# Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems

Gregory Diamos
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
gregory.diamos@gatech.edu

Andrew Kerr
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
arkerr@gatech.edu

Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
sudha@ece.gatech.edu

Nathan Clark
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
ntclark@cc.gatech.edu

## ABSTRACT

Ocelot is a dynamic compilation framework designed to map the explicitly data parallel execution model used by NVIDIA CUDA applications onto diverse multithreaded platforms. Ocelot includes a dynamic binary translator from Parallel Thread eXecution ISA (PTX) to many-core processors that leverages the Low Level Virtual Machine (LLVM) code generator to target x86 and other ISAs. The dynamic compiler is able to execute existing CUDA binaries without recompilation from source and supports switching between execution on an NVIDIA GPU and a many-core CPU at runtime. It has been validated against over 130 applications taken from the CUDA SDK, the UIUC Parboil benchmarks [1], the Virginia Rodinia benchmarks [2], the GPU-VSIPL signal and image processing library [3], the Thrust library [4], and several domain specific applications.

This paper presents a high level overview of the implementation of the Ocelot dynamic compiler highlighting design decisions and trade-offs, and showcasing their effect on application performance. Several novel code transformations are explored that are applicable only when compiling explicitly parallel applications and traditional dynamic compiler optimizations are revisited for this new class of applications. This study is expected to inform the design of compilation tools for explicitly parallel programming models (such as OpenCL) as well as future CPU and GPU architectures.


*PACT'10,* September 11–15, 2010, Vienna, Austria.


## Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*processors, compilers*; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*multiple data stream architectures*

## General Terms

Design, Experimentation, Performance

## 1. INTRODUCTION

The mainstream adoption of many-core architectures has resulted in an excess of applications that can no longer fully utilize the resources available in modern processors. Applications that could once leverage frequency and ILP scaling to transparently improve performance are now confined to a single core: they are restricted to an amount of chip area that is shrinking with Moore's law. The problem of designing applications that perform well on modern as well as future multi-core architectures has been pushed up the stack into programming languages, execution models, and compilation chains. Much of the progress made towards this goal has been pioneered by the graphics community, which has almost uniformly adopted a bulk-synchronous [5] programming model [6] coupled to architectures that abandon global cache coherence and strong memory consistency in favor of coarse- and fine-grained parallelism [6–8].

Unfortunately, moving to an explicitly parallel, bulk-synchronous programming model significantly changes the problem presented to a compiler; the hardware target may have a different degree of parallelism relative to what is expressed in the programming model. Whereas compilers for sequential or implicitly parallel programming models were required to automatically extract instruction or thread level parallelism from applications, *compilers for explicitly parallel applications must* ***reduce*** *the degree of parallelism* to match the resources available in a given processor. This

fundamental change in the problem definition coupled with new ISA semantics for dealing with parallel execution will effect a significant change in the design of future compilers.

Scalable bulk-synchronous applications will also require dynamic compilation and binary translation. At least three independent studies have shown that the most efficient processor for a given application is dependent both on application characteristics and input-data [9–11]. Binary translation allows a runtime to select high-throughput accelerators if they are available or fall back on slower processors for cross-platform compatibility or load balancing. Whether binary translation is done using hardware decoders as in modern Intel and AMD x86 processors or in software as in NVIDIA and AMD GPUs or Transmeta CPUs [12], the evolution of the processor micro-architecture in response to circuit and technology constraints will mandate changes in the hardware/software interface. Binary translation will be necessary to bridge this gap, but it will not be enough alone to ensure high performance on future architectures. Dynamic and life-long program optimization will be needed to traverse the space of program transformations and fit the degree of parallelism to the capabilities of future architectures that have yet to be designed.

This paper is written in the context of these challenges. It makes the following contributions:

- A model for the translation of explicitly parallel bulk synchronous applications to a multi-threaded execution model.
- A description of the complete implementation and detailed performance evaluation of the Ocelot dynamic compiler, which can generate code for both CPU and GPU targets at runtime.
- Empirical evidence that there are significant differences in the most efficient code transformations for different architectures. For example, thread-serializing transformations that result in sequential memory accesses perform significantly better on CPU platforms whereas approaches that result in strided, coalesced accesses are more appropriate when targeting GPUs.
- A sensitivity evaluation of existing compiler optimizations applied to CUDA applications.

Ocelot is an open source project that is intended to provide a set of binary translation tools from PTX to diverse many-core architectures. It currently includes an internal representation for PTX, a PTX parser and assembly emitter, a set of PTX to PTX transformation passes, a PTX emulator, a dynamic compiler to many-core CPUs, a dynamic compiler to NVIDIA GPUs, and an implementation of the CUDA runtime. The emulator, many-core code generator, and GPU code generator support the full ptx1.4 specification and have been validated against over 130 CUDA applications. Although Ocelot is designed to target a variety of possible architectures, this paper focuses on compilation for shared-memory multi-core CPUs. The intent is to expose the unique problems associated with compiling explicitly parallel execution models to many-core architectures and evaluate several potential solutions.

# 2. MODEL FORMULATION

This paper addresses the problem of compiling explicitly parallel programs to many-core architectures. Rather than providing an abstract and generic solution, this paper evaluates a real implementation for the specific case of PTX programs and many-core x86 CPUs. The approach advocated by this paper is to 1) start with a highly parallel, architecture independent, specification of an application, 2) perform architecture specific, parallel to serial, transformations that fit the amount of parallelism to the resources available in hardware, 3) generate code and execute the application utilizing all of the available hardware resources, and 4) collect performance information at runtime and possibly apply different transformations to under-performing applications. It is shown in Figure 1. The implementation described in this paper, Ocelot, begins with pre-compiled PTX/CUDA applications and targets CPUs with multiple cores, a shared global memory space, coherent caches, and no on-chip scratch-pad memory. To the best of our knowledge, this implementation is the first example of a dynamic compiler from a bulk-synchronous programming model to an x86 many-core target processor.

This section covers the salient of features of NVIDIA's Parallel Thread Execution (PTX) ISA that make it a suitable intermediate representation for many-core processors as well as related work that influenced the design of Ocelot.

## 2.1 A Bulk-Synchronous Execution Model

One could speculate that PTX and CUDA grew out of the development of Bulk-Synchronous Parallel (BSP) programming models first identified by Valiant [5]. PTX defines an execution model where an entire application is composed of a series of multi-threaded *kernels*. Kernels are composed of parallel work-units called Cooperative Thread Arrays (CTAs), each of which can be executed in any order subject to an implicit barrier between kernel launches. This makes the PTX model similar to the original formulation of the BSP programming model where CTAs are analogous to BSP tasks.

The primary advantage of the BSP model is that it allows an application to be specified with an amount of parallelism that is much larger than the number of physical cores without incurring excessive synchronization overheads. The expectation is that global synchronization (barriers) will eventually become the fundamental limitation on application scalability and that their cost should be amortized over a large amount of work. In the case of PTX, a program can launch up to $2^{32}$ CTAs per kernel. CTAs can update a shared global memory space that is made consistent at kernel launch boundaries, but they cannot reliably communicate within a kernel. These characteristics encourage PTX and CUDA programmers to express as much work as possible between global synchronizations.

As a final point, PTX extends the BSP model to support efficient mapping onto SIMD architectures by introducing an additional level of hierarchy that partitions CTAs into threads. Threads within a CTA are grouped together into logical units known as *warps* that are mapped to SIMD units using a combination of hardware support for predication, a thread context stack, and compiler support for identifying reconverge points at control-independent code [13]. In contrast with other popular programming models for SIMD architectures that require vector widths to be specified explicitly, the aforementioned techniques allow warps to be automatically mapped onto SIMD units of different sizes. The next section briefly revisits the topic of how these ab-

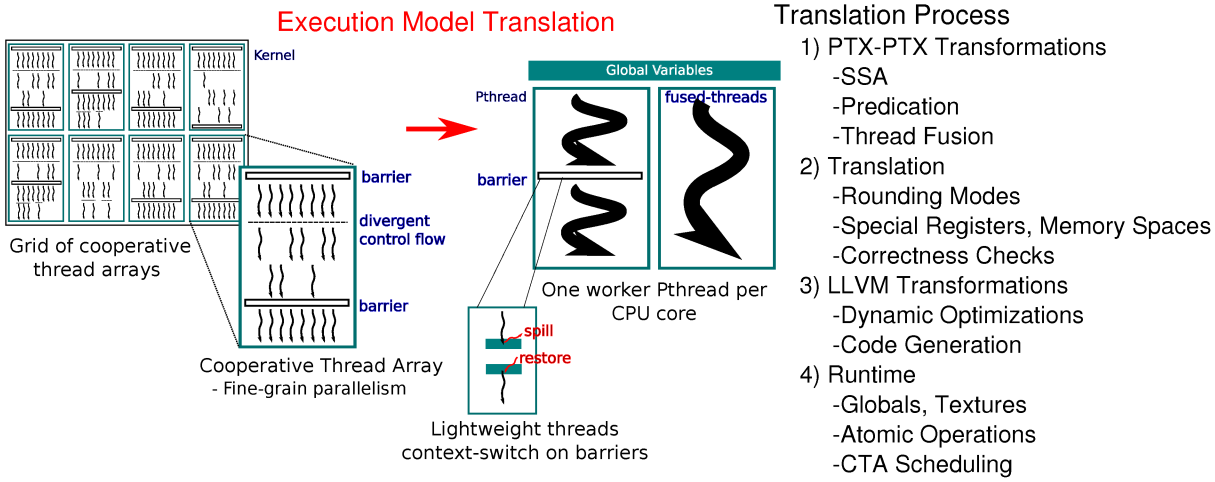


**Figure 1: An overview of the translation process from PTX to multi-threaded architectures.**

stractions, which were intended to scale across future GPU architectures, can be mapped to many-core CPU architectures as well.

## 2.2 Mapping The Model To A Machine

The goal of Ocelot is to provide a just-in-time compiler framework for mapping the PTX BSP model onto a variety of many-core processor architectures. The existence of such a framework allows for the same high level representation of a program to be executed efficiently on different processors in a heterogeneous system. It also enables the development and evaluation of a new class of parallel-aware dynamic compiler optimizations that operate directly on an IR with explicit notions of threads, synchronization operations, and communication operations. Rather than focusing on a single core and relying on a runtime layer to handle processors with multiple cores, the compiler can treat a many-core processor as a single device: it is responsible for generating code that fully utilizes all of the core and memory resources in a processor.

This topic has previously been explored from three complementary perspectives: 1) a static compiler from CUDA to multi-core x86 described in Stratton et al. [14] and extended by the same authors in [14], and 2) a dynamic compiler from PTX to Cell by Diamos et al. [15], and 3) a characterization of the dynamic behavior of PTX workloads by Kerr et al. [16]. From this body of work, the following insights influenced the design of the Ocelot dynamic compiler.

- From MCUDA: PTX threads within the same CTA can be compressed into a series of loops between barriers using *thread-fusion* to reduce the number of threads to match the number of cores in a processor.
- From the PTX to Cell JIT: Performing the compilation immediately before a kernel is executed allows the number and configuration of threads to be used to optimize the generated code.
- From PTX Workload Characterization: Dynamic program behavior such as branch divergence, inter-thread data-flow, and activity factor can significantly influence the most efficient mapping from PTX to a particular machine.

## 2.3 Thread Fusion

Mapping CTAs in a PTX program onto set of parallel processors is a relatively simple problem because the execution model semantics allow CTAs to be executed in any order. A straightforward approach can simply iterate over the set of CTAs in a kernel and execute them one at a time. Threads within a CTA present a different problem because they are allowed to synchronize via a local barrier operation, and therefore must be in-flight at the same time. In MCUDA, Stratton et al. suggest that this problem could be addressed by beginning with a single loop over all threads and traversing the AST to apply "deep thread fusion" at barriers to partition the program into several smaller loops. Processing the loops one at a time would enforce the semantics of the barrier while retaining a single-thread of execution. Finally, "universal" or "selective" replication could be used to allocate thread-local storage for variables that are alive across barriers.

MCUDA works at the CUDA source and AST level, whereas Ocelot works at the PTX and CFG level. However, Ocelot's approach applies the same concept of fusing PTX threads into a series of loops that do not violate the PTX barrier semantics and replicating thread local data.

## 2.4 Just-In-Time Compilation

With the ability of the compiler to i) fuse threads together [14], ii) redefine which threads are mapped to the same SIMD units [13], iii) re-schedule code to trade off cache misses and register spills [17], and iv) migrate code across heterogeneous targets [15], recompiling an application with detailed knowledge of the system and a dynamic execution profile can result in significant performance and portability gains.

The first industrial implementations of dynamic binary translation were pioneered by Digital in FX32! [18] to execute x86 binaries on the Alpha microarchitecture. Transmeta extended this work with a dynamic translation framework that mapped x86 to a VLIW architecture [12]. Several Java compilers have explored translating the Java Virtual Machine to various backends [19]. Even the hardware schemes that translate x86 to microops used in AMD and Intel x86 processors can be considered to be completely hardware forms of binary translation. These forms of binary translation are used to enable compatibility across architectures with different ISAs.

Another complementary application of binary translation is for program optimization, instrumentation, and correct-

ness checking. The Dynamo [20] system uses runtime profiling information to construct and optimize hot-paths dynamically as a program is executed. Pin [21] allows dynamic instrumentation instructions to be inserted and removed from a running binary. Valgrind [22] guards memory accesses with bounds checking and replaces memory allocation functions with book-keeping versions so that out-of-bounds accesses and memory leaks can be easily identified.

Several challenges typically faced by dynamic binary translation systems were simplified by the CUDA programming model and exploited in the design of Ocelot. Most significantly, 1) kernels are typically executed by thousands or millions of threads, making it significantly easier to justify spending time optimizing kernels, which are likely to be the equivalent of hot paths in serial programs; 2) the self-contained nature of CUDA kernels allows code for any kernel to be translated or optimized in parallel with the execution of any other kernel, without the need for concerns about thread-safety; 3) code and data segments in PTX are kept distinct and are registered explicitly with the CUDA Runtime before execution, precluding any need to distinguish between code and data by translating on-the-fly.

## 2.5 Profile-Aware Compilation

Effective dynamic compilation requires low overhead and accurate predictions of application performance to apply optimizations intelligently. Kerr et al. have recently identified several metrics that can be used to characterize the behavior of PTX applications [16]. Example metrics include the amount of SIMD and MIMD parallelism in an application, control flow divergence, memory access patterns, and inter-thread data sharing. Bakhoda et al. [23] and Collange et al. [24] take a more architecture-centric approach by showing the impact of caches, interconnect, and pipeline organization on specific workloads. Taken together, this body of work provides basis for identifying memory access patterns, control flow divergence, and data sharing among threads as key determinants of performance in PTX programs. Ocelot's many-core backend focuses on efficiently handling these key areas.

In addition to revisiting the insights provided by previous work, the design of Ocelot exposed several other problems not addressed in prior work, most significantly 1) on-chip memory pressure, 2) context-switch overhead, and 3) variable CTA execution time.

# 3. IMPLEMENTATION

This section covers the specific details of Ocelot's PTX to x86 many-core dynamic compiler. At a high level, the process can be broken down into the following operations: 1) performing transformations at the PTX level to create a form that is representable in LLVM, 2) translation from PTX to LLVM, 3) LLVM optimizations and native code generation, 4) laying out memory, setting up the execution environment, and initializing the runtime that executes the program on a many-core processor. LLVM was chosen due to the similarities between the LLVM ISA and the PTX ISA, the stability and maturity of the codebase, and the relative complexity of back-end code generators for x86. The key issues and solutions for addressing each of the preceding operations are shown in the right of Figure 1, they are described in the following section.

## 3.1 Building The PTX IR

After PTX assembly programs are extracted from CUDA binaries and registered with the Ocelot runtime, each PTX program is parsed into an abstract syntax tree (AST). Once the AST has been generated, a Module is created for each distinct AST. Ocelot borrows the concept of a Module from LLVM [25] which contains a set of global variables and functions. Similarly, the concept of a Module in Ocelot contains a set of global data and texture variables which are shared among a set of kernels. The portions of the AST belonging to distinct kernels are partitioned and the series of instructions within each kernel are used to construct a control flow graph for each kernel.

## 3.2 PTX to PTX Transformations

During translation to the LLVM ISA, concepts associated with the PTX thread hierarchy such as barriers, atomic operations, votes, as well as the exact number and organization of threads are lost. Subsequently, parallel-aware optimizations are performed at this stage.

An optimization pass interface was designed where different optimization "Passes" can be applied to a Module, a Kernel, or a basic block. A similar design is used in LLVM. It is motivated by the idea that a manager can orchestrate the execution of a series of optimization passes in a way that improves the performance of generated code or improves the performance of the optimizer, which is critical to optimizations that are applied dynamically. For example, the optimizer could apply the series of passes to each block before moving on to the next one to improve the locality of data accessed. Two PTX optimizations were implemented to reverse if-conversion (since the LLVM IR does not support predication) and modify the control flow structure such that the semantics of a PTX barrier were satisfied even when executing the program with a single thread.

**PTX SSA Form**. For any individual PTX kernel in Ocelot, a data-flow graph mirrors the control-flow-graph and retains information at the basic block level in the form of live-in and live-out register sets. These sets are computed using iterative data-flow analysis. The control-flow-graph and data-flow-graph are kept separate and computed lazily to reduce the overheads of performing full iterative data-flow for targets that do not require it (such as the Ocelot PTX emulator).

A PTX kernel begins in partial SSA form (infinite registers but no phi nodes). Conversion to full SSA form is useful for some optimizations and necessary for translation to LLVM. It is done using the live-in and live-out sets for each basic block where each live-in register with at least two predecessors is converted into a phi node. As PTX does not have a concept of a PHI node, these are maintained separately in the data-flow-graph rather than the control-flow-graph.

**Reversing If-Conversion**. LLVM does not support predication. Instead it includes a conditional select instruction similar to the PTX *selp* instruction. In order to handle PTX code that uses predicated instructions that update variables (as opposed to predicated branches which do not conditionally update registers), it is necessary to convert from predicated instructions in PTX to select instructions in LLVM. However, SSA form significantly complicates the conversion from predication to conditional selection.

Consider the example PTX code shown in the upper left of Figure 2 before converting into SSA form. After convert-

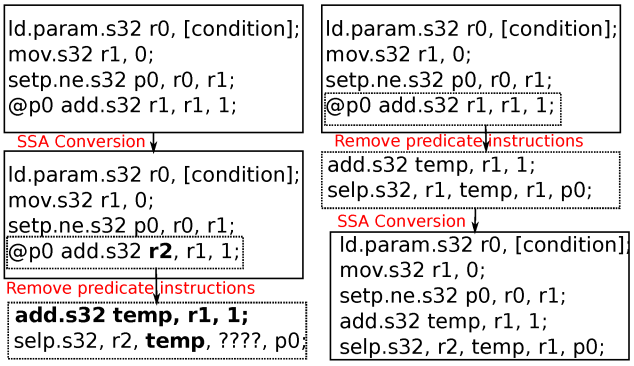


**Figure 2: An example of removing predicated code in PTX.**

ing into SSA form (middle left of the figure), the destination of the add instruction is assigned a new register (r2). Now, converting the predicated add instruction to a regular add followed by a conditional select instruction becomes problematic. The original predicated add instruction could map to a non-predicated add paired with a select as shown in the bottom left of the figure. However, it is not possible to easily determine the new value of r2 if the predicate is not true. It is much simpler to insert conditional select instructions before converting into SSA form, as in the upper right of the figure. In which case it is simple to determine that r1 should be the value of r2 if the predicate condition is false (bottom right). This example reiterates the point that SSA form is complicated by predication. From an engineering perspective, it is difficult to move between SSA forms that support predication (PTX in Ocelot) and those that do not (LLVM).

**Deep Thread-Fusion**. The semantics of the PTX barrier instruction state that all threads execute up to the barrier before any thread executes beyond the barrier. In order to handle this case, each kernel is broken into sub-kernels beginning at either the program entry point or a barrier, and ending at either a barrier or the program exit point. The solution is to iterate over each sub-kernel to ensure that the semantics of a barrier are retained. However, it is still necessary to handle registers that are live across the barrier because they represent thread-local state that would otherwise be lost during a context switch.

Live registers are maintained by creating a register spill area in local memory for each thread. For each kernel exit point ending in a barrier, all live registers are saved to the spill area before exiting the kernel. For every kernel entry point beginning with a barrier, code is added that restores live registers from the spill area. The definition of local memory ensures that the spill area will be private for each thread, so this transformation can be applied directly at the PTX level.

Figure 3 shows an instructive example of this process. The left kernel contains a single basic block with a barrier in the middle. The right figure shows the program control flow graph after removing barriers. The immediate successor of the entry block decides whether to start from the original kernel entry point or the barrier resume point. The left successor of this block is the original kernel entry point and the right block is the barrier resume point. Note that two live registers are saved at the end of the left-most node. They are restore in the rightmost block. During execution, all threads will first execute the leftmost block then they will execute the rightmost block and exit the kernel.

For a series of barriers, multiple resume points will be created. After translation, the LLVM optimizer is used to convert the chain of entry blocks into a single block with an indirect jump. This approach is logically equivalent to deep-thread-fusion as described by Stratton et al. [14], although it works with the PTX control-flow-graph rather than the CUDA AST. [1]

## 3.3 Translation to LLVM

Figure 4 shows the translated LLVM assembly for a simple PTX program. The basic approach is to perform naive translation as fast as possible and rely on subsequent optimization passes to generate more efficient code. PTX instructions are examined one at a time and an equivalent sequence of LLVM instructions is generated[2]. It is assumed that PTX transformations have been applied to a kernel and that it has been converted into full SSA form. Translation begins by creating an LLVM function for each PTX kernel. This function is passed a single parameter which contains the context for the thread being executed. This makes generated code inherently thread-safe because each thread context can be allocated and managed independently.

Once the function has been created, the PTX control flow graph is walked and each basic block is examined. For each basic block, the data-flow-graph is examined to obtain the set of PHI instructions and one LLVM PHI instruction is emitted for each PTX PHI instruction. Each PTX instruction in each block is dispatched to a translation function for that instruction, which generates an equivalent sequence of LLVM instructions. This simpler approach was chosen (rather than translating multiple instructions concurrently) to reduce translator complexity.

Some special translation cases are mentioned below.

**Rounding Modes**. PTX supports all of the IEEE754 rounding modes (to nearest, to infinity, to -infinity, to zero). However, LLVM only supports rounding to the nearest int. Support for infinity and to -infinity is emulated by respectively adding or subtracting 0.5 before rounding a number. This introduces one extra instruction of overhead. To zero is supported by determining if the number is greater or less than zero and conditionally adding or subtracting 0.5. This introduces three extra instructions of overhead. None of these transformations conform to the IEEE standard, and do affect the precision of generated code. Modifications to the LLVM ISA or high overhead emulation would be required for full standards compliance.

**Special Registers**. Special Registers in PTX are used to provide programs with a fast mechanism of obtaining status information about the thread that is currently executing. They allow single instruction access to the thread's id, the CTA's id, the CTA dimensions, the kernel dimensions, several performance counters, and the thread's mapping to a

9[1] Stratton et al. have recently extended their approach to eliminate the need for the scheduler block at the program entry point [14], and this approach could also be used in Ocelot in future work. However, it is not included in this implementation.

9[2] It is interesting to note that this makes the processing of translating each PTX instruction completely independent and suitable for data-parallel translation. Future work could explore writing translation passes in a data parallel language like CUDA and offloading them to accelerators.

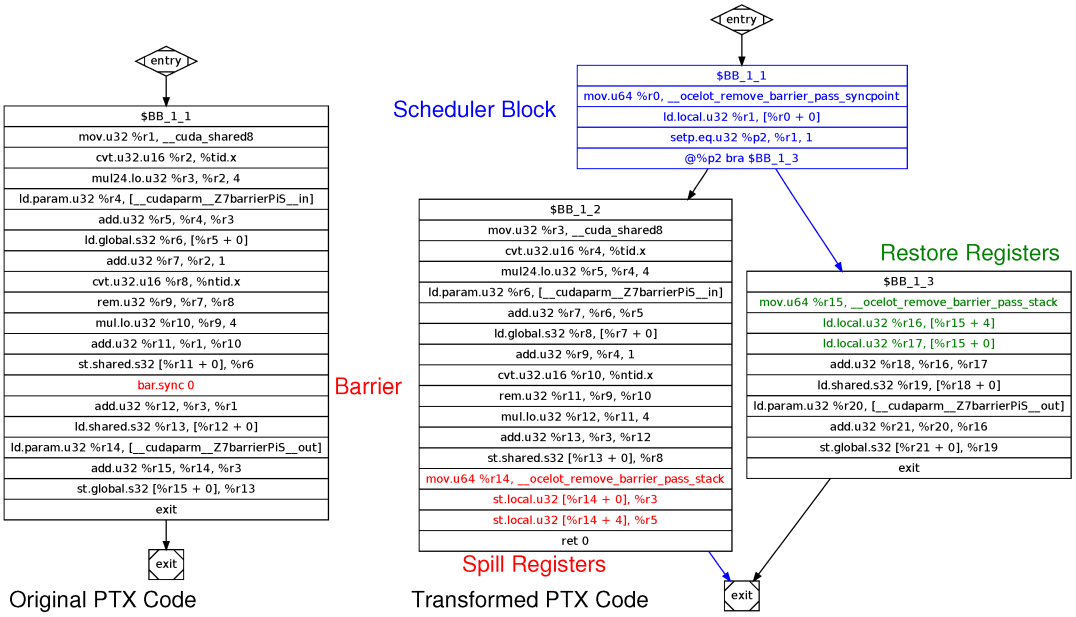


**Figure 3: Example of PTX Barrier Conversion. Blue edges are branch targets and black edges are fall-through targets. The left CFG is split at the barrier, all live registers are spilled before the split point, a scheduler block is inserted to direct threads to the resume point, and all live registers are restored at the resume point.**

warp. A lightweight thread context is allocated for each host thread; it is updated to reflect the currently executing thread on a context-switch via a pointer bump. For PTX instructions that try to read from a special register, loads are issued to the corresponding field in the thread's context.

**Special Functions**. Special Functions such as sin, cos, exp, log, inverse, texture interpolation, etc are supported as instructions in PTX and are accelerated by dedicated hardware in GPUs. CPUs generally do not have equivalent support so these instructions are translated into function calls into an emulation library. This introduces a significant overhead when executing these instructions on CPU platforms.

**Different Memory Spaces**. PTX supports six distinct memory spaces: parameter, local, shared, global, constant, and texture. These spaces are handled by including a base pointer to each space in the context of the current thread and adding this base pointer to each load or store to that space. The texture, constant, parameter, and global memory spaces are shared across all CTAs in a kernel. However, each CTA has a unique shared memory space and each thread has a unique local memory space. These spaces present a problem because allocating separate shared memory to each CTA and local memory to each thread would consume an excessive amount of memory as described by Diamos [26].

This problem is addressed by only allowing a single CTA to be executed by each CPU core at a time. Only enough memory to support $threads_{per-cta} * CPU_{cores}$ is allocated at a time and this memory is shared across different CTAs and threads in a kernel. In order to avoid excessive allocation/deallocation operations, the shared and local memory spaces are never deallocated and only reallocated when they increase in size.

**Memory Errors**: Out of bounds memory accesses, misaligned memory accesses, or memory races are common programmer errors that are difficult to debug on GPU platforms. Array bounds checking, misalignment detection, and memory race detection are supported in Ocelot by adding guards to translated memory operations which check against allocated memory regions, which are managed by Ocelot. These checks are selectively enabled or disabled at different optimization levels, which control whether or not the checking operations are inserted into the translated code.

## 3.4 LLVM Transformations

LLVM provides a very comprehensive library of optimizing compiler transformations as well as either a static code emitter or a lazy code emitter. The static emitter will generate x86 instructions for the entire kernel before executing it. The lazy emitter will generate x86 instructions dynamically, as the program is being executed by trapping segfaults generated when the program tries to jump into a block that has not yet been compiled. The lazy emitter is complicated by the fact that Ocelot executes kernels in parallel using multiple host threads. At the very least, the x86 code generator would need to lock the generated code region so that only a single thread could update it at a time. This would introduce overhead into the compilation and execution of a kernel. PTX kernels are typically small with good code coverage and are cooperatively executed by thousands of threads so Ocelot uses the static emitter by default.

The initial implementation included the basic optimization passes available in OPT, the LLVM optimizer, for O1, O2, Os, and O3. After several experiments, it was found that the inter-procedural optimizations included in OPT were not relevant for optimizing single PTX kernels and they were subsequently removed. Section 4.2 explores this topic in more detail by examining the sensitivity of CUDA applications to individual optimization passes.

## 3.5 Linker and Runtime Issues

Translating the instructions from PTX to LLVM to x86 is only part of the process of executing a PTX program on a many-core processor. It also involves allocating space for statically allocated variables, propagating the locations

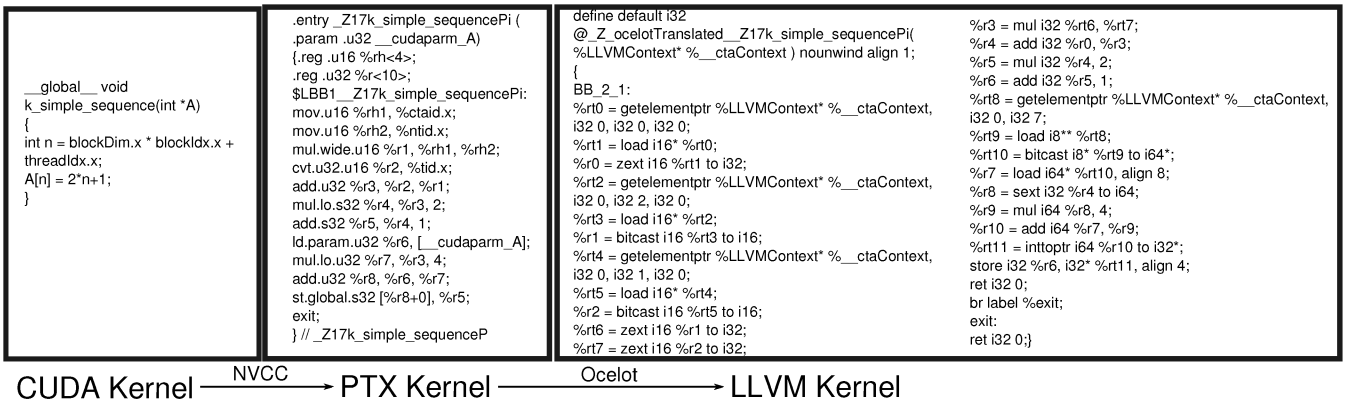


**Figure 4: Sample CUDA source, PTX assembly, and LLVM assembly. There is a significant amount of code expansion, mainly due to explicit casts which are required in LLVM, but not in PTX. Much of this redundancy is lost during LLVM to x86 code generation.**

of these variables to references in the program, as well as allocating OpenGL buffers and variables bound to textures.

**Global Variables**. Global variables in PTX present a problem from a compilation perspective because they can conditionally be linked to dynamic memory allocations declared externally from the PTX program and bound at runtime using the CUDA Runtime API. Ocelot handles global variables in translated code via a primitive runtime linker. Before translation, the linker scans though the instructions in the kernel and replaced accesses to these variables with static offsets into a module-local memory region. This handles the private variables. External variables are declared as globals in LLVM and their identifiers are saved in a list. The LLVM code generator is then used to compile the kernel without linking the external variables. Upon executing a kernel, existing mappings for these variables are cleared and the LLVM linker is used to bind references to the most currently mapped memory for that variable.

**Texture Interpolation**. Graphics applications rely heavily on the process of texture mapping - intuitively this is the process of wrapping a 2D image around a 3D geometry using interpolation. Most modern GPUs include hardware support for texture mapping in the form of floating point units that perform load operations from floating point addresses. These addresses are wrapped or clamped to the dimensions of a 1D or 2D image bound to a texture. For addresses that do not fall on integer values, nearest point, linear, or bilinear interpolation is used to compute a pixel value from the surrounding pixels. For non-graphics applications, textures can be used to accelerate interpolation for image or signal processing.

In PTX, textures are exposed in the ISA using instructions that sample different color channels given a set of floating point coordinates. Modern CPUs do not have hardware support for interpolation. Furthermore, this operation is complex enough that it cannot be performed using a short sequence of LLVM instructions. In order to reduce the complexity of the LLVM translator, a texture interpolation library was implemented to emulate the interpolation operations in software.

Library support is also required (and was implemented) for OpenGL and multi-threaded applications. A detailed description is omitted here for brevity.

**Runtime**. Once a kernel has been translated, it must be executed in parallel on all of the cores in a CPU. The Hydrazine threading library [27] (which itself wraps pthreads on Linux) is used to bind one worker thread to each CPU core. When the first kernel is executed on a CPU device, all of the worker threads are started. The main thread will assign a subset of CTAs to each thread and signal each worker to begin executing the kernel. The main thread will then block until all workers have completed in order to preserve the global barrier semantics of the bulk-synchronous execution model.

In this implementation, care was taken to reduce the number of synchronization routines used during kernel execution. Only one condition variable broadcast is issued from the main thread when the kernel is launched and one condition variable signal is used per worker thread when the kernel completes. The overhead of creating/destroying worker threads is mitigated by reusing the same threads to execute a series of kernels.

Special considerations were also given to handle atomic memory operations and CTAs with variable execution times.

**Atomic Operations**. Atomic operations in PTX are useful for performing commutative operations with low overhead across CTAs in a program. For example, they can be used to implement an efficient reduction across a large number of CTAs. As useful as they are, atomic operations introduce some difficulties when being executed by multiple worker threads. Straightforward solutions involving locking access to atomic operations may introduce an excessive amount of overhead as locks can involve much higher overhead than atomic operations supported by hardware in the GPU. LLVM alternatively supports a series of intrinsic operations that expose hardware support for the atomic operations in PTX. Ocelot's implementation of atomic operation uses locks rather than atomic operations for simplicity. This decision is justified empirically in Section 4.1.

**CTA Scheduling**. The initial implementation of the Ocelot runtime used a static partitioning scheme where the 2D space of CTAs was projected onto a 1D space and divided equally among the worker threads. This scheme proved effective for many applications where the execution time of CTAs was constant. However, several applications, particularly the SDK Particles example, exhibited variable execution time for each CTA leading to cases where some worker

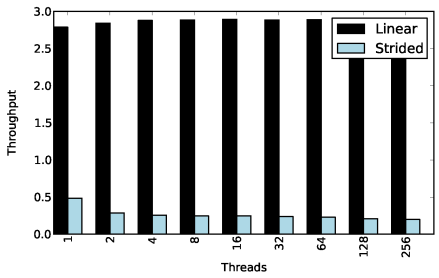


**Figure 5: Memory Bandwidth.**

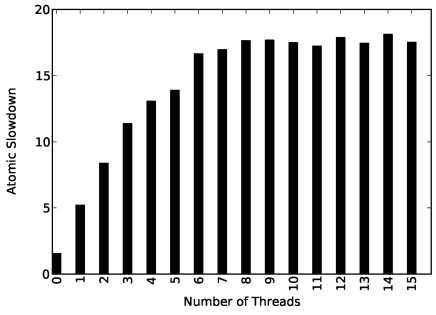


**Figure 6: Atomics Throughput.**

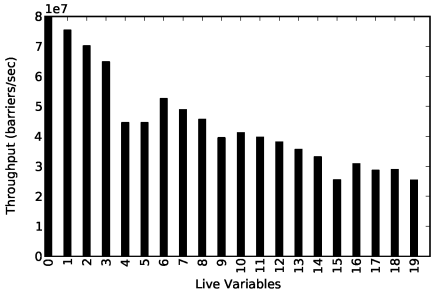


**Figure 7: Barrier Throughput.**

threads would finish their set of CTAs early and sit idle until the kernel completed.

To address this problem, the classical work stealing approaches as well as different static partitioning schemes were considered. Eventually a locality-aware static partitioning scheme was selected due to perceived overheads associated with work stealing. For several applications, the execution time of a CTA was strongly correlated with that of its neighbors. In the Particles example, this is the case because neighboring CTAs process neighboring particles, which are more likely to behave similarly. An interleaved partitioning scheme was implemented where the 2D space was still mapped onto a 1D space, but the space was traversed beginning at an offset equal to the worker thread's id, and incremented by the total number of worker threads. This made it more likely that each worker thread would be assigned a set of CTAs with a similar distribution of execution times.

# 4. RESULTS

This Section covers a preliminary analysis of the performance of several CUDA applications when translated to x86 and executed on a quad-core CPU. The system configuration of an Intel i920 CPU with 8GB RAM is used for all of the experiments in this section. NVCC 2.3 is used to compile all CUDA programs, Ocelot-1.0.432, and LLVM-2.7svn are used for all experiments. Results from a set of microbenchmarks are presented first. They explore the performance limits of the Ocelot CPU backend. A second set of experiments attempt to characterize the sources and significance of overheads in Ocelot. A third set of experiments cover the scalability of several full applications using multiple cores. A final experiment evaluates the sensitivity of CUDA applications to common compiler optimizations available in LLVM. The measurements presented in this section 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. In response, the sample mean was computed from at least 100 samples per experiment and are presented in the form of bar charts with 95% confidence intervals.

## 4.1 Microbenchmarks

Microbenchmarks provide an upper bound on the real-world performance of PTX applications that are translated to an x86 system using Ocelot. In order to avoid artifacts introduced by the NVIDIA CUDA to PTX compiler, Ocelot was extended to accept inlined PTX assembly via a set of new CUDA API calls. Microbenchmarks for memory bandwidth, atomic operation throughput, context-switch overhead, instruction throughput, and special function throughput were written using this new interface.

**Benchmark:** *Memory Bandwidth.* The first microbenchmark explores the impact of memory traversal patterns on memory bandwidth. This experiment is derived from prior work into optimal memory traversal patterns on GPUs, 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 by thread-fusion and coalesced accesses are transformed into strided accesses. Figure 5 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 that would be coalesced on the GPU.

**Insight:** *Compiler Optimizations Impact Memory Traversal Patterns.* It is very significant that the strided access pattern is over 10x slower using the CPU backend when compared to the linear access pattern. This indicates that the optimal memory traversal pattern for a CPU is completely different than that for a GPU. PTX transformations, such as thread-fusion used in MUCDA [14], that change the memory traversal pattern of applications should be designed with this in mind.

**Benchmark:** *Atomic Operations.* The next experiment details the interaction between the number of host worker threads and atomic operation overhead. This experiment involves 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 with a single thread that incremented a single variable in memory until it reached the same threshold without using atomics. Figure 6 shows that the overhead of atomic operations is less than 20x the overhead of non-atomic equivalents.

**Benchmark:** *Context-Switch Overhead.* This experiment explores the overhead of a context-switch when a thread hits a barrier. The test 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

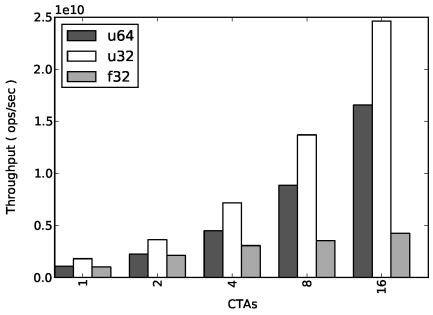


**Figure 8: Instruction Throughput**

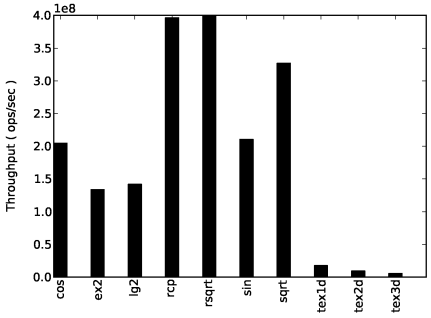


**Figure 9: Special Op Throughput**

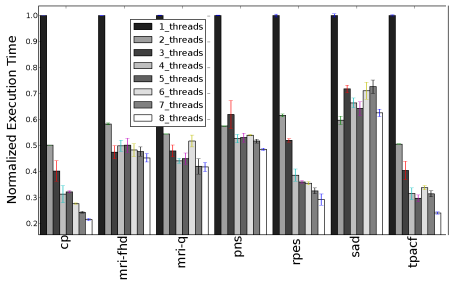


**Figure 10: Multi-Core Scaling**

the effect of barriers on a single thread, only one thread in one CTA is launched. The thread will hit the barrier, exit into the Ocelot thread scheduler, and be immediately scheduled again. Figure 7 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.

**Insight:** *Memory Pressure Matters.* One of the assumptions behind the PTX programming model is that all threads in a CTA are alive at the time that the CTA is executed. This implicitly assumes that there are enough on-chip resources to accommodate all threads at the same time to avoid a context-switch penalty. For GPU style architectures, this puts pressure on the register capacity of a single multiprocessor; if the total number of registers needed by all of the threads in a CTA exceeds the register file capacity, the compiler must spill registers to memory. For CPU style architectures, this puts pressure on the cache hierarchy; all live registers must be spilled on a context-switch which will hopefully hit in the L1 data cache. If the total number of live registers needed by all threads in a CTA exceeds the cache capacity, a CTA-wide context-switch could flush the entire L1 cache. Programs without barriers have an advantage from the perspective of memory pressure because there is no need to keep more than one thread alive at a time.

The PTX model indirectly addresses this problem with the concept of a CTA. This reduces the number of threads that must be alive at the same time from the total number of threads in a kernel to the CTA size. This partitioning maps well to the hardware organization of a GPU and most CPUs which have local memory per core (either a shared register file or L1 cache). Future architectures may introduce additional levels of hierarchy to address increasing on-chip wire latency. A scalable programming model for these architectures should extend to a multi-level hierarchy of thread groups, and the compiler should be able to map programs with deep hierarchies to architectures with more shallow memory organizations. Concepts similar to Sequoia memory hierarchies [28] or Habanero hierarchical place trees [29] could be applied to this problem.

**Insight:** *Context-Switch Overhead Is Significant.* This microbenchmark demonstrates that there is a non-trivial overhead associated with context-switching from one thread to another. This suggests that the compiler should actively try to reduce the number of context switches. Ocelot does this by deferring switches until barriers are encountered. However, it may be possible to reduce the number of context-switches more aggressively by identifying threads that can never share data and allowing disjoint sets of threads to pass through barriers without context-switching. This could be done statically using points-to analysis or dynamically by deferring context-switches to loads from potentially shared state. Additionally, it may be possible to reduce the context-switch overhead by scheduling independent code around the barrier to reduce the number of variables that are alive across the barrier.

**Benchmark:** *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 instructions were tested, the results of which are shown in Figure 8. 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 slower, only achieving 4Gflops on all 4 cores. This is slower than the peak performance of the test system, and could be the result of the generated code schedule or use of x87 for floating point operations. These results suggest that code translated by Ocelot will be relatively fast when performing integer operations, and slow when performing floating point operations.

**Benchmark:** *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. The benchmarks consist of a single unrolled loop per thread where the body consists simply of a series of independent instructions. The number of threads and CTAs were varied to maximize throughput. 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 9 shows the maximum sustainable throughput for each special function. The throughputs of these operations are relatively consistent when run on the GPU, which uses hardware acceleration to quickly provide approximate results. Ocelot implements these operations with standard

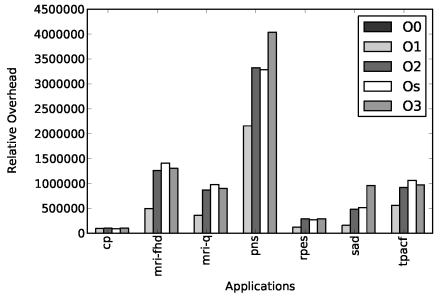


**Figure 11: Optimization Overhead**

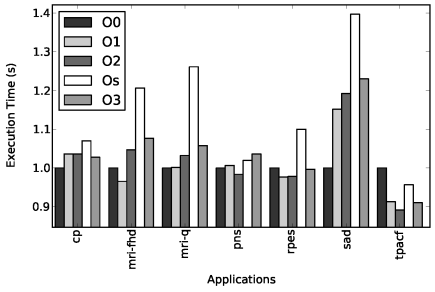


**Figure 12: Optimization Scaling**

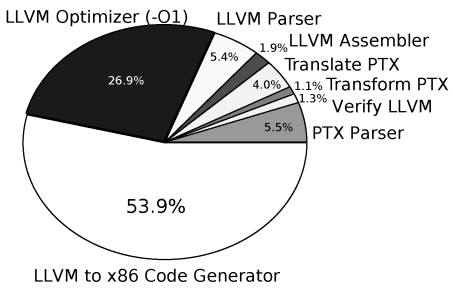


**Figure 13: Sources of Overhead**

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 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. Texture operations are the slowest, nearly 30x slower than the Rcp baseline. Consequently, texture-intensive kernels should be biased towards GPU architectures.

## 4.2 Runtime Overheads

The next set of experiments are designed to identify overheads that limit total application performance. The kernel startup cost is measured first, followed by 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.

**Benchmark:** *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. The mean overhead across the Parboil benchmark suite was found to be approximately 30us which is on the order of the precision of the timers. These are negligible compared to the overheads of translation and optimization. Future work may explore more dynamic work distribution mechanisms such as work stealing that take advantage of this headroom.

**Benchmark:** *Optimization Overhead.* In order to determine the relative overhead of applying different levels of optimization at runtime, the optimization passes in Ocelot were instrumented to determine the amount of time spent in optimization routines. The total amount of time spent in optimization routines is shown in Figure 11 for different applications in the Parboil benchmark suite. Notice that the total optimization time depends significantly on the application. Furthermore, the relative overhead of O3 compared to O2 is also significantly application dependent.

A second experiment measures the total execution of the Parboil benchmarks using different optimization levels, which are shown in Figure 12. This experiment includes overheads of optimization as well as speedups due to executing more optimized code. 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. Note that the LLVM to x86 code generator 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.

**Insight:** *Parallel-Aware Optimizations Are Necessary.* A significant amount of effort in our implementation was spent dealing with barriers and atomic operations in PTX, and that all of the compiler transformations available in LLVM were oblivious to these program semantics. In the future, there could be significant progress made in developing compiler optimizations that are aware of the PTX thread hierarchy and primitive parallel operations. For example, sections of threads that are independent of the thread id could be computed by one thread and then broadcast to others, barriers could be reorganized to reduce the number of context-switches, and threads that take the same control paths could be fused together into a single instruction stream.

**Benchmark:** *Component Contribution.* As a final experiment into the overheads of dynamic translation, callgrind [22] was used to determine the relative proportion of time spent in each translation process. Note that callgrind records basic block counts in each module, which may be different than total execution time. Figure 13 shows that the vast majority of the translation time is spent in the LLVM code generator. The decision to use a new LLVM IR 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 removing if-conversion and barriers from PTX takes less than 2% of the total translation time. These results justify many of the design decisions made when implementing Ocelot.

## 4.3 Full Application Scaling

Moving on from micro-benchmarks to full applications, the ability of CUDA applications to scale to many cores on a multi-core CPU was studied. The test system includes a processor with four cores, each of which supports 2-way simultaneous multi-threading (SMT). 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.

| Application | ConstantPropagation | DeadInstElimination | DeadCodeElimination | DeadStoreElimination | AggressiveDCE | InductionVariableSimplify | InstructionCombining | LoopInvariantMotion | LoopStrengthReduce | LoopUnswitch | LoopUnroll | LoopRotate | TailDuplication | JumpThreading | CFGSimplification | BlockPlacement | GlobalValueNumber | GEPSplitter | SCCVN | All Optimizations |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CP | 1.03 | 1.04 | 1.01 | 1.03 | 1.02 | 1.01 | 1.01 | 1 | 1.02 | 1.04 | 1.01 | 1.03 | 1.02 | 1.02 | .96 | 1.04 | 1 | .99 | 1.02 | .99 |
| MRI-FHD | .89 | 1.14 | .92 | .92 | .95 | .9 | .77 | 1 | 1 | .92 | 1.02 | .88 | .9 | .81 | .97 | .94 | .86 | .88 | .74 | .99 |
| MRI-Q | 1.04 | .92 | .88 | 1.06 | 1.03 | .9 | 1.03 | .88 | .98 | 1.01 | .9 | 1.18 | 1.01 | 1 | 1.01 | .92 | 1 | .97 | .93 | .99 |
| PNS | 1.02 | .97 | 1.01 | .98 | 1.02 | .97 | 1.02 | 1.02 | .98 | 1 | 1.02 | 1 | 1 | .95 | 1 | 1.01 | .97 | .98 | 1.02 | 1 |
| RPES | 1 | 1.05 | 1.01 | 1 | .99 | 1.01 | 1.03 | 1 | 1 | 1 | 1.01 | 1.02 | 1 | 1.02 | 1.03 | 1 | 1 | .99 | .91 | .98 |
| SAD | 1.15 | 1.21 | 1.18 | 1.16 | 1.12 | 1.47 | 1.18 | 1.2 | 1.13 | 1.12 | 1.18 | 1.15 | 1.15 | 1.16 | 1.1 | 1.53 | 1.32 | 1.12 | 1.2 | .98 |
| TPACF | .84 | .86 | .76 | .88 | .87 | .89 | .89 | .89 | .85 | .86 | .8 | .8 | .78 | .84 | .87 | .83 | .88 | .84 | .9 | .85 |
| Average | .98 | .99 | .95 | .98 | .97 | .98 | .99 | .98 | .96 | .98 | .96 | .96 | .95 | .97 | .98 | .98 | .97 | .96 | .95 | .97 |

**Table 1: Normalized execution time of different LLVM passes compared to the baseline with no optimization.**

**Benchmark:** *Parboil Scaling.* The Parboil benchmarks were used as examples of real CUDA applications with a large number of CTAs and threads; previous work shows that the Parboil applications launch between 5 thousand and 4 billion threads per application [16]. Figure 10 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 SMT. Conversely, SAD slows down when the number of threads is increased beyond two. Previous work by Kerr et al. [16] have found PNS and SAD to be highly memory intensive, and likely to be constrained by a processor's off-chip memory bandwidth rather than its core count. These applications may be more suitable for GPU architectures, which focus on high bandwidth rather than low latency. These results motivate the need for a dynamic compiler like Ocelot that can direct applications to the most efficient architecture in a heterogeneous system.

**Insight:** *Variable CTA Execution Time.* Several of the applications in this paper demonstrate the importance of evenly distributing CTAs across cores in a CPU or GPU. These results suggest that work distribution schemes must simultaneously deal with two constraints that follow from locality among CTAs: 1) neighboring CTAs are likely to have similar execution times, and 2) neighboring CTAs are likely to access similar memory locations. In other words, mapping neighboring CTAs to the same processor core will improve memory locality, but lead to uneven work distributions. Conversely, random partitioning schemes will hurt memory locality, but even out work distributions. There is a clear need for additional work that addresses this problem using static analysis as well as runtime adaptive mapping encapsulated in the translator.

## 4.4 Sensitivity Analysis

This paper concludes with an analysis of the sensitivity of individual Parboil applications to single LLVM optimization passes shown in Table 1. These results also include normalized runtimes with all optimizations enabled. No overheads are included in this experiment, only time spent executing translated code is counted. On average, LLVM optimizations improve execution time by 1% to 5%. Some applications, such as TPACF, universally benefit from optimization (possibly due to optimizations being performed in the code generator) while others, such as SAD, uniformly slow down. It is also clear that certain optimizations are more suitable to specific applications. For example, MRI-FHD benefits the most from instruction combining, which negatively impacts the performance of PNS. In several applications, optimizations such as *ConstantPropagation* and *InductionVariableSimplify* achieve faster execution individually than when all optimization passes are applied. Yet, this relationship does not hold for every application. Overall these per-thread optimizations yield relatively minor improvements in execution time, indicating that the optimizations performed statically by NVCC, the during code generation by LLVM, and dynamically by the CPU instruction schedulers are already highly tuned. Future work into dynamic optimization may have more success by shifting focus away from single-threads to address system-wide issues such as improving memory access patterns via better thread/CTA schedules or eliminating redundancy via inter-thread analysis.

# 5. CONCLUSIONS

This paper presents a detailed overview of Ocelot, including a dynamic compiler from PTX to Multi-core x86 CPUs. Through the study of Ocelot using several microbenchmarks and full applications, on-chip memory pressure, context-switch overhead, and variable CTA execution time were identified as fundamental issues that impact performance when compiling highly parallel programs to systems with few hardware resources. In the future, these issues will have to be addressed as systems continue to migrate towards many-core architectures, and developers seek programming models that can scale to them.

# 6. ACKNOWLEDGEMENT

This research was supported by NSF under grant CCF-0905459, IBM through an OCR Innovation award, LogicBlox Corporation, and an NVIDIA Graduate Fellowship.

# 7. REFERENCES

[1] IMPACT, "The parboil benchmark suite," 2007.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, October 2009.

[3] A. Kerr, D. Campbell, and M. Richards, "Gpu vsipl: High-performance vsipl implementation for gpus," in *HPEC'08: High Performance Embedded Computing Workshop*, Lexington, MA, USA, 2008.

[4] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2009, version 1.2.

[5] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[6] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009.

[7] NVIDIA, "Nvidias next generation cuda compute architecture: Fermi," NVIDIA Coporation, Tech. Rep., 2009.

[8] AMD, "R600/r700/evergreen assembly language format," Tech. Rep., 2009.

[9] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling gpu-cpu workloads and systems," in *Third Workshop on General-Purpose Computation on Graphics Procesing Units*, Pittsburg, PA, USA, March 2010.

[10] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO'09*. New York, USA: IEEE, devember 2009.

[11] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," in *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 19–33.

[12] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 15–24.

[13] W. W. L. Fung, I. Sham, G. Yuan, and T. 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420.

[14] J. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei Hwu, "Efficient compilation of fine-grained spmd-threaded programs for multicore cpus," in *CGO 2010*, Toronto, Canada, April 2010.

[15] G. Diamos, A. Kerr, and M. Kesavan, "Translating gpu binaries to tiered simd architectures with ocelot," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-01, January 2009.

[16] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *IISWC09: IEEE International Symposium on Workload Characterization*, Austin, TX, USA, October 2009.

[17] C. Madriles, P. Lopez, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez, "Anaphase: A fine-grain thread decomposition scheme for speculative multithreading," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 15–25.

[18] A. Chernoff and R. 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*. Berkeley, CA, USA: USENIX Association, 1997.

[19] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, 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.*, vol. 44, no. 2, 2005.

[20] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2000.

[21] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*. New York, NY, USA: ACM, 2004, p. 22.

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[23] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Boston, MA, USA, April 2009.

[24] S. Collange, D. Defour, and D. Parello, "Barra, a modular functional gpu simulator for gpgpu," Tech. Rep. hal-00359342, 2009.

[25] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.

[26] G. Diamos, "State explosion: An obvious limitation to strong scaling," NFinTes, Tech. Rep., 2009.

[27] ——, "Hydrazine: A high performance library for c++ and cuda," November 2009.

[28] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[29] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, october 2009.