redundancy is an efficient method for error-resiliency. Hydra replaces each CUDA "kernel" of GPU computation with redundant computations, each of which can be executed on the GPU or the CPU. Redundant computation can therefore be spread over the available resources for performance tailored to the specific computation and application. To support redundant kernel computations, Hydra functions implement wrappers of a subset of CUDA functions. These wrappers support the memory operations necessary for redundant computation, and are designed to minimize the effort necessary to instrument existing CUDA codes for redundancy. These wrappers involve C++ templating.

3.1 Mapping CUDA Computation to Hydra

To understand Hydra, it is first necessary to understand the basic structure of a CUDA GPU computation. Because the GPU and CPU memory spaces are completely disjoint, several steps are necessary in any CUDA GPU computation. First, memory must be allocated in the GPU memory space. Second, data must be copied onto the GPU. Third, a CUDA kernel can be executed as discussed in Chapter 2. Fourth, the results must be copied back to the CPU. Fifth, the GPU pointer can now be freed. Figure 3.1 shows the memory control structures surrounding a kernel invocation. Each of these actions will need to be modified by Hydra to implement redundant computation.

3.2 The `Handle` as Standard Datatype

Implementing redundant GPU computation by hand is a tedious process, and the resulting code is much more difficult to reason about than the original. We will therefore automate the redundancy; one of our goals is to minimize the the work necessary to instrument existing CUDA code for redundancy using Hydra. To accomplish this, the programmer must be given some manner in

```
1    float* xg;
2    float* yg;
3    cudaMalloc(&xg,bsize);
4    cudaMalloc(&yg,bsize);
5
6    cudaMemcpy(xg,x,bsize,cudaMemcpyHostToDevice);
7    cudaMemcpy(yg,y,bsize,cudaMemcpyHostToDevice);
8
9    saxpy<<<dimGrid,dimBlock>>>(a,xg,yg);
10
11   cudaMemcpy(z,yg,bsize,cudaMemcpyDeviceToHost);
12
13   cudaFree(xg);
14   cudaFree(yg);
```

Figure 3.1: An example of CUDA code from the SAXPY application, showing the memory control functions surrounding the kernel invocation. The kernel is defined elsewhere in the program.

which to access multiple replicas of a CUDA computation analogous to how one would access a single computation in a CUDA application. Hydra implements an opaque datatype called a `Handle`. Every time the CUDA programmer would create a pointer on the GPU using `cudaMalloc()`, the Hydra programmer calls `hydraMalloc`. The `hydraMalloc` function allocates a `Handle` structure and returns a pointer to it. The `Handle` is implemented opaquely to the programmer—that is, if the programmer would have used an integer pointer in CUDA, he or she uses an integer pointer in Hydra, and at no point does the programmer need to know that the pointer is in fact a `Handle`. The `Handle` in turn points to a `Context` and an array of integer pointers. The `Context` consists of the number of GPUs and the number of CPUs to use for this computation. For each GPU used in the computation, there is a pointer in the array that points into the GPU memory space, and for each CPU used in the computation there is a pointer in the array that points into the CPU memory space. Figure 3.2 shows the logical image of a `Handle` with a `Context` of 5 GPUs and 4 CPUs.

When the CUDA programmer would move data into the GPU using

`cudaMemcpy(dest,src,size,cudaMemcpyHostToDevice),`

the Hydra programmer calls

`hydraMemcpy(dest,src,size,cudaMemcpyHostToDevice).`

The latter function merely copies the `size` bytes of data in `src` into every pointer in the array of pointers pointed to by the `Handle dest`.

The `Handle` datatype makes redundant freeing similarly simple to implement, as the `hydraFree()` function wraps the `cudaFree()` function and frees each of the pointers the `Handle` points to, the `Context` it points to, and the `Handle` itself.

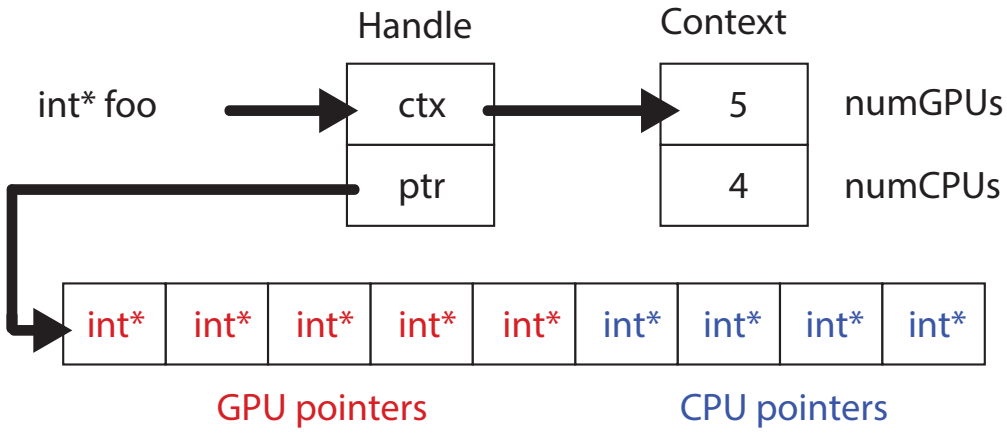

Figure 3.2: An overview of the Hydra `Handle` datatype.

The other two actions discussed earlier, kernel invocation and returning data to the CPU memory space, are more complicated.

3.3 Kernel Invocation

The variety of the kernels that can be called in CUDA or Hydra provide a challenge to opaquely implementing kernel invocation in Hydra. We could resolve this challenge in one of two ways.

The satisfying resolution is to create a variadic function `kernelInvoke()` which takes as its arguments the kernel name, the parameters of the GPU computation (block size, grid size, etc.), and the arguments of the kernel named. This requires variadic templating, as the kernel requires arguments of the specified type. This single function then invokes the kernel in both GPU and CPU the correct number of times, accessing the `Context` through the `Handles` of the arguments passed to the kernel. Within this function, invocations the kernel on the CPU are parallelized using OpenMP. Invocations of the kernel on the GPU need not be parallelized, as all GPU kernel invocations are asynchronous, so whenever there is sufficient space on the GPU the kernels will naturally be parallelized. Unfortunately, CUDA does not yet support variadic templates, which are a new feature as of C++11, so for now this solution is unfeasible. We have therefore written the function to implement as soon as variadic templates are supported by the CUDA compiler.

The current resolution to this challenge is to require the programmer to invoke the kernel separately for each GPU and CPU involved in the computation, and to explicitly make the CPU kernels execute in parallel if that is desired.

Invoking the kernel for each replica of the computation requires the programmer to access the `Handle` directly. Since the `Handle` should be completely opaque, we have provided a set of macros that implement all of the program logic for invoking the kernel. No such macros are provided for

```
1    Context context = Context(2,1);
2    float* xg;
3    float* yg;
4    hydraMalloc(&xg,bsize,context);
5    hydraMalloc(&yg,bsize,context);
6
7 do{
8      hydraMemcpy(xg,x,bsize,cudaMemcpyHostToDevice);
9      hydraMemcpy(yg,y,bsize,cudaMemcpyHostToDevice);
10
11     int i;
12     FORALL_GPU(i,context){
13       saxpy<<<dimGrid,dimBlock>>>(a,xg,yg);
14     }
15     FORALL_CPU(i,context){
16       cpusaxpy(a,xg,yg,size);
17     }
18   }while(hydraMemcpy(z,yg,bsize,cudaMemcpyDeviceToHost)!=hydraSuccess);
19
20   hydraFree(xg);
21   hydraFree(yg);
```

Figure 3.3: An example of Hydra code from the SAXPY application, showing the Hydra wrappers of memory control functions. This is an instrumentation of the code in figure 3.1. For this example the CPU kernel is hand-defined elsewhere.

parallelizing the CPU replicas, as the `#pragma omp` used to initiate a parallel region of code in OpenMP cannot be included in a macro due to the octathorpe (#) involved.

3.4 Returning Data to the CPU

In CUDA, data is returned to the CPU memory space using the command

`cudaMemcpy(dest,src,size,cudaMemcpyDeviceToHost).`

To translate to Hydra, this command is replaced by

`hydraMemcpy(dest,src,size,cudaMemcpyDeviceToHost).`

This function copies all of the GPU replica data back to CPU (using the CUDA command) and by default implements a voting scheme to merge the results. If there is a majority result for each element of the data, the data is accepted. If there is no majority result for some element of data, the program throws an error. The programmer is expected to handle that error, either by terminating the program or re-executing the computation.

From Ferreira et al. [3] we expect throwing an error for a result unresolvable by the voting scheme to be incredibly rare. Should such an error occur, nothing needs to be done to return the application to pre-kernel state. Because we have implemented the voting scheme in a non-destructive manner, the programmer can re-execute by simply copying the data from the CPU to the GPU again and invoking the kernel again. Figure 3.3 shows the Hydra instrumentation of the code from figure 3.1. The `do-while` loop implements the re-execution of the kernel following an error unresolvable by the voting scheme.

3.5 Summary

Hydra executes redundant copies of a CUDA kernel concurrently in GPU and CPU for resiliency and uses a voting scheme to resolve differences in the results. In the rare case in which the voting scheme is insufficient, Hydra enables the programmer to easily restart the failed computation. For performance, the ratio of GPU and CPU computation can be controlled by the programmer to balance the workload between resources. Hydra implements wrappers of CUDA memory manipulation functions to manage the redundant computations. This allows the redundancy to be implemented opaquely to the programmer, so that CUDA programs can be instrumented for redundancy with minimal effort.

Chapter 4

Heterogeneous Computing Modes

The important contribution Hydra makes is the implementation of redundancy for GPGPU computation through CUDA's standard kernel expression. As we have seen in Chapter 2, it is often the case that GPU and CPU calculations can be used interchangeably, albeit with some cost in terms of performance or expressibility. In Chapter 1 we noted that redundancy is simply a restricted form of parallelism, and in Chapter 3 we presented a programming framework to support redundancy in GPU calculations. One of the goals of Hydra is to support direct heterogeneous computation that effectively leverages the performance of the underlying CUDA code; to do so we will make use of our framework for redundant computation. Hydra provides systems for managing direct heterogeneous computation. Each will be an example of the scatter-gather paradigm. Scatter-gather is a method of managing parallelism that takes data from a single source, "scatters" it to discrete processing units, and then "gathers" the results to a single destination. In both systems, GPU memory allocation and kernel invocation directly use the framework for Hydra redundancy. As is true with most parallel computation models, there are many competing approaches. In this chapter, we consider two Hydra-based approaches to parallelism.

4.1 Partition/Concatenate

The first computational idiom Hydra provides for direct heterogeneous programming is partition/concatenate. Partition/concatenate is designed for direct heterogeneous computation of embarrassingly parallel applications. With an invocation of the templated function

```
template<typename T>
hydraScatterPartition(T* dest, T* src, size_t size);
```

the programmer sends a portion of the `size` bytes of data at `src` to each of the locations pointed to by the `Handle dest`[1]. Each location receives an equal-sized contiguous portion of the data. The

[1] Chapter 3 discusses the opaque implementation of `Handles`. That implementation is why `dest` can be passed to the function as a pointer to an object of type `T` but treated as a pointer to a `Handle`.

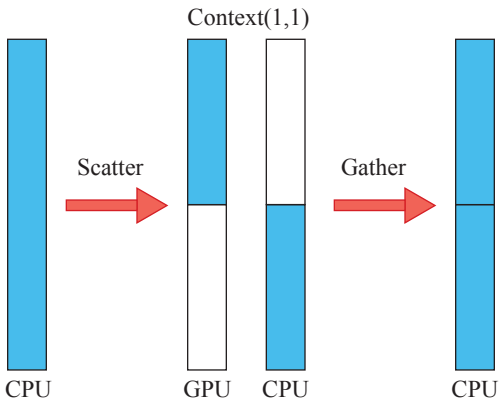

Figure 4.1: Image of data transfer in partition/concatenate mode. The vertical length of the bars represent separate elements of an array.

programmer can control the portion of computation done in GPU vs. CPU by manipulating the `Context`. The command

```
template<typename T>
hydraGatherConcatenate(T* dest, T* src, size_t size);
```

concatenates the data in each of the locations pointed to by the `Handle src` into the array located at `dest`. This mode is summarized in figure 4.1. The `size` parameter gives the size (in bytes) of the result array, to allow the programmer to reason about the scattered data as a single entity.

4.2 Split/Merge

The second idiom is split/merge. Split/merge is designed for direct heterogeneous programming of applications that exhibit data-level parallelism. It uses a user-defined function to separate each each element of the data into some number of pieces. The templated function

```
template <typename T>
hydraScatterSplit(T* dest, T* src, size_t size, void(*split)(int, T*, T**))
```

creates an array for each compute resource in the `Context` associated with the `Handle dest` and iterates over the data, applying the `split` function to each element. It then copies each array to its relevant processor. Since the pieces of data, in the order imposed by the `split` function, are copied to GPU for as many GPUs as are specified by the `Context`, and the rest to CPU, the programmer has total control, by way of the `split` function over which portions of the application are executed on which architecture. After the kernel executes, the programmer returns data to the CPU using the templated function

```
template <typename T>
hydraScatterMerge(T* dest, T* src, size_t size, void(*merge)(int, T*, T**)).
```

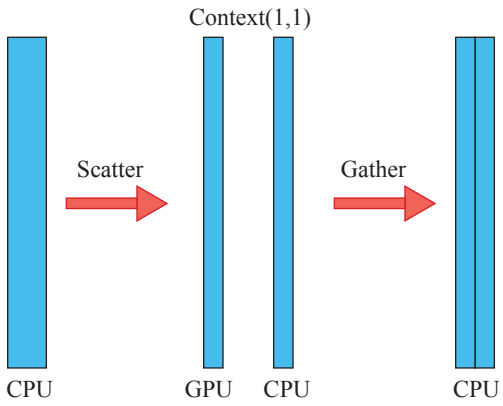

Figure 4.2: Image of data transfer in split/merge mode. The vertical length of the bars represent separate elements of an array. The horizontal length of the bars represent separable attributes of each element of the array.

This function copies the data from every compute resource in the `Context` associated with the `Handle src` to the CPU. It then invokes the `merge` function on each element of the resulting arrays, which merges their data into a single element of data that is stored into `dest`. The data flow of this mode is summarized in 4.2. Once again, `size` refers to the size in bytes of the result stored into `dest`, so that the programmer can reason about the scattered data as a single entity.

4.3 Summary

The same framework that enables redundant programming allows us to perform other types of parallel computing. In particular, it allows us to implement direct heterogeneous programming, allowing the programmer to schedule a single task to be spread between CPU and GPU computation. Hydra includes two new modes for direct heterogeneous programming designed for efficiency. The partition/concatenate mode effectively leverages embarrassing parallelism, while the split/merge mode effectively leverages data-level parallelism. These two types of parallelism are the best fit to express using the framework used for redundancy, because they require neither tight coupling nor communication between the parallel components.

Chapter 5

Hydra Performance

We have previously been interested primarily in the expressibility of redundancy through Hydra. We now turn our attention to evaluating the performance of Hydra for redundant computation. As we have seen in our discussion of [3] in Chapter 2, as long as the overhead is close to minimal—that is, close to an n times slowdown for n-way redundancy—redundancy outperforms checkpoint-restart for error-resiliency, while simultaneously solving the problem of silent bit errors. We will show that Hydra demands minimal overhead.

5.1 Machine Setup

All of our timing was done on a single 8-core Intel x86 machine running Ubuntu version 12. The machine had one NVIDIA GeForce GT640 GPU, a common off-the-shelf graphics processing unit. All tests were performed in isolation. Each test was run 10 times, and the results were averaged.

5.2 Benchmarks

We tested the performance of the Hydra redundancy framework using two benchmarks.

LULESH

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a benchmark from Lawrence Livermore National Laboratory that implements a portion of a hydrodynamics code. It is a mesh-based code using a hexahedral mesh with two centerings. For each time step of the code there are three major steps. The first step uses the vertices of the mesh for its centering, and advances the kinematic variables, such as position and location. The second step uses the centers of the mesh as its centering, and advances the thermodynamic variables, such as energy and pressure. The third step calculates the maximum time-step that can be covered in the next step of the algorithm [10].

Instrumentation of the LULESH benchmark was relatively straightforward. To avoid the necessity of handwriting CPU versions of the various kernels in the application, we used only GPU

Size	LULESH	3x	5x
5	0.181116	0.459664	0.711914
10	0.457633	1.711108	2.459876
15	1.973951	5.878158	8.975858
20	4.681939	13.815759	21.696995
25	9.874054	28.067038	45.105018
30	15.373150	44.905106	75.261955
35	24.165369	70.654150	117.934506
40	35.935525	106.487010	177.793042
45	53.278535	158.455741	263.707747

Table 5.1: LULESH timing data for 3x and 5x redundancy using Hydra (in seconds)

Size	LULESH	3x	5x
5	0.181116	2.54	3.93
10	0.457633	3.74	5.38
15	1.973951	2.98	4.55
20	4.681939	2.95	4.63
25	9.874054	2.84	4.57
30	15.373150	2.92	4.90
35	24.165369	2.92	4.88
40	35.935525	2.96	4.95
45	53.278535	2.97	4.95

Table 5.2: Uninstrumented LULESH run time (in seconds) alongside the slowdown for 3x and 5x redundancy using Hydra

computing for our computation context. Otherwise, the instrumentation was exactly as described in Chapter 3. We used CUDA version 6.5 for the LULESH benchmark.

NAS LU

Warwick NAS LU is a MPI/CUDA port from the University of Warwick of a psuedo application benchmark from NASA's Advanced Supercomputing division. The benchmark implements the Lower-Upper Gauss-Seidel Solver. The Lower-Upper Gauss-Seidel Solver is an iterative method for solving a system of linear equations [14].

The Warwick NAS LU code uses two Hydra-ized CUDA functions that not discussed in Chapter 3. To instrument NAS LU, we used a `hydraMemset` wrapper for the `cudaMemset` function. We did not wrap the `cudaMemcpyToSymbol` function, as symbols are passed by reference in CUDA. Thus each GPU in the computation context can use the same symbol. Similarly, since the GPU symbol is copied from some symbol in CPU space, each CPU in the computation context can use the same symbol[1]. For this benchmark as well we used only GPU computing for our computation context, and the rest of the instrumentation was exactly as described in Chapter 3. The Warwick NAS LU benchmark is incompatible with CUDA 6.5, so we used CUDA version 4.0.11 for this benchmark.

5.3 Performance

We gathered timing data for LULESH uninstrumented, instrumented using a computation context of three GPUs, and instrumented using a computation context of five GPUs. The default problem size of LULESH for a single processor is 45 mesh points in each dimension, and is designed to keep the problem within the memory specifications of the machine. We ran LULESH with every

[1]CUDA uses symbols to take advantage of the shared memory on GPUs. While scalar values are generally passed to the GPU by value in each kernel invocation, if a scalar will be accessed by many kernels it is more efficient to store it in the GPUs shared memory.

Size	NAS LU	2x	3x	5x
S	7.714422	15.391651	23.146085	38.624439
A	19.538068	39.008706	58.502194	97.362675

Table 5.3: Warwick NAS LU timing data for 2x, 3x, and 5x redundancy using Hydra.

Size	NAS LU	2x	3x	5x
S	7.714422	1.995	3.000	5.007
A	19.538068	1.997	2.994	4.983

Table 5.4: Warwick NAS LU slowdown for 2x, 3x, and 5x redundancy using Hydra

multiple of 5 up to 45 used as the number of mesh points in each dimension. Table 5.1 shows the run times for each problem size and computation context used. Table 5.2 shows the run time for the uninstrumented code alongside the relative slowdown for a 3x or 5x redundant computation context for each problem size using Hydra.

Note that for every problem size other than 10, the slowdown for 3x redundancy is less than 3 and the slowdown for 5x redundancy is less than 5. This means that the overhead of Hydra beyond that inherent in redundancy is dominated by the CPU portions of the code, and in fact some overhead of the computation is regained by using Hydra code. We would therefore expect to see benefits from this level of redundancy in accordance with Ferreira et al. [3].

For the Warwick NAS LU benchmark varying the problem size was more challenging. The benchmark comes with 4 default sizes, “S”, “A”, “B”, and “C”. Unfortunately, for problem sizes B and C a redundant Hydra program would exceed the memory capabilities of our GPU. We therefore restricted ourselves to the “S” and “A” problem sizes.

We gathered timing data for the NAS LU code uninstrumented, and using 2x, 3x, and 5x Hydra redundancy for each of the usable problem sizes. Table 5.3 shows the run time for each combination of redundancy and problem size for NAS LU. Table 5.4 shows the run time for uninstrumented code along with the relative slowdown for a 2x, 3x, or 5x redundant computation context for the NAS LU code.

Again note that for the larger problem size the slowdown for 2x redundancy is less than 2, the slowdown for 3x redundancy is less than 3, and the slowdown for 5x redundancy is less than 5. This means that we would expect to see benefits from this level of redundancy in accordance with Ferreira et al. [3] in this case as well.

5.4 Summary

From our performance testing of both benchmarks we see that Hydra performance for n-way redundancy consistently outperforms a slowdown of n times. When the GPU is utilized heavily, n way redundancy approaches an n times slowdown. However, as in [3], this overhead can be eliminated by introducing additional graphics cards, leaving only I/O overhead. Hydra can easily be extended to

take advantage of additional GPU resources. Combining our performance results with the analysis done by Ferreira et al., we can conclude that Hydra will outperform checkpoint-restart resiliency schemes for the exascale class machines discussed in [3]. Thus Hydra can improve performance of parallel applications on heterogeneous architectures of sufficient scale for which the GPU error rate dominates the overall error rate. Additionally, there are situations in which accuracy is a matter of life and death or in which error-rates are unnaturally high. Space travel is an example of both; computing for space travel must be perfectly accurate and must account for high levels of error due to cosmic radiation. In such circumstances 5-way or even higher levels of redundancy could be used to eliminate silent errors and check CPU and GPU computation against each other using separately programmed kernels.

Chapter 6

Hydra Parallel Performance

As we did in Chapter 5 for redundancy, we now turn our attention to the performance of the Hydra direct heterogeneous programming modes. In Chapter 4 we introduced these modes and discussed how they work to express parallelism. This chapter will explore the efficiency with which they extract that parallelism, by comparison among Hydra heterogeneous programming implementations, optimized serial implementations, and CUDA implementations.

6.1 Partition/Concatenate

As discussed in Chapter 4, the Hydra partition/concatenate model is designed to take advantage of embarrassing parallelism. To test this model we instrumented two simple SAXPY programs with Hydra partition/concatenate. SAXPY stands for Single-precision aX Plus Y. SAXPY programs take a floating point value a and two vectors of floating points X and Y and compute the vector $aX + Y$.

Size	Serial	CUDA	Hydra-9-1	Hydra-5-0
100000	1.848591	3.854450	6.319272	5.116759
200000	3.684758	7.413493	12.230523	9.880144
400000	7.344762	14.345334	19.070938	16.833618
800000	14.724253	27.966975	32.822460	30.528899
1600000	29.520793	55.363893	60.573154	57.912479
3200000	58.843567	109.960220	115.195342	112.239030
6400000	117.884251	217.575002	223.571298	220.323850
12800000	235.929288	434.433974	409.429708	437.708541
25600000	471.690017	866.230390	818.639645	804.664639

Table 6.1: Time data for serial, CUDA, and two Hydra implementations of the first SAXPY program (in seconds). The Serial and CUDA implementations perform the computation entirely on the respective processors. Hydra-9-1 performs 90% of the computation on GPU and 10% of the computation on CPU. Hydra 5-0 computes entirely on the GPU, but in five separate pieces.

Size	Serial	CUDA	Hydra-9-1	Hydra-5-0
100000	1.848591	0.48	0.29	0.36
200000	3.684758	0.50	0.30	0.37
400000	7.344762	0.51	0.39	0.44
800000	14.724253	0.53	0.45	0.48
1600000	29.520793	0.53	0.49	0.51
3200000	58.843567	0.54	0.51	0.52
6400000	117.884251	0.54	0.53	0.54
12800000	235.929288	0.54	0.58	0.54
25600000	471.690017	0.54	0.58	0.59

Table 6.2: Time data for the serial implementation of the first SAXPY program (in seconds), alongside the relative speedup for the CUDA and two Hydra implementations.

Our first simple SAXPY program generates random values for a, X, and Y, and then executes the SAXPY computation. It executes this entire process, including the vector generation, 1000 times. Our second SAXPY program generates random values for a, X, and Y, and then executes the SAXPY computation 1000 times, using the Y from the previous computation as X and the result of the previous computation as Y. These two programs simulate the differences between a memory-bound program (program 1) and a computation bound program (program 2). We compared the Hydra parallelism performance both to a serial implementation and to the similar CUDA implementation. We timed all three implementations for 100k, 200k, 400k, 800k, 1.6M, 3.2M, 6.4M, 12.8M, and 25.6M float arrays. The machine setup was the same as in Chapter 5, and as in Chapter 5 we timed each size 10 times and averaged the results.

CUDA programs naturally need some tuning for optimal performance, and Hydra heterogeneous programming codes need additional tuning to find the optimal ratio of CPU and GPU computation. We used simple tuning for the CUDA variables that apply to both the CUDA and Hydra implementations, and we kept that tuning constant across all implementations. We used two Hydra implementations, one with a computation context of 9 GPUs and 1 CPU, and one with a computation context of 5 GPUs and no CPUs. This explores the balance between the relative speed of GPU and CPU computation and the overhead of splitting the problem into increasing numbers of pieces. We did not exhaustively tune the program to arrive at these values, but the performance is sufficient to prove the value of this Hydra parallelism model, so we let it stand as is.

Table 6.1 shows the run time of each code (serial, CUDA, and two Hydra implementations) for the first SAXPY program. Table 6.2 shows the same data with the CUDA and Hydra run times expressed in terms of speedup[1] over the serial implementation. Table 6.3 shows the run time of each code (serial, CUDA, and two Hydra implementations) for the second SAXPY program. Table 6.4 shows the same data with the CUDA and Hydra run times expressed in terms of speedup over the

[1]Here we use speedup instead of the slowdown, as we did in Chapter 5, because we are optimistic that Hydra will outperform serial code at scale.

Size	Serial	CUDA	Hydra-9-1	Hydra-5-0
100000	0.036908	0.153402	1.447690	0.682800
200000	0.074784	0.173633	1.542223	0.789841
400000	0.147071	0.163771	1.309977	0.684844
800000	0.278392	0.154891	1.287306	0.599973
1600000	0.522551	0.155969	1.458051	0.545828
3200000	1.008859	0.175461	1.922888	0.545100
6400000	10.864234	0.225375	2.808778	0.581445
12800000	—	0.333917	4.557887	0.683162
25600000	—	0.554712	8.057994	0.890564

Table 6.3: Time data for serial, CUDA, and two Hydra implementations of the second SAXPY program (in seconds).

serial implementation. Note that the serial implementation of the second SAXPY program timed out for the 12.8M and 25.6M sizes, so we have no data for those sizes, nor do we have comparative speedup for those sizes. In each case, the Hydra-5-0 performance functions as a sort of sanity check; by executing in 5 segments in GPU only, its performance should closely track that of the CUDA code, except that it pays the overhead of splitting the data 5 ways. For the first SAXPY program, Hydra-9-1 outperforms CUDA, but both are outperformed by serial code. Surprisingly, Hydra-5-0 also outperforms CUDA and Hydra-9-1 in this case for the largest problem size. We suspect that this is due to peculiarities of our sub-optimal tuning of these programs, as CUDA tuning is an difficult process. We executed each CUDA and Hydra program with the same tuning parameters, which may have accidentally skewed the results in favor of some programs at certain sizes. Serial code outperforming both CUDA and Hydra in this case is unsurprising, because the I/O overhead of bringing data to the GPU dominates the performance. For the second SAXPY program, both Hydra and CUDA significantly outperform serial code (with many time speedups in performance) and both Hydra implementations are—as might be expected—outperformed by the CUDA implementation. We can clearly see from the data that through the smallest few sizes the

Size	Serial	CUDA	Hydra-9-1	Hydra-5-0
100000	0.036908	0.24	0.03	0.05
200000	0.074784	0.43	0.05	0.09
400000	0.147071	0.90	0.11	0.21
800000	0.278392	1.80	0.22	0.46
1600000	0.522551	3.35	0.36	0.96
3200000	1.008859	5.75	0.52	1.85
6400000	10.864234	48.21	3.87	18.68

Table 6.4: Time data for the serial implementation of the second SAXPY program (in seconds), alongside the relative speedup for the CUDA and two Hydra implementations.

Size	Serial	Hydra
100000	0.371554	0.158831
200000	0.730093	0.182260
400000	1.461013	0.192990
800000	2.745313	0.193177
1600000	5.121474	0.214497
3200000	9.862550	0.289065
6400000	122.535971	1.375257

Table 6.5: Time data for the serial and Hydra implementations of the SAXPY and sum program (in seconds).

Size	Serial	Hydra
100000	0.371554	2.34
200000	0.730093	4.01
400000	1.461013	7.57
800000	2.745313	14.21
1600000	5.121474	23.88
3200000	9.862550	34.12
6400000	122.535971	89.10

Table 6.6: Time data for the serial implementation of the SAXPY and sum program alongside the relative speedup of the Hydra implementation

overhead of the Hydra-5-0 implementation is being amortized, leading to the lower run times for increasing sizes. Beginning around 3.2M, however, the computation time comes to dominate, and the run time increases with further increases in the problem size.

6.2 Split/Merge

The Hydra split/merge model is designed to take advantage of data-level parallelism. To test this model we instrumented an addition to our second SAXPY code from Section 6.1. The new application invokes the 1000 iterations of iterative SAXPY as before, but also computes the sum of the entries in the original two arrays. There is therefore a data-level parallelism in the first two arrays between their use in the SAXPY computations and in the summation. The Hydra implementation of this code used a computation context of one GPU and one CPU. The split function for this implementation merely sends each datum to both the GPU and CPU, and the merge function merely copied back data from the GPU, as the CPU result was separately stored in a global variable.

We compared the Hydra split/merge implementation to a serial implementation for 100k, 200k, 400k, 800k, 1.6M, 3.2M, and 6.4M float arrays. The machine setup was the same as in Chapter 5, and as in Chapter 5 timed each size 10 times and averaged the results.

Table 6.5 shows the run times for each implementation for each of the sizes tested. Table 6.6 shows the same data, but with the Hydra run times expressed in terms of speedup over the serial implementation. We can see from this data that the Hydra split/merge model is expressing parallelism in a manner that can be effectively implemented by the heterogeneous architecture.

6.3 Summary

We have demonstrated that both direct heterogeneous computation modes provided with Hydra express parallelism in ways that vastly improve performance over serial computation. Additionally, for some problem types Hydra partition/concatenate mode outperforms native CUDA implementations. Tuning and the particulars of the problem can have surprisingly large affects on the performance. As we saw with the Hydra-5-0 implementation of the first SAXPY program, proper tuning can have an outsized effect on run times. Additionally, the exact nature of the problem effects the optimal implementation. Our two simple SAXPY programs are extreme in their attributes—one is extremely I/O bound and the other is extremely computation bound. We hypothesize on the basis of this data that for more moderate problems there may be a sweet spot in which Hydra may outperform both serial and CUDA implementations. Future work may include experimenting to find such a sweet spot.

Chapter 7

Future Work

Much of this work is proof-of-concept, necessarily falling short of a complete approach to eliminating soft errors in computation. This chapter describes techniques and ideas that might be investigated to make Hydra more resilient, efficient, and complete.

7.1 Refinements of the Hydra System

One obvious element of future research could be to implement wrappers for the remaining CUDA functions that target device data. We have only implemented wrappers for a subset of CUDA necessary to test the viability and performance of the project.

Additionally, in Chapter 3 we discussed the proof of concept of the `hydraInvoke` function for invoking kernels. This function will need to be tested and added to the program once the NVIDIA `nvcc` compiler supports variadic templates.

We have implemented Hydra as a CUDA program that can be compiled with the target program. Hydra cannot currently be implemented as a shared library because of the templating of some of the wrapper functions. One element of future work may be to implement Hydra as several libraries for the various data types that may be necessary with an additional C program that can be used for user-defined data types.

There are many types of parallelism available to expose in various applications. Some types of parallelism are taken advantage of naturally by CUDA, and other types are taken advantage of by the partition/concatenate and split/merge models of Hydra. Future research should include evaluations of other plausible forms of available parallelism and construction of the models that will take advantage of those forms of parallelism.

7.2 Performance Evaluations

More detailed studies of the performance of Hydra redundant computations and Hydra parallelism at increasing scale are needed. Additionally, if an error-injection tool such as BIFIT [12] can be

extended to heterogeneous architectures, Hydra redundancy performance under various plausible error-rates can be modeled without the expense of testing at the wide variety of scales present in the current HPC work space.

Hydra parallelism performance includes the ease of programming given by the uniform programming model. Additional studies should be performed on the programmer time saved by Hydra programming for direct heterogeneous computation over more time-intensive programming of indirect heterogeneous computation.

7.3 Hydra and Transactional Memory

Hydra computations aborted due to failure of the voting scheme in some ways resemble the aborted transactions of transactional memory in a variety of important ways. Appendix A discusses transactional memory and its potential applications to Hydra. One piece of future research should be to implement explicit software transactions for Hydra that abstract the rollback mechanism from the programmer.

More sophisticated additional research could include the implementation of hardware transactions for Hydra. Current best-effort hardware transactional memory systems cannot support transactions that include a kernel invocation. Introducing hardware that supports such transactions could greatly improve the speed of rollback for Hydra in the cases in which voting fails.

We would note that although such research is valuable, its improvements to Hydra's performance may be negligible due to the rarity with which we expect to see failures of the Hydra voting scheme for redundant computation.

7.4 Kernel Compilation for CPU

In Chapter 2 we discussed the MCUDA project for compiling CUDA GPU kernels for CPU using OpenMP. Unfortunately MCUDA is not currently maintained at production levels. The NVIDIA compiler (discussed in Chapter 2 as well) compiles only device inline functions, but not kernels, for CPU computation. There are no academic projects of which we are aware that maintain updated code for source-to-source translation between CUDA GPU code and CPU code.

The PGI compiler purports to be able to do compilation of CUDA GPU code for CPU execution, but due to its cost we have not worked with it in any capacity.

Whether through the PGI compiler or through another source, a tool for compiling CUDA GPU kernels for CPU execution should be maintained. Hydra needs to be updated to include automatic translation of GPU kernels for CPU in the presence of computation contexts including CPU computation.

Chapter 8

Conclusion

Resilience is of increasing concern as supercomputer sizes and node counts increase. Poor resiliency scaling is particularly problematic in the context of GPGPU computations involving thousands of lightweight processors per node. Current techniques for managing resiliency are inefficient and do not protect against soft errors. We have presented the design of the Hydra programming model for heterogeneous architectures. Hydra is based on NVIDIA's CUDA runtime, and is designed to ease programmer transition and leverage CUDA's efficient representation of parallelism. It is implemented by providing opaque wrappers to a variety of CUDA functions to manage redundancy. It additionally provides scatter-gather models for other types of direct heterogeneous computation.

Hydra redundancy protects the device computation in a heterogeneous architecture and allows for a transactional-memory-style approach to resolve fatal failures. The assumption that device computation is significantly more susceptible to error than host computation is supported by the architecture of GPU devices and the existing methods of protecting host computation, such as error-correcting codes in main memory. We have also shown that Hydra redundancy requires negligible overhead beyond that which is obviously inherent in redundant computation. This is sufficiently little overhead as to be an efficient form of resiliency on sufficiently large machines, as suggested by Ferreira et al. [3].

Hydra redundancy also protects against silent faults, which are not detected by the currently standard checkpoint-restart resiliency protocol. As error rates increase, silent faults will become increasingly detrimental to the accuracy of scientific computations, and future resiliency schemes will need to detect such faults. Resiliency to silent faults is also important in contexts in which error-rates are already high and accuracy is important, such as in extra-terrestrial computation, where cosmic radiation increases error-rates. Silent faults may also be of particular concerns at high-altitude supercomputer installations, such as the Department of Energy's Los Alamos National Laboratory.

Hydra parallelism models also support the abstractions of direct heterogeneous computation. The partition/concatenate model expresses embarrassing parallelism, while the split/merge model supports data-level parallelism. Our results show that both models effectively extract the paral-

lelism that they express. In both modes, the computation context becomes a parameter that can be tuned for optimal performance, along with tuning the native CUDA code. For problems that have approximately equal run times for GPU and CPU implementations, Hydra may outperform both by leveraging the natural parallelism of a heterogeneous system.

Appendix A

Transactional Memory and Hydra

A.1 Error Handling

All of the applications discussed in the previous chapter were run on too small a machine and had too short a run-time for soft faults to be likely to manifest. For these applications we simply had Hydra throw an error in the event of a fault from which it could not recover, and said error was never thrown. For larger and longer running applications, however, we would need a scheme by which to recover from errors that exceed the error-correction capabilities of the level of redundancy at which the application is run.

The goal of such a scheme is to restart the GPU calculation from the point at which the relevant data was moved into the GPU. Our goal with such a scheme is to mimic the effects of Software Transactional Memory.

A.2 Transactional Memory

Transactional memory is a programming coherency paradigm first introduced by Herlihy and Moss in 1993 [8]. They proposed a hardware solution to coherency issues canonically solved by software-based locks. The idea behind transactional memory is that memory accesses should behave like database transactions; that is, they should be atomic, consistent, and isolated (durability, the other concern of database transactions, is not relevant to transactional memory). Instead of acquiring a lock, executing a critical section, and releasing a lock, a process could begin a transaction and continue executing. If the transaction is not aborted, the process can finalize the transaction after the critical section without concern for coherence, as the transaction interface ensures atomicity of all transactions. Herlihy and Moss' system modifies existing cache coherence protocol, as any protocol that can track accessibility conflicts can track transaction conflicts at the same time. Lines of the transaction cache are maintain lists of the transactions which read them while in a "shared" state in a MOESI protocol, and on a write abort and roll back all such transactions except the writing transaction, which tags the data as "owned." To implement transaction rollback,

the transaction cache maintains two copies of all data; one copy (tagged "xcommit") contains the original value, and one copy (tagged "xabort") contains the modified value. On transaction commit, the xcommit value is deleted (set to "empty") and the xabort copy is tagged as "normal." On transaction abort, the xabort value is deleted and the xcommit value is tagged as "normal." Herlihy and Moss investigated transactional memory primarily for its ease of programming, but in simulations they found it out-performed locking mechanisms by a significant margin, since it only required a transaction rollback if two transactions accessed the same word of memory, rather than the same variable. This makes transactional memory of significant interest to programmers in HPC environments.

Transactional memory can more easily be implemented in software, at some cost of overhead. Herlihy et al. [7] extended software transactional memory (STM) to dynamically sized data, creating dynamic software transactional memory (DSTM), a key step in proving the usefulness of transactions as a programming paradigm. Previous software transactional memory systems had required statically defined transactions and data sizes, similar to the implementation of Herlihy and Moss' hardware transactional memory [8] which aborted for all transaction whose sizes exceeded the small transaction cache. Herlihy et al. allow for dynamically sized data by creating a transactional object class which is a container for regular data. This container class maintains the pointers necessary to implement transactional memory, such as a pointer to the last transaction to open it in write mode, the old version of the object, and the new version of the object. Since anything can then be put in the container, this allows for dynamically sized objects in transactions. In addition to managing dynamically sized data, DSTM has a modular contention manager that specifies the back-off strategy. This allows the programmer to easily tailor the back-off strategy of particular applications to their data use pattern for maximum efficiency. This is important as Herlihy et al. provide a pair of example applications between which it is necessary to change back-off strategies to achieve reasonable efficiency, as an efficient strategy for one is highly inefficient for the other. Like any other STM, DSTM is highly inefficient (approximately 4x overhead) and therefore unlikely to be used on its own in any applications which require efficiency.

A.3 Hydra Transactions

Hydra programs develop errors not from coherency problems, but from transient hardware faults. However, we can mentally recast the transient faults as the result of a rogue process that always has higher priority than the running process—that is, some process is corrupting the computation, and the corrupted process cannot force the corrupting process to be part of the solution. We can model this with software transactional memory. Each GPU computation is a transaction; Hydra's voting scheme is the method by which it detects that another process (the rogue "error process") has corrupted its data.

As we discussed before, when Hydra computation detects an error from which it cannot naturally recover, returns an error code and aborts before changing any of the data allocated directly by the

C/C++ host code. For small applications it may be suitable to abort in this case. However, for larger applications the programmer can handle the error by jumping to a suitable location in the code. Since Hydra has not modified host data, simply re-running all of the GPU-side computation should be sufficient to overcome the error. This is equivalent to aborting a Hydra transaction because of the effects of the rogue "error process" and restarting the transaction.

For general transactional memory some sort of scheme to ensure progress is necessary. However, this is unnecessary in the case of Hydra because progress can only be prevented by GPU computation errors occurring at a very high rate. Thus by modeling Hydra after transactional memory we see that recovery from what might otherwise be a fatal error is not difficult.

A.4 Summary

Transactional memory is a programming model designed to eliminate the need for locking mechanism. It does so by ensuring atomicity of each memory transaction. Transactions are an effective way to think about failed Hydra computations, when errors are modeled by a lack of atomicity. Hardware support for transactions spanning a GPU kernel invocation could therefore greatly speed Hydra recovery in the face of errors exceeding the correcting ability of the redundant computation.

Bibliography

[1] Bronevetsky, G., and de Supinski, B. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22Nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 155–164.

[2] Diamos, G., Kerr, A., and Kesavan, M. Translating gpu binaries to tiered simd architectures with ocelot. Tech. Rep. 0901, January 2009.

[3] Ferreira, K., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 44:1–44:12.

[4] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 78:1–78:12.

[5] Fung, W. W. L. Dynamic warp formation: exploiting thread scheduling for efficient mimd control flow on simd graphics hardware.

[6] Garland, M., Kudlur, M., and Zheng, Y. Designing a unified programming model for heterogeneous machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (Nov 2012), pp. 1–11.

[7] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2003), PODC '03, ACM, pp. 92–101.

[8] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1993), ISCA '93, ACM, pp. 289–300.

[9] Hong, C., Chen, D., Chen, W., Zheng, W., and Lin, H. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 217–226.

[10] Karlin, I., Bhatele, A., Chamberlain, B. L., Cohen, J., Devito, Z., Gokhale, M., Haque, R., Hornung, R., Keasler, J., Laney, D., et al. Lulesh programming model and performance ports overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2012).

[11] Lee, S., Min, S.-J., and Eigenmann, R. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not. 44*, 4 (Feb. 2009), 101–110.

[12] Li, D., Vetter, J. S., and Yu, W. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 57:1–57:11.

[13] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6*, 2 (Mar. 2008), 40–53.

[14] Pennycook, S. J., Hammond, S. D., Jarvis, S. A., and Mudalige, G. R. Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark. *ACM SIGMETRICS Performance Evaluation Review 38*, 4 (2011), 23–29.

[15] Shantharam, M., Srinivasmurthy, S., and Raghavan, P. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 152–161.

[16] Stratton, J. A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., and Hwu, W.-m. W. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 111–119.

[17] Stratton, J. A., Stone, S. S., and Hwu, W.-M. W. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30.