Translating GPU Binaries to Tiered SIMD Architectures with Ocelot

Gregory Diamos and Andrew Kerr and Mukil Kesavan
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
gregory.diamos@gatech.edu arkerr@gatech.edu mukil@cc.gatech.edu

***Abstract*—Parallel Thread Execution ISA (PTX) is a virtual instruction set used by NVIDIA GPUs that explicitly expresses hierarchical MIMD and SIMD style parallelism in an application. In such a programming model, the programmer and compiler are left with the not trivial, but not impossible, task of composing applications from parallel algorithms and data structures. Once this has been accomplished, even simple architectures with low hardware complexity can easily exploit the parallelism in an application.**

With these applications in mind, this paper presents Ocelot, a binary translation framework designed to allow architectures other than NVIDIA GPUs to leverage the parallelism in PTX programs. Specifically, we show how (i) the PTX thread hierarchy can be mapped to many-core architectures, (ii) translation techniques can be used to hide memory latency, and (iii) GPU data structures can be efficiently emulated or mapped to native equivalents. We describe the low level implementation of our translator, ending with a case study detailing the complete translation process from PTX to SPU assembly used by the IBM Cell Processor.

I. Introduction

Motivation. The dramatic transition from frequency scaling driven by hardware pipelining and aggressive speculation to concurrency scaling riding massive hardware multithreading and computationally dense SIMD demands an equally dramatic programming model shift. GPU architectures have neatly skirted the power and frequency walls impeding the progress of general purpose CPU architectures by targeting a class of applications that can be expressed using parallel algorithms and data structures – forgoing the complexities of out of order superscalar style scheduling [1] and inefficiencies of multilevel caches [2]. The heaviest load in this new many-core arena falls on the shoulders of the programmer to leverage programming models that harness the computational capability of future many-core architectures.

Parallel Languages. Hwu et al. warn that as market requirements and fabrication capabilities enable the development of increasingly parallel architectures like GPUs, an irresistable temptation to impose explicitly parallel programming models on these architectures will follow [3]. This trend is apparent in the introduction of the CUDA programming model for NVIDIA GPUs where MIMD and SIMD style parallelism [4] is exposed directly to the programmer. However, rather than impeding development, the incredible volume of work concerning ports of applications from a variety of domains including graphics [5], signal processing [6]–[8], computational finance [9], physics simulation [10]–[12], streaming media [13], and others [14] to the CUDA programming model suggests architectures that leverage explicitly parallel representations of programs can dramatically impact the field of general purpose computing.

CUDA. Given the success of the CUDA programming model for GPU architectures and the lack of comparable models for general purpose many-core processors (the closest competitors being threading libraries like Pthreads [15], OpenMP [16], and MPI [17]), it is a natural extension to support CUDA programs on other architectures. This approach has been shown to be possible for x86 architectures via emulation [4] and efficient via source to source translation [18]. The expectation is that many-core architectures will be able to leverage the explicit data parallelism in CUDA programs.

PTX. In order to provide a generic and efficient approach, we would like to be able to move down a level of abstraction; rather than compiling CUDA programs to distinct architectures, we would like to generate an intermediate representation that retains the explicit data parallelism in a CUDA application, while easing translation to generic architectures. Luckily, NVIDIA already provides such a representation: a virtual instruction set called PTX is first generated by the CUDA compiler before a just in time (JIT) translator is used to target a specific GPU architecture before a program is executed. If we can develop an equivalent translator for architectures other than NVIDIA GPUs, it becomes possible to run existing CUDA applications efficiently on generic architectures.

Ocelot. To enable the efficient migration of existing CUDA applications across diverse many-core architectures, this paper introduces a set of translation techniques, implemented in a framework called Ocelot, for mapping low level GPU specific operations to many-core architectures. Based on the feature set of the target architecture, we show how (i) high level PTX operations can be selectively translated to hardware primitives or emulated in software, (ii) GPU specific data structures can be mapped to native equivalents or emulated in software, and (iii) introduce program analysis techniques required to translate the data parallel representation of PTX. We show how

these generic techniques can be applied to the translation of PTX to the IBM Cell Specialized Processing Unit (SPU) ISA, and provide a case study covering the complete translation process of a simple but nontrivial example.

Organization. This paper is organized as follows: Section II gives background on PTX; Section III presents Ocelot; Section IV describes translation to Cell; Section V walks through an example translation; and Section VI covers related work.

II. Background - PTX

As a virtual instruction set, PTX was designed to be translated. Rather than adopting the competing philosophy that a single invariant instruction set is a necessity for backwards compatibility, PTX allows the underlying architecture to be changed without the need for the hardware decoders present in many modern processors [19], [20]; the approach is similar to that taken by Transmeta where x86 is treated as a virtual instruction set that is dynamically translated to a propriety VLIW architecture [21]. The advantage of using a new instruction set rather than x86 is that it allows the use of a set of abstractions that are agnostic by design to the underlying architecture. In PTX, this design goal of architecture agnosticism is achieved through (i) a set of constructs in the language that ease translation, (ii) a relaxed set of assumptions about the capabilities of the architecture, and (iii) a hierarchical model of parallelism.

A. Facilitating Translation

Typed SSA. In order to enable translation, PTX is stored in a format that is amenable to program analysis. Specifically, instructions use an infinite set of typed virtual registers in partial SSA form [22] as operands. Partial SSA in this context means that registers have a single defs, multiple uses, and no explicit Φ functions. Directly, this partial SSA form eliminates a redundant register allocation stage in the high level compiler, and ensures that the translator can easily infer the program data flow without performing memory disambiguation to determine aliases of variables spilled to memory. Indirectly, the typing system enhances a variety of program analyses. Virtual registers in PTX are explicitly typed in terms of minimum precision (8-64 bits) and basic type (bitvector, bool, signed/unsigned int, and floating point), with support for weakly-typed languages [23] provided via explicit convert instructions. This typing system is remarkably similar to the system used in LLVM [24], which was designed to facilitate array dependence analysis and loop transformations in addition to field-sensitive points-to analyses, call graph construction, scalar promotion of aggregates, and structure field reordering transformations which do not require explicit typing [25]–[27].

PTX vs LLVM. Other virtual instruction sets like LLVM further enable program analysis though explicit memory allocation [24] instructions, a feature that is noticeably lacking in PTX. We speculate that the lack of this feature is closely related to the lack of support for heap based memory allocation in any current NVIDIA GPU [4]. In general, it is more difficult to implement heap based allocation for parallel architectures [28] than static allocation, and this process is further complicated by the weak-consistency memory model assumed by PTX. It is not clear whether adding a similar instruction to PTX would provide any benefit.

SIMD. Support for SIMD architectures is provided with the inclusion of predication and select instructions in the PTX ISA. Starting with the ability of threads to take arbitrary control paths, predication allows simple control flow to be compressed into a series of predicated instructions. The PTX translator can selectively implement these instructions as if-then-else style branch intensive code on architectures with narrow SIMD width or hardware predicates on architectures that support them.

Parallelism. Additionally, PTX includes directives for explicitly handling fundamental parallel operations like barriers, reductions, atomic updates, and votes. Specifying these operations at a high level allows the translator to leverage specialized hardware as in [29] if it exists, fall back on common primitives like semaphores that are available to the majority of modern architectures, or convert parallel sections with synchronization to equivalent sequential loops between synchronization points as in [18].

B. A Hierarchical Model of Parallelism

Threads and Warps. The basic unit of execution in PTX is a light-weight thread. All threads in a PTX program fetch instructions from the same binary image, but can take distinct control paths depending on the values of pre-set id registers with unique values for each thread. The first level of hierarchy groups threads into SIMD units (henceforth warps). The warp size is implementation dependent, and available to individual threads via a pre-set register.

In order to support arbitrary control flow as a programming abstraction while retaining the high arithmetic density of SIMD operations, NVIDIA GPUs provide hardware support for dynamically splitting warps with divergent threads and recombining them at explicit synchronization points. PTX allows all branch instructions to be specified as divergent or non-divergent, where non-divergent branches are guaranteed by the compiler to be evaluated equivalently by all threads in a warp. For divergent branches, targets are checked across all threads in the warp, and if any two threads in the warp evaluate the branch target differently, the warp is split. PTX does not support indirect branches, limiting the maximum ways a warp can be split to two in a single instruction. Fung et. al. [30] show that the PTX translator can insert synchronization points at post-dominators of the original divergent branch where warps can be recombined.

CTAs. As shown in figure 1, the second level of hierarchy in PTX groups warps into concurrent thread arrays (CTAs). The memory consistency model at this point changes from sequential consistency at the thread level to weak consistency with synchronization points at the CTA level. Threads within a CTA are assumed to execute in parallel, with an ordering constrained by explicit synchronization points. These synchronization points become problematic when translating to sequential

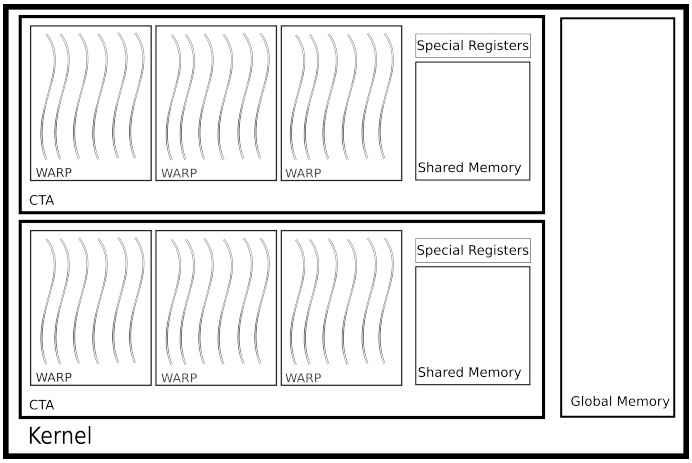

Fig. 1. PTX Thread Hierarchy

architectures without software context switching: simple loops around CTAs incorrectly evaluate synchronization instructions. Stratton et. al. [18] show that CTAs with synchronization points can be implemented without multithreading using loops around code segments between synchronization points. CTAs also have access to an additional fixed size memory space called shared memory. PTX programs explicitly declare the desired CTA and shared memory size; this is in contrast to the warp size, which is determined at runtime.

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

Scalable Parallelism. Collectively, these abstractions allow PTX programs to express an arbitrary amount of data parallelism in an application: the current implementation limits the maximum number of threads to 33554432 [4]. The requirements of weak consistency with no synchronization among CTAs in a kernel may seem overly strict from a programming perspective. However, from a hardware perspective, they allow multiprocessors to be designed with non-coherent caches, independent memory controllers, wide SIMD units, and hide memory latency with fine grained temporal multithreading: they allow future architectures to scale concurrency rather than frequency. In the next section, we show how a binary translator can harness the abstractions in PTX programs to generate efficient code for parallel architectures other than GPUs.

III. Ocelot

PTX as an IR. PTX provides a description of an application at a similar level of abstraction to the intermediate representation of a compiler. Because of this, the process of translating a PTX application to another architecture is remarkably similar to the process of compiling a CUDA application to that architecture after the intermediate representation has been generated by the compiler front end. At a very high level, the translator must examine a set of PTX instructions before generating an equivalent set of native instructions. This process can potentially create new control flow paths and introduce additional variables requiring the translator at the very least to then perform data flow analysis to determine register live ranges before allocating registers and generating traslated code.

Trivial Translation. An incredibly simple translator targeting only single threaded PTX applications that avoided using GPU specific shared memory could be implemented with only the aforementioned operations. However, the PTX thread hierarchy and the lack of support on many architectures for scratchpad memory significantly complicates the translation process for non-trivial programs. In Ocelot, this complexity is divided between modifications to the code generated by the translator and a runtime component that interacts with the translated application as it is executing.

A. Runtime Support

Thread Hierarchy. In Ocelot, a runtime component is used to provide services to the translated application that are not provided by the target architecture. It should be noted that even PTX applications translated to NVIDIA GPUs using the CUDA tool chain are coupled with a runtime component that handles resource allocation and a first pass of thread scheduling. Similarly, Ocelot's runtime component performs a first pass of thread scheduling that is required because no hardware platform that we are aware of today including GPUs can manage the massive number of threads available to PTX in hardware. The first pass of thread scheduling is performed by looping over the number of CTAs in a PTX program, and launching them on the hardware as resources become available. After this first pass of scheduling, a second pass is required on architectures that do not provide hardware support for multithreading to the degree possible within a CTA (up to 512-way). SIMD architectures that do not provide hardware support for warp splitting and recombining require a third pass where software checks are inserted after potentially divergent branches. All of the synchronization operations available in PTX without native hardware equivalents must also be handled at this level.

GPU Hardware Emulation. In addition to thread scheduling, the runtime component must also emulate GPU specific data structures (shared memory and special registers), and GPU specific instructions that are not directly translated. PTX includes instructions for high level math and trigonometric functions like sin, atan, and exp. For architectures without instruction level implementations of these operations, either the translator must inline their implementation, or call equivalent functions. In regards to accesses to special registers and shared memory, the translator can insert traps into the runtime

that redirect accesses to preallocated data structures in main memory. Additionally, the translator can wrap accesses to main memory with conditional traps into the runtime implementing software caching in scatchpad memory for architectures without hardware caches.

B. Thread Compressing Loops

Thread Compression. The degree of hardware support for multi-threading in modern processors varies from none in single core architectures to 16 in Intel Nehalem [31] to 32 in Sun Niagra 2 [32]. None of these architectures can natively support the maximum of 512 threads within a CTA available to PTX programs, and require the use of either software multithreading or a technique that we refer to as thread compression to handle PTX programs with more than a few threads per CTA. To our knowledge, this idea was first introduced in [18] – though it is conceptually similar to the blocked loop unrolling used in OpenMP [16] and programming model intrinsics used by Intel Larabee [33]; it consists of emulating fine grained PTX threads with loops around unsynchronized sections of code, compressing several PTX threads into a single hardware thread. In the context of a translator, we implement thread compression by examining the control flow graph of the application and finding all synchronization instructions. For every synchronization instruction starting from the root of the dominator tree of the application, we create a loop starting at the earliest unsynchronized dominator of the block with the synchronization instruction and ending immediately before the instruction.

Multithreading. In our complete partitioning solution, the number of PTX threads is divided among the hardware threads available in a single core. A loop over PTX threads is assigned to each hardware thread or further divided among software threads if the overhead of doing so is low enough. Intuitively, implementing software multithreading rather than thread compressing loops is advantageous only if there are opportunities to conditionally context switch at boundaries other than synchronization instructions. A motivating example can be found in an architecture with a software managed cache implemented in scratchpad memory. In such an architecture, cache misses could incur a high latency penalty and could also be made visible to a software thread scheduler. If the context switch overhead was lower than the direct and indirect cache miss penalties, it would be advantageous to switch to another thread and while a cache miss was being serviced.

C. Emulating Fine Grained TMT

Temporal Multithreading. In the lightweight threading model of PTX, threads have distinct register sets, program counters, and little else. PTX does not explicitly support a stack, an entire application shares a virtual address space, and special registers are handled by the runtime. In this model, a context switch involves check-pointing the register file and program counter, selecting a new thread, and restoring the new thread's live registers and program counter. The major direct cost of this operation is the checkpoint and restore of the register file, and the major indirect cost is the displacement of the working set of a thread on architectures with caches. Being a binary translator, Ocelot has an opportunity to reduce these overheads before the application is ever executed.

Context Switches. We begin by noticing that PTX programs have a relatively large variance in the number of live registers at any time as can be seen in Figure 2. For sections of the application where the number of live registers is much smaller than the size of the architected register file, multiple versions of the same section can be generated using non-overlapping sets of registers. Threads can be distributed evenly across these orthogonal register sets; if there are N such sets of registers, then N threads can be stored in the register file at any given time. A single extra register can be added to each thread's state pointing to the program counter of the next thread, and the process of a context switch involves saving the program counter of the current thread and jumping to the program counter of the next thread. This can be implemented using a single move instruction and a single branch instruction on many architectures – making context switches among threads stored in the register file nearly free. Note that assigning fewer registers to each thread allows more contexts to be stored at any given time, but increases the probability of having to spill any given register. This phenomenon creates a trade off between context switch and register spill overhead.

D. Software Warp Formation

Dynamic Warp Formation. The fact that PTX was simultaneously designed for SIMD architectures and arbitrary control flow creates significant challenges for architectures that implement SIMD instruction sets without hardware support for dynamically reforming warps. Fung [34] shows a 4x worst case, 64x best case, and 32x average increase in IPC on a 16-wide SIMD architecture with no warp recombination across eight sample CUDA applications wehn compared to a serial architeture. This implies that there is at least the potential for speedup on architectures that support SIMD instructions, but do not have hardware support for detecting divergence within a SIMD set. In order to provide the ability to exploit SIMD instructions in architectures that support them, we extend Fung's work to address software dynamic warp formation.

A Software Implementation. Ocelot uses a combination of binary translation and runtime support to implement dynamic warp splitting. We optimistically assume that PTX threads can be packed into warps and statically assign them during translation. We then run a pass through the control flow graph and find branch instructions that are specified as possibly divergent. For these instructions, we break the conditional branches into three operations: (i) a SIMD computation of the branch target, (ii) a SIMD check to see if all branch targets are equal, and (iii) either a trap into the runtime if the check fails or the original branch instruction if the check succeeds. In the case that the check succeeds, the program runs N*M threads at a time where N is the warp size and M is the degree of hardware multithreading. If the check fails, the runtime splits the warp into two partially full warps, each taking a different

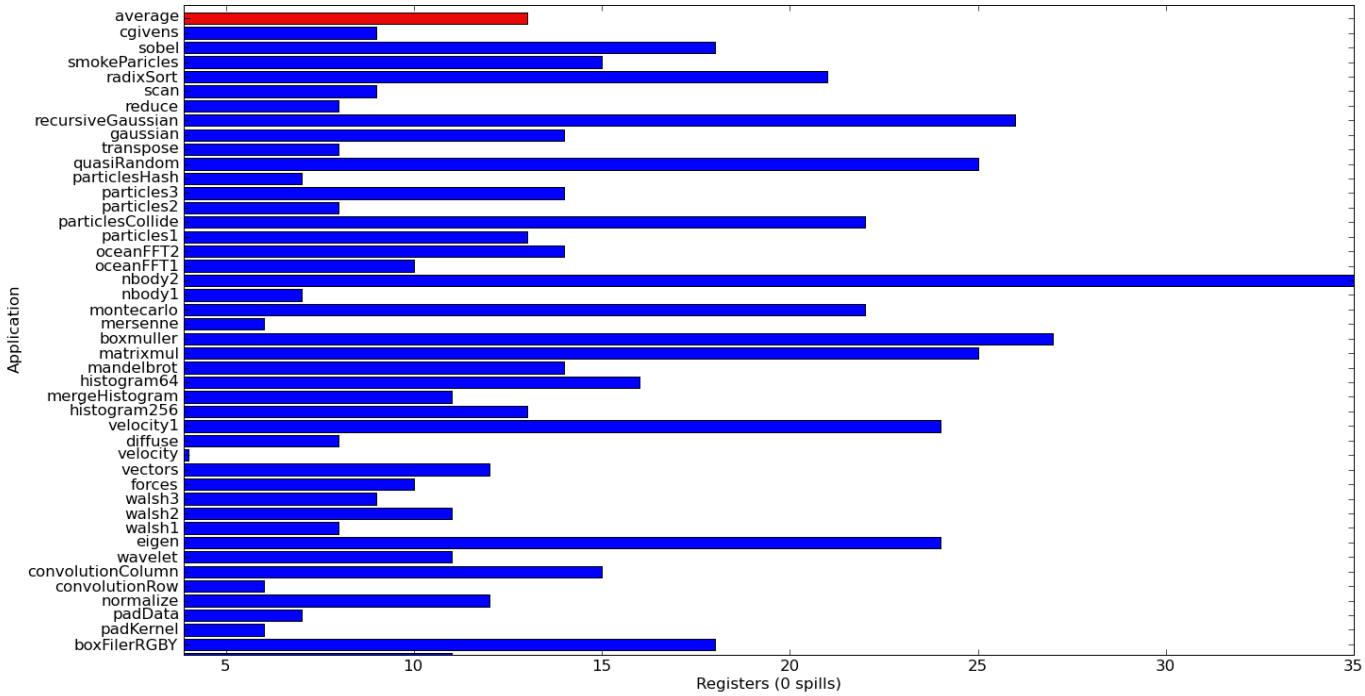

Fig. 2. Sample CUDA Register Usage

path. For partially full warps, either predication is used to disable inactive ways in the warp if it is supported, or loads and stores from those ways are ignored by the runtime.

Warp Reformation. Ocelot implements dynamic warp reforming in a similar fashion. We note that all threads will eventually return to the same program counter at synchronization points. Recall from [18] that thread loops end at synchronization points. If Ocelot inserts a trap into the runtime after each synchronization point, the runtime can scan the entire set of PTX threads assigned to a single hardware thread, deleting any newly created warps and enabling all of the threads in the remaining warps. This reverts the assignments of threads to warps to the static assignment specified by Ocelot when the program was translated. The reader should note that this operation creates a trade off between the loop and runtime overhead needed to reform warps and the performance lost from running partially full warps. Fung suggests that reformations can should place at the immediate post dominator of all divergent branches for hardware implementations which achieve a 44.7% average speedup over not recombining. Combining this with our approach would required loops around all divergent control paths. A more thorough evaluation is needed before a similar conclusion can be reached for our proposed software implementation.

E. Dynamic Translation

GPUs. Until now, all of the techniques that we describe can be implemented using static translation before a PTX application is executed, which is the approach used by the CUDA translator as of version 2.1 [4]. Given the prominent examples of dynamic translation for x86 [35]–[37], Java Bytecode [38], and VLIW architectures [21], [39] the reader might be surprised by its omission here. Dynamic translation in the previously mentioned examples is typically done at the super-block level or lower. Multihtreading significantly complicates the process of dynamic translation as threads can take divergent control flow paths, simultaneously requiring the services of the translator. On GPUs, PTX provides no mechanism for self-modifiable code, making dynamic translation at the PTX level impossible without delving into the undocumented and unsupported realm of GPU specific binary translation. Additionally, the weak memory consistency model assumed by PTX across CTAs would make updating a shared code cache difficult to implement. Finally, the sheer magnitude of distinct control paths running on modern NVIDIA GPUs at any given time (up to 15360 [4]) would make lock based update mechanism to a shared code cache prohibitively expensive – these updates would need to be made by a translator written in CUDA or PTX running on the GPU; we are not aware of any attempt to port generic compiler/translator operations to wide SIMD architectures. Luckily, this restriction is not overly cumbersome: PTX programs are typically smaller than complete applications and are launched like function calls using the CUDA runtime API. Multiple launches of the same PTX program can be translated once and cached for subsequent executions similar to method-level dynamic translation used in [38].

Other Architectures. Though dynamic translation does not seem practical for GPU architectures, other architectures may offer a different perspective. For example, a four-core

superscalar processor supporting four wide SIMD as in Intel's [31] will only have at most four distinct control flow paths, where each core offers fast single threaded performance when compared to a GPU. Each thread can build a chain of superblocks as it encounters code that has not yet been translated and update a protected global code cache in main memory. These are the architectures where previous examples of dynamic translation have found success, and we plan to explore dynamic implementations of Ocelot when targeting them.

F. Integration With CUDA

Intercepting API Calls. As mentioned briefly in the previous section, PTX programs are typically called from larger C applications using the CUDA runtime API. Along with providing functions for calling PTX programs, the API allows the C application to query to system to determine the number of and types of GPUs, bind to a specific GPU, allocate and deallocate memory on the GPU, and perform DMA copies from the virtual address space of the C program to the address space of the GPU. In order to target architectures other than GPUs, we would like to be able to emulate the functionality of some of these calls – particular memory allocation and transfer operations – and use Ocelot rather than NVIDIA's translator when a PTX application is called. To accomplish this, we use a technique similar to that used by Gupta. et. al. for GPU virtualization [40]: we provide a dynamically linked library that replaces cudart.so and intercepts all CUDA API calls, selectively redirecting them to either native CUDA calls or Ocelot equivalents. This enables us to conditionally choose between NVIDIA's translator when launching a PTX program on an NVIDIA GPU, or Ocelot when launching a program on another supported architecture.

IV. Targeting IBM Cell

Cell Background. This section assumes that the reader is familiar with the architecture of the IBM Cell Processor. We suggest the following reference [41] to readers who are new to this architecture. We adopt the terms **SPU** to refer to the either the individual cores in Cell or the instruction set used by those cores, **local store** to refer to SPU specific scratchpad memory, and **mailbox** to refer to the dedicated communication channels between adjacent cores in Cell.

Translation. Now that the reader is familiar with the general process of translating a PTX application to a new architecture, we narrow the scope of the paper and consider translating specifically to the IBM Cell Processor. At this time, Cell remains the only Ocelot target with a functioning implementation and the following discussion is informed by the lessons learned during that implementation. We began with the Cell processor because of its novelty as a target for translation and the similarities between its architecture and that of NVIDIA GPUs. At a high level, both architectures provide an array of relatively simple processors with SIMD capabilities – Shader Modules (SMs) for GPUs and Synergistic Processing Units (SPUs) for Cell – suitable for mapping the thread hierarchy expressed in PTX. We begin with a detailed description of this mapping, moving on to a discussion of emulating shared memory and special registers in the SPU local store, software caching and temporal multithreading, warp resizing, Cell's lack of predication, and ending with a discussion of register management conforming to the Cell ABI.

A. Mapping the Thread Hierarchy

The mapping of the PTX thread hierarchy to Cell mirrors the set of hardware units in Cell:

- Warp - SIMD Unit
- CTA - SPU
- Kernel - CTA Array

Kernels and CTAs. Ocelot first considers the number of CTAs in a PTX program, creating a queue of all CTAs in a data structure on the Power Processing Unit (PPU). CTAs are launched on SPUs as they become available. This approach was chosen rather than static assignment in order to maintain efficient work distribution of CTAs in the case that they have variable and nondeterministic runtimes – which has been problematic for applications like ray tracing with secondary rays [5]. Ocelot does not split CTAs from the same group across SPUs in order to take advantage of the weak-consistency with no synchronization assumption among groups: caches may be used by each SPU without the need for a coherence protocol, and even though Cell provides sequential consistency among all SPUs, it is not necessary to ensure the correctness of any translated application. Compared to static assignment, Ocelot's queue based approach does incur additional latency for the SPU to notify the PPU that its CTA has finished and to be issued a new one. Using the mailbox primitives available in Cell, we believe that this overhead can be made smaller than a DMA copy by the SPU to main memory – in other words, we believe that the overhead is negligible for non-trivial CTAs.

Warps. Once an SPU has been assigned a CTA, it is partitioned into warps. Most operations can be packed into SIMD instructions in the SPU ISA. However, particularly lacking are scatter and gather type operations required to implement loads and stores to distinct addresses in SIMD. Instructions such as these are serialized as a sequence of load and store operations. Given the high latency of memory operations, we do not believe that this solution creates any significant overhead compared to an explicit scatter or gather instruction, which would have to perform the same underlying operations. Warps are composed of either two or four threads depending on the floating point precision used by the PTX program. SPUs currently use 128-bit registers capable of holding either four single precision floating point numbers, or two double precision numbers. We note that it would be possible to dynamically switch between a warp size of two and four by inserting explicit runtime traps before entering a section of code using a different precision. A simple approach would be to make this distinction after each synchronization instruction signifying the start of a new thread loop because it could be combined with the warp reformation operation. However, a more complex analysis would be required to

determine the most efficient positions to insert these warp precision transitions.

Threads. The first partitioning stage reduces the number of threads within a CTA by a factor of either two or four depending on the warp size. The remaining warps must then be implemented in software, as SPUs have no hardware support for simultaneous multithreading. For this implementation, we use a combination of thread compression and software TMT as described in Section III-B. Thread compression is used to reduce the amount of state associated with each thread, and software TMT is used to hide latency in the case of a long latency operation. Without thread compression, the runtime would have to maintain the live register set of each warp in the SPU local store. Thread compression by a factor of N reduces the number of times that this live register set must be saved and loaded from the local store by at least a factor of N. However, this must be balanced against the number of software threads in order improve the chance of finding an available new thread when a high latency operation is encountered. Ocelot places a minimum on the number of software threads as the number of threads that can be stored completely in the register file, and it places a maximum of the maximum latency of any operation divided by the average context switch latency. This takes advantage of the lower overhead of switching to a thread whose live registers are already in the register file, also acknowledging that in the worst case where every thread immediately encounters an operation with the greatest possible latency, the process of cycling through all threads and starting their operation will gives the first thread's operation time to finish. If there are options between the minimum and maxmimum number of threads, Ocelot reverts to a user defined parameter, floored to the maximum and ceilinged to the minimum.

B. Software Caching and Threading

Context Switch On Cache Miss. Software TMT allows for a context switch to hide high latency operations. In Ocelot, the only high latency operation considered is an access to main memory. This operation is accomplished by trapping into the runtime component which maintains a software managed cache in the SPU local store memory. On every access to main memory, the runtime checks to determine if the specified address is present in the cache. If so, it performs a load or store to the copy in the local store. If the address is not present, it first checks for pending operations on the same cache line, initiates an asynchronous DMA copy from main memory if there are none, and performs a context switch to a ready thread. The check for pending operations to the same cache line is needed to prevent deadlock situations where one warp kicks out data referenced by another, which then proceeds to kick out the first warp's data before it has a chance to use it. If a cache access fails due to a miss, the program counter of the current warp is rolled back to the previous instruction so that the access will be replayed when the warp is scheduled next.

Thread Scheduling. Context switching in software TMT is managed by the runtime. A set of warps is stored in the register file and the remaining warps are mirrored in the local store. The runtime maintains a ready flag for each warp that determines whether or not it can be scheduled. The ready flag begins set for each warp, is cleared when the warp encounters a cache miss or hits the next synchronization point, and is reset when a cache line is returned from main memory. Divergent warps manipulate the same registers as their original counterparts, and are scheduled independently by the runtime. However, threads within the warp that are not active do not update registers.

Scheduling Algorithm. The goal of scheduling in this context is to provide the lowest total runtime of the total set of warps, rather than provide any fairness guarantees to the set of warps. This goal is best accomplished by minimizing the number of context switches and keeping the working set of a particular warp warm in the cache as much as possible, so our policy selects the most recently used ready warp, giving preference to those already in the register file. Note that the static partitioning of the register sets used by warps means that any two warps using the same registers cannot be present in the register file at the same time. When conflicts between the most recently used warp and warps already in the register file exist, we choose a warp already in the register file based on the intuition that the benefits gain from avoiding a register checkpoint and restore will be greater than those from selecting a thread whose working set is more likely to be in the cache. Empirical studies have shown the opposite behavior in the context of traditional process level multithreading where indirect misses due to replacement and reordering have higher penalties than register checkpointing [42]. These results are not applicable to PTX threads which share data, run in near lockstep, and are designed with high spatial locality to exploit memory coalescing [43].

Managing Registers. Note that the number of live registers can potentially change between synchronization instructions, allowing more or less warps to be cached in the register file at any given time. The Ocelot translator takes as a parameter the number of registers desired per thread. Any required registers over this limit are spilled to a dedicated local memory region in the local store, but any fewer registers can be assigned to other warps. This potentially complicates transitions across synchronization points because the registers assigned to the same warp in the previous section might not correspond exactly to threads in the new section. Ocelot provides two approaches to handling this case. The first approach examines the total number of registers required by each section, and selects the maximum for each, while the second approach, inspired by [35], provides compensation code at synchronization points to copy live values in previous registers into the newly allocated registers. The first approach potentially allows fewer threads to be cached in the register file, but avoids overheads associated with compensation code. Ocelot provides both approaches to the user specifiable via a configuration option.

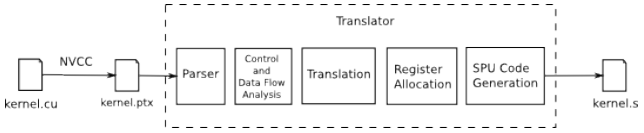

Fig. 3. Ocelot Translation Process

C. Emulating GPU Memory Structures

In addition to the resources provided by the IBM Cell architecture, PTX assumes the existence of scratchpad memory referred to as shared memory, and special registers.

Special Registers. Special registers can be handled in two ways by Ocelot: in the translator or in the runtime. Ignoring recently introduced special registers tied to GPU performance counters [4], special registers make the CTA dimensions, warp size, and number of CTAs available to PTX threads. To handle these instructions in the translator, Ocelot maps them to SSA registers that are initialized when the thread is started and exist for the lifetime of the application. Subsequent data-flow analysis and dead code removal eliminates registers that are never referenced, while ensuring that physical registers are shared among SSA registers that do not have overlapping live ranges. The second approach replaces all references to special registers with traps into the runtime which determine the correct value based on which thread is currently executing. This is possible because all special registers are derived from a combination of the active warp, CTA, and the static dimensions of the PTX program. Either of these approaches are available to the user via a configuration parameter.

Shared Memory. Handling shared memory is potentially more complicated than special registers because of its overlap in functionality with the SPU local store which is utilized by the software managed cache of global memory. Shared memory is accessible by all warps in a CTA, and therefore it can allocated directly in the local store. However, it is not immediately clear that shared memory should be directly implemented in the SPU local store because it would reduce the size of the software managed cache. The alternative would be to implement it in global memory using the software managed cache to reduce its access time. In the first case, loads and stores to shared memory could be directly translated into loads and stores to the local store, whereas in the second case, they would be replaced by traps into the runtime, adding a performance penalty. In PTX, the assumption is that shared memory is significantly faster than global memory, and applications that relied on this would benefit from the first technique. Depending on the application, the additional shared memory latency may or may not be offset by the reduction in cache misses and context switches. Ocelot implements both techniques and again allows the user to choose one or the other.

D. Translating PTX Instructions

SPU vs PTX. When compared to PTXs's lowly 51 instructions, the SPU's 201 would seem to indicate that the set of operations available on Cell would subsume those available in PTX, not so. PTX's typing system allows the same instructions to specify different operations based on the types of the operations supplied, whereas SPU includes a distinct instruction for each operand type, making PTX the more comprehensive ISA. In particular, SPU does not implement predication, parallel primitives (vote, reduction, barrier), high level math operations (sin, cos, atan, exp), or atomic accesses to global or shared memory[1]. These instructions require extra attention by the translator to implement an equivalent sequence of SPU instructions.

Predication. Predication is used in PTX to express control flow in a more compatible form to SIMD operations – depending on the value of a specified predicate register, the destination register of a predicated instruction is conditionally updated. No concept of predication exists in SPU. However, SPU does include a conditional select instruction that picks bits from one of two possible registers depending on the values of a third register. Implementing predication using these instructions involves first generating a temporary select mask whenever a PTX predicate is updated and storing it in an SPU register. When a predicated PTX instruction is encountered, a temporary SSA SPU register is created to hold the result. The conditional select SPU instruction is used to update the result of the PTX predicated instruction with either the result of the instruction stored in the temporary register or the original value of the register. Rather than introducing dedicated registers for each PTX predicate, SPU registers are only allocated for the live ranges of PTX predicates. Note that this process adds an additional instruction for each PTX predicated instruction. If active masks for warps are implemented using a similar method, full warps will achieve a peak IPC of two rather than four, warps with two active threads will break even with a serial implementation, and warps with a single thread will operate at half speed. This observations suggests that for Cell it is more beneficial to create disjoint register sets when warps are split so that the entire set can be updated while the warp executes without needed conditional select instructions for instructions that are not explicitly predicated.

Parallel Operations. Parallel primitives might seem like they must be handled by traps into the runtime since they share data across threads within a CTA, and this is true for atomic and reduction operations. However, Ocelot handles barrier and vote by inserting translated code instead. For barrier, this is possible because the instruction implies a synchronization point and is handled trivially via the formation of thread compressing loops. For vote, which is an any or all operation across predicates of threads within a warp, Ocelot can insert instructions that scan the complete value of the indicated predicate register masked by the warp's active mask and immediately determine a value. Atomic and reduction instructions require read-modify-write style operations to global and shared memory. These can be handled via locks available to SPU

[1]As atomic accesses are a new feature of PTX, they are not supported in the current implementation of Ocelot

```
Givens __device__ CGivens(complex f, complex g) {
    complex c, s;

    if (g.x == 0 && g.y == 0) {
        c.x = 1;
        c.y = 0;
        s.x = 0;
        s.y = 0;
    }
    else if (f.x == 0 && f.y == 0) {
        c.x = c.y = 0;
        s = sign(g);
    }
    else {
        float f2 = f.x * f.x + f.y * f.y;
        float g2 = g.x * g.x + g.y * g.y;
        float inv_sq_f2_g2 = 1.0f/sqrt(f2 + g2);
        float inv_f2 = 1.0f/sqrt(f2);

        c.x = sqrt(f2) * inv_sq_f2_g2;
        c.y = 0;

        f.x *= inv_f2;
        f.y *= inv_f2;

        s = cjmul(f, g);
        s.x *= inv_sq_f2_g2;
        s.y *= inv_sq_f2_g2;
    }

    return make_float4(c.x, c.y, s.x, s.y);
}
```

Fig. 4. CGivens CUDA Source

```
    .entry _Z9k_CGivensP6float2S0_
    {
    .reg .u32 $r1,$r2,$r3,$r4,$r5,$r6,$r7,$r8,$r9,
        $r10,$r11,$r12,$r13,$r14;
    .reg .f32 $f1,$f2,$f3,$f4,$f5,$f6,$f7,$f8,$f9,
        $f10,$f11,$f12,$f13,$f14,$f15,$f16,$f17,$f18,$f19,
        $f20,$f21,$f22,$f23,$f24,$f25,$f26,$f27,$f28;
    .reg .pred $p0,$p1,$p2;
    .param .u32 __cudaparm__Z9k_CGivensP6float2S0__A;
    .param .u32 __cudaparm__Z9k_CGivensP6float2S0__G;
    .loc    13  100 0
$LBB1__Z9k_CGivensP6float2S0_:
__cudaparm__Z9k_CGivensP6float2S0__A+0x0
    ld.global.v2.f32    {$f1,$f2}, [$r1+0]; //
    ld.global.f32   $f3, [$r1+8];   //  id:213
    ld.global.f32   $f4, [$r1+12];  //  id:214
    mov.f32     $f5, 0f00000000;        //  0
    set.eq.u32.f32  $r2, $f3, $f5;  //
    neg.s32     $r3, $r2;               //
    mov.f32     $f6, 0f00000000;        //  0
    set.eq.u32.f32  $r4, $f4, $f6;  //
    neg.s32     $r5, $r4;               //
    and.b32     $r6, $r3, $r5;          //
    mov.s32     $r7, 0;                 //
    setp.eq.s32     $p1, $r6, $r7;      //
    @$p1 bra    $Lt_0_16;               //
    mov.f32     $f7, 0f00000000;        //  0
    mov.f32     $f8, 0f00000000;        //  0
    mov.f32     $f9, 0f3f800000;        //  1
    bra.uni     $Lt 0 15:               //
```

Fig. 5. CGivens PTX Source

programs, requiring traps into the runtime to execute.

V. A Case Study : CGivens

Translating CGivens. As a practical example of the translation approach used in Ocelot, we captured the intermediate representation generated after each stage in the translation process of a simple application as shown in figure 3. The program is specified in CUDA augmented C and compiled using the NVIDIA compiler NVCC V0.2.1221, which generates a corresponding PTX assembly file. Before execution, the PTX assembly file is loaded by Ocelot which creates a control flow graph (CFG) of basic blocks in the program. The freshly loaded control flow graph begins in the partial SSA form used by PTX. Ocelot immediately converts the CFG to complete SSA form, which eases optimization at the PTX level. A dominator tree is computed using reverse iterative dataflow which allows Ocelot to determine the boundaries for thread compressing loops. The resulting CFG is then translated to an equivalent CFG of SPU instructions and runtime traps. Again the CFG is verified to be in complete SSA form and optimizations are selectively applied. After the optimization phase, data flow analysis is used to determine live ranges of SSA register values. Finally, Ocelot performs a register allocation operation to map SSA registers to SPU equivalents. Ocelot outputs an assembly file of SPU instructions which is subsequently passed to the SPU assembler to generate an SPU object file. This is finally linked against the statically compiled Ocelot runtime and driver application, generating an invokable binary.

A. CUDA Source

Compilation. The application chosen for this evaluation was the simplest non-trivial application in our library of CUDA applications so that operations at the assembly level would simple enough to grasp by the reader. This application (CGivens) computes the Givens rotation matrix, which is a common operation used in parallel implementations of linear algebra operations like QR decomposition. The inner kernel of the application is shown in Figure 4. Note that this operation is completely data-parallel, where each PTX thread computes a different matrix in parallel. The code provides divergent control flow paths, but there are no explicit synchronization points.

B. PTX Assembly

PTX IR. The CUDA compiler is used to generate the PTX representation of CGivens, of which the first basic block is shown in Figure 5. Note the explicit typing of each SSA register, the reliance on predication to perform conditional branching, the memory space qualifiers for load and store operations, and the .uni suffix on unconditional branches. Ocelot begins by parsing a program in this form and generating a control flow graph of PTX instructions.

C. Control Flow Graph

CFG. Figure 6 provides a snapshot of a node in CGivens before and after conversion to complete SSA form. Note that Φ functions are only generated for live registers using partial data-flow analysis as in [44] creating an efficient pruned SSA representation. Though this example does not perform any, Ocelot provides the infrastructure to selectively apply optimizations at this stage.

Dominator Tree. Figure 7 shows the dominator tree of the CGivens application. In this case the lack of synchronization instructions precludes the need for explicit thread loops to enable thread compression. Thread compression in this case

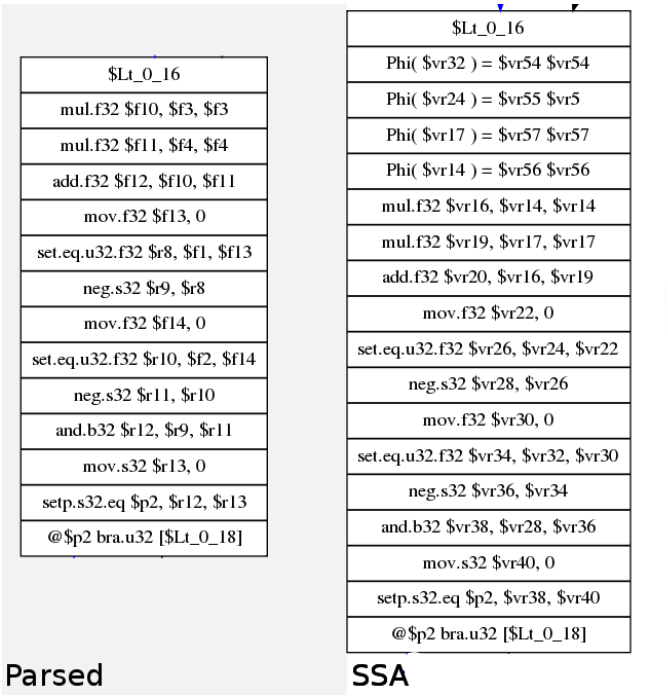

Fig. 6. SSA Form

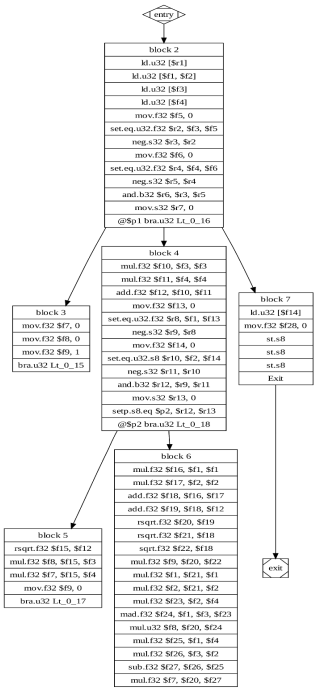

Fig. 7. CGivens Dominator Tree

would degenerate into a single loop around the entire application. As mentioned in section III-D, a more complex analysis could potentially reduce the amount of time that warps remain divergent by placing additional synchronization instructions at the post dominators of potentially divergent branches – in the figure, the first divergent branch occurs at the end of block 2 which is post dominated by block 7. However, this implementation of Ocelot does not include that optimization.

D. Translation

PTX to SPU. The translation process is then performed on the optimized PTX control flow graph. In this implementation, PTX instructions are examined one at a time and converted to equivalent sequences of SPU instructions. Figure 8 compares the translation of two different types of memory accesses. Parameter memory is mapped to structures in the SPU local store, and is therefore replaced by a simple load from the local store. Global memory is cached by the runtime, so a load from global memory is replaced by a function call to the runtime. Note that calls to the runtime must abide by the semantics of the SPU application binary interface [45], which specifies which registers can be used for passing arguments during function calls. At this point, the translator is dealing with virtual registers in SSA form, so a mapping must be created from any registers used as arguments to function calls to their corresponding physical registers. These registers are referred to as preallocated, and are given preference during the final register allocation phase.

Predicates. Another interesting example of translation occurs when predicates are used in PTX. Figure 9 shows a PTX instruction that updates a predicate and a series of SPU instructions that emulate the behavior. A new SSA register is created to temporarily store the predicate value, which is eventually consumed by a conditional branch in SPU, rather than a predicated branch in PTX. Note that this implementation of Ocelot has many opportunities for optimization in the translation process. An obvious example is in the extra add with zero instruction used to move the predicate after it was generated. The temporary register is needed in case the result of the comparison operation in the PTX SETP instruction is applied to a boolean operation to generate the final result. If, as in this case, it is not applied to a boolean function, it is copied directly to the result. It would be possible to detect this case in the translator and eliminate the redundant copy instruction by saving the output of the comparison directly to the result. There are likely to be many such opportunities for optimizations in the current implementation of Ocelot.

E. Data and Control Flow Analysis

Restoring SSA. After the translation process, the program is stored in a CFG of SPU instructions. Additional variables introduced in the translation process require an additional pass to convert the CFG to complete SSA form. Once in SSA form again, further optimizations can be easily applied to the SPU version of the program. Our current implementation does not support include any optimizations other than dead code elimination at this stage. However, this example does not have any dead instructions so the code is not changed during the optimization phase.

```
# ld.u32.param {$vr48} [$vr46 + 0]
  ilhu $82, cta_mem_param_ptr@h
  iohl $82, cta_mem_param_ptr@l
  lqd $80, 0($82)

# ld.f32.global {$vr69, $vr72} [$vr48 + 0]
  ai $3, $80, 4
  brasl $lr, cta_load_word
  ori $83, $3, 0
```

Fig. 8. Translated Memory Operations

```
# setp.s32.eq P1, $vr66, $vr68
  ceq $82, $80, $87
  a $80, $81, $82

# @$p1 bra.u32 [$Lt_0_16]
  brnz $80, .L3
```

Fig. 9. Translated Predicate Instructions

Register Live Ranges. After optimizations have been performed, Ocelot iteratively evaluates a set of data flow equations as in [46] to determine the live ranges of registers. For each basic block, Ocelot creates a function that consecutively applies the definitions and uses of each register. Performing this function in reverse for each basic block, adding a register to the current live set upon it's first use and removing it after encountering it's definition, effectively computes register live ranges for an individual block. Registers that are still alive after traversing the entire block are propagated to the live set of the block's predecessors, whose live sets are scheduled to be recomputed. This process is repeated until the live sets of all blocks stabilize, and overlapping live ranges for each SSA register can be determined.

F. Register Allocation and Code Generation

Register Allocation. In the last phase of translating the CGivens application, the register live ranges computed in the previous section along with the set of preallocated registers from the translation stage are used to perform register allocation. The Cell ABI reserves 47 callee saved registers and 71 caller saved registers out of the 128 SPU registers for use by SPU applications. The Ocelot register allocator chooses callee saved registers first, falling back to caller saved registers, and finally spilling excess registers to the stack. The allocator implements the well known Chattain-Briggs algorithm which expresses the allocation process as a graph coloring problem. Note that the allocator is able to color the graph using 9 distinct registers, enabling between 11 and 44 separate contexts to be stored in the register file at any time depending on the number of divergent warps.

Code Generation. Once the register allocation stage has finished, the application is passed through a code generation stage in Ocelot that either creates a binary object file or outputs an assembly text file that can be assembled using the Cell Toolchain. A section of the resulting assembly file is shown in Figures 8 and 9. Observe that the corresponding PTX instructions are inlined as comments to ease debugging the translation process.

Correctness. In order to test the validity of out translator, we link the generated SPU assembly file with the previously compiled Ocelot runtime which handles loading the application and running it on the SPUs in Cell. We run the complete application for a number of input parameters, generating distinct Givens rotation matrices. We then compare these results to the equivalent application generated with the CUDA toolchain run on an NVIDIA GPU. In terms of functionality for this application, both the NVIDIA toolchain and Ocelot produce identical results.

VI. Related Work

A. GPGPU Programming

CUDA. Numerous examples of applications ported to the CUDA programming model provide the motivation for seamlessly migrating CUDA applications to other many-core architectures using Ocelot. Sungupta et. al., Govindaraju et. al., and Krüger et. al. provide basic building blocks of many algorithms with efficient GPU implementations of Scan (prefix sum) [47], database primitives [48], and linear algebra operations [49] respectively. Cederman and Tsigas provide an efficient implementation of quicksort using CUDA, able to sort 16 million floating point numbers in under half a second on an NVIDIA 8800GTX [50]. Higher level applications have also been shown to have efficient explicitly parallel CUDA implementations [12]. Ocelot is able to leverage all of these implementations for architectures other than GPUs.

B. CUDA Compilation

MCUDA and Larabee. Ocelot builds on the emerging body of knowledge concerning compiling explicitly parallel programs (like CUDA) to architectures other than GPUs. In the CUDA toolchain, NVIDIA includes an emulator that compiles CUDA applications directly to native code without first generating a PTX representation [4]. For each PTX thread, a corresponding native thread is created, relying on mutex primitives for implementing synchronization points. As the number of threads in a PTX application grows, the overheads associated with this approach grows to an unacceptable level as most native threading libraries were not designed to handle several thousand or more threads. Stratton et. al. offer an alternative approach at the compiler level by introducing a source to source translator that replaces CUDA API calls with native equivalents and inlines thread compressing loops at CUDA synchronization function boundaries. The result of the translation is ANSI C source than can be compiled with a native compiler. SIMD level parallelism is extracted by the compiler which makes use of OpenMP style pragmas around thread compressing loops. Similarly, the programing model used by Intel Larabee Renderer [33] uses the concepts of lightweight *fibers* and batch processing rather than many

hardware threads to perform parallel rendering on an array of x86 cores.

Ocelot. Ocelot differs from these approaches which operate above the compiler level by utilizing binary translation. Rather than requiring code to be recompiled from source when targeting a new architecture, existing applications can be translated as their PTX source is loaded. Ocelot does have two distinct advantages over these approaches that rely on static compilation, (1) the dimensions of the PTX thread hierarchy are determined at runtime, allowing a translator to change the partitioning of threads among warps, thread compressing loops, software threads, and hardware threads before a PTX application is launched, and (2) runtime information about the execution of an application can be used to optimize hot paths [51], target more efficient architectures, or reallocate runtime resources.

C. Binary Translation

Examples. Binary translation has been successfully applied to a variety of problems including optimizing Java byte-code [38] or x86 [35], running x86 on RISC [36] and VLIW processors [21], [39], instrumenting applications [37], and even by NVIDIA for ensuring compatibility across generations of GPUs [52].

Liquid SIMD. Perhaps the most similar example to Ocelot is Clark's liquid SIMD approach to migrating SIMD operations to future architectures by expressing them in terms of scalar operations that can be converted back to SIMD operations by a binary translator. Again, thread compressing loops are used to express SIMD instructions as a sequence of smaller scalar operations. These loops are expressed in an implicitly parallel form that can be easily recognized by a compiler and converted back to SIMD. This approach differs from the representation of SIMD operations in PTX as it assumes that ways in a SIMD set are never divergent, and thus cannot support the arbitrary control flow expressible in PTX.

The Liquid SIMD approach focuses on being able to convert back to SIMD operations from scalar equivalents. Ocelot leverages the explicit specification of parallelism in PTX to avoid the need for this operation: if a program has been compressed down to a small number of threads, the original PTX can be referred to again when targeting an architecture supporting a wider SIMD width.

VII. Conclusions

Summary. This paper proposed the use of binary translation to enable flexible and efficient migration of CUDA applications to platforms other than NVIDIA GPUs, leveraging the explicit data parallelism painstakingly extracted by the programmer and compiler. Ocelot provides a complete methodology for translating CUDA applications to parallel architectures that may not be equipped with the same hardware capabilities as GPUs. Ocelot presents translation techniques enabling: (i) software warp resizing, (ii) latency hiding via software context switches, and (iii) runtime overhead reduction via thread compression. These new techniques are used as part of a complete translation framework incorporating both novel and well known operations to translate PTX applications to the IBM Cell Processor.

We examined the considerations needed to map GPU specific data structures to the Cell Processor, then showed how the thread hierarchy in PTX could be mapped to the processing elements available in Cell. Our walk through of the complete translation of the CGivens application provided insight into the low level considerations required to implement the previously described techniques.

Future Work. Ocelot, in its current form, provides a useful set of tools for analyzing PTX applications and running CUDA programs on Cell. Eventually, we would like to explore the integration of Ocelot with other ISAs like LLVM, enabling translation to a variety of many-core targets other than Cell and GPUs. Runtime dynamic translation in particular could enable a system containing multiple Ocelot targets to selectively translate PTX functions to different architectures depending on their predicted efficiency, or fall back on a generic many-core processor if no other targets were present. Of course, runtime platforms for heterogeneous systems like Harmony [53], Merge [54], or Seqouia [55] could benefit from the ability to dynamically translate PTX applications rather than being forced to rely on previously compiled libraries.

Acknowledgement

We gratefully acknowledge Nate Clark for giving us the opportunity to work on this project and providing valuable feedback and suggestions, and NVIDIA for providing equipment grants, the CUDA toolchain, and comprehensive documentation on PTX.

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1997, pp. 206–218.

[2] C.-L. Su and A. M. Despain, "Cache design trade-offs for power and performance optimization: a case study," in *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*. New York, NY, USA: ACM, 1995, pp. 63–68.

[3] W. mei Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Implicitly parallel programming models for thousand-core microprocessors," in *DAC '07: Proceedings of the 44th annual conference on Design automation*. New York, NY, USA: ACM, 2007, pp. 754–759.

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

[5] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 703–712, 2002.

[6] D. Campbell, "Vsipl++ acceleration using commodity graphics processors," in *HPCMP-UGC '06: Proceedings of the HPCMP Users Group Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 315–320.

[7] K. Moreland and E. Angel, "The fft on a gpu," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 112–119.

[8] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision," in *MM '08: Proceeding of the 16th ACM international conference on Multimedia*. New York, NY, USA: ACM, 2008, pp. 1089–1092.

[9] I. Buck, "Gpu computing with nvidia cuda," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, USA: ACM, 2007, p. 6.

[10] J. Yang, Y. Wang, and Y. Chen, "Gpu accelerated molecular dynamics simulation of thermal conductivities," *J. Comput. Phys.*, vol. 221, no. 2, pp. 799–804, 2007.

[11] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *J. Comput. Phys.*, vol. 227, no. 18, pp. 8290–8313, 2008.

[12] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande, "N-body simulation on gpus," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 188.

[13] B. Han and B. Zhou, "Efficient video decoding on gpus by point based rendering," in *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. New York, NY, USA: ACM, 2006, pp. 79–86.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 511–524.

[15] IEEE, "Ieee std. 1003.1c-1995 thread extensions . ieee 1995, isbn 1-55937-375-x formerly posix.4a. now included in 1003.1-1996."

[16] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[17] Message, "MPI-2: Extensions to the Message-Passing Interface," University of Tennessee, Knoxville, TN, USA, Tech. Rep., July 1997.

[18] 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

[19] R. Colwell and R. Steck, "A 0.6-m bicmos microprocessor with dynamic execution," in *IEEE International Solid-State Circuits Conference*, 1995, pp. 176–177.

[20] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[21] A. Klaiber, "The technology behind crusoe processors," Santa Clara, CA, USA, Tech. Rep., 2000.

[22] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[23] B. C. Pierce, *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0262162288

[24] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[25] M. Hind and A. Pioli, "Which pointer analysis should i use?" in *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2000, pp. 113–123.

[26] C. Lattner and V. Adve, "Data structure analysis: A fast and scalable context-sensitive heap analysis," Tech. Rep., 2003.

[27] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1999, pp. 13–24.

[28] C. S. Ellis and T. J. Olson, "Algorithms for parallel memory allocation," *Int. J. Parallel Program.*, vol. 17, no. 4, pp. 303–345, 1988.

[29] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1990, pp. 180–189.

[30] 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.

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

[32] T. Johnson and U. Nawathe, "An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2)," in *ISPD '07: Proceedings of the 2007 international symposium on Physical design*. New York, NY, USA: ACM, 2007, pp. 2–2.

[33] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[34] W. W. L. Fung, "Dynamic warp formation : exploiting thread scheduling for efficient mimd control flow on simd graphics hardware," Master's thesis, University of British Columbia, 2329 West Mall Vancouver, BC Canada V6T 1Z4, September 2008.

[35] 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.

[36] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "Fx!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[38] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen, "Implementing jalape no in java," in *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1999, pp. 314–324.

[39] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 529–548, 2001.

[40] V. G. and, "Unkown," Georgia Institute of Technology, Tech. Rep., October 2008.

[41] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[42] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker, "Characterizing and modeling the behavior of context switch misses," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 91–101.

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

[44] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[45] I. Systems and T. Group, *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*, 1st ed., IBM Corporation, Hopewell Junction, NY, August 2007.

[46] G. A. Kildall, "A unified approach to global program optimization," in *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1973, pp. 194–206.

[47] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 97–106.

[48] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international*

conference on Management of data. New York, NY, USA: ACM, 2004, pp. 215–226.

[49] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003, pp. 908–916.

[50] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in *Proceedings of the 16th Annual European Symposium on Algorithms*, vol. 5193. Springer-Verlag, 2008, pp. 246–258.

[51] R. Cohn and P. G. Lowney, "Hot cold optimization of large windows/nt applications," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 80–89.

[52] NVIDIA, *NVIDIA Compute PTX: Parallel Thread Execution*, 1st ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[53] 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.

[54] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 287–296.

[55] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.