An Execution Model and Runtime For Heterogeneous Many-Core Systems

Gregory Diamos

School of Electrical and Computer Engineering

Georgia Institute of Technology

gregory.diamos@gatech.edu

January 10, 2011

Contents

1 Summary of Research Performed

1.1 Introduction

The emergence of heterogeneous and many-core architectures presents a unique opportunity to deliver order of magnitude performance increases to high performance applications by matching certain classes of algorithms to specifically tailored architectures. However, their ubiquitous adoption has been limited by a lack of programming models and management frameworks designed to reduce the high degree of complexity of software development inherent to heterogeneous architectures. This proposal introduces Harmony, an execution model for heterogeneous systems that provides: (1) semantics for simplifying heterogeneity management, (2) dynamic scheduling of compute intensive kernels to heterogeneous processor resources, and (3) online monitoring driven performance optimization for heterogeneous many core systems. This work focuses on simplifying development and ensuring binary portability and scalability across system configurations and sizes. Results from a prototype implementation of the Harmony execution model demonstrate binary compatibility with systems with different GPU and CPU configuration. Results are also presented from an integrated dynamic compiler that allows the same low-level PTX representation of a kernel to be executed on both CPU and GPU processors.

1.2 Origin and History of the Problem

1.2.1 The Rise and Fall of General Purpose Computing

The driving forces of many-core scaling and heterogeneous acceleration have the potential to revolutionize general purpose computing. Recently, graphics accelerators from NVIDIA and AMD have been released with up to 4.14 TFlops of single precision throughput and 256GB/s of off-chip memory bandwidth [2, 57]. At the same time, Intel Research has demonstrated a 48-core tiled IA32 processor that can boot Linux [34]. These processors are the culmination of tremendous advances in semiconductor manufacturing, VLSI, circuit design, and micro-

architecture. However, they are also a significant departure from the Von Neumann model and greatly increase the complexity of software design.

Though the evolution of the computing industry has included forays into parallel processing, vector architectures, and application specific accelerators, the last few decades have been dominated by general purpose computing and high volume, commodity processors. Commodity general purpose processors were successful because they were able to steadily increase performance while maintaining a static hardware-software interface. Applications could be written and compiled once, for a single ISA, and frequency and ILP scaling would ensure that new processors would not only correctly execute an application, they would do so with greater performance. However, as power constraints began to limit frequency scaling and the amount of available ILP in existing applications was exhausted, general purpose processors began to again move towards parallel and domain specific architectures.

1.2.2 The Many-Core and Heterogeneous Revolution

Many-Core Processors Even before technology constraints forced the industry migration away from aggressive super scalar architectures, several research efforts began developing architectural solutions to the increased wire delay, limited off-chip bandwidth, growing power consumption even in the face of voltage scaling, and mounting design and verification complexity as Moore's Law scaling enabled the production of billion transistor processors [9]. These efforts spanned the spectrum from aggressive super-scalar designs to reconfigurable fabrics of parallel components. They were polarized between two opposing design philosophies. Some focused on working within the constraints of the existing hardware/software interface. These included advanced superscalar designs that extended the upper bound on ILP from 2-4 to 16-32 [65], trace and multi-scalar architectures that attempted to execute bundles rather than individual instructions speculatively [68], culminating in compiler-assisted hardware transactional memory and speculative threading [31, 67]. The alternative philosophy focused on technology driven designs that could refine or even redefine the hard-

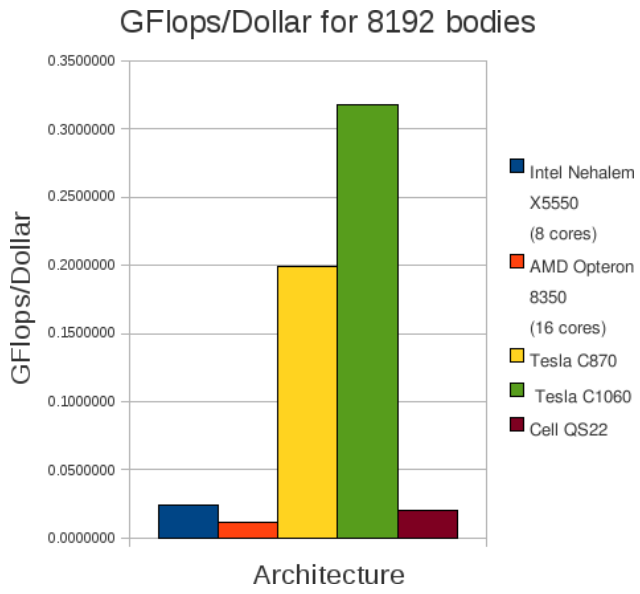

Figure 1: GFPLOPs per dollar for different multi-core platforms when running the Nbody application developed by Arora et al. [5].

ware/software interface. Some prominent examples included processors built around simultaneous multi-threading [23], vector and SIMD architectures [44], chip-multiprocessors [61], and polymorphic and tiled architectures [10, 69, 79]. Multi-threaded and multi-core architectures retained a uniform ISA, but required applications to coordinate communication and synchronization across multiple threads. Ultimately, a hardware/software interface defined by a single processor with a uniform memory space and a static ISA became a liability and the general purpose computing industry moved to chip-multiprocessors and domain specific accelerators. One of the most daunting challenges in computing today is dealing with the fallout from this decision.

Heterogeneous Systems General purpose architectures invariantly sacrifice efficiency for programming simplicity and generality. Although the prohibitive cost of designing and manufacturing a processor limits the economic feasibility of ASICs, certain application domains that are limited by the efficiency and capability of general purpose processors can still support the design of custom architectures if they are driven by a large enough market. These conditions exist in domains such as computer graphics, signal processing, and embedded

computing where accelerators have been designed that offer significant performance and efficiency advantages over general purpose architectures. Figure 1 compares the evolution of graphics accelerators and general purpose processors in terms of FLOPs/Watt. These architectures are not heterogeneous individually. However, the ability to integrate many processors together onto the same die presents the opportunity to compose truly heterogeneous system on chips, with different processors optimized for different classes of workloads. These architectures are becoming increasingly important as designs become more power constrained.

Heterogeneous processors can be grouped into three categories: heterogeneous micro-architectures, ISA extensions, and heterogeneous cores. At the micro-architecture level, allocating hardware resources differently can affect the performance of certain classes of applications. For example, Najaf-abadi et al. propose a scheme for architecture contesting where several cores, each with a different micro-architecture, are included in a multi-core processor and used to accelerate workloads with different characteristics [53]. At the ISA level, it is possible to add additional instructions to the base ISA for accelerating domain specific operations. For example, SIMD extensions to the x86 instruction sets allow vector operations to be accelerated on some general purpose processors [66], and approximate transcendental and texture interpolation instructions are included in NVIDIA GPU ISAs [14]. Finally, cores with completely different ISAs and micro-architectures can be incorporated into the same multi-core processor. Wong et al. have prototyped a combined Intel GMA X4500 GPU and Intel IA32 CPU on the same FPGA [80]. Recently released designs from Apple [4], Intel [38], and NVIDIA [59] along with announced projects from AMD [40] pair CPU and GPU cores together on the same processor. It should go without saying that both ISA extensions and heterogeneous cores require significant changes in the hardware/software interface as illustrated in Figure 2.

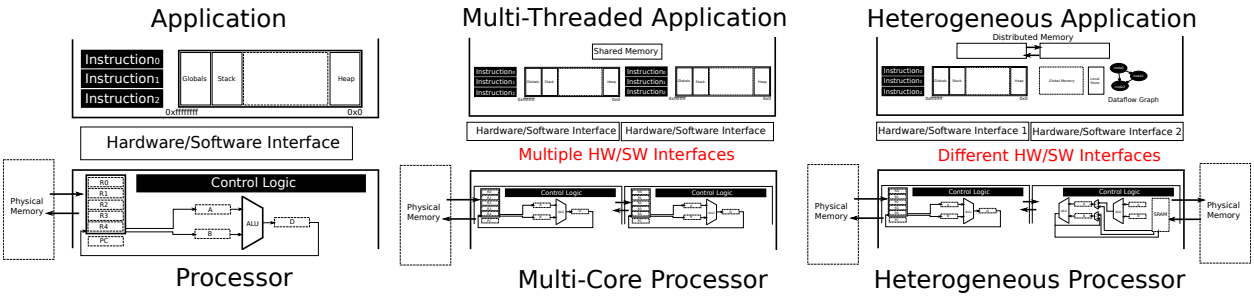

Figure 2: Multi-core and heterogeneous processors significantly increase the complexity of software design because they change the hardware/software interface. Multi-core processors require applications to explicitly coordinate among multiple instances of the same interface. Heterogeneous processors require applications to coordinate among multiple *different* interfaces.

1.2.3 Programming Languages for Parallel and Heterogeneous Systems

Though the return to parallel and heterogeneous architectures has allowed performance scaling to continue, the accompanying hardware/software interface changes have ended the forward scalability and, in some cases, the forward compatibility of existing applications. These problems have been pushed higher up the stack, into compilers, runtime systems, programming languages, and even algorithms and data structures. The many-core revolution has forced programmers to address parallelism, and several industry and research efforts have developed implicitly and explicitly parallel models for multi-core programming. Beyond typical message passing and threading models, streaming and bulk-synchronous-parallel languages have emerged as viable programming models for many-core processors. Several efforts have also addressed architecture heterogeneity with new language semantics and operating system and runtime support.

Implicitly and Explicitly Parallel Programming Following the introduction of multi-core processors, the first languages and programming models to take advantage of the new capability were threading models that had previously been used to program multi-processor systems with shared memory [55] and shared nothing models like MPI that used message passing for communication on clusters [62]. In these explicitly parallel models, the program-

mer is responsible for launching threads, mapping them to cores, and coordinating inter-thread communication and synchronization. An alternative philosophy quickly emerged that relied on programmer inserted hints and intelligent compilers to automatically extract parallel regions from sequential applications and map them onto threads. Today, OpenMP [11] is probably the most popular realization of this philosophy. However, a number of research efforts identified several potential problems with these solutions to parallel programming. Most notably, 1) that programs written for a fixed number of processes in MPI or threads in Pthreads could not be scaled without modification to future processors with higher core counts [8], and 2) that the ability to perform operations with side-effects and unstructured, non-deterministic, synchronization and communication significantly increased the complexity of writing parallel applications [41]. Cilk [8], Intel TBB [13], and Ct [37] are programming languages that address the problem of scalability by expressing applications in terms of large groups of light-weight threads coupled with runtime components that provide low overhead synchronization and communication operations. Alternatively, Charm++ [41] adopts a message passing model and focuses on the problem of complexity by requiring threads to execute encapsulated functions without side effects and explicitly specifying channels for communication. Finally, streaming and bulk-synchronous-parallel languages are emerging models for parallel programming that are significantly different from traditional threaded or shared memory models. They are discussed in more detail in the following sections.

Stream Programming Stream computing is driven by a popular class of applications that operate on signals or series of data. Though there are many subtle variations for different languages, programs are typically composed of directed data-flow graphs of 'kernels' that operate on explicit input and output streams and perform some transformation on elements from the input streams to produce elements on the output streams. Prominent examples of streaming languages include StreamIt [75], the Stream Virtual Machine [46], and Brook [35]. Streaming languages have well defined semantics that ease their migration onto parallel

architectures. In the simplest case, kernels can be processed concurrently by exploiting pipeline parallelism [74]. This technique has been used to map streaming languages to the IBM cell processor [81], multi-core x86 processors [45], NVIDIA GPUs [76], AMD GPUs [1], and several others [27]. Recently, compiler transformations have been proposed that map streaming languages onto vector SIMD processors [33]. Although streaming languages have significant advantages, some applications, particularly those with complex control flow, do not map well to the streaming model.

Bulk-Synchronous-Parallel Languages Bulk-Synchronous-Parallel (BSP) programming is based around the idea that the cost of global synchronization will eventually limit the scalability of parallel applications. Applications are composed of many parallel tasks that execute independently and can communicate via either message passing or shared memory. However, messages are only guaranteed to be received and shared memory is only guaranteed to be consistent after periodic global barriers. Applications are encouraged to launch as many tasks as possible between barriers to ammortize the global synchronization overhead. As tasks cannot reliably communicate between barriers, there are no constraints on the scheduling of tasks; tasks can be executed in parallel on different processors, or serialized on the same processor and executed to completion without a context switch. Since the original formulation of the BSP model by Valiant [77], several industry initiatives have adopted variants for general purpose programming on GPUs. CUDA was the first example such example, released by NVIDIA in 2007 [56]. OpenCL followed with a specification in 2008 [28] and the first compilers began to emerge in late 2009 [3, 58]. Both CUDA and OpenCL express programs in terms of a series of compute kernels, which are composed of highly multi-threaded tasks (blocks in CUDA terminology). Tasks cannot synchronize within kernels. However, a shared memory space is made consistent across kernel launches, which are effectively global barriers. A significant addition to the BSP model made by OpenCL and CUDA is that tasks are not persistent across kernel launches. This limits the amount of state needed to represent

a kernel; it is reduced to the total number of active tasks rather than the total number of tasks, which can be potentially overwhelming in cases where millions or billions of tasks are launched [17].

Heterogeneous-Aware Languages Though there has been comparatively less work addressing heterogeneity than there has been on reducing the complexity and increasing the scalability of parallel applications, there are a few notable exceptions. Linderman et al. propose the Merge framework for writing applications for heterogeneous systems consisting of CPUs and GPUs. Programs in Merge are expressed as a series of kernels with 'predicates', which are characteristics of the processors that the kernel can execute on. A Merge program is executed by a runtime that uses explicit dependencies between kernels coupled with predicates to determine the set of possible mappings from active kernels to available processors and choose a good mapping. Merge offers one approach for handling micro-architecture hierarchy. However, it does not directly address systems where processors each have their own memory hierarchies. Fatahalian et al. address the problem of memory system heterogeneity with Sequoia [25], another runtime supported programming model that expresses an application as a tree of kernels with explicit data dependencies. The structure of a Sequoia application allows it to be recursively partitioned into different levels of DRAM and cache in a complex memory hierarchy. A common feature of these approaches is the expression of an application in terms of kernels, and a runtime component that dynamically maps kernels to available hardware resources. The use of a runtime introduces the need to schedule kernels effectively given the fact that the same kernels can perform significantly better or worse on a given processor. Jimenez et al. explore several existing scheduling techniques in heterogeneous systems with GPUs and CPUs [39]. They conclude that predictive models that attempt determine the expected execution time of each kernel on each available processor are necessary for efficient kernel scheduling. Noticeably lacking from this body of work are the problems of 1) executing the same kernels on processors with different ISAs, 2) migrating

from existing languages to these new frameworks, and 3) static and dynamic optimizations that can be performed on kernel representations of programs.

1.2.4 Dynamic and Heterogeneous Compilers

Even before the widespread adoption of truly heterogeneous systems, the use of different ISAs in competing commodity processors prevented applications from being executed on multiple platforms and restricted the design of new micro-architectures. Dynamic compilation (also known as just-in-time compilation) was born out of this lack of application portability. Previous work in dynamic compilation has been concerned mainly with enabling portability and performance across Von-Neumann architectures with different ISAs. However, there is an independent body of work that addresses static compilation from bulk-synchronous-parallel languages to a variety of Von-Neumann, Vector, and even FPGA architectures. One of the major contributions of this work is to show that these complementary bodies of work can be combined to provide transparent portability of BSP languages across Von-Neumann, Vector, and FPGA architectures.

Dynamic Compilers Though virtual instruction sets and interpreters have existed for decades, their poor performance has historically precluded their widespread adoption. In 1999, DEC released FX32!, a dynamic binary translator from x86 to the Alpha instruction set; FX32! allowed x86 applications to be executed on the 21364 series of Alpha processors [12]. Rather than translating an entire application at once and then executing it, FX32! translated a program as it executed by trapping into the compiler when untranslated code was encountered and saving the translated instructions in a translation cache. Following this project, several languages that were originally compiled into virtual ISAs and interpreted, for example, Java and CLR [50], moved to dynamic compilers that translated from virtual ISAs to the native ISA of a particular processor. Whereas the typical overheads of interpretation resulted in 40x or greater slowdowns over native versions of the same ap-

plication, dynamic compilation and just-in-time translation made virtual ISAs competitive with native compilers. In fact, the ability to use runtime information to inform compiler optimizations spawned several projects such as Dynamo [6] and Jalepeno/Jikes [7] that were able to out-perform native compilers for some applications. The success of these projects lead researchers to explore problems that were even more ambitious, for example, Hydra [60] showed that dynamic compilation could be applied to VLIW architectures, and Transmeta introduced an entire line of VLIW processors coupled with a dynamic compiler that could execute native x86 applications [15]. In the context of this proposal, the rapid development of dynamic compilation frameworks culminated in the release of the NVIDIA PTX virtual ISA and dynamic compiler, which provided a low level ISA for expressing bulk-synchronous-parallel applications and a compiler for mapping this model onto multi-core vector processors (GPUs) [14, 29].

Heterogeneous Compilers One of the primary limitations of heterogeneous systems is the existence of significantly different programming models and languages for different accelerators. Typically an application written using a single programming model will function on only a single type of accelerator, and it is usually not possible to even evaluate the suitability of the same programming model for multiple accelerators. However, there have recently been a few attempts to map parallel programming models typically used for multi-core CPUs to GPUs, for example source-to-source translators from OpenMP [48] and MPI [72] to CUDA have been proposed. Other research efforts have explored mapping GPU programming models to other processors. Stratton et al. have developed MCUDA, a source-to-source translator from CUDA to C that allows the same applications to execute on any CPU with a C compiler and any NVIDIA GPU with a CUDA compiler [71]. This work was extended by Papakonstantinou et al. in FCUDA to include a source-to- source compiler from CUDA to AutoPilot, which is a synthesizable subset of the C language that can be mapped onto FPGAs [64]. This line of work is very interesting because the performance of

MCUDA and FCUDA are competitive with existing languages for FPGAs and multi-core CPUs. It suggests that the CUDA programming model is a good match for several classes of processors, not only GPUs. Furthermore, it suggests that it should be possible to develop a compiler with multiple back-ends that allows applications to be executed on systems with CPUs, GPUs, and FPGAs.

Optimizations for Heterogeneous Compilers With the growing availability of CUDA compilers for different accelerators, it is becoming increasingly important to optimize their performance. Although some existing compiler optimizations can be applied directly to CUDA applications, for example, Murthy et al. have explored loop-unrolling [52], and Dominguez et al. have covered register allocation [22], the explicit notions of parallelism and synchronization create unexplored opportunities for parallel-aware optimizations. For example, Stratton et al. offer a preliminary exploration into the trade-offs between performing redundant computations in parallel and or assigning them to a single thread and then broadcasting the results to all other threads [70]. Another interesting opportunity presented by the CUDA execution model is the ability to map a SIMT (multi-threaded) execution model onto a SIMD (vector) machine. Fung et al. present a hardware optimization that dynamically regroups threads into vector bundles (warps) to improve the utilization of SIMD processors [26]. Tarjan et al. extend this mechanism to split warps on cache misses to hide memory latency [73]. These optimizations have great potential and yet have not been evaluated in CUDA compilers.

1.3 Preliminary Research

In the context of these prior efforts, this proposal introduces the Harmony execution model to address the gap between software complexity and hardware performance in heterogeneous systems. Rather than forcing the developer to manage multiple cores per processor and multiple processors with different micro-architectures and ISAs, Harmony applications are

expressed in an intermediate representation (IR) that is mapped to processors in a heterogeneous system at runtime. The IR is intended to be the target of a high level compiler. It consists of a control and data flow graph of encapsulated data-parallel tasks, which are referred to as kernels, and managed re-sizable regions of memory, which are referred to as variables. Kernel-level-parallelism is used to distribute kernels among multiple processors and data-parallelism is used to map kernels to multi-core processors. The model is based on dynamic detection and tracking of data and control dependencies(which are exposed in the IR), and a decoupling of kernel invocation and kernel scheduling/execution. The approach is inspired by solutions to instruction scheduling and management in out-of-order (OOO) superscalar processors, where those solutions are now adapted to schedule kernels on diverse cores. Flow and control dependencies are first inferred for a window of kernels that have yet to execute and then used as constraints to a scheduler that attempts to minimize execution time or power consumption while satisfying these dependencies. A dynamic compiler is used to generate binaries for available processors and an online performance monitoring framework is used to collect information about kernels as they are executing and refine the kernel scheduler.

1.3.1 The Harmony Execution Model

The core of the Harmony execution model is relatively simple. Programs are expressed in terms of compute kernels, control decisions, and variables, which are analogous to arithmetic instructions, branch instructions, and registers in most ISAs. Programs are specified as a sequence of kernels and control decisions, which can be expressed in a control-flow graph (CFG) form.

Compute Kernels are analogous to functions or procedure calls in imperative programming languages. However, this model places restrictions on compute kernels that enable fast inference of parallelism and independence from processor architecture. First, all input and

output variables must be known when the kernel is invoked, i.e., no pointers embedded in the input/output arguments. This ensures that executing a kernel will not have any side-effects. Second, a kernel must use a single processor exclusively; it cannot, for example, distribute itself across a CPU and a GPU and manage GPU-CPU communication itself. Third, kernels must adhere to the PTX execution model [14]. This allows kernels to include threads and perform local synchronization between groups of threads. However, it forbids global synchronization between groups of threads. Fourth, all input variables must be initialized before a kernel is executed. Finally, kernels may dynamically allocate local memory for scratchpad computations, but this memory may not be persistent across kernel executions. Persistent memory should be specified explicitly as a kernel output variable.

These restrictions give compute kernels the following properties:

- *No Side-Effects* : Kernels are only allowed to modify variables that are declared as outputs prior to the execution of the kernel. The execution of a kernel cannot modify any other state in the system.

- *Atomicity* : From the perspective of the Harmony runtime, kernels are atomic operations that require the existence of a set of input variables and change the state of a set of output variables. The state of the output variables is considered inconsistent until a kernel finishes execution.

- *Exclusivity* : Kernels are executed exclusively by a single processor with a single memory space. They are able to use all of the resources of that processor without interference from other kernels.

- *Determinism* : Kernels are defined in a specific order subject to explicit control decisions. Although kernels may be executed out-of-order, any valid execution of a Harmony program must produce results that are identical to a sequential execution of the program. Coupled with the requirement that kernels may not read uninitialized

data, this ensures that Harmony programs always execute deterministically as long as individual kernels are also deterministic.

Control Decisions are syntactic structures for specifying control-dependent kernel execution and are analogous to branches in that they can potentially change the sequential flow of a Harmony application. Control decisions take in a set of input variables and determine the next compute kernel or control decision to be executed. Control decisions can be conditional or unconditional, and can include multiple possible targets. However, unlike indirect branches in most ISAs, control decisions must specify all possible targets explicitly. The explicit specification of such control dependencies enables transparent optimizations such as speculation and predication to improve concurrency at the level of compute kernels in the same manner that branch prediction improves instruction level parallelism.

Variables in Harmony are managed explicitly to enable runtime support for variable renaming, migration across address spaces, and space optimizations. Variables can be single elements, variably sized arrays, or complex data structures. However, in order to move variables between address spaces, complex data structures require opaque allocation, construction, and copy operators to be implemented application developers or intelligent. The ability to manage variables explicitly in the runtime allows for copy-on-write behavior to improve concurrency, imposing additional dependencies on outstanding kernels to conserve memory, and performing static memory optimizations on the Harmony IR.

The Kernel Control Flow Graph is the low level internal representation of Harmony programs. It is a directed cyclic graph where nodes are a series of interleaved kernel calls terminated by a control decision. Edges originate at control decisions and end at possible targets of a given control decision. Only the first kernel in each node may be the target of a control decision and each node may have at most one control decision. This ensures that nodes have a single entry point and a single exit point. In this representation, the common

```
1   int* p_hmaxs = h_maxs; // initial input data
2   float* p_hvars = h_vars; // initial input data
3
4     // Launch the device computation threads!
5   for (i = 0; i<t-block_num; i+=block_num)
6   {
7       // Launches a kernel
8       PetrinetKernel<<< grid, threads>>>(g_places, g_vars, g_maxs, N,
9           s, 5489*(i+1));
10      // Copies results from the Device to the Host
11      CopyFromDeviceMemory(p_hmaxs, g_maxs, block_num*sizeof(int));
12      CopyFromDeviceMemory(p_hvars, g_vars, block_num*sizeof(float));
13      // Selects the next inputs
14      p_hmaxs += block_num;
15      p_hvars += block_num;
16  }
```

Figure 3: CUDA source code for the inner loop of PNS. CUDA requires applications to be manually partitioned into GPU kernels and C++ statements that are executed sequentially. Harmony treats all kernels, C++ statements, and memory transfer operations as encapsulated kernels that execute out of order subject to data and control dependencies.

term basic block (BB) is used to refer to a node. The only difference between this notion of a BB and the classical notion is that these BBs contain kernel calls rather than instructions.

Harmony Example The execution of an application implementing this model may be viewed as logically equivalent to that of a sequential application executing on a modern superscalar, out-of-order (OOO), processor with instructions replaced by kernels and functional units replaced by heterogeneous processors. During execution, the runtime walks the CFG in program order. It also scans a window of kernels that have yet to be executed using either branch prediction or predication. Dependence-driven scheduling over kernels in a *dispatch window* deploys kernels on available processors. Optimizations such as speculative kernel execution[1] are supported via the separation of kernel execution (affecting local state) and commitment of the results of execution (affecting global state).

The following example is used to illustrate the advantages and potential of this model of kernel level parallelism, as well as identify opportunities and challenges. It focuses on an

[1]Speculative kernel execution is explored in detail in prior Harmony work [19].

example of a sample execution of two iterations through PNS, a petrinet simulation. This application was originally implemented in CUDA as part of the Parboil benchmark suite [36]. In this example, CUDA kernels, memory copy operations, and host code sections are mapped to Harmony kernels by hand to demonstrate how the process could be performed by a high level compiler. The high level structure of the application is visible in the original CUDA source code shown in Figure 3. In CUDA, the program loops over a range of input data and makes blocking kernel calls for each set of data. CUDA interleaves calls to compute kernels, which are dispatched to a selected GPU, with C++ statements, which are executed directly on a CPU. In Harmony, both the CUDA kernels and the C++ statements are mapped to Harmony kernels, control structures (the for-loop in this example) are mapped to control decisions, and regions allocated with cudaMalloc are mapped to Harmony variables.

Using Harmony, the pns source code would be compiled into an intermediate representation that expressed the application only in terms of kernels, control decisions, and variables. Figure 4 shows the basic building blocks of the application represented as a control flow graph of kernels. Control dependencies are expressed explicitly in this representation, which eases their resolution at runtime. PNS is typically used to model distributed systems. This particular version is composed of a loop around a single kernel and memory copy that updates global state.

When the Harmony runtime begins executing this program, it scans through the kernels as they are encountered in the sequential flow of the application without blocking. This process implicitly creates a window of kernels that have been encountered, but not yet executed. As the window is created, the runtime builds a graph representing all of the data dependencies among kernels in the window. The data dependency graph after one iteration through the loop is shown in Figure 5. Control decisions limit the number of kernels that can be scanned without blocking because they potentially change the flow of the application. This limitation can be avoided using techniques like prediction or predication to speculatively scan kernels beyond a control decision as long as there is a method for recovering from misspectulations.

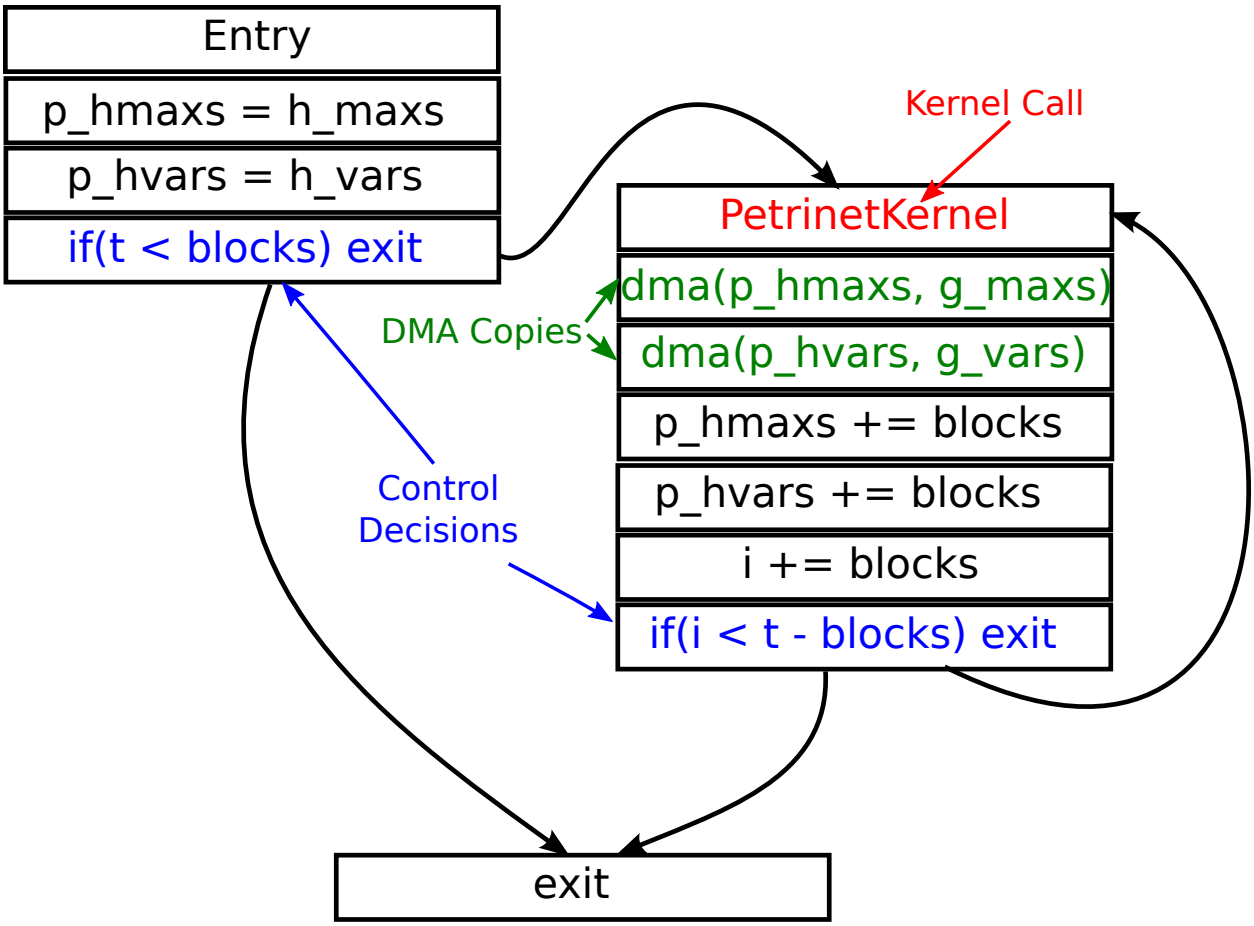

Figure 4: The control flow graph for the PNS application. This is the representation that Harmony programs are compiled into before being executed.

In Figure 6, the kernels from the second iteration have been invoked speculatively.

Another issue arise when two kernels share the same memory, but do not have a producer/consumer relationship. In Figure 4, all of the temporary variables are reused through multiple iterations of the loop. However, the petrinet kernels do not have explicit flow dependencies. These are directly analogous to output and anti dependencies in instruction scheduling. Consequently, variable renaming is used to eliminate non-flow dependences: every time a non flow dependency is found between two kernels, the runtime allocates a new variable to be used instead, effectively breaking the dependency at the cost of increasing the memory footprint of the application. In the dependency graph shown in Figure 6, all of the variables denoted as red nodes have been renamed and replicated.

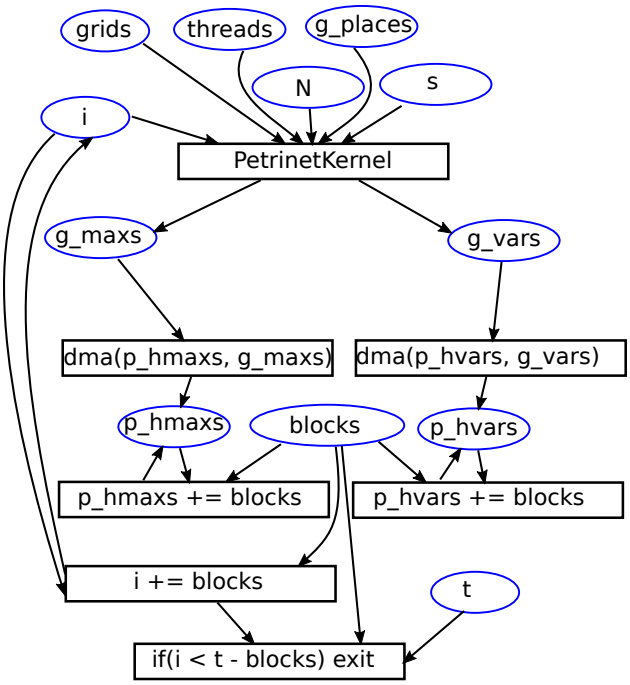

Figure 5: The data flow graph for one iteration through the PNS loop. The black nodes represent kernels and the blue nodes represent variables that are managed by the runtime. Edges represent data dependencies.

The runtime utilizes a machine model description (e.g., number and type of cores) paired with a dynamic compiler (Ocelot) to generate implementations for each kernel for at least one (but possibly many) ISAs. As kernels complete execution, dependencies between kernels in the dispatch window are updated and the schedule is revised. Figure 7 shows an example schedule for the PNS application using a system with a CPU, and two GPUs with different micro-architectures. This scheduling phase creates a dynamic mapping from kernels to heterogeneous architectures, while the existence of binaries for multiple machines permits execution on systems with radically different processor configurations, and performance to scale as more resources are added to a system until kernel level parallelism is exhausted.

Note that the execution time of individual kernels varies across architectures and can even be data dependent (in Figure 7, the petrinet kernel runs slower on the second GPU), yet being able to accurately predict this time a-priori improves the quality of the schedule as pointed out in the first Harmony publications [20] as well as several newer studies performed

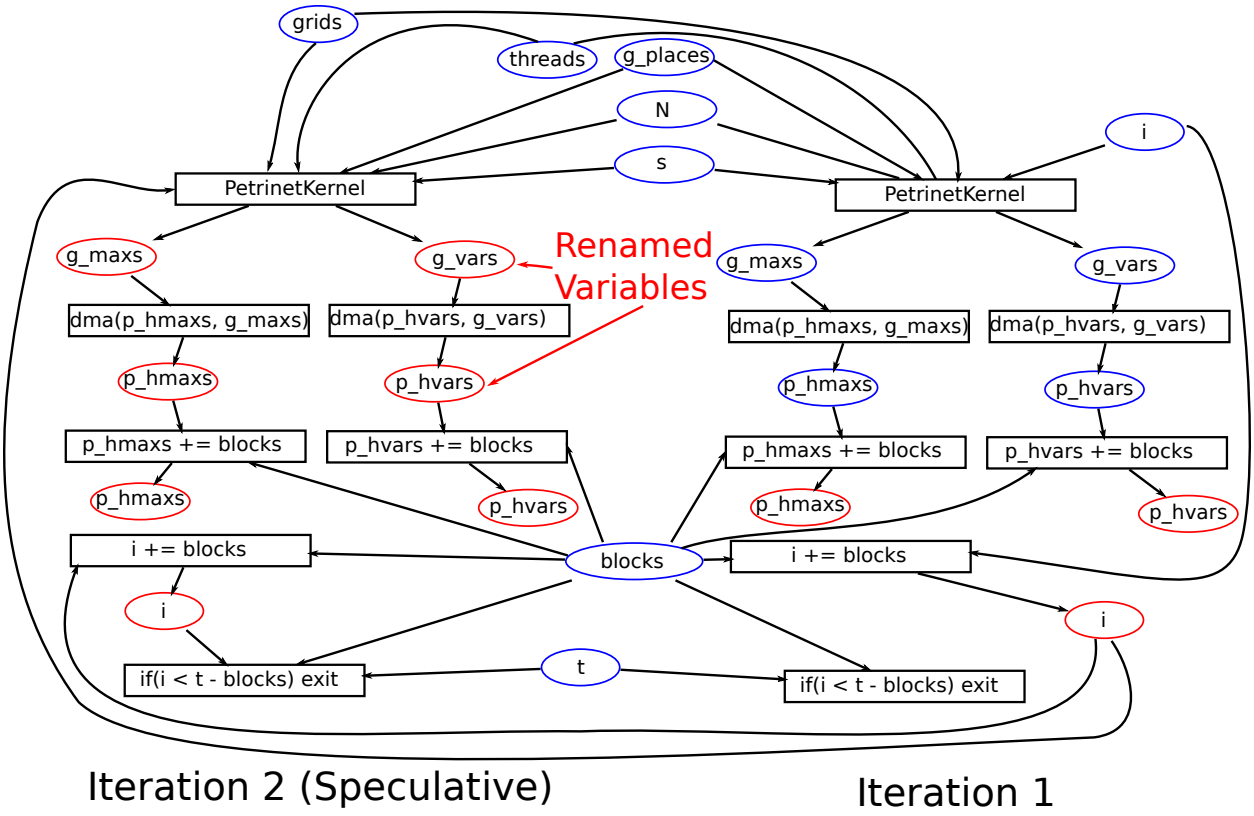

Figure 6: The data flow graph for two iteration through the PNS loop. Kernels from the second iteration have been invoked speculatively and red variables have been renamed to eliminate false data dependencies.

independently [39, 49]. To address this issue, online monitoring is used for measuring the execution time of each kernel as well as a set of kernel characteristics. It is then possible to construct kernel and input dependent models of execution time as a function of target core and utilize these to make more effective scheduling decisions. Work on this topic in the context of the Harmony execution model has been explored in studies on on workload characterization [42] and performance modeling [43], which are uniquely leveraged to build these models.

1.3.2 The Ocelot Dynamic Compiler

Harmony relies on either the existence of multiple binaries for each kernel or a dynamic compiler that can generate code for new processors on demand. The lack of a commercially

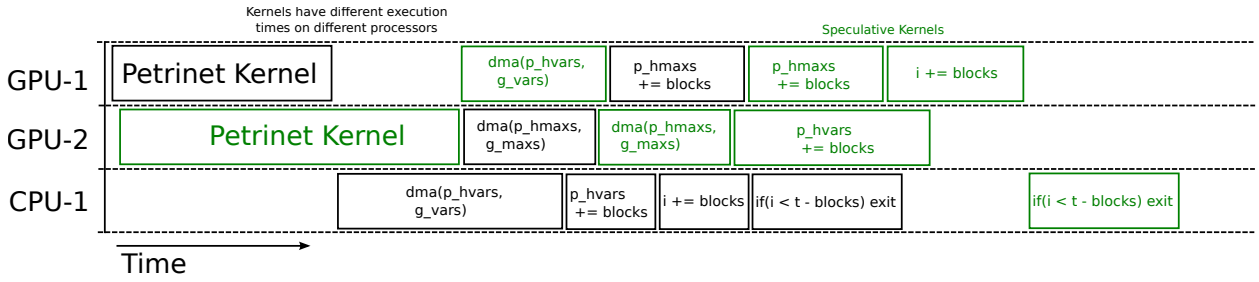

Figure 7: A possible schedule for two iterations of the PNS application on a system with two different GPUs and a CPU. The green kernels in the figure are from the second, speculative, iteration. Note that speculation significantly increases the parallelism in the application. It is also worthwhile to note that the same kernel can have different execution times on different processors.

available dynamic compilation environment[2] lead to the development of Ocelot, a dynamic compilation framework that provides back-end targets for the NVIDIA PTX virtual ISA [14]. Ocelot currently supports NVIDIA GPUs and multi-core CPUs and has been validated on over 130 applications. Using Ocelot as a dynamic compiler, Harmony applications can be written as a collection of CUDA kernels, where the existing NVIDIA compilation chain is leveraged to compile CUDA kernels into PTX binaries. This section details the design of the Ocelot compiler and how the PTX execution model that was originally intended to be executed on NVIDIA GPUs can be mapped to multi-core CPUs, enabling Harmony applications to execute on systems with NVIDIA GPUs and multi-core CPUs.

The PTX Execution Model PTX is a virtual instruction set that explicitly includes semantics for management of parallelism and synchronization. Rather than requiring multiple kernels execute on different cores in the same processor, PTX allows applications to be specified in terms of a large amount of parallelism that is intended to fully utilize the resources of a single parallel processor. The basic unit of execution in PTX is a light-weight thread. All threads in a PTX program fetch instructions from the same binary image, but can take distinct control paths depending on the values of pre-set id registers with unique values for each thread. The first level of hierarchy in PTX groups threads into SIMD units

[2]OpenCL had not been released at the time that the Ocelot project was started.

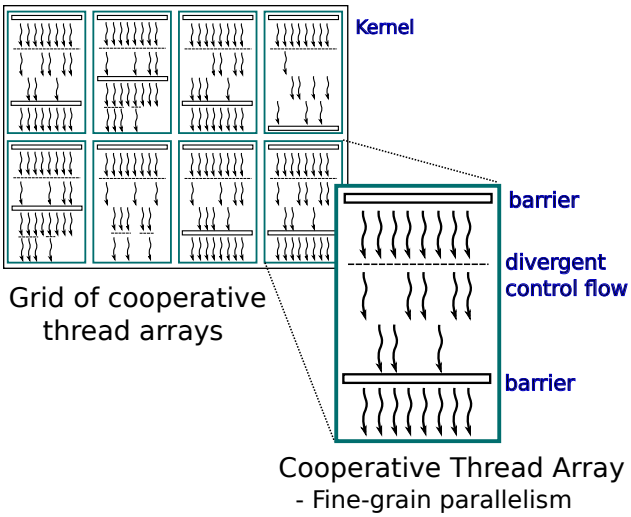

Figure 8: PTX thread hierarchy.

(henceforth warps). The warp size is implementation dependent, and available to individual threads via a pre-set register.

In order to support arbitrary control flow as a programming abstraction while retaining the high arithmetic density of SIMD operations, NVIDIA GPUs provide hardware support for dynamically splitting warps with divergent threads and recombining them at explicit synchronization points. PTX allows all branch instructions to be specified as divergent or non-divergent, where non-divergent branches are guaranteed by the compiler to be evaluated equivalently by all threads in a warp. For divergent branches, targets are checked across all threads in the warp, and if any two threads in the warp evaluate the branch target differently, the warp is split. PTX does not support indirect branches, limiting the maximum ways a warp can be split to two in a single instruction. Fung et. al. [26] show that the PTX dynamic compiler can insert synchronization points at post-dominators of the original divergent branch where warps can be recombined.

As shown in Figure 8, the second level of hierarchy in PTX groups warps into concurrent thread arrays (CTAs). The memory consistency model at this point changes from sequential consistency at the thread level to weak consistency with synchronization points at the CTA level. Threads within a CTA are assumed to execute in parallel, with an or-

dering constrained by explicit synchronization points. These synchronization points become problematic when translating to sequential architectures without software context switching: simple loops around CTAs incorrectly evaluate synchronization instructions. Stratton et. al. [71] show that CTAs with synchronization points can be implemented without multithreading using loops around code segments between synchronization points. CTAs also have access to an additional fixed size memory space called shared memory. PTX programs explicitly declare the desired CTA and shared memory size; this is in contrast to the warp size, which is determined at runtime.

The final level of hierarchy in PTX groups CTAs into kernels. CTAs are not assumed to execute in parallel in a kernel (although they can), which changes the memory consistency model to weak consistency without synchronization. Synchronization at this level can be achieved by launching a kernel multiple times, but PTX intentionally provides no support for controlling the order of execution of CTAs in a kernel. Communication among CTAs in a kernel is only possible through shared read-only data structures in main memory and a set of unordered atomic update operations to main memory.

Collectively, these abstractions allow PTX programs to express an arbitrary amount of data parallelism in an application: the current implementation limits the maximum number of threads to 2^{41} [56]. The requirements of weak consistency with no synchronization among CTAs in a kernel may seem overly strict from a programming perspective. However, from a hardware perspective, they allow multiprocessors to be designed with non-coherent caches, independent memory controllers, wide SIMD units, and hide memory latency with fine grained temporal multithreading: they allow future architectures to scale concurrency rather than frequency. The next section shows how a dynamic compiler can harness the abstractions in PTX programs to generate efficient code for parallel architectures other than GPUs.

Mapping PTX to CPUs Compiling a PTX kernel for execution on multicore CPUs requires first translating the kernel to the desired instruction set then transforming the kernel's execution semantics from PTX's thread hierarchy to multiple host threads. Ocelot could conceivably execute a PTX kernel by translating PTX instructions into native instructions, launching one kernel-level host thread per thread in the CTA and relying on OS-level support for barriers and multi-threading to provide concurrency and context switching. This approach is similar to CUDA emulation mode [56] except the underlying kernel representation is the same PTX representation that would be executed on the GPU. However, CUDA programs are most efficient on GPUs when many light-weight threads are launched to hide memory latency. Launching such a large number of heavy-weight host threads would lead to excessive thread-creation, operating system scheduling, and memory requirements [17].

The strategy adopted by Ocelot is to first translate PTX instructions into the Low Level Virtual Machine [47] and then to map the thousands of logical CUDA threads onto a few host threads in a compile-time transformation that inserts procedures to perform light-weight context switching at synchronization barriers within the kernel, similar to thread fusion described in [71]. A scheduler basic block is inserted at the entry point of the kernel that selects branch targets depending on the currently selected thread and its progress through the kernel. When a thread reaches an explicit synchronization point, control jumps back to the scheduler which stores live variables that have been written, updates the resume point, and jumps to the new thread's next instruction. Live variables are loaded via compiler-inserted stack management instructions as needed and execution continues.

This method does not require function calls or heavy-weight context switches. Thread creation at the start of the CTA incurs no incremental overhead other than the fixed costs of allocating shared and local memory which each worker thread completes once when a kernel is launched. Context switches require spilling live state and an indirect branch. Spilling live registers is the most significant overhead because the indirect branch will be taken identically by all threads in a CTA, which is a behavior that can be correctly predicted by modern CPU

branch predictors. Threads are scheduled in a last-in first-out method to maximize the probability that load instructions used to restore live registers will hit in the cache.

Some CUDA programs are written in a manner that assumes the warp size for the executing processor is at least a particular value, typically 32 threads for current CUDA GPU architectures. While this assumption is stronger than what the execution model guarantees, existing architectures execute such kernels correctly, and the additional clock cycles needed to execute synchronization instructions may be saved. The multicore execution model translation technique described here assumes warp size is equal to 1 thread, so applications must be recompiled with synchronization points following those statements expected to be executed simultaneously on architectures with a larger warp size. This is mentioned here because even though this practice violates the semantics of the PTX execution model, a significant number of existing CUDA applications rely on this behavior and require modification to use with Ocelot.

When an application launches a kernel, a multi-threaded runtime layer launches as many worker threads as there are hardware threads available in addition to a context data structure per thread. This context consists of a block of shared memory, a block of local memory used for register spills, and special registers. Worker threads then iterate over CTAs in the kernel. The execution model permits any ordering of CTAs and any mapping to concurrent worker threads.

PTX defines several instructions which require special handling by the runtime. PTX compute capability 1.1 introduces atomic global memory operators which implement primitive transactional operations such as exchange, compare and swap, add, increment, and max, to name a few. Global memory is inconsistent until a kernel terminates in the execution model, so there were several options for implementing atomic accesses. The simplest option was to place a global lock around global memory. This was a relatively simple to implement compared to emitting explicit atomic operations in the native ISA. However, it turns out that the overheads of this operation were insignificant for even hand-crafted appli-

cations designed to stress the atomic operation throughput of the system as shown in Section 1.4.2. Therefore, more complex alternatives were not considered.

GPUs typically provide hardware support for texture sampling and filtering. PTX defines a texture sampling instruction which samples a bound texture and optionally performs interpolation depending on the GPU driver state. As CPUs do not provide hardware support for texture sampling or interpolation, Ocelot performs nearest and bilinear interpolation in software by translating PTX instructions into function calls which examine internal Ocelot data structures to identify mapped textures, compute addresses of referenced samples, and interpolate accordingly.

Additionally, PTX includes several other instructions that do not have trivial mappings to LLVM instructions. These include transcendental operators such as *cos* and *sin* as well as parallel reduction. These are also implemented by calls into the Ocelot runtime which in turn call C standard library functions in the case of the floating-point transcendentals. Operations such as reductions and votes perform distributed computations across all threads in a warp. These operations are defined for a particular warp size; in the translation to multicore, the warp size is a single thread, so they can be implemented without a context switch.

Dynamic Optimization The CPU back-end currently leverages the LLVM framework for optimization passes that are applied after the parallel PTX execution model is transformed into the sequential LLVM execution model. However, it currently does not take advantage of the SIMD parallelism inherent in PTX to generate vector instructions for use in CPU SSE/Altivec units. This could provide up to a 4x speedup over the current implementation and up to an 8x improvement when AVX is introduced. This transformation will be explored in future work. Finally, Ocelot includes an optimization framework for PTX that supports state of the art compiler optimizations that can consider the synchronization primitives, communication between threads, and the explicit memory hierarchy exposed in PTX.

Application	Description	Problem Size	Control Flow	Model
AES	Encrypts and decrypts a large document using 256-bit AES	3.2 MB Text File	For Loops	Harmony
MonteCarlo	Gaussian Quadrature estimates the area under a normal function	1 Precision value	While Loops	Harmony
MatrixMultiply	Dense matrix multiplication using subblocks	4096x4096 matrices	Nested Loops	Harmony
CapModel3	Risk analysis for adding a new asset to an existing loan portfolio	1000000 Samples	Nested Loops	Harmony
Random	A regression test for Harmony that constructs a CFG of simple kernels with random edges subject to a completion constraint	10 Variables 100 Average Iterations	Random Structure	Harmony
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
MRI-FHD	Computation of an image-specific matrix FHd, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
CP	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed.	40000 Atoms in a 512x512 grid	For Loops	CUDA
SAD	Sum of absolute differences kernel, used in MPEG video encoders.	50KB image	None	CUDA
TPACF	Measures the probability of finding an astronomical body at a given angular distance from another astronomical body.	100 Random Numbers 4096 Points	None	CUDA
PNS	Implements a generic algorithm for Petri net simulation.	2000x2000 matrix	For Loops	CUDA
RPES	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.	20000 molecules	For Loops	CUDA

Table 1: Application Characteristics

1.4 Results

This section briefly presents results from two existing studies. The first study focuses on identifying the amount of kernel-level-parallelism (KLP) in Harmony and CUDA applications, while the second study focuses on the performance of PTX kernels that are compiled for x86 CPUs using Ocelot.

1.4.1 Kernel Level Parallelism in Harmony Applications

This study develops a theoretical formulation of the upper bound of KLP in Harmony and CUDA applications. It shows how this upper bound is impacted by speculation and renaming for the benchmark applications in Table 1. The Harmony applications were written from scratch by the author of this proposal and the CUDA applications were taken from the UIUC Parboil benchmark suite [36].

Application	Kernels	KLP	MIMD	SIMD
CP	10	9.85	256	128
MRI-Q	4	3.91	97.5	320
MRI-FHD	7	6.96	110.57	292.57
SAD	3	2.6	594	70.28
TPACF	1	1.0	156.63	206.11
PNS	112	111.03	17.99	248.88
RPES	71	70.42	64757	40.5799

Table 2: Parallelism in CUDA Applications

At a high level, KLP should express the amount of parallelism within a Harmony or CUDA application in the same way that ILP expresses the amount of instruction level parallelism within a single threaded application. KLP is difficult to formulate exactly as different kernels typically have different, data-dependent execution times as shown in Diamos et al. [20] and Luk et al. [49]. Furthermore, these execution times can be dependent on input data and are typically strongly dependent on processor architecture. They can even include non-deterministic components when run on systems with stateful hardware units like caches, variable performance characteristics caused by DVFS, and software artifacts such as operating system scheduling or dynamic binary translation and optimization.

With these concerns in mind, *kernel level parallelism is defined for an application on a heterogeneous system as the speedup of a parallel execution on a system with an infinite number of accelerators over a sequential execution on the same system where each kernel is run on the accelerator that gives the lowest execution time.* In order to account for possible non-determinism in the execution time of a kernel, the average execution time from the accelerator with the lowest such average time is used.

For Harmony applications, KLP is computed by analyzing traces of the kernels and control decisions launched by an application as it executed. These traces express the average kernel execution time and the input and output variables of each kernel. For CUDA applications, Ocelot [18] was used to instrument all load and store instructions from every kernel in an application. A set of all memory locations written by kernels was maintained along

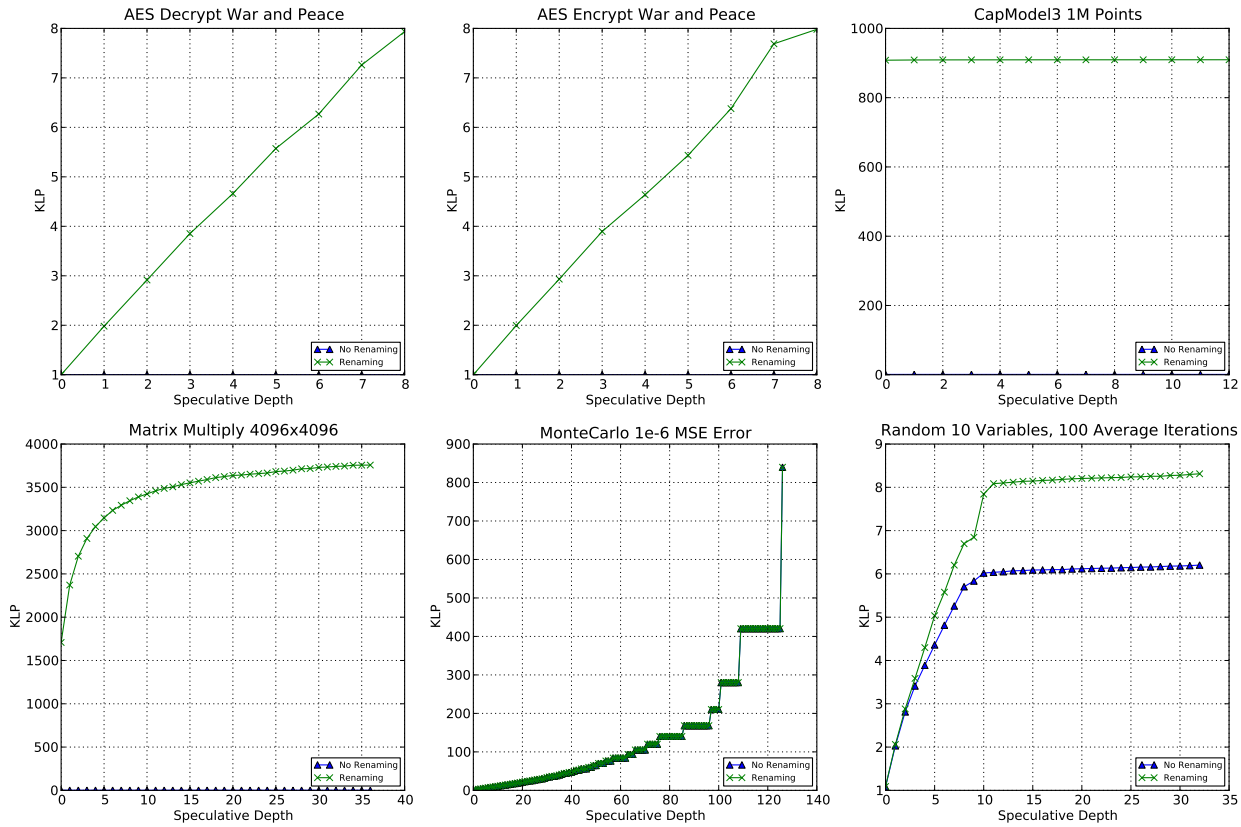

Figure 9: Upper bound on kernel-level-parallelism in Harmony applications. The blue line represents KLP when variable renaming is not allowed. The green line represents KLP with renaming enabled.

with the id of the last kernel to write to that location. If a kernel ever loaded a value that was stored by a previous kernel, a dependency was created between the two kernels. This information was used to create a dependency graph for the entire application. Ocelot did not have the ability to identify control decisions in CUDA applications because it could only analyze PTX kernels, not the complete application, so the best case KLP assuming that all control decisions could be removed via perfect speculation is reported in Table 2. The average MIMD and SIMD parallelism as defined in Kerr et al. [42] within each kernel is also presented for comparison.

Figure 9 shows the computed upper bound on KLP for all of the Harmony applications in our test suite. In this figure, speculative depth refers to the maximum number of control decisions that can be outstanding at a time. These results show a significant amount of KLP

CPU	Intel i920 Quad-Core 2.66Ghz
Memory	8GB DDR-1333 DRAM
CPU Compiler	GCC-4.4.0
GPU Compiler	NVCC-2.3
OS	64-bit Ubuntu 9.10

Table 3: Test System

within all applications tested. For all of the applications except Monte Carlo, renaming provides the most significant boost to KLP and indeed most applications without renaming do not show any improvement from speculation. Monte Carlo stands out because it explicitly uses different variables for the input seed and output result of each Monte Carlo simulation; this demonstrates that it is possible for the programmer to do the equivalent of renaming. However, once renaming has been enabled, being able to remove control decisions via speculation greatly improves KLP. Over all of the applications, it extends the upper bound of KLP by an average of 3.6x over renaming alone. The KLP saturates around a speculative depth of about 10 except for Monte Carlo which continues to scale up to a speculative depth of 133.

For the CUDA applications, KLP is comparable to that of the Harmony applications in all cases except for SAD, TPACF and MRI-Q, which simply do not launch a significant number of kernels. It is possible that increasing the data set size for these benchmarks would improve their KLP. It is also possible that the kernels in these applications would have to be split to expose additional KLP, which would limit the potential benefits of speculation. Kerr et al. [42] show that this is not the common case for CUDA applications, which typically have tens to hundreds of kernels, but it still represents a problem that will have to be addressed in future work in order to apply kernel level speculation to certain CUDA applications.

1.4.2 The Ocelot CPU Backend

The second study covers a preliminary analysis of the performance of several CUDA applications when translated to LLVM. Note that this section is intended to merely provide several

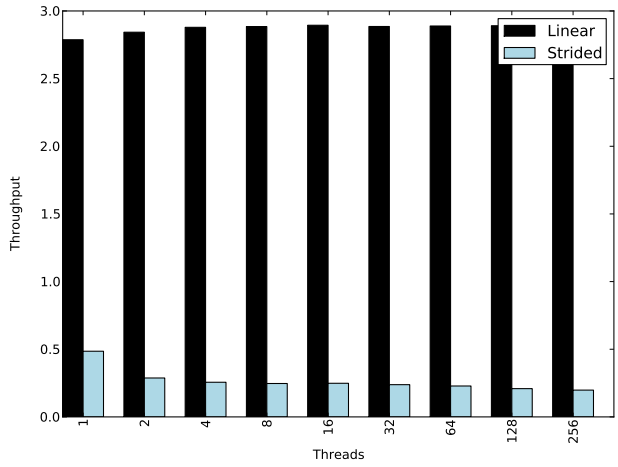

Figure 10: Memory throughput using different access patterns. Note that strided accesses, which result in efficient coalesced accesses on GPUs, are inefficient on the CPU.

distinct points of reference of the scaling and throughput possible when translating CUDA applications to x86. All of the experiments in this section use the system configuration given in Table 3. The section begins with a set of experiments exploring the performance limits of Ocelot using a set of microbenchmarks, moving on to a classification of runtime overheads, and ending with a study of the scalability of several full applications using multiple cores.

Microbenchmarks The first set of experiments focus on several low level PTX benchmarks that were designed to stress various aspects of the system. In order to write and execute PTX programs outside of the NVIDIA compilation chain, which does not accept inlined assembly, the CUDA Runtime API was extended with two additional functions to allow the execution of arbitrary PTX programs. The function **registerPTXModule** allows inserting strings or files containing PTX kernels at runtime and **getKernelPointer** obtains a function pointer to any registered kernel that can be passed directly to **cudaLaunch**.

```
void registerPTXModule( std::istream& module, const std::string& moduleName );
const char* getKernelPointer( const std::string& kernelName,
    const std::string& moduleName );
```

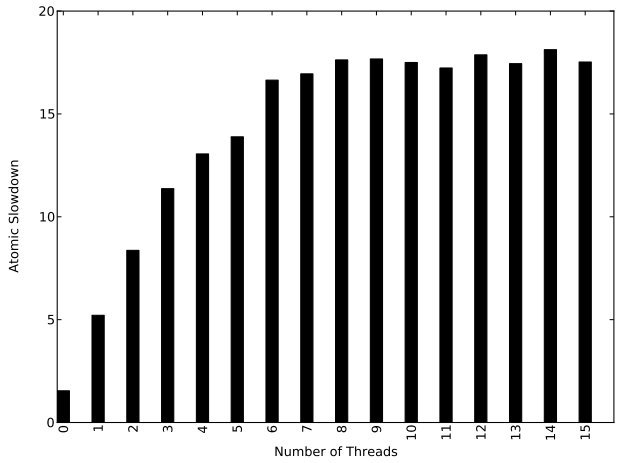

Figure 11: The throughput of atomic operations decreases as the number of host worker threads increases. The overhead of using locks only introduces at worst 20x the overhead over lock-less operations.

Using this infrastructure, memory bandwidth, atomic operation throughput, context-switch overhead, instruction throughput, and special function throughput were explored. These measurements were taken from a real system, and thus there is some measurement noise introduced by lack of timer precision, OS interference, dynamic frequency scaling, etc. These results were taken from the same system and include at least 100 samples per metric. The sample mean is presented in the form of bar charts with 95% confidence intervals for each metric.

Memory Bandwidth. The microbenchmark explores the impact of memory traversal patterns on memory bandwidth. This experiment is based on prior work into optimal memory traversal patterns on GPUs [56], which indicates that accesses should be coalesced into multiples of the warp size to achieve maximum memory efficiency. When executing on a GPU, threads in the same warp would execute in lock-step, and accesses by from a group of threads to consecutive memory locations would map to contiguous blocks of data. When translated to a CPU, threads are serialized and coalesced accesses are transformed into strided accesses. Figure 10 shows the performance impact of this change. The linear access pattern represents partitioning a large array into equal contiguous segments and having each thread traverse a single segment linearly. The strided access pattern represents a pattern

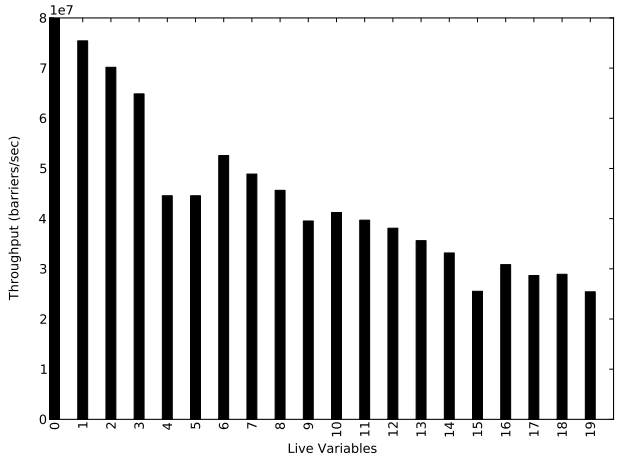

Figure 12: The overheads of context switches are primarily due to spilling live registers. Applications with few live variables are likely to perform well on CPUs.

that would be coalesced on the GPU. It is very significant that the strided access pattern is over 10x slower when translated to the CPU. This indicates that the optimal memory traversal pattern for a CPU is completely different than that for a GPU.

Atomic Operations. The next experiment details the interaction between the number of host worker threads and atomic operation overhead. This experiment uses an unrolled loop consisting of a single atomic increment instruction that always increments the same variable in global memory. The loop continues until the counter in global memory reaches a preset threshold. As a basis for comparison, the same program was run using a single thread to increment a single variable in memory until it reached the same threshold without using atomic operations. Figure 11 shows the slowdown of the atomic increment compared to the single-thread version for different numbers of CPU worker threads. These results suggest that the overhead of atomic operations in Ocelot are not significantly greater than on GPUs.

Context-Switch Overhead. The third experiment explores the overhead of a context-switch when a thread hits a barrier. It consists of an unrolled loop around a barrier, where several variables are initialized before the loop and stored to memory after the loop completes. This ensures that they are all alive across the barrier. In order to isolate the effect of barriers on a single thread, only one thread was and one CTA were launched for this benchmark.

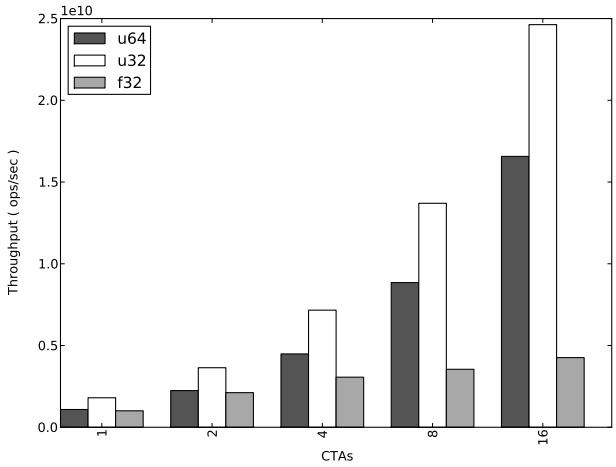

Figure 13: CPUs are more efficient when performing integer operations.

A thread will hit the barrier, exit into the Ocelot thread scheduler, and be immediately scheduled again.

Figure 12 shows the measured throughput, in terms of number of barriers processed per second. Note that the performance of a barrier decreases as the number of variables increases, indicating that a significant portion of a context-switch is involved in saving and loading a thread's state. In the same way that the number of live registers should be minimized in GPU programs to increase the number of thread's that can be active at the same time, programs translated to the CPU should actively try to minimize the number of live registers to avoid excessive context-switch overhead.

Instruction Throughput. The fourth microbenchmark attempts to determine the limits on integer and floating point instruction throughput when translating to a CPU. The benchmark consists of an unrolled loop around a single PTX instruction such that the steady state execution of the loop will consist only of a single instruction. 32-bit and 64-bit integer add, and floating point multiply-accumulate were tested. The results are shown in Figure 13. The theoretical upper bound on integer throughput in the test system is 3 integer ALUs $*$ 4 cores $*$ 2.66 $*$ 10^9 $cycles/s$ = 31.2 $*$ $10^9 ops/s$. 32-bit adds come very close to this limit, achieving 81% of the maximum throughput. 64-bit adds achieve roughly half of the maximum throughput. 32-bit floating point multiply-accumulate operations are much

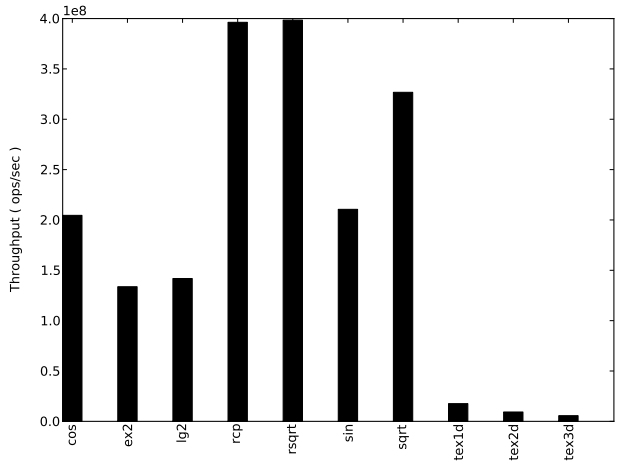

Figure 14: Special instruction throughput. Texture operations are significantly slower without hardware support.

slower, only achieving 4GFLOPs on all 4 cores. This is slower than the peak performance of the test system, and more exploration into the generated x86 machine code is needed to understand exactly why. These results suggest that code translated by Ocelot will be relatively fast when performing integer operations, and slow when performing floating point operations.

Special Function Throughput. The final microbenchmark explores the throughput of different special functions and texture sampling. This microbenchmark is designed to expose the maximum sustainable throughput for different special functions, rather than to measure the performance of special functions in any real application. To this end, the benchmarks launch enough CTAs such that there is at least one CTA mapped to each worker thread. Threads are serialized in these benchmarks because there are no barriers, so the number of threads launched does not significantly impact the results. The benchmarks consist of a single unrolled loop per thread where the body consists simply of a series of independent instructions. To determine the benchmark parameters that gave the optimal throughput, the number of iterations and degree of unrolling was increased until less than a 5% change in measured throughput was observed. Eventually, the configuration yielding the greatest throughput was determined to be 16 CTAs, 128 threads per CTA, and 2000 iterations

Application	Startup Latency (s)	Teardown Latency (s)
CP	4.45843e-05	6.07967e-05
MRI-Q	3.48091e-05	8.55923e-05
MRI-FHD	3.62396e-05	8.4877e-05
SAD	4.14848e-05	5.45979e-05
TPACF	3.48091e-05	8.70228e-05
PNS	4.48227e-05	8.53539e-05
RPES	4.17233e-05	6.12736e-05

Table 4: Kernel Startup And Teardown Overhead

each of which contains a body of 100 independent instructions. Inputs to each instruction were generated randomly using the Boost 1.40 implementation of Mersenne Twister, with a different seed for each run of the benchmark. The special functions tested were reciprocal (rcp), square-root (sqrt), sin, cos, logarithm base 2 (lg2), 2^{power} (ex2), and 1D, 2D, and 3D texture sampling.

Figure 14 shows the maximum sustainable throughput for each special function. The throughputs of these operations are comparable when run on the GPU, which uses hardware acceleration to quickly provide approximate results. Ocelot implements these operations with standard library functions, incurring the overhead of a fairly complex function call per instruction in all cases except for rcp, which is implemented using a divide instruction. Rcp can be used as a baseline, as it shows the throughput of the hardware floating point divider. Based on these results, the special operation throughput using Ocelot is significantly slower than the GPU, even more so than the ratio of theoretical FLOPs on one architecture to the other. Additionally, the measurements include a significant amount of variance due to the random input values. This is a different behavior than the GPU equivalents, which incur a constant latency per operation.

Runtime Overheads The next set of experiments were designed to measure the startup cost of each kernel, the overhead introduced by optimizing LLVM code before executing it, and finally the contribution of various translation operations to the total execution time of a program.

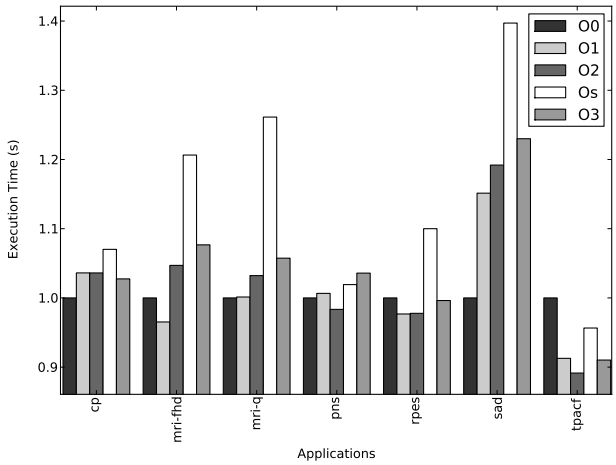

Figure 15: Performance of the same applications with different levels of optimization. The overheads of performing optimizations at runtime must be weighed against the speedup provided by executed more optimized code. The best optimization level is application dependent.

Kernel Startup and Teardown. The use of a multi-threaded runtime for executing translated programs on multi-core CPUs introduces some overhead for distributing the set of CTAs onto the CPU worker threads. Ocelot was instrumented using high precision linux timers to try to measure this overhead. Table 4 shows the measured startup and teardown cost for each kernel. Note that the precision of these timers is on the order 10us, thus the results indicate that the startup and teardown costs are less than the precision of the timers. These are completely negligible compared to the overheads of translation and optimization. They suggest that there is room for dynamic work distribution mechanisms such as work stealing that have greater overheads.

Optimization Overhead. The next experiment attempts to measure the relative overhead of applying different levels of optimization at runtime by instrumenting the optimization passes in Ocelot to determine the amount of time spent in optimization routines. The experiment was run on every application in the parboil benchmark suite to identify any differences in optimization time due to the input program's structure. Figure 15 shows that O3 is never more than 2x slower than O1. Optimization for size is almost identical to O2 in all cases, and O3 is only significantly slower than O2 for pns and sad.

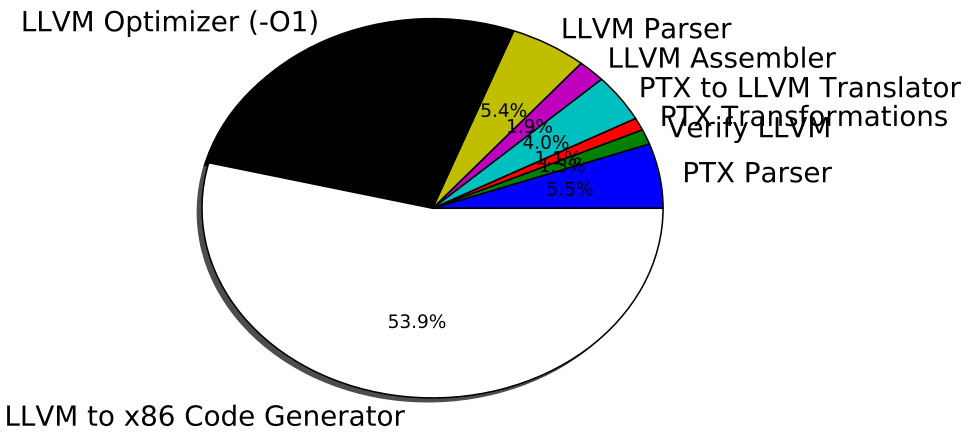

Figure 16: The breakdown of time spent in different Ocelot modules. x86 code generation is the most significant source of overheads.

The best optimization level depends significantly on the application. For CP, MRI-Q, and SAD, the overhead of performing optimizations can not be recovered by improved execution time, and total execution time is increased for any level of optimization. The other applications benefit from O1, and none of the other optimization levels do better than O1 (except for O2 on pns). Note that the LLVM to x86 JIT always applies basic register allocation, peephole instruction combining, and code scheduling to every program regardless of optimizations at the LLVM level. These may make many optimizations at the LLVM level redundant, not worth dedicating resources to at execution time. Also, note that the optimizations used here were taken directly from the static optimization tool OPT, which may not include optimizations that are applicable to dynamically translated programs. A more comprehensive study is needed to identify optimizations are applicable to applications that are sensitive the optimization complexity.

Component Contribution. As a final experiment into the overheads of dynamic translation, callgrind [54] was used to determine the relative proportion of time spent in each translation process. Note that callgrind records basic block counts in each process, which may be different than total execution time. Figure 16 shows that the vast majority of the translation time is spent in the LLVM code generator. The decision to use a new LLVM

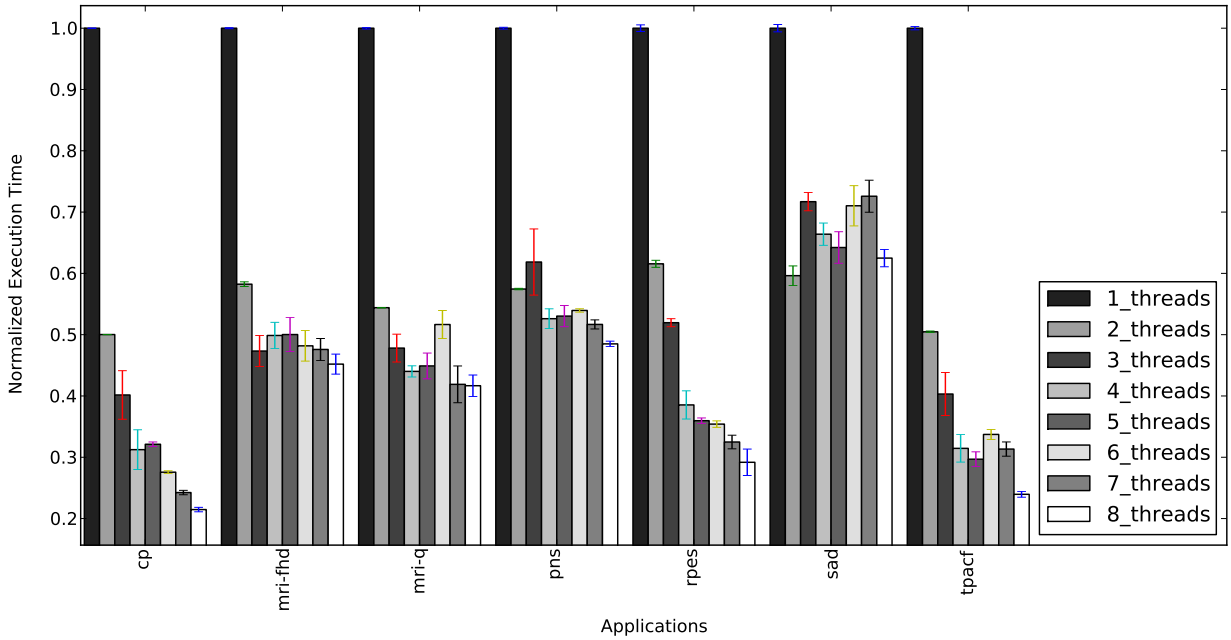

Figure 17: The scaling of applications from the Parboil benchmark suite on 4 cores and 2-way simultaneous multi-threading.

IR in Ocelot only accounts for 6% of the total translation overhead. The time it takes to translate from PTX to LLVM is less than the time needed to parse either PTX or LLVM, and the speed of Ocelot's PTX parser is on par with the speed of LLVM's parser. LLVM optimizations can be a major part of the translation time, but performing PTX-to-PTX transformations in Ocelot takes less than 2% of the total translation time. These results justify many of the design decisions made when implementing Ocelot's CPU back-end.

Full Application Scaling Moving on from micro-benchmarks to full applications, this experiment studies the ability of CUDA applications to scale to many cores on a multi-core CPU. Test system includes a processor with four cores, each of which supported hyper-threading. Therefore, perfect scaling would allow performance to increase with up to 8 CPU worker threads. This is typically not the case due to shared resources such as caches and memory controllers, which can limit memory bound applications.

The Parboil benchmarks were used as examples of real CUDA applications with a large number of CTAs and threads; previous work on Ocelot shows that the Parboil applications

launch between 5 thousand and 4 billion threads per application [42]. Figure 17 shows the normalized execution time of each application using from 1 to 8 CPU worker threads. All of the applications scale well to two threads, but not necessarily beyond that. The CP benchmark is able to achieve better than a 4x speedup using 8 threads, indicating that it is probably compute bound and is able to benefit from hyperthreading. Conversely, SAD slows down when the number of threads is increased beyond two, indicating that the additional threads are probably competing for memory bandwidth and cache occupancy. These results suggest that some applications are significantly more suited to execution on multi-core CPUs than others. In future work, it would be interesting to determine if these trends hold on GPUs as well, or if there are some applications that scale well on GPUs, but poorly on CPUs and visa versa.

1.4.3 Conclusion

This proposal introduces the Harmony execution model and the Ocelot dynamic compiler as tools for reducing the complexity of designing applications for heterogeneous systems. Applications are written in a high level language using kernels similar to CUDA and OpenCL and then compiled into kernels, control decision, and variables primitives in the Harmony execution model. Harmony programs are executed by a runtime that creates a dynamic mapping between kernels and available processors. The runtime uses the Ocelot dynamic compiler to generate different binaries for heterogeneous processors. Preliminary results show that there is a significant amount of kernel level parallelism in Harmony applications that can be used to scale performance as processors are added to a system. Additional results show that the same Bulk-Synchronous-Parallel representation of a kernel can be efficiently compiled to CPUs and GPUs.

2 Statement of Dissertation Topic

2.1 Proposed Research

2.1.1 Objective

An Execution Model For Heterogeneous Many-Core Systems *The objective of the proposed work is to mitigate the growing complexity of programming parallel and heterogeneous systems through the design and evaluation of a novel execution model and runtime system.* The execution model will provide a single interface to systems with multiple heterogeneous-ISA processors. It will include semantics for expressing applications in terms of encapsulated compute kernels and memory objects with explicit data and control dependencies that can be cleanly mapped to loosely coupled hardware resources. The runtime system will map the execution model onto available hardware resources such that application throughput will be increased as more resources are added and correct functionality will be maintained with a minimal set of hardware. The runtime will employ novel dynamic optimization techniques such as kernel level speculation, dynamic kernel re-compilation, and statistical models for performance prediction to optimize an application as it executes. The proposed work extends existing models for streaming and data parallel computing by introducing abstractions that directly address ISA and micro-architecture heterogeneity. It will significantly reduce the complexity of designing applications for heterogeneous systems, and allow more applications to benefit from the combined power efficiency and compute throughput of domain specific accelerators.

2.1.2 Research Challenges

The impending emergence of heterogeneous and many-core systems presents a unique opportunity to revisit the realization of high impact computing problems on more capable hardware. It also significantly complicates software development by exposing many distinct points of interaction between systems and software as well as allowing these points of in-

terfaces to vary across system configurations. In summary, the mainstream adoption of heterogeneous and many-core platforms is precluded by the following research challenges:

- **Execution Models:** The vast majority of existing execution models manage systems at the granularity of a single core in a single processor. Processors with multiple cores require the interaction of different instances of the same execution model. Systems with heterogeneous processors require interaction between different execution models. This state of the art over-complicates the hardware/software interface and creates a pressing need for models that can be used to reason about the resources available in a system collectively.

- **Portability:** Heterogeneous systems vary significantly in scale as well as capability. Re-partitioning, re-tuning, and even re-compiling applications for each instance of a heterogeneous system is an excessive and ultimately impractical burden to place on developers. Systems that dynamically map applications to available hardware resources are essential.

- **Performance Scaling:** Moore's law technology scaling is expected to drive the explosion of core and memory resources as well as the inclusion of additional domain specific accelerators. Ideally, a new execution model would enable the performance of existing applications to transparently scale with the capabilities of future hardware similar to frequency and ILP scaling in existing models. The end of frequency and ILP scaling necessitate new solutions.

2.1.3 Research Contributions

In the context of the challenges presented in the previous section, this work makes the following research contributions to address each challenge:

- **For Execution Models:** This work introduces the Harmony execution model, which provides a single interface between applications and heterogeneous many-core systems.

Applications are expressed in terms of encapsulated compute intensive kernels which are in turn expressed as bulk-synchronous-parallel computations. Kernels are constrained by explicit data and control flow analogous to instructions in traditional ISAs. The similarity between the kernel model of Harmony and existing ISAs allows a large body of existing optimizations to be applied directly and the bulk-synchronous form of kernels allows them to be mapped to parallel processors.

- **For Application Portability:** The runtime component of Harmony addresses application portability by dynamically mapping and dynamically compiling kernels to available processors on-demand. System administrators can remove processors for maintenance or install upgrades transparently to the application.

- **For Performance Scaling:** Performance scaling is accomplished by 1) using dynamic compiler optimizations that tailor synchronization, concurrency, and data movement operations in the bulk-synchronous kernel model to the capabilities of the selected hardware, and 2) dynamically extracting kernel-level-parallelism from complete applications that can be mapped to integrated processors or discrete accelerators.

2.1.4 Work Remaining

There has already been a significant amount of work completed that explores the design and evaluation of the proposed Harmony execution model. A prototype implementation of the Harmony runtime has been released publicly under an open-source license that supports GPU and CPU targets using static binaries for each kernel [16]. A stable version of the Ocelot dynamic compiler has been completed and validated against 131 applications [18]. Several workload characterization studies have been performed using Ocelot [42, 43] that identify program characteristics correlated with performance on different processors. Speculative kernel execution and variable renaming have been enabled in the Harmony runtime and studied for a variety of applications [19]. Additionally, a limit study on the feasibility of

off-loading Harmony kernels to remote processors has yielded promising results [63].

In order to complete this body of work, I would like to finish the integration of the Harmony runtime with the Ocelot dynamic compiler and add support for the remote execution of kernels to the Harmony runtime. The integration of Harmony and Ocelot will allow for existing PTX kernels to be used for both CPU and GPU platforms, enabling the evaluation of model-driven dynamic optimizations for existing CUDA applications. Remote execution of kernels will allow a system to be assembled dynamically, composed of neighboring processors in a heterogeneous cloud or cluster. These systems are becoming increasingly important with the introduction of accelerator clouds [32] and accelerator clusters such as Keeneland [78] and Lincoln [24].

Harmony-Ocelot Integration Harmony and Ocelot currently exist as separate code bases. Harmony requires two identical versions of each kernel, one x86 binary and one PTX binary, in order to fully utilize a system with GPUs and CPUs. The Harmony backend needs to be extended to use Ocelot API functions to translate PTX binaries into GPU and CPU binaries and execute them on different processors. Ocelot also requires modification. The current API only supports the execution of CUDA applications. Functions like concurrent kernel execution, PTX to LLVM translation, and dynamic kernel re-optimization are not supported via this API and are instead handled internally in Ocelot. A new Ocelot API needs to be created that exposes this functionality to Harmony. Finally, Ocelot has not been rigorously tested with applications that attempt to use multiple GPU/CPU devices concurrently, and there are likely to be bugs within the existing implementation. Some effort will have to be devoted to testing and stabilizing this concurrent kernel execution functionality.

Predictive Performance Models have been identified by our prior work [20] and other independent studies [30,39] as being necessary for efficient kernel scheduling in heterogeneous systems. Our recent work has culminated in the development of a statistical framework based on principal component analysis, clustering, and regression modeling for building models that

are able to accurately predict expected kernel execution times on different processors based on a set of program and processor metrics [43]. Harmony currently includes a basic model for performance prediction based on kernel input size. However, our recent work has identified five program characteristics, only one of which is input size, that significantly influence kernel execution time. The integration of Harmony and Ocelot will allow the other four program characteristics to be collected at runtime and incorporated into the models and scheduler used in Harmony.

Distributed Systems are a natural fit for the Harmony execution model. The difficulties normally associated with programming distributed systems such as non-uniform address spaces and message passing based communication are also applicable to most heterogeneous system configurations. A simple extension of Harmony to support distributed systems would involve spawning Harmony clients on remote systems to identify and manage remote processors. The interface used to allocate memory, perform DMA transfers, and launch kernels on directly attached processors would have to be extended to fall back on message passing when referencing a remote processor. The Harmony performance models would need to be updated to accurately model the overheads of transferring data and code remotely so that the scheduler would only launch kernels remotely if the overheads could be offset by exploiting additional kernel-level-parallelism. Finally, it would be interesting to explore offloading fused groups of kernels remotely and then unpacking them recursively as in a distributed diffusing computation [21, 51]. However, this is probably not feasible in the timeframe allocated for this work. It may be possible instead to evaluate this technique analytically instead.

2.2 Tasks

This section provides an overview of the tasks required to complete the proposed work.

2.2.1 Harmony-Ocelot Integration

- Ocelot API
  - Create functions exposing memory management, kernel translation, and kernel optimization.
  - Expect Duration: 2 weeks
- Concurrent Kernel Execution
  - Add API support for launching multiple kernels in Ocelot concurrently.
  - Expect Duration: 3 weeks
- Harmony Backend
  - Modify the Harmony backend to launch kernels via Ocelot rather than directly.
  - Expect Duration: 2 weeks
- Experimentation/Analysis
  - Test the performance of low level benchmarks and full applications using the new functionality.
  - Expect Duration: 2 weeks
- Writing
  - Summarize the results into a coherent report.
  - Expect Duration: 2 weeks

2.2.2 Distributed Systems

- Device Discovery
  - Implement a remote Harmony client that runs on remote machines and services queries as to the number and type of available processors.
  - Expect Duration: 1 weeks
- Remote Device Management
  - Create a message passing API for allocating memory on remote devices, transferring kernel binaries, invoking kernels, and receiving acknowledgements when kernels complete.
  - Expect Duration: 3 weeks
- Remote Kernel Execution
  - Add runtime support for selecting remote devices as targets for Harmony kernels.

  - Expect Duration: 3 weeks

- Scheduler Enhancements
  - Add models for remote processors, including message passing overheads for signaling and data transfers. Include new performance models developed in prior work [43].
  - Expect Duration: 3 weeks

- Experimentation/Analysis
  - Assemble a distributed system. Test the performance of micro-benchmarks and full applications with and without remote execution.
  - Expect Duration: 2 weeks

- Writing
  - Summarize the results into a coherent report.
  - Expect Duration: 2 weeks

2.3 Schedule

This section presents a tentative schedule for the aforementioned tasks.

- **April 2010** – *This proposal.*
- **August 2010** – Ocelot API development, unit tests, validation. Concurrent Kernel Execution development finished, unit tests written.
- **September 2010** – Concurrent Kernel Execution tests pass. Harmony Backend development, unit tests, validation. Experiment design, benchmarking.
- **October 2010** – Experiment data collection and analysis.
- **November 2010** – Writing Summary.
- **December 2010** – Harmony on Distributed Systems. Device Discovery implementation, unit tests, validation. Remote Device Management implementation, unit tests, validation.
- **January 2011** – Remote device management implementation, unit tests, validation. Remote kernel execution implementation.
- **February 2011** – Remote kernel execution unit tests and validation. Scheduler enhancements implementation, unit tests, and validation.
- **March 2011** – Experiments and analysis.
- **April 2011** – Writing.

2.4 Facilities Needed

GPU Workstations There are currently 3 GPU workstations that can be used as test platforms that are representative of single nodes in a distributed heterogeneous system.

1. **WorkStation 1:**
   - *CPU* - Intel Core i7 920 - 2.667 Ghz
   - *Memory* - 6 GB
   - *GPU1* - GeForce 8800 GTS 320MB - 12 Cores - 900 Mhz
   - *GPU2* - Tesla C1060 4096 MB - 30 Cores - 1.15Ghz
2. **WorkStation 2:**
   - *CPU* - AMD Phenom 9550 - 2.2 Ghz
   - *Memory* - 8 GB
   - *GPU1* - Tesla C1060 4096 MB - 30 Cores - 1.15Ghz
   - *GPU2* - GeFore 280GTX 1024 MB - 30 Cores - 1.2 Ghz
3. **WorkStation 3:**
   - *CPU* - Intel Core2 E8500 - 2.66 Ghz
   - *Memory* - 4 GB
   - *GPU1* - 2 GeForce 9800GX2 512 MB - 16 Cores - 1.3 Ghz
   - *GPU2* - 4 Tesla s870 - 16 Cores - 1.1 Ghz

Embedded Systems Although the bulk of this work considers the benefits of heterogeneity from a high-performance perspective, it would also be interesting to evaluate platforms that were constrained by power consumption, battery life, or size. These platforms are becoming increasingly important in the consumer space, where Apple [4], NVIDIA [59], and Intel [38] have all released low power platforms with tightly coupled CPU-GPU processors. Having an NVIDIA Ion platform available for testing Harmony and Ocelot applications would allow comparisons of power-efficiency and energy-efficiency as well as execution time in the high performance systems. These are relatively cheap, and readily available, but will be needed to be purchased.

Keeneland Keeneland is a $12 million Track 2 award from NSF for an experimental high-performance computing (HPC) system incorporating a large number of GPU nodes. A small number of Keeneland nodes are expected to be to be made available at Georgia Tech for application prototyping and these can be used to represent large scale heterogeneous systems. The Keeneland nodes are expected to consist of:

- **Hewlett Packard Nodes**
  - Dual socket Intel 2.8 GHz Nehalem-EP
  - 24 GB Main memory per node
- **NVIDIA Servers**
  - Fermi GPUs
  - InfiniBand 4x QDR w/ full bisection interconnect

References

[1] AMD. Brook. `http://developer.amd.com/gpuassets/AMD-Brookplus.pdf`, 2007.

[2] AMD. R600/r700/evergreen assembly language format. Technical report, 2009.

[3] AMD. Opencl: The open standard for parallel programming of gpus and multi-core cpus. `http://ati.amd.com/technology/streamcomputing/opencl.html`, 2010.

[4] APPLE. ipad technical specifications and accessories for ipad. http://www.apple.com/ipad/specs/, 2009.

[5] Nitin Arora, Aashay Shringarpure, and Richard W. Vuduc. Direct n-body kernels for multicore platforms. In *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, pages 379–387, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[7] B. Banerjia, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Dolby, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[9] Doug Burger and James R. Goodman. Billion-transistor architectures. *Computer*, 30(9):46–49, 1997.

[10] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[11] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[12] Anton Chernoff and Ray Hookway. Digital fx!32 running 32-bit 86 applications on alpha nt. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 2–2, Berkeley, CA, USA, 1997. USENIX Association.

[13] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[14] NVIDIA Corp. Ptx: Parallel thread execution. `http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`, 2009.

[15] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Gregory Diamos. Harmony: A runtime for heterogeneous multi-core systems. `http://code.google.com/p/harmonyruntime/`, 2009.

[17] Gregory Diamos. State explosion: An obvious limitation to strong scaling. Technical report, NFinTes, 2009.

[18] Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. Ocelot: A dynamic compilation framework for ptx. `http://code.google.com/p/gpuocelot/`, 2009.

[19] Gregory Diamos and Sudakhar Yalamanchili. Speculative execution on Multi-GPU systems. In *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[20] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *HPDC'08*, Boston, Massachusetts, USA, June 2008. ACM.

[21] Edsger W. Dijkstra and C.S.Scholten. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1–4, 1980.

[22] Rodrigo Dominguez, David R. Kaeli, John Cavazos, and Mike Murphy. Improving the open64 backend for gpus. Technical report, Dept. of Electrical and Computer Engineering, 2009.

[23] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[24] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[26] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[28] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2009.

[29] Vinod Grover, Sean Lee, and Andrew Kerr. Plang: Translating nvidia ptx language to llvm ir machine, 2009.

[30] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Programming gpu accelerators with aspects and code transformation. In *First Workshop on Programming Models for Emerging Architectures (PMEA)*, 2009.

[31] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[32] Hoopoe. Hoopoe: Cloud services for gpu computing, 2010.

[33] Amir Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Macross: Macro-simdization of streaming application. In *ASPLOS-XV: Proceedings of the 15th international conference on Architectural support for programming languages and operating systems*, Pittsburg, PA, USA, 2010.

[34] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan1, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC10: IEEE International Solid-State Circuits Conference*, 2010.

[35] Buck Ian, Foley Tim, Horn Daniel, Sugerman Jeremy, Fatahalian Kayvon, Houston Mike, and Hanrahan Pat. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[36] IMPACT. The parboil benchmark suite, 2007.

[37] Intel. Ct: C for throughput computing.

[38] Intel. Intel atom processor n400 series. http://download.intel.com/design/processor/datashts/322847.p 2009.

[39] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, and S. Naffziger. An x86-64 core implemented in 32nm soi cmos. In *ISSCC10: IEEE International Solid-State Circuits Conference*, 2010.

[41] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Champaign, IL, USA, 1993.

[42] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[43] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Submitted to Third Workshop on General-Purpose Computation on Graphics Procesing Units*, Pittsburg, PA, USA, March 2010.

[44] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion-transistor era: Iram. *Computer*, 30(9):75–78, 1997.

[45] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2008. ACM.

[46] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, Washington, DC, USA, 2004. IEEE Computer Society.

[47] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[49] ChiKeung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO'09*, New York, USA, devember 2009. IEEE.

[50] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the common language runtime, 2000.

[51] Jayadev Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 4(1):37–43, 1982.

[52] Giridhar Sreenivasa Murthy, Muthu Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for gpgpu programs. In *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[53] Hashem H. Najaf-abadi and Eric Rotenberg. Architectural contesting: exposing and exploiting temperamental behavior. *SIGARCH Comput. Archit. News*, 35(3):28–35, 2007.

[54] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[55] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[56] NVIDIA. Cuda reference manual. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf`, 2009.

[57] NVIDIA. Nvidias next generation cuda compute architecture: Fermi. Technical report, 2009.

[58] NVIDIA. Nvidia opencl jumpstart guide. `http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf`, 2010.

[59] NVIDIA. User guide: Tegratm 200 series. `http://developer.download.nvidia.com/tegra/docs/Tegra%20200_Series_DevBoard_UserGuide_DG04927001v01.pdf`, 2010.

[60] Kunle Olukotun, Jules Bergmann, Kun Chang, and Basem A. Nayfeh. Rationale, design and performance of the hydra multiprocessor. Technical report, Stanford, CA, USA, 1994.

[61] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[62] Peter S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[63] Sudnya Padalikar and Gregory Diamos. Exploring the latency and bandwidth tolerance of cuda applications. Technical report, NFinTes, 2009.

[64] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. High-performance cuda kernel execution on fpgas. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 515–516, New York, NY, USA, 2009. ACM.

[65] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, 1997.

[66] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, 2000.

[67] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30(9):68–74, 1997.

[68] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[69] H. Spaanenburg. Multi-core/tile polymorphous computing systems. pages 1 –4, may 2008.

[70] John Stratton, Vinod Grover, Jaydeep Marathe, Baastian Aarts, Mike Murphy, Ziang Hu, and Wen mei Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO 2010*, Toronto, Canada, April 2010.

[71] John Stratton, Sam Stone, and Wen mei Hwu. Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[72] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[73] David Tarjan, Jiayuan Meng, and Kevin Skadron. Increasing memory miss tolerance for simd cores. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[74] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[75] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[76] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[77] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[78] Jeffrey Vetter, Jack Dongarra, Karsten Schwan, Richard Fujimoto, and Thomas Schulthess. Keeneland: National institute for experimental computing, 2010.

[79] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: The raw machine. Technical report, Cambridge, MA, USA, 1997.

[80] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 52–61, New York, NY, USA, 2008. ACM.

[81] Xin David Zhang. A streaming computation framework for the cell processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2007.