The Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86

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

***Abstract*—Ocelot is a dynamic compilation framework designed to map the explicitly parallel PTX execution model used by NVIDIA CUDA applications onto diverse many-core architectures. Ocelot includes a dynamic binary translator from PTX to many-core processors that leverages the LLVM code generator to target x86. The binary translator is able to execute CUDA applications without recompilation and Ocelot can in fact dynamically switch between execution on an NVIDIA GPU and a many-core CPU. It has been validated against over 100 applications taken from the CUDA SDK [1], the UIUC Parboil benchmarks [2], the Virginia Rodinia benchmarks [3], the GPU-VSIPL signal and image processing library [4], and several domain specific applications.**

This paper presents a detailed description of the implementation of our binary translator highlighting design decisions and trade-offs, and showcasing their effect on application performance. We explore several code transformations that are applicable only when translating explicitly parallel applications and suggest additional optimization passes that may be useful to this class of applications. We expect this study to inform the design of compilation tools for explicitly parallel programming models (such as OpenCL) as well as future CPU and GPU architectures.

I. Introduction

A blind embrace of the many-core revolution has resulted in a surplus 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. A pursuit of this lost scalability has lead to a frantic transition towards many-core architectures with only an afterthought given to the programming models that will follow behind.

Most parallel programming models are not up to the task. Those that rely on heavy-weight threads and coherent shared memory will not be scalable due to excessive synchronization overheads [5] and per-thread state explosion. Yet these are the most commercially prevalent models.

Designing applications that perform well on modern as well as future architectures can no longer be an afterthought that is passed off to processor architects and circuit designers. It must be a principle requirement that is addressed explicitly in the programming language, execution model, compilation chain, and the hardware architecture. Much of the progress made towards this goal has been pioneered by the graphics community, which has almost uniformly adopted a bulk-synchronous [6] programming model [7]–[9] coupled with architectures that simultaneously mextend many-core to its limits while abandoning global cache coherence and strong memory consistency [9]–[12]. Unfortunately, this body of work has largely ignored the impact of these new programming paradigms and architectures on the aspects of compiler design.

Moving to an explicitly parallel, bulk-synchronous programming model significantly changes the problem presented to a compiler; selecting a many-core architecture with a varying distribution of hardware resources significantly changes its target. 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 the additional constructs for dealing with parallel execution that are made visible to the compiler will effect a significant change in the design of future compilers.

Scalable bulk-synchronous applications will also require just-in-time compilation and binary translation. Whether binary translation is done using hardware decoders as in modern Intel and AMD x86 [13], [14] processors or in software as in NVIDIA GPUs [7], the evolution of the processor michroarchitecture 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 compilation 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.

Ocelot provides a framework for evaluating extent of these changes to compiler design by leveraging the explicitly parallel PTX execution model. We have completed a comprehensive dynamic compiler infrastructure for PTX including PTX to PTX transformations as well as backend targets for NVIDIA GPUs and many-core CPUs. In this paper, we explore the detailed design and implementation of our many-core CPU backend. Our intent is to expose the unique problems presented while compiling explicitly parallel execution models to many-core architectures and evaluate several potential solutions.

Organization. This paper is organized as follows. Section II gives background on Ocelot. Section III describes the implementation of the PTX to x86 dynamic compiler. Section IV presents performance results using the x86 backend. Section VI reviews the most significant lessons learned. Section V briefly covers related work and Section VIII concludes with suggestions for future compilers.

II. Ocelot and PTX

Ocelot is an open source project developed by the authors of this paper. It 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. Our emulator, many-core compiler, and GPU compiler support the full ptx1.4 specification and have been validated against over 100 CUDA applications.

This section covers the salient of features of PTX that make it a suitable intermediate representation for many-core CPUs and dynamic compilation.

A. A Bulk-Synchronous Execution Model

We speculate that PTX and CUDA grew out of the development of Bulk-Synchronous Parallel (BSP) programming models first identified by Valiant [6]. 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 Concurrent-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 incredibly similar to the original formulation of the BSP programming model.

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 processors without incurring excessive synchronization overheads. 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 across kernel launches, but they cannot reliably communicate within a kernel. Like BSP programs, these characteristics ensure that PTX programs can express a large amount of parallelism that can be efficiently mapped onto a machine with a smaller number of processors while only periodically having to incur an expensive global barrier to make memory consistent.

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 can be mapped with relative ease onto a hardware SIMD pipeline using a combination of hardware support for predication, a thread context stack, and compiler support for identifying reconverge points at control-independent code [15]. In contrast with other popular programming models for SIMD architectures which require vector widths to be specified explicitly, the aforementioned techniques allow PTX to be automatically mapped onto SIMD units of different sizes. In the next section, we describe how these abstractions, which were intended to scale across future GPU architectures, can be mapped to many-core CPU architectures as well.

B. 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. This topic has previously been explored from two complementary perspectives: 1) a static compiler from CUDA to multi-core x86 described in Stratton et al. [16] and extended by the same authors in [17], and 2) our previous work exploring the dynamic translation of PTX to Cell [18], and our characterization of the dynamic behavior of PTX workloads [19]. From this body of work, we drew the following insights that influenced the design of our PTX to x86 dynamic compiler:

- From MCUDA: PTX threads within the same CTA can be compressed into a series of loops between barriers.
- From our PTX to Cell work: 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 our analysis of PTX kernels: Dynamic program behavior such as branch divergence, memory intensity, inter-thread data-flow, and activity factor can influence the optimal mapping from PTX to a particular machine.

C. Thread Fusion

Mapping CTAs in a PTX program onto set of parallel processor 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. In MCUDA, Stratton et al. suggested 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, while our implementation works at the PTX and CFG level. However, our 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.

D. Just-In-Time Compilation

The diversity of possible many-core architectures makes static compilation for PTX at the very least overly general: any change in architecture or input data could change the optimal

thread mapping scheme, schedule of generated code, or layout of variables in memory. With the ability of the compiler to fuse threads together [16], [20], redefine which threads are mapped to the same SIMD units [15], re-schedule code to trade off chache misses and register spills [21], and migrate code across heterogeneous targets [18], recompiling an application with detailed knowledge of the system being executed on and a dynamic profile can result in significant performance and portability gains.

In the context of this paper, dynamic compilation is used primarily from a portability perspective to execute the same CUDA programs on both NVIDIA GPUs and x86 CPUs. We also highlight the significant differences between optimization for CPU and GPU architectures that can be used in future work dynamically map the same program to one of several possible architectures.

E. Profile-Aware Compilation

In previous work, we identified several metrics that can be used to characterize the behavior of PTX applications [19]. For example, we identified the amount of SIMD and MIMD parallelism in an application, control flow divergence, memory access patterns, and inter-thread data sharing. Bakhoda et al. [22] and Collage et al. [23] 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. Our implementation of Ocelot's many-core backend focuses on efficiently handling these key areas.

In addition to revisiting the insights provided by previous work, our implementation 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.

III. Implementation

This section covers the specific details of our PTX to x86 many-core dynamic compiler. At a high level, the process can be broken down into the following operations: 1) extracting the PTX binary from a CUDA application, 2) performing transformations at the PTX level to create a form that is representable in LLVM, 3) translation from PTX to LLVM, 4) LLVM optimizations, 5) laying out memory and setting up the execution environment, and 6) initializing the runtime that executes the program on a many-core processor.

A. PTX Binary Extraction

The first step in executing a CUDA program using Ocelot is to extract the PTX binaries from the CUDA program so that they can be translated. This process is very convoluted as an artifact of the design of CUDA. It is also not currently documented at the time that this paper was written; our implementation was done by reverse engineering NVIDIA's

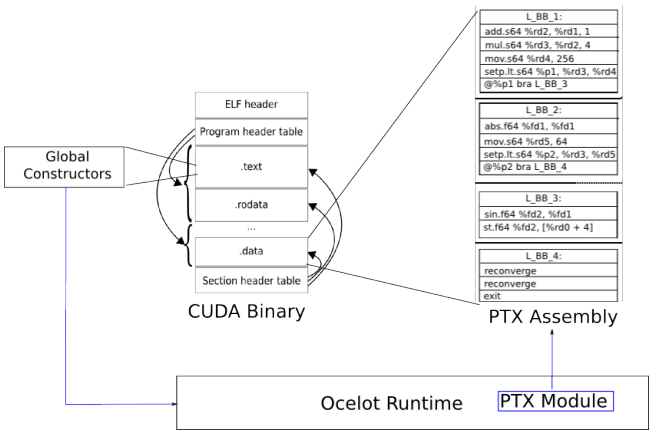

Fig. 1. Extracting a Binary From a CUDA Application

reference libcudart.so. We include a detailed description here for the benefit of others.

Rather than emitting PTX in binary format stored in the .text section of an ELF or Window binary, the CUDA compiler (NVCC) performs a source-to-source transformation that converts CUDA source files into C++ source files. All of the CUDA kernels are compiled by NVCC into PTX and all of the native C++ code is copied directly over. NVCC creates a global variable (referred to as a *fatBinary* in the CUDA headers) containing PTX assembly code for each CUDA kernel and inserts it directly into the generated C++ source file. It also creates one global constructor for each CUDA kernel that makes a call into the CUDA Runtime API to register the global variable. The native C++ compiler is then invoked to generate a native binary.

This approach has the advantage of not requiring a different backend to support different binary formats: the native compiler is responsible for generating the binary. However, it also makes it nearly impossible to statically extract a PTX binary from a CUDA program because the assembly code will be stored in the program data section and it will be impossible to distinguish between PTX binaries and generic global variables until the program is executed. Ocelot includes an implementation of the CUDA Runtime API which intercepts the global constructors that register each CUDA kernel. This process is shown in Figure 1.

Unfortunately, as a consequence of the design of CUDA, it is not currently possible for us to either modify the PTX stored in a CUDA program nor insert translated code into a preexisting binary. This forces us to re-translate each kernel every time the program is executed. This is common practice for most binary translators [24]–[27], but it does introduce some overhead each time a program is executed.

B. Building The PTX IR

After PTX assembly programs are registered with the Ocelot runtime, our implementation will parse each PTX program and build an abstract syntax tree (AST) for the file. The AST is incredibly simple because the PTX assembly language

only supports a few syntactical constructs with no nesting. However, it is helpful for simplifying the implementation of the parser which can handle multiple passes by first generating the AST in a single pass and then traversing the AST to resolve labels and variable names.

Once the AST has been generated, a Module is created for each distinct AST. Ocelot borrows the concept of a Module from LLVM [28] which contains a set of global variables and functions. Similarly, our concept of a Module 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.

Once this process has finished, there will be one Module for each fat binary. Within each module, there will be one control flow graph for each kernel. Our implementation of a control flow graph does not contain any detailed analysis information. If any of the later transformations require data-flow, dominance, or mappings from identifiers to register values these analyses will be lazily performed upon their first use.

C. PTX to PTX Transformations

Once we have constructed the previously defined data-structures for program analysis, it becomes possible to perform code transformations on PTX programs. During translation to LLVM, we lose concepts associated with the PTX thread hierarchy such as barriers, atomic operations, votes, as well as the exact number and organization of threads. Although we do not perform any additional optimizations using this information, we wanted our design to support doing these optimizations at this stage rather than within the LLVM infrastructure.

In order to support PTX transformations, we created an optimization pass interface similar to that used by LLVM where different optimization "Passes" can be applied to a Module, a Kernel, or a basic block. This design 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. 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.

Using this infrastructure, we implemented two PTX optimizations that eased the translation process to LLVM. In particular, we found it difficult to support predication and barriers after translation to LLVM. Instead, we perform transformations at the PTX level to convert predication to conditional selection and modify the control flow structure such that the semantics of a barrier are satisfied even when executing the program with a single thread.

PTX SSA Form. For any individual PTX kernel, the data-flow graph 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. A PTX kernel begins in partial SSA form (infinite registers but no phi nodes). Conversion to full SSA form is done using the live-in and live-out sets for each basic block where each live in register is converted into a phi instruction. As PTX does not have a concept of a PHI instruction, these are maintained in the separately in the data flow graph rather than the control flow graph.

LLVM requires kernels to be in full SSA form, so all translated kernels are converted into this form before translation. Alternatively, it would be possibly to avoid this conversion stage using the LLVM Mem2Reg pass and treat all registers as local variables. We chose not to use this optimization due to the desirability of performing optimizations at the PTX level, which could benefit from SSA form. Mem2Reg claims to be extremely high performance (possibly better than our implementation) so we may investigate this in the future if the performance of SSA conversion is ever determined to be significant.

Reversing If-Conversion. LLVM does not support predication at all. 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), we must 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 following example PTX code before converting into SSA form.

```
ld.param.s32 r0, [condition];
mov.s32 r1, 0;
setp.ne.s32 p0, r0, r1;
@p0 add.s32 r1, r1, 1;
```

After converting into SSA form, the destination of the add instruction is assigned a new register (r2).

```
ld.param.s32 r0, [condition];
mov.s32 r1, 0;
setp.ne.s32 p0, r0, r1;
@p0 add.s32 r2, r1, 1;
```

Now, converting the predicated add instruction to a regular add followed by a conditional select instruction becomes problematic.

```
@p0 add.s32 r2, r1, 1;
```

The orignal prediicated add instruction could map to a non-predicated add paired with a select.

```
add.s32 temp, r1, 1;
selp.s32, r2, temp, ????, p0;
```

However, it is not possible to easily determine what to set the value of r2 to if the predicate is not true. It is much simpler to insert conditional select instructions before converting into SSA form.

```
ld.param.s32 r0, [condition];
```

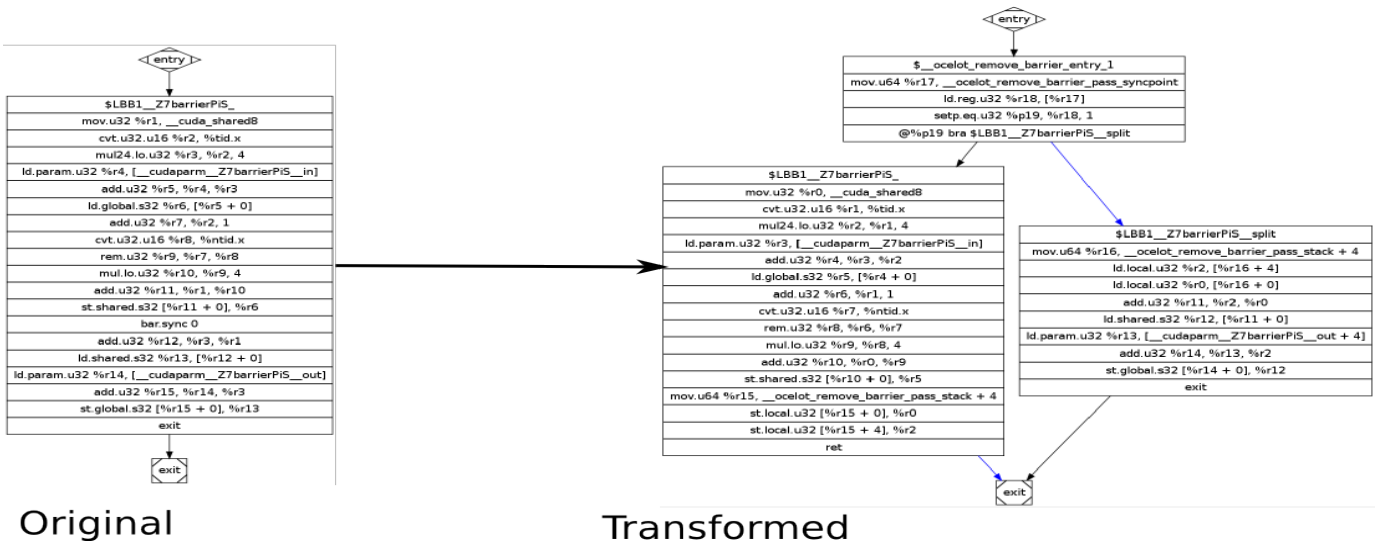

Fig. 2. Example of PTX Barrier Conversion

```
mov.s32 r1, 0;
setp.ne.s32 p0, r0, r1;
add.s32 temp, r1, 1;
selp.s32, r2, temp, r1, p0;
```

In which case it is simple to determine that r1 should be the value of r2 if the predicate condition is false. SSA form is universally praised in literature as simplifying compiler analysis. This is one example at least where it is more difficult to deal with.

Handling Barriers. A simple way of executing a PTX program on a single-threaded architecture would be to just loop over the program and execute it once for each thread. Unfortunately this violates the semantics of the PTX barrier instruction which assume that all threads execute up to the barrier before any thread executes beyond the barrier. In order to handle this case, we break each kernel into sub-kernels beginning at either the program entry point or a barrier, and ending at either a barrier or the program exit point. We can then loop over each sub-kernel one at a time to make sure that the semantics of a barrier are retained. However, we still have to handle registers that are alive across the barrier.

We handle live registers by creating a barrier spill area in local memory for each thread. For each program exit point ending in a barrier, we save all live registers to the spill area before exiting the program. For every program entry point beginning with a barrier, we add code that restores live registers from the spill area. The definition of local memory ensures that the spill area will be private for each thread.

Figure 2 shows a simple example of this process. The left program 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 program entry point or the barrier resume point. The left successor of this block is the original program 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 program.

For a series of barriers, multiple resume points will be created. After translation, we use the LLVM optimizer to convert the chain of entry blocks into a single block with an indirect jump.

D. The LLVM IR

LLVM ships with a full featured IR that is integrated with their concept of a control flow graph. This representation is rich enough to support all of the LLVM optimization transformations directly. We chose not to use this representation for two reasons:

1) It is too heavy weight
   - A standalone translator would require linking against much of the LLVM codebase which is significantly large at this point. This would increase compilation time and generated binary size.
   - We do not need any concepts of a LLVM control flow graph, just a way to emit generated instructions.
2) It is actively being developed
   - LLVM is constantly being changed and we assumed that the internal representation would change more frequently than the ISA[1].

Based on these factors, we decided to implement our own IR for the LLVM language. This IR is intended to be able to represent any LLVM program, but be lightweight in the sense that it is divorced from the control-flow graph and optimization

[1]Actually this turned out to be only partially true. During the implementation of Ocelot, LLVM actually changed the semantics of the Add and Sub instructions and added new FAdd and FSub instructions.

passes. In this sense it is closer to an AST representation of an LLVM assembly program. During translation, we create an LLVM program in terms of this representation and emit it as an assembly file which is then passed to the LLVM toolchain.

E. Translation to LLVM

Figure 3 shows the translated LLVM assembly for a simple PTX program.

Our basic approach here is to do naive translation, meaning that we examine PTX instructions one at a time and then generate an equivalent sequence of LLVM instructions.

More precisely, we assume that PTX transformations have been applied to a kernel and that it has been converted into full SSA form. We begin 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, we walk the PTX control flow graph and examine each basic block. For each basic block, we examine the dataflow graph to get the set of PHI instructions and emit one LLVM PHI instruction for each live in register. We then iterate over each PTX instruction and dispatch to a translation function for that instruction, which generates an equivalent sequence of LLVM instructions. Note that it would be possible to consider multiple PTX instructions at a time and generate a smaller sequence of LLVM instructions. We instead choose to generate an excessive number of instructions during translation and rely on the LLVM peephole optimizer to remove redundant code.

Most PTX instructions correspond exactly to equivalent LLVM instructions and require trivial translation. However, some special 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. We support to infinity and to -infinity 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.

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 warp. We allocate a lightweight thread context for each host thread and update it to reflect the currently executing thread on a context-switch. For PTX instructions that try to read from a special register, we issue loads 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 standard library implementations. This introduces a significant overhead when executing these instructions.

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. We currently allow a single CTA to be executed a time by each host pthread and all of the threads in those CTAs to be executed. Only enough memory to support $threads * pthreads$ 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, we only resize the shared and local memory spaces when they increase in size. There is currently no bounds checking implemented in our LLVM translator, and this implementation may allow out of bounds memory accesses to silently succeed for some Kernels. However, it would be relatively easy to add checking code to each memory access in a future revision of Ocelot because we have the entire map of allocated memory for each space at runtime[2].

Shared Memory Wrapping. While verifying our translator we noticed that some older versions of the NVIDIA compiler would add large offsets to registers containing addresses that were eventually used to access shared memory. We can only speculate as to why this was done. However, NVIDIA's implementation of CUDA is able to execute these programs without making out of bounds accesses by masking off the upper bits of the address. In order to support older CUDA programs we also needed to mask off these bits using explicit LLVM instructions before performing any shared memory access. This introduced some overhead to all shared memory accesses.

Memory Alignment. PXT requires all memory accesses to be aligned to an address that is evenly divisible by the access size. This implicitly relies on the fact that all memory spaces will begin at addresses that are aligned to the largest possible access size (currently 32 bytes in PTX). We map some PTX memory accesses to vector operations in LLVM, which also must conform to alignment restrictions. Unfortunately, standard library implementations of new/malloc do not strictly enforce alignment requirements. In order to handle these cases, we wrapped malloc and new to always allocate memory in units of (bytes + 32) when allocating memory for a PTX memory space and then manually align the returned values

[2]The PTX emulator included with Ocelot actually does bounds checking by default

Fig. 3. Sample PTX Application and Corresponding LLVM

to 32-byte boundaries.

Uninitialized Registers. A consequence of the SSA form used in LLVM is that the definition of all values must dominate all uses. The NVIDIA compiler will sometimes generate code where registers can potentially be used undefined as in the following code example:

```
int value;
if( condition )
{
  value = 1;
}
return value;
```

In this case, even if *condition* is always true, the compiler may not be able to prove this statically during compilation and there exists a path in the program where *value* used uninitialized. Or, equivalently, *value = 1;* does not dominate *return value;*. In order to handle these cases, we rely on data flow analysis which will generate an alive-in register set at the entry to the program with one entry for each register that is possibly used uninitialized. During translation, we add a new block immediately after the program entry point that sets all of these registers to 0.

F. 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 we execute 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 of a kernel. PTX kernels are typically small with good code coverage and are cooperative executed by multiple threads so we use the static emitter by default.

Dynamic compilers typically have to trade off time spent optimizing code against the additional time spent executing unoptimized code. For CUDA applications, kernels are executed over and over by thousands of threads, making it seem like investment in early optimization would pay off over time. Our initial implementation included the basic optimization passes available in opt, the llvm optimizer for O0, O1, O2, and O3. After several experiments, we found that the inter-procedural optimizations included in OPT were not relevant for optimizing single PTX kernels and subsequently removed them. Figure 9 shows the impact of different optimizations passes on selected benchmarks. Note that for long running benchmarks, the advantages of applying higher optimizations can outweigh the overhead of spending more time in the optimizer, but this is not necessarily the case for shorter applications. In the future we plan to use a decision model to use static information about the number of threads and code size of each kernel to determine the optimization level to apply.

G. Environment Setup

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 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. Alternatively, they can be private to the PTX program and not shared across kernel launches.

We considered one approach to handling global variables that involved replacing instructions that load or store to global identifiers with calls that would dynamically lookup the address of the variable before execution the memory operation. This approach would return either the address of the bound variable if the variable was declared externally, or the address of a local copy. The approach would allow a program to be translated and linked a single time, but it would also introduce the overhead of an indirect lookup for every memory access to a global variable.

In order to avoid this overhead, we chose to implement a different mechanism. We first allocate a memory region for each kernel large enough to accommodate all kernel private variables. We then scan though the instructions in the kernel and replaced accesses to these variables with static offsets into this 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 emitter 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.

OpenGL Interoperability. CUDA provides support for OpenGL interoperability on linux via a set of API functions that allow CUDA kernels to write directly into OpenGL buffers. This process involves registering an OpenGL buffer with the CUDA runtime and then obtaining a pointer to the base of the buffer. When running on a GPU, writes to the area of memory will be forwarded directly to a specific OpenGL buffer. If this buffer is also stored in GPU memory, avoiding a round trip from GPU memory to CPU memory back to GPU memory can provide a significant performance advantage. This trade-off is reversed when kernels are run on the CPU. We use the OpenGL API to obtain pointers to a host memory region that is mapped to an OpenGL buffer. These are passed directly as parameters to kernels. The OpenGL runtime forwards writes to this memory region to the memory on the GPU, possibly introducing overhead compared to running the same kernel natively on a GPU.

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, we implemented a texture interpolation library to emulate the interpolation operations in software. Unfortunately, there is no formal specification for the floating point format/precision or the exact interpolation method used by NVIDIA GPUs, making it difficult to validate our implementation. During the design, we created several unit tests that compared the results of texture operations run using NVIDIA's implementation to our library. We refined our implementation until the results were within a small enough margin or error to pass the built-in regression tests in the CUDA SDK.

In terms of performance, our library executes a function call with a significant number of instructions. The GPU implementation can perform the interpolation in conjunction with the memory access. This severely limits the performance of applications that execute a significant number of texture instructions when executing on CPUs. As an example of this behavior, we tested the performance of a Sobel filter application which includes equivalent implementations, one using texture interpolation and one without. From a series of simple test, we determined that the texture interpolation implementation was relatively faster on the GPU, but slower on the CPU.

H. Runtime

Once a kernel has been translated, it must be executed in parallel on all of the cores in a CPU. We use the Hydrazine threading library [29] (which itself wraps pthreads on Linux) to bind one worker thread to each CPU core. Upon program initialization, all of the worker threads are started and put to sleep until a kernel is executed. When a kernel is executed, 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 semantics of the bulk-synchronous execution model.

In our implementation of the runtime, we were very careful about the number of synchronization routines used during kernel execution and were able to reduce it to one condition variable broadcast when the kernel is launched and then one condition variable signal per worker threads when the kernel completes. The overhead of creating/destroying worker threads is mitigated by reusing the same threads to execute a series of kernels.

This approach to distributing CTAs to worker threads was sufficiently efficient for many applications. However, special considerations were needed 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

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 I
TEST SYSTEM

operations may introduce an excessive amount of overhead as locks can involve much higher overhead than atomic operations supported by hardware. LLVM alternatively supports a series of intrinsic operations that expose hardware support for the atomic operations in PTX. In order to determine the performance impact of locks compared to atomic operations, we ran several microbenchmarks to determine the execution time of simple PTX kernels that exclusively performed a long series of atomic operations, using pthread mutexes for atomic operations. We found that the average time to acquire a highly contested lock, perform an atomic operation, and release the lock was less than 20x slower than simply performing the same operation with one thread and no locks. In other words, locking and unlocking a contested lock was only 20x slower than issuing a load and then a store instruction. This was a significantly better result than we were expecting and thus we did not move on to use LLVM atomic intrinsics.

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

To address this problem we considered the classical work stealing approaches as well as different static partitioning schemes. We eventually settled on a different static partitioning scheme due to excessive synchronization overheads associated with work stealing. We noticed that 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. We implemented an interleaved partitioning scheme where the 2D space was still mapped onto a 1D space, but the space was traversed beginning at an offset equal to the worker threads 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.

IV. RESULTS

This Section 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 distinct points of reference of the scaling and throughput possible when translating CUDA applications to x86. We plan to follow up this study with a detailed characterization of the performance of CUDA applications across a range of GPU and CPU platforms with the intent of identifying program characteristics that are more suited to one style of architecture. For all of the experiments in this section, we use the system configuration given in Table I. We begin with a set of experiments exploring the performance limits of our compiler 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.

A. Microbenchmarks

In order to quickly evaluate the performance limitations of our implementation, we wrote several low level PTX benchmarks 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, we extended the CUDA Runtime API 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 );
```

Using this infrastructure, we explored memory bandwidth, atomic operation throughput, context-switch overhead, instruction throughput, and special function throughput. 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. We present the sample mean in the form of bar charts, and 95% confidence intervals for each metric.

Memory Bandwidth. Our first microbenchmark explores the impact of memory traversal patterns on memory bandwidth. This experiment is based off of prior work into optimal memory traversal patterns on GPUs [7], 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 4 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. It is very significant that

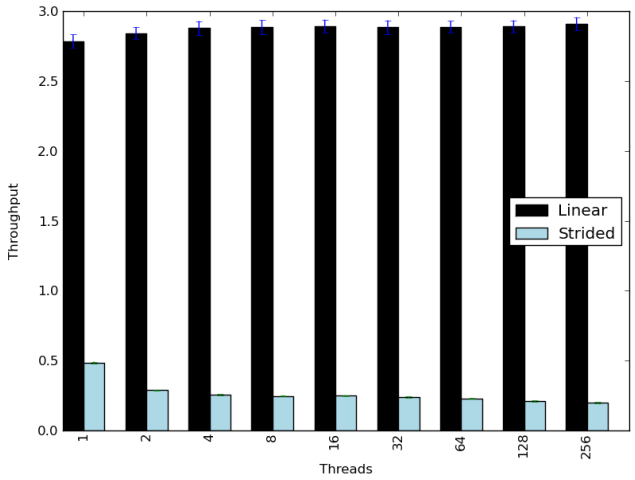

Fig. 4. Maximum Bandwidth for Strided and Linear Accesses

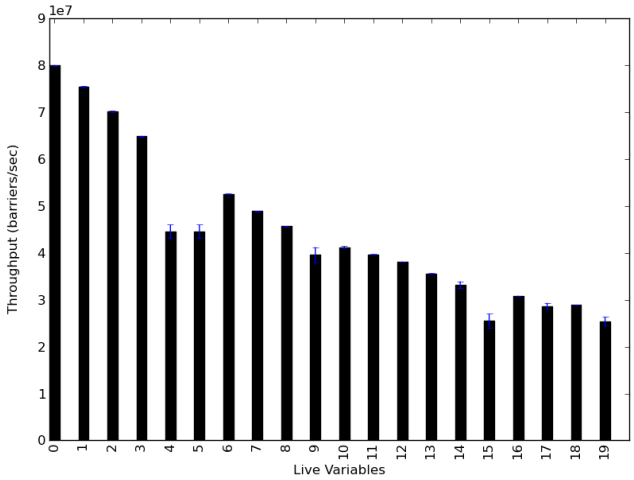

Fig. 6. Barrier Throughput

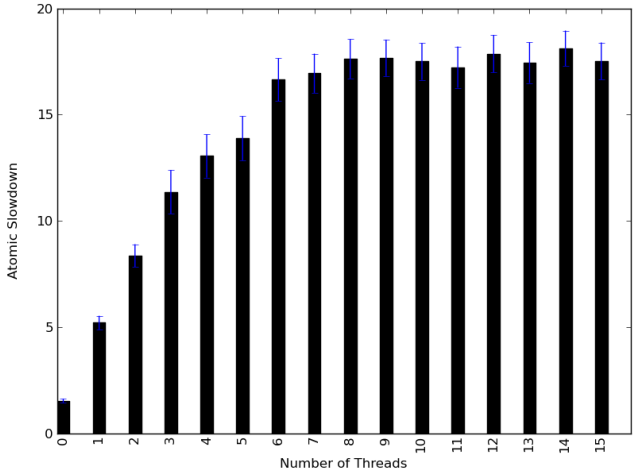

Fig. 5. Atomic Operation Slowdown

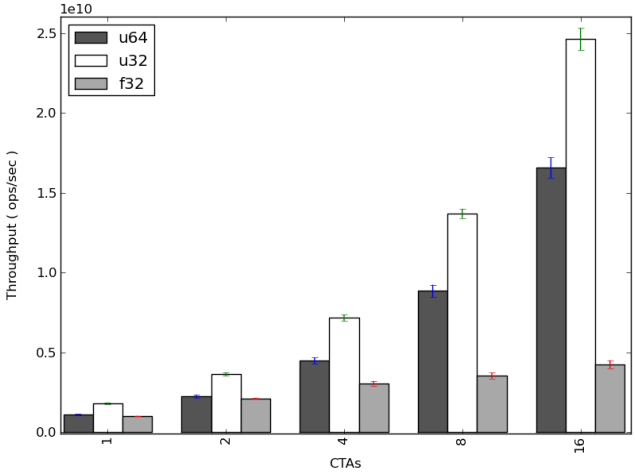

Fig. 7. Basic Integer and Floating Point Instruction Throughput

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 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, we ran the same program where a single thread incremented a single variable in memory until it reached the same threshold. Figure 5 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. This experiment explores the overhead of a context-switch when a thread hits a barrier. Our 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 the effect of barriers on a single thread, we only launched one thread and one CTA for this benchmark. In this case, a thread will hit the barrier, exit into the Ocelot thread scheduler, and be immediately scheduled again.

Figure 6 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

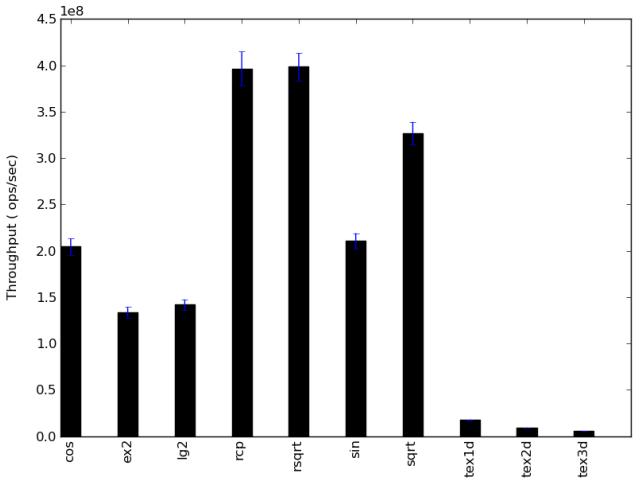

Fig. 8. Special Function Throughput

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. We tested 32-bit and 64-bit integer add, and floating point multiply-accumulate, the results of which are shown in Figure 7. The theoretical upper bound on integer throughput in our 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 our test system, and we need to explore the generated x86 machine code 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. We eventually settled on 16 CTAs, 128 threads per CTA, 2000 iterations 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 8 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 divider. Based on these results, we conclude that 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.

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 II
KERNEL STARTUP AND TEARDOWN OVERHEAD

B. Runtime Overheads

In order to tune our implementation and identify bottlenecks that limited the total application performance, we designed an experiment 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.

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. We instrumented Ocelot using high precision linux timers to try to measure this overhead. Table II 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 our timers. These are completely negligible compared to the overheads of translation and optimization. In the future, we may explore more dynamic work distribution mechanisms such as work stealing that take advantage of this headroom.

Optimization Overhead. In order to determine the relative overhead of applying different levels of optimization at runtime, we instrumented the optimization pass in Ocelot to determine the amount of time spent in optimization routines.

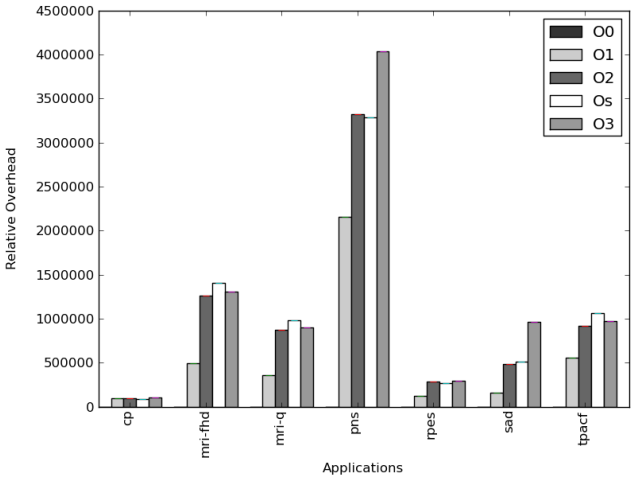

Fig. 9. Optimization Overhead

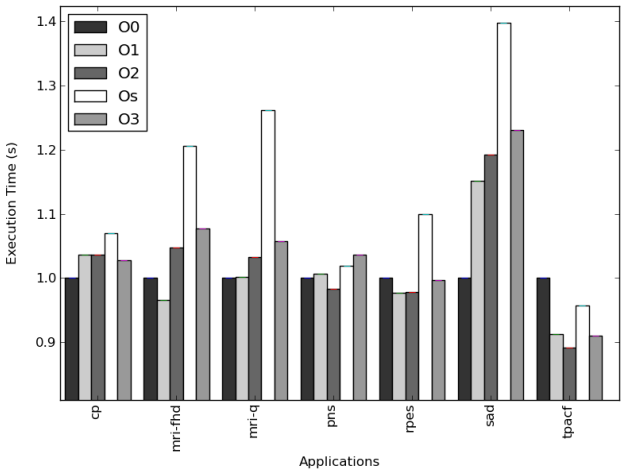

Fig. 10. Parboil Scaling with Different Optimization Levels

We ran the experiment on every application in the parboil benchmark suite to identify any differences in optimization time due to the input program's structure. Figure 9 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.

To determine the impact of different optimization levels on the total execution time of different applications, we measured the execution time of each of the Parboil benchmarks with different levels of optimization. Figure 10 shows these results, where 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. 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

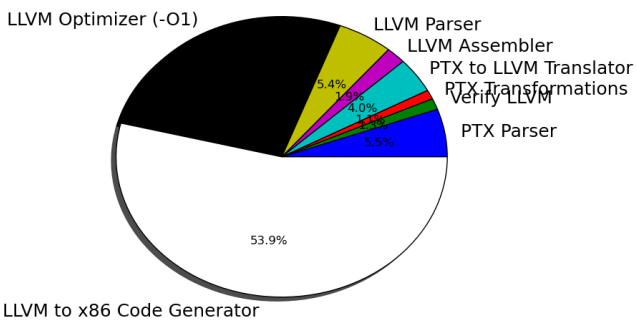

Fig. 11. Contribution of Each Source of Overhead

the LLVM level. These may make many optimizations at the LLVM level redundant, not worth dedicating resources to at execution time. Also, we should 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, we used callgrind [30] 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 11 shows that the vast majority of the translation time is spent in the LLVM code generator. The decision to use our own 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 our 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's translator.

C. Full Application Scaling

Moving on from micro-benchmarks to full applications, we studied the ability of CUDA applications to scale to many cores on a multi-core CPU. Our test system includes a processor with four cores, each of which supported hyperthreading. 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.

We used the Parboil benchmarks as examples of real CUDA applications with a large number of CTAs and threads; our previous work shows that the Parboil applications launch between 5 thousand and 4 billion threads per application [19]. Figure 12 shows the normalized execution time of each

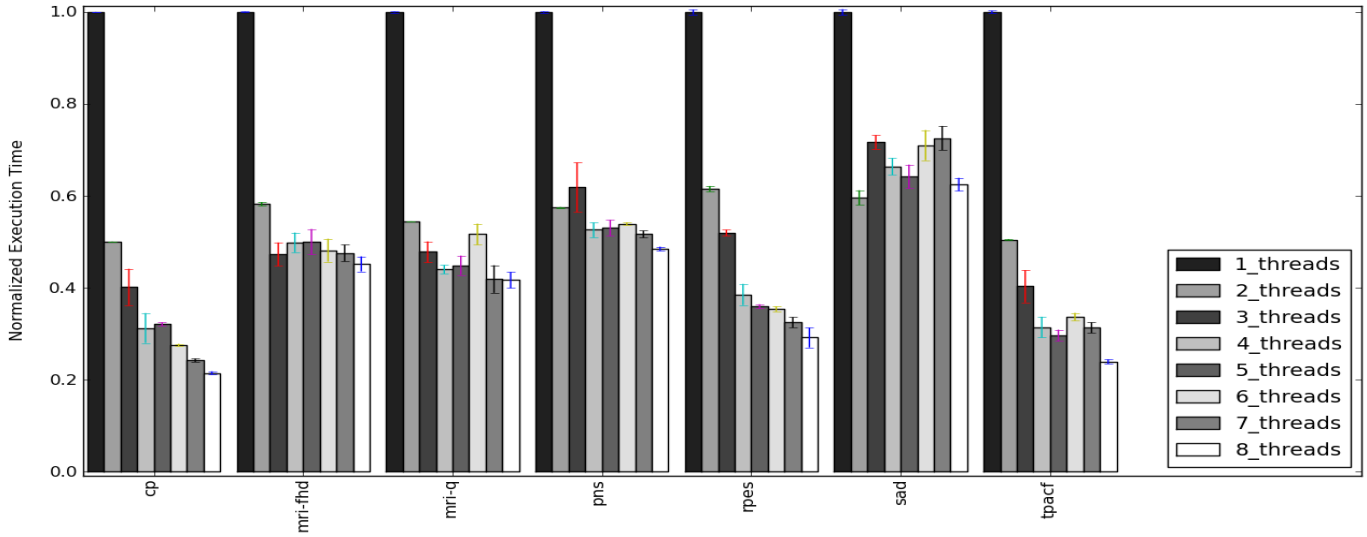

Fig. 12. Parboil Scaling with Number of Worker Threads

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. We are particularly interested in determining 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.

V. Related Work

This paper focuses on comparing Ocelot's dynamic compiler with two main bodies of work: 1) compilation frameworks from CUDA to architectures other than GPUs, and 2) dynamic binary translators.

A. From CUDA to CPUs

The attractive features coupled with the growing acceptance of the CUDA programming model have driven an emerging body of work intent on mapping CUDA to architectures other than GPUs. To the best of our knowledge, MCUDA [16] introduced by Stratton et al. was the first to address this problem. A key difference between MCUDA and other related work is the direction of mapping: MCUDA provides a mapping from CUDA applications to CPUs, while OpenMP to GPU [31], MPI to CUDA [32], and Hadoop using CUDA [33] map from domain specific languages to GPUs. This concept has been extended even so far as to allowing CUDA programs to generate synthesizable RTL for FPGAs by Papakonstantinou et al. [34]. As industry looks for a scalable solution for programming multi-core CPUs and many-core architectures such as Rigel [9], IRAM [35], RAW [36], and Trips [37], CUDA and OpenCL remain the only industry-accepted implementations of explicitly parallel, BSP [6] programming models. The principle idea behind MCUDA, allowing CUDA to be mapped to architectures other than GPUs, enables a single body of bulk-synchronous applications to be brought to bear to the significant challenge of programming many-core architectures.

B. Dynamic Binary Translation

The first industrial implementations of dynamic binary translation were pioneered by Digital in FX32! [38] to execute x86 binaries on the Alpha microarchitecture. Transmeta extended this work with a dynamic translation framework that mapped x86 to a VLIW architecture [39]. Several Java compilers have explored translating the Java Virtual Machine to various backends [24]. Even the hardware schemes that translate x86 to microops used in AMD [14] and Intel x86 [13] 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 correctness checking. The Dynamo [27] system uses runtime profiling information to construct and optimize hot-paths dynamically as a program is executed. Pin [26] allows dynamic instrumentation instructions to be inserted and removed from a running binary. Valgrind [30] 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.

Ocelot employs both forms of binary translation: PTX is translated to LLVM and then to x86 for compatibility, while program optimizations are selectively applied to CUDA kernels. However, several challenges typically faced by dynamic binary translation systems are significantly simplified by the

CUDA programming model, which we took advantage of 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 normal 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 cleanly distinct and registered explicitly with the CUDA Runtime before execution, precluding any need translate on-the-fly.

VI. Insights Gained

These preliminary results allowed us to make the following recommendations about the design of future compilers and binary translators for explicitly parallel BSP languages: 1) managing on-chip memory pressure must be a primary concern, 2) there must be a low overhead mechanism for context-switches, which should occur as infrequently as possible, 3) the work distribution mechanism should tolerate variable CTA execution time, and 4) compiler optimizations should be aware of the thread hierarchy and parallel intrinsics (atomics, barriers, etc).

A. On-Chip Memory Pressure

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 multi-processor; 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.

B. Context-Switch Overhead

The results from our barrier micro-benchmark show 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. In our implementation, we only context-switch on barriers. However, it may be possible to reduce the number of context-switches 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.

C. 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. From these results and our implementation, we believe 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. We would like to see additional work that addresses this problem using static analysis as well as runtime adaptive mapping.

D. Parallel-Aware Optimizations

A significant amount of effort in our implementation was spent dealing with barriers and atomic operations in PTX, and that all of the compiler transforms available in LLVM were oblivious to these program semantics. In the future, we believe that 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.

VII. Open Problems and Future Work

An open problem facing efforts like this is finding the right balance between abstractions that hide architecture features and semantics that expose program characteristics to dynamic compilers and hardware optimizers. Stratton et al. implicitly argue in MCUDA that the CUDA programming model is a step in the right direction, and we agree. However, the relative performance differences between equivalent CPU and GPU implementations shown in this work indicate that CUDA is

only an adequate, not a perfect solution. Memory mapping and traversal patterns, synchronization overhead, and optimal instruction mix are some of potentially many hardware features that are not abstracted away from the programmer. Memory pressure, non-uniform work distribution, thread divergence, and thread-fission/fusion are opportunities for dynamic optimization that are lost in state-of-the art compilation chains. It is our opinion that the changes required to solve this problem conclusively would require an impossibly automatic mapping from the definition of a problem to its efficient hardware solution.

The infrastructure developed by the authors and described in this paper is geared towards solving a more modest problem: can we identify those program characteristics that make a program more or less suited to execution on a particular style of architecture? Even if we are far away from automatically changing the memory traversal pattern of a program, can we identify those architectures for which a given pattern is efficient? In future work immediately succeeding this study, we plan to explore this problem in detail by: 1) identifying key application characteristics that impact relative performace on GPUs vs CPUs, and 2) coupling this information with dynamic mapping schemes proposed in our prior work [40] to automatically map CUDA kernels to one of many potential GPUs or CPUs in a heterogeneous system.

VIII. Conclusions

This paper presents a detailed overview of Ocelot's binary translator from PTX to Multi-core x86 CPUs. Through the study of our translator using several microbenchmarks and full applications, we were able to identify on-chip memory pressure, context-switch overhead, and variable CTA execution time as fundamental issues that must be addressed 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 architetcures, and developers seek programming models that can scale to them.

Acknowledgement

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc. and NVIDIA Corp. both through research grants, fellowships, and technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp.

References

[1] NVIDIA, *NVIDIA CUDA SDK 2.1*, 2nd ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[2] IMPACT, "The parboil benchmark suite," 2007. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/parboil.php

[3] 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, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[4] 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.

[5] G. Contreras and M. Martonosi, "Characterizing and improving the performance of the intel threading building blocks runtime system," in *International Symposium on Workload Characterization (IISWC 2008)*, September 2008. [Online]. Available: http://www.gigascale.org/pubs/1350.html

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

[7] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, 2nd ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[8] K. O. W. Group, *The OpenCL Specification*, December 2008. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[9] 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, pp. 140–151.

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

[11] AMD, "Amd stream computing user guide," August 2008.

[12] Intel, "G45 express chipset." http://code.google.com/p/hydrazine/.

[13] ——, "First the tick, now the tock: Next generation intel microarchitecture (nehalem)," Intel Coporation, Tech. Rep., 2008.

[14] O. Liu, "Amd technology: power, performance and the future," in *CHINA HPC '07: Proceedings of the 2007 Asian technology information program's (ATIP's) 3rd workshop on High performance computing in China*. New York, NY, USA: ACM, 2007, pp. 89–94.

[15] 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.

[16] J. Stratton, S. Stone, and W. mei Hwu, "Mcuda: An efficient implementation of cuda kernels on multi-cores," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-08-01, March 2008. [Online]. Available: http://www.gigascale.org/pubs/1278.html

[17] 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.

[18] 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. [Online]. Available: http://hdl.handle.net/1853/27246

[19] 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.

[20] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid simd: Abstracting simd hardware using lightweight dynamic mapping," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 216–227.

[21] 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.

[22] 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.

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

[24] 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, pp. 399–417, 2005.

[25] C. Wang, S. Hu, H.-S. Kim, S. R. Nair, M. B. Jr., Z. Ying, and Y. Wu, "Stardbt: An efficient multi-platform dynamic binary translation system." in *Asia-Pacific Computer Systems Architecture Conference*, ser. Lecture Notes in Computer Science, L. Choi, Y. Paek, and S. Cho,

Eds., vol. 4697. Springer, 2007, pp. 4–15. [Online]. Available: http://dblp.uni-trier.de/db/conf/aPcsac/aPcsac2007.html#WangHKNBYW07

[26] 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.

[27] 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, pp. 1–12.

[28] 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.

[29] G. Diamos, "Hydrazine: A high performance library for c++ and cuda," http://code.google.com/p/hydrazine/, November 2009.

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

[31] S. Lee, S.-J. Min, and R. 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*. New York, NY, USA: ACM, 2009, pp. 101–110.

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

[33] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.

[34] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "High-performance cuda kernel execution on fpgas," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 515–516.

[35] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: a modular reconfigurable architecture," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000, pp. 161–171.

[36] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 2.

[37] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed microarchitectural protocols in the trips prototype processor," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 480–491.

[38] 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, pp. 2–2.

[39] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing™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.

[40] G. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *HPDC'08*. Boston, Massachusetts, USA: ACM, june 2008.