A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot

Naila Farooqui[1], Andrew Kerr[2], Gregory Diamos[3], S. Yalamanchili[4], and K. Schwan[5]
College of Computing[1,5], School of Electrical and Computer Engineering[2,3,4]
Georgia Institute of Technology
Atlanta, GA
{naila[1], schwan[5]}@cc.gatech.edu, {arkerr[2], gregory.diamos[3], sudha[4]}@gatech.edu

ABSTRACT

In this paper we present the design and implementation of a dynamic instrumentation infrastructure for PTX programs that procedurally transforms kernels and manages related data structures. We show how performing instrumentation within the GPU Ocelot dynamic compiler infrastructure provides unique capabilities not available to other profiling and instrumentation toolchains for GPU computing. We demonstrate the utility of this instrumentation capability with three example scenarios - **(1)** performing workload characterization accelerated by a GPU, **(2)** providing load imbalance information for use by a resource allocator, and **(3)** providing compute utilization feedback to be used online by a simulated process scheduler that might be found in a hypervisor. Additionally, we measure both **(1)** the compilation overheads of performing dynamic compilation and **(2)** the increases in runtimes when executing instrumented kernels. On average, compilation overheads due to instrumentation consisted of 69% of the time needed to parse a kernel module, in the case of the Parboil benchmark suite. Slowdowns for instrumenting each basic block ranged from 1.5x to 5.5x, with the largest slowdowns attributed to kernels with large numbers of short, compute-bound blocks.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Heterogeneous (hybrid) systems; D.3.4 [**Programming Languages**]: Retargetable compilers; D.3.4 [**Programming Languages**]: Run-time environments; D.4.8 [**Operating Systems**]: Measurements

General Terms

GPU Computing, Instrumentation, Dynamic Binary Compilation

Keywords

CUDA, OpenCL, Ocelot, GPGPU, PTX, Parboil, Rodinia

1. INTRODUCTION

Dynamic binary instrumentation is a technique in which application binaries are modified by an instrumentation tool to insert additional procedures into the existing execution path. Such instrumentation provides access to the run-time state of the application and enables sophisticated actions to take place. For example, code can be inserted for correctness checks such as memory bounds checking, or to improve performance such as insertion of pre-fetch instructions. Instrumentation takes place at the instruction level and offers inspection opportunities not exposed at a higher level such as source-level assertions. Dynamic compilation frameworks offer the additional capability of adding and removing instrumentation at runtime to avoid performance costs unless instrumentation is needed.

Toolchains such as OpenCL [1] and CUDA [2] have tremendously enhanced the state of the art for developing high-performance applications targeting GPU architectures. Both platforms provide a C-like language for expressing data-parallel kernels and an API for launching them on GPU accelerators as well as managing associated resources such as textures and device memory. However, GPUs are typically designed with vendor-specific instruction set architectures, and the toolchains do not facilitate direct inspection and modification of native application binaries. Consequently, developers of GPU compute applications are deprived much of the flexibility and power that dynamic binary instrumentation has brought to traditional architectures.

In this paper, we discuss techniques for dynamically instrumenting CUDA kernels at the PTX level [3] and present enhancements to a toolchain - GPU Ocelot - for *transparently* instrumenting *unmodified* CUDA applications. Specifically, we discuss the use of GPU Ocelot's [4] pass manager for applying PTX-to-PTX transformations to loaded modules before they are dispatched to the GPU driver or other backends. We define an instrumentation model and expose APIs to enable construction of user-defined, custom instrumentation tools. Additionally, we present a framework within Ocelot's implementation of the CUDA Runtime API for managing data structures associated with instrumentation. Our tool inserts instrumentation via a transformation pipeline which exists upstream of each of Ocelot's supported processor backends. Consequently, procedural instrumentation may be inserted and utilized transparently for each of the Ocelot backends: NVIDIA GPUs, x86 multicore CPUs, PTX emulation, and (under development) AMD Radeon GPUs. The details of translation to multicore x86 are described in [5], and translation to AMD GPUs are described by Dominguez, et al. in [6]. We also anticipate the possibility of an OpenCL API front-end to Ocelot that would extend the reach of the toolchain described in this paper to OpenCL applications.

GPGPU-4 Mar 05-05 2011, Newport Beach, CA, USA

We demonstrate Ocelot's instrumentation capability with several example use cases revealing application behaviors that would take up to 1000x longer if run on Ocelot's PTX emulator. To the best of our knowledge, this is the first implementation of a dynamic transformation and instrumentation tool for PTX.

This paper provides a background of GPU computing and of GPU Ocelot in Section 2. In Section 3, we discuss the design of GPU Ocelot's PTX-to-PTX pass manager, facilities for modifying kernels at the PTX level, and the hooks into Ocelot's CUDA Runtime API implementation. The PTX-to-PTX pass manager is not new to GPU Ocelot, but integrating it as part of a transformation pipeline upstream of each device backend is a new capability. Enhancements to Ocelot's API for managing instrumentation procedures and extracting results is also a novel capability not discussed in previous works. In Section 4, we describe several metrics and gather results from applications from the CUDA SDK and Parboil Benchmark suite. We show how a process scheduler might use this information to enforce a fairness target, and we characterize overheads of dynamic compilation and instrumentation.

2. GPU COMPUTING

NVIDIA's CUDA [2] toolchain is a programming language and API that enables data-parallel kernels to be written in a language with C++-like semantics. Computations are performed by a tiered hierarchy of threads. At the lowest level, collections of threads are mapped to a single *stream multiprocessor* or SM and executed concurrently. Each SM includes an L1 data cache, a shared scratch-pad memory for exchanging data between threads, and a SIMD array of functional units. This collection of threads is known as a *cooperative thread array* (CTA), and kernels are typically launched with tens or hundreds of CTAs which are oversubscribed to the set of available SMs. A work scheduler on the GPU maps CTAs onto individual SMs for execution, and the programming model forbids global synchronization between SMs except on kernel boundaries.

2.1 GPU Ocelot

GPU Ocelot [4] is a dynamic compilation and binary translation infrastructure for CUDA that implements the CUDA Runtime API and executes PTX kernels on several types of backend execution targets. Ocelot includes a functional simulator for offline workload characterization, profiling, and correctness checking. A translator from PTX to LLVM provides efficient execution of PTX kernels on multicore CPU devices with the addition of a runtime execution manager. To support Ocelot's NVIDIA GPU device, PTX kernels are emitted and invoked via the CUDA Driver API. Ocelot has the unique capability of inspecting the state of the application as it is running, transforming PTX kernels before they are executed natively on available GPU devices, and managing additional resources and data structures needed to support instrumentation.

Figure 1 illustrates Ocelot's relationship with CUDA applications. Ocelot replaces the CUDA Runtime API library that CUDA applications link with. API calls to CUDA pass through Ocelot providing a layer of compilation support, resource management, and execution. Ocelot may modify CUDA kernels as they are registered and launched as well as insert additional state and functionality into the host application. Consequently, it is uniquely positioned to *transparently* instrument applications and respond to data-dependent application behaviors which would not be possible with static transformation techniques. For example, Ocelot may implement random sampling by inserting instrumentation into a kernel as an application is running, profiling for a brief period, then re-issuing the original kernel without instrumentation. A more sophisticated approach might use the results of profiling to perform optimizations of the kernel, although the application of profile-directed optimizations is beyond the scope of this work.

2.2 PTX Instruction Set and Internal Representation

Parallel Thread eXecution, or PTX, is the RISC-like virtual instruction set targeted by NVIDIA's CUDA and OpenCL compilers and used as an intermediate representation for GPU kernels. PTX consists of standard arithmetic instructions for integer and floating-point arithmetic, load and store instructions to explicitly denoted address spaces, texture sampling and graphics related instructions, and branch instructions. Additionally, special instructions for interacting with other threads within the CTA are provided such as CTA-wide barrier instructions, warp-wide vote instructions, and reduction operations, to name several. Implementing the so-called Single-Instruction, Multiple-Thread (SIMT) execution model, PTX specifies the execution of a scalar thread and the hardware executes many threads concurrently. PTX is decoupled from actual hardware instantiations and includes an abstract Application Binary Interface (ABI) for calling functions and managing a local parameter memory space while leaving the actual calling convention semantics to the native ISA implementation.

CUDA sources compiled by `nvcc`, NVIDIA's CUDA compiler, become C++ programs with calls to register PTX kernels to the active CUDA context via the CUDA Runtime API. The kernels themselves are stored as string literals within the program, and kernel names are bound to objects the host program may then refer to when configuring and launching kernel grids. Ocelot implements the CUDA Runtime API's `__cudaRegisterFatBinary()` and parses PTX kernels into an internal representation. Control-flow analysis partitions instructions into basic blocks connected by edges indicating control dependencies. An optional data-flow analysis pass transforms the kernel into *static single-assignment* form [7] which enables register re-allocation. Ocelot's PTX internal representation (PTX IR) covers the entire PTX 1.4 ISA [1] specification and provides a convenient programming interface for analyzing kernels and adding new instructions.

3. DESIGN AND IMPLEMENTATION

In this section, we discuss the specific enhancements made to Ocelot to add externally-defined instrumentation procedures, apply them to PTX modules, and extract profiling information during application execution.

Ocelot defines an interface for implementing PTX instrumentation tools and provides an externally visible API for attaching instrumentation passes to Ocelot before and during the execution of GPU compute applications. As described in previous work [8], Ocelot replaces NVIDIA's CUDA Runtime API library (`libcudart` on Mac and Linux, `cudart.dll` on Windows) during the link step when CUDA applications are compiled. To insert third party instrumen-

[1]Ocelot 2.0.969 supports PTX 2.1 (Fermi) and has become the main trunk as of January 31, 2011.

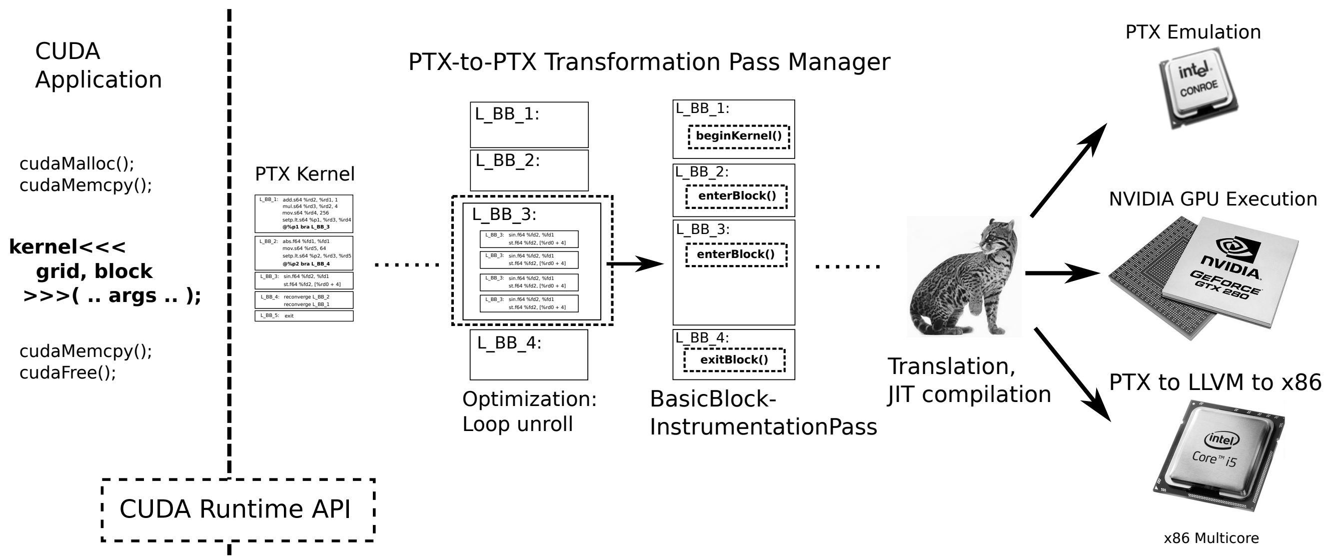


Figure 1: Overview of the GPU Ocelot dynamic compilation infrastructure.

tation procedures, applications can be modified to explicitly add and remove instrumentors between kernel launches of the program via Ocelot's `add` and `remove` APIs. Alternatively, instrumentation tools built as an additional library and linked with the application may add themselves when the library is initialized. This approach means application sources do not need to be modified or recompiled.

The instrumentation tools themselves are C++ classes that consist of two logical components: (1) an instrumentor class derived from the abstract base class `PTXInstrumentor`, and (2) an instrumentation pass class derived from Ocelot's `Pass` abstract class. A class diagram in Figure 2 illustrates an example instrumentation tool, `BasicBlockInstrumentor`, and presents the class structure relationship graphically. The instrumentor is responsible for performing any static analysis necessary for the instrumentation, constructing instrumentation-related data structures, instantiating a PTX transformation pass, extracting instrumentation results, and cleaning up resources. The PTX pass applies transformations to PTX modules which are presented to it via Ocelot's PTX Internal Representation (IR).

3.1 Instrumentation and PTX Passes

The host application calling the CUDA Runtime API is responsible for registering PTX modules which are then loaded on the selected CUDA device. Under the Runtime API layer, within Ocelot, modules are loaded by parsing and analyzing PTX, applying procedural transformations, then translating for execution on the target device. Registered `PTXInstrumentor` instances are applied during the procedural transformation step by invoking their `instrument()` method on each module.

PTX modules contain a set of global variables and functions and must be modified to include a global variable pointing to additional data structures to receive and output instrumentation results. This module-level change is performed by a `ModulePass` specific to each `PTXInstrumentor` instance. Subsequently, kernels within the module are instrumented by either a kernel pass or basic block pass.

Between the parse phase and the translation phase, a pass manager is positioned to apply a sequence of procedural transformations to the PTX kernels before they are loaded onto the selected device. This approach is largely inspired by LLVM's [9] pass manager framework in which transformations may be made at the module level, the function level, and the basic block level depending on the scope of the transformation. The manager orchestrates pass application to ensure both the sequential order of passes is preserved while taking advantage of locality of reference. Figure 3 illustrates this method of restricting pass scope and coalescing passes by type.

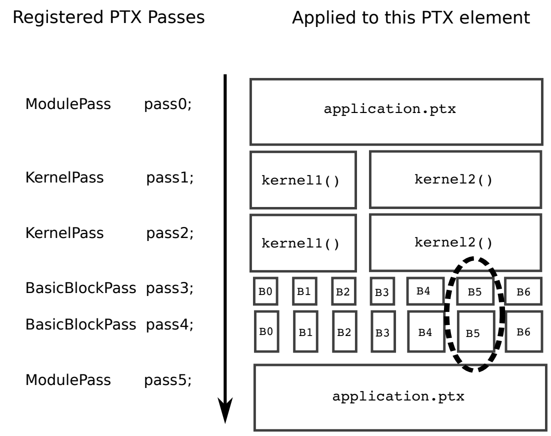


Figure 3: PTX Pass manager showing application of transformations to a PTX module. The circle indicates `BasicBlockPasses` `pass3` and `pass4` are each applied to basic block `B5` before moving on to other blocks.

Passes themselves are C++ classes that implement interfaces derived from Ocelot's `analysis::Pass` abstract class. These include `ImmutablePass` for performing static analysis consumed later during the transformation pipeline, `ModulePass` for performing module-level modifications, `KernelPass` for changing the control structure of the kernel, and `BasicBlockPass` for applying transformations within

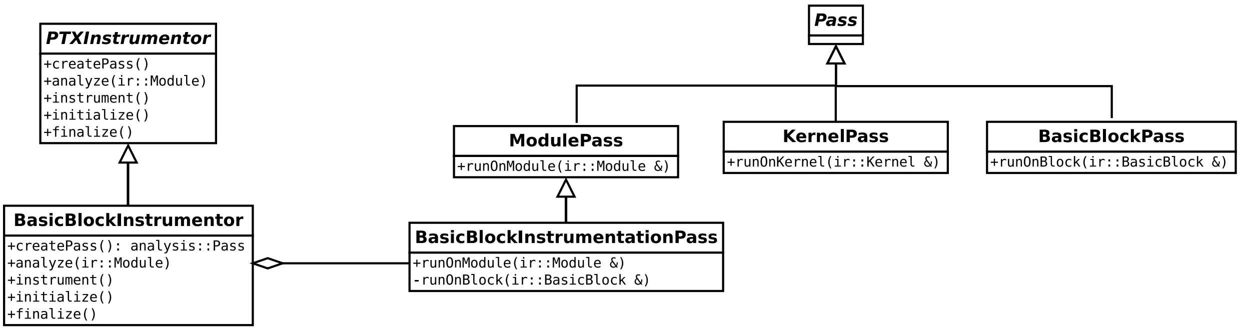


Figure 2: Class diagram for instrumentation passes.

the scope of one basic block at a time. PTX passes may access the kernel's control-flow graph, dominator tree, and data-flow graph. The data-flow graph is updated when new instructions are added that create new values, and the dominator trees are recomputed when kernel and module passes change the control-flow of kernels.

3.2 Instrumentor APIs and CUDA Runtime

Certain instrumentations may require inspection of the kernel's CFG to obtain necessary information required by the CUDA Runtime API to properly allocate resources on the device. In general, any actions that must be performed prior to allocating resources on the device, are encapsulated in the `analyze()` method. For our basic block execution count instrumentation, we obtain the CFG of each kernel to determine the total number of basic blocks.

Before launching a kernel, memory on the device must be allocted and initialized to store the instrumentation results. Ocelot calls each registered instrumentation pass's `initialize()` method which may allocate memory and transfer data to and from the selected device. After the kernel has been launched, each instrumentor's `finalize()` method is invoked to free up allocated resources and extract instrumentation results into an instance of `KernelProfile`. The `KernelProfile` class outputs results either to a file or database, or it may channel instrumentation results to other components or applications that link with Ocelot. External applications can access the `KernelProfile` instance via the `kernelProfile()` API within Ocelot.

3.3 Example Instrumentation Tools

The `ClockCycleCountInstrumentationPass` inserts instrumentation to read the clock cycle counter exposed by PTX's special register `%clock` which corresponds to a built-in hardware clock cycle counter. Instrumentation is inserted to the beginning of the kernel to record the starting clock cycle number and at the end, along with a barrier waiting for all threads to finish, to record the CTA's ending time. In addition to storing runtimes, the PTX register `%smid` is accessed to determine which streaming multiprocessor each CTA was mapped to.

The `BasicBlockInstrumentationPass` constructs a matrix of counters with one row per basic block in the executed kernel and one column per dynamic PTX thread. By assigning one basic block counter per thread, the instrumentation avoids contention to global memory that would be experienced if each thread performed atomic increments to the same block counter. Instrumentation code added via a `BasicBlockPass` loads a pointer to the counter matrix from a global variable. The instrumentation pass then adds PTX instructions to each basic block that compute that thread's counter index and increments the associated counter using non-atomic loads and stores. Counters of the same block for consecutive threads are arranged in consecutive order in global memory to ensure accesses are coalesced and guaranteed to hit the same L1 cache line. PTX instructions added to the beginning of each basic block appears in Listing 1. This code is annotated with pseudocode illustrating the purpose of each instruction sequence. At runtime, this instrumentation pass allocates the counter matrix sized according to the kernel's configured block size.

Listing 1: Instrumentation inserted into each basic block for `BasicBlockInstrumentationPass`.

```
// get pointer to counter matrix
mov.u64 %r28, __ocelot_basic_block_counter_base;
ld.global.u64 %r13, [%r28 + 0];
add.u64 %r13, %r13, %ctaoffset;

// idx = nthreads * blockId + threadid
mad.lo.u64 %r14, %nthreads, 5, %threadid;

// ptr = idx * sizeof(Counter) + base
mad.lo.u64 %r15, %r14, 8, %r13;

// *ptr ++;
ld.global.u64 %r12, [%r15 + 0];
add.u64 %r12, %r12, 1;
st.global.u64 [%r15 + 0], %r12;
```

The C++ `BasicBlockInstrumentationPass` class uses PTX IR factories to construct instrumentation code inserted into the kernel. The `BasicBlockInstrumentationPass` implements the `runOnBlock` method to add the relevant PTX at the beginning of every basic block. Listing 2 shows a code snippet of the corresponding C++ code in the `runOnBlock` method for creating the PTX to update the global basic block counter index. This code corresponds to the last 3 lines of the PTX shown in Listing 1.

Listing 2: Corresponding C++ code for inserting instrumentation into each basic block for `BasicBlockInstrumentationPass`.

```
PTXInstruction ld( PTXInstruction::Ld );
PTXInstruction add( PTXInstruction::Add );
PTXInstruction st( PTXInstruction::St );

ld.addressSpace = PTXInstruction::Global;
ld.a.addressMode = PTXOperand::Indirect;
ld.a.reg = registerMap["counterPtrReg"];
```

```
ld.d.reg = registerId;
ld.d.addressMode = PTXOperand::Register;

add.addressSpace = PTXInstruction::Global;
add.d = ld.d;
add.a = ld.d;
add.b.addressMode = PTXOperand::Immediate;
add.b.imm_int = 1;

st.addressSpace = PTXInstruction::Global;
st.d.addressMode = PTXOperand::Indirect;
st.d.reg = registerMap["counterPtrReg"];
st.a.addressMode = PTXOperand::Register;
st.a.reg = registerId;

// Inserts at the beginning of the basic block,
// as the 5th, 6th, and 7th statements
// (first statement indexed at 0).

kernel->dfg()->insert( block, ld, 4 );
kernel->dfg()->insert( block, add, 5 );
kernel->dfg()->insert( block, st, 6 );
```

4. EVALUATION

The above instrumentation passes were implemented as a branch of Ocelot version 1.1.560 [4]. To evaluate the usefulness and performance impact of these instrumentation passes, the following experiments were performed on a system with an Intel Core i7 running Ubuntu 10.04 x86-64 and equipped with an NVIDIA GeForce GTX480. Benchmark applications were chosen from the NVIDIA CUDA Software Development Kit [2] and the Parboil Benchmark Suite [10].

Experiment 1 - Hot Region Detection. To determine the most frequently executed basic blocks within the kernel, we use our `BasicBlockInstrumentationPass`. Figure 4 is a heat map visualizing the results from this experiment for the Scan application from the CUDA SDK. Basic blocks are colored in intensity in proportion to the number of threads that have entered them. The hottest region consists of blocks `BB_001_007`, `BB_001_008`, `BB_001_009` corresponding to this kernel's inner loop. This metric captures architecture-independent behavior specified by the application. A similar instruction trace analysis offered by Ocelot's PTX emulator provides the same information but at the cost of emulation. By instrumenting native PTX and executing the kernels natively on a Fermi-class NVIDIA GTX480 [11], a speedup of approximately 1000x over the emulator was achieved. This is an example of workload characterization accelerated by GPUs.

Experiment 2 - Overhead of Instrumentation. The basic block execution count instrumentation contributes a per-block overhead in terms of memory bandwidth and computation. Blocks in the hottest region make numerous accesses when incrementing their respective per-thread counters and displace some cache lines from the L1 and L2 caches. Clock cycle count instrumentation inserts instructions to read clock cycles at the beginning of the kernel and then to store the difference into a counter in global memory. All forms of instrumentation can be expected to perturb execution times in some way. This experiment measures runtimes of sample applications with and without each instrumentation pass. Slowdowns for selected applications from the CUDA SDK and Parboil appear in Figure 5. These applications cover a spectrum of structural properties related to basic block instrumentation. Properties include number of operations per basic block, number of kernels launched, and whether they are memory- or compute-bound.

All applications perform consistently well with clock cycle count instrumentation, achieving minimal slowdown (less

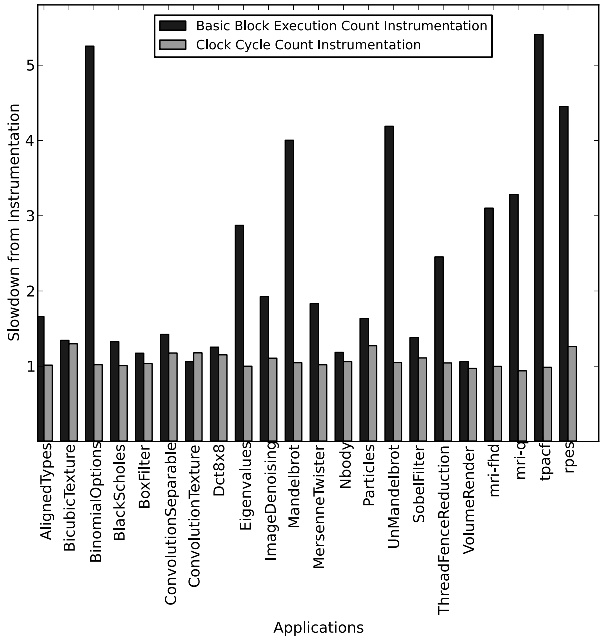


Figure 5: Slowdowns of selected applications due to `BasicBlockInstrumentor` and `ClockCycleCountInstrumentor`.

than 1.3x). Compute intensive applications with many instructions per basic block, such as BicubicTexture, BlackScholes, BoxFilter, and Nbody achieve the least slowdown from `BasicBlockInstrumentor`, as the costs of accessing memory are amortized or hidden entirely. Applications with a large number of short basic blocks, such as BinomialOptions, tpacf, and rpes, exhibit the largest slowdowns from `BasicBlockInstrumentor`. Other applications that exhibited a mixture of large and small basic blocks but still had either a significantly large number of total basic blocks per kernel or many kernel invocations, such as Mandelbrot, UnstructuredMandelbrot, EigenValues, and ThreadFenceReduction, had a slowdown between 2x and 4x.

In order to verify the cause of the largest slowdown in our experiments, BinomialOptions, we use our basic block instrumentation data to determine the most frequently executed basic blocks within the BinomialOptions kernel. An analysis of the associated PTX depicts that this kernel consists of many compute-intensive, short basic blocks with very few memory accesses (less than 11% of the total instructions, without instrumentation, are `ld` and/or `st` instructions). The basic block instrumentation contributes a large additional bandwidth demand as well as a significant fraction of dynamic instructions, resulting in a 5.5x slowdown for this application. As an optimization, a more sophisticated instrumentation pass could use registers for counter variables and ellide registers for blocks that could be fused.

Experiment 3 - CTA Load Imbalance. CUDA execution semantics specify coarse-grain parallelism in which cooperative thread arrays (CTAs) may execute concurrently but independently. By excluding synchronization across CTAs from the execution model, GPUs are free to schedule the execution of CTAs in any order and with any level of concurrency. Moreover, the programmer is encouraged to

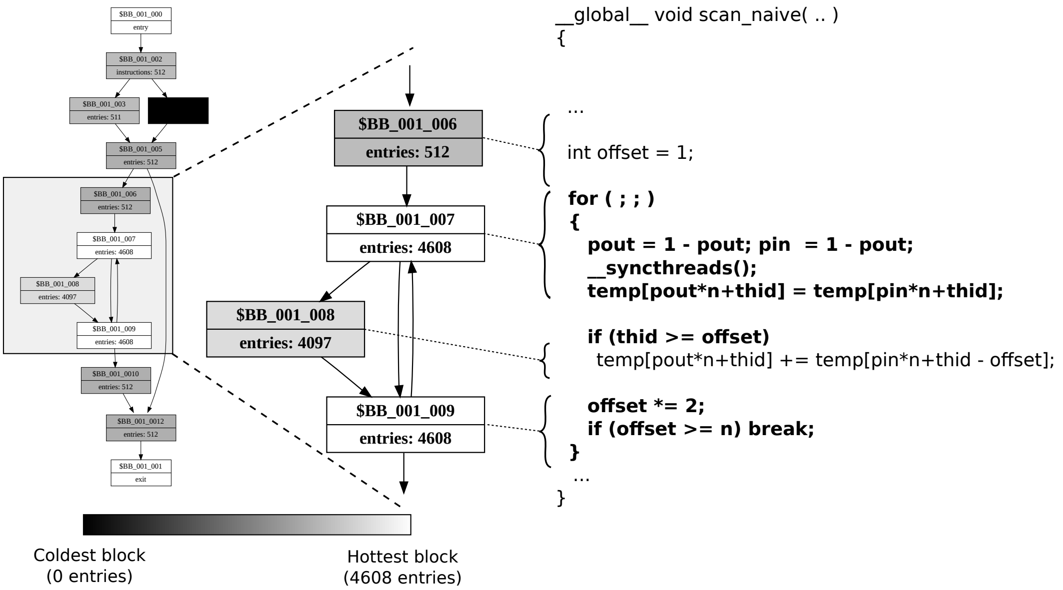


Figure 4: Hot region visualization of CUDA SDK Scan application profiled during native GPU execution. Each block presents a count of the number of times a thread entered the basic block and is color coded to indicate computational intensity. The magnified portion of the control-flow graph illustrates a loop, the dominant computation in the kernel.

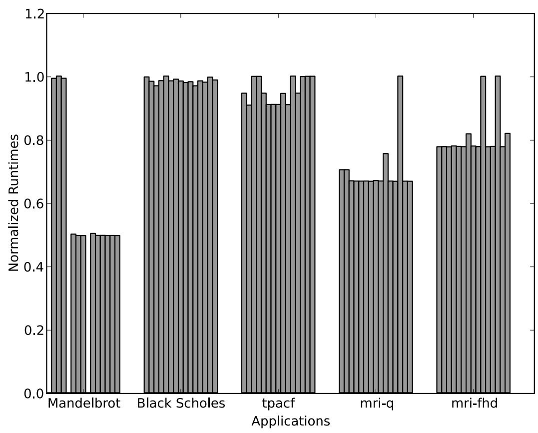


Figure 6: Normalized runtimes for each Streaming Multiprocessor for several workloads. Kernel runtime is the maximum number of cycles over all SMs, and some SMs are less heavily used than others.

specify the number of CTAs per kernel grid as a function of problem size without consideration of the target GPU architecture. This is one approach toward parallelism scalability, as the hardware is free to map CTAs to streaming multiprocessors (SMs) as they become available. An application's performance may scale as it is run on a low-end GPU with 2 SMs to a high-end GPU with 30 SMs. Unfortunately, this approach may also result in load imbalance as more work may be assigned to some CTAs than others.

`ClockCycleCountInstrumentor` records CTA runtimes and mapping from CTA to SM and determines whether the number of CTAs and corresponding workloads leave any SMs idle for extended periods or whether the workload is well-balanced. Figure 6 plots runtimes of selected applications for each SM normalized to total kernel runtime. The Mandelbrot application exhibits zero clock cycles for SMs 0, 4, and 8, implying they are idle, yet SMS 1, 2, and 3 have runtimes that are nearly twice as long as the other SMs. This implies a rather severe load imbalance in which nearly 80% of the GPU is unutilized for half of the kernel's execution. This level of feedback may hint to the programmer to reduce the amount of possible work per CTA to enable the hardware work scheduler to assign additional, shorter running CTAs to the unutilized SMs. The other applications exhibit a balanced workload with all SMs utilized for over 75% of the total kernel runtime.

Experiment 4 - Online Credit-based Process Scheduling. In this experiment, we demonstrate Ocelot's API for querying the results of profiling information gathered from dynamic binary instrumentation by simulating an on-line process scheduler. This contrasts with the NVIDIA Compute Visual Profiler [12] which only offers offline analysis. The experimental configuration is derived from work published by Gupta, et. al. [13] in which a hypervisor attempts to balance contention for a physical GPU among a collection of guest operating systems in a virtualized environment. By instrumenting kernels at the hypervisor level, scheduling decisions may be informed by data collected without modifying the actual applications.

A credit-based scheduler illustrated in Figure 7 assigns a kernel launch rate to each process according to a history of previous execution times, and rates are adjusted to achieve utilization targets for each process. For our demonstration, we wrote two CUDA applications, each performing matrix multiplication on distinct data sets. Application A performs matrix multiplication on a randomly-generated data set that is intentionally smaller than Application B's randomly-generated data set, resulting in disparate GPU utilizations. To properly balance compute resources across both processes, the hyperviser must transparently deter-

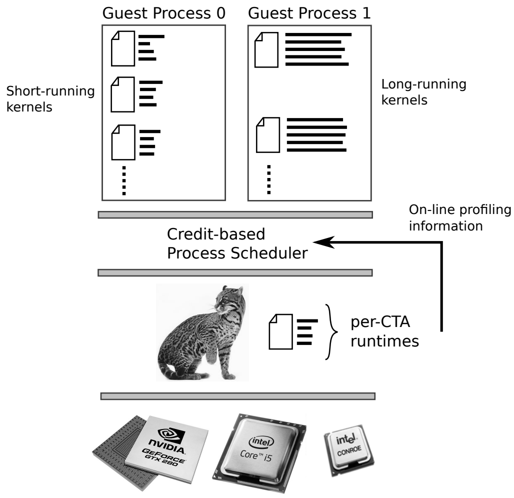


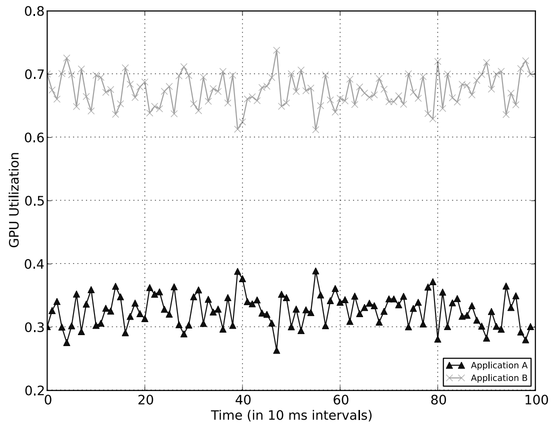
Figure 7: Credit-based process scheduler utilizing online profile information to make control decisions.

mine kernel runtimes and assign credits accordingly. We capture kernel runtimes via the `ClockCycleCountInstrumentor`, and adjust each application's kernel launch rate to achieve the desired 50% GPU utilization target. The scheduler initially launches each application's kernel with period of 10 ms and adjusts the rates every 50 ms.

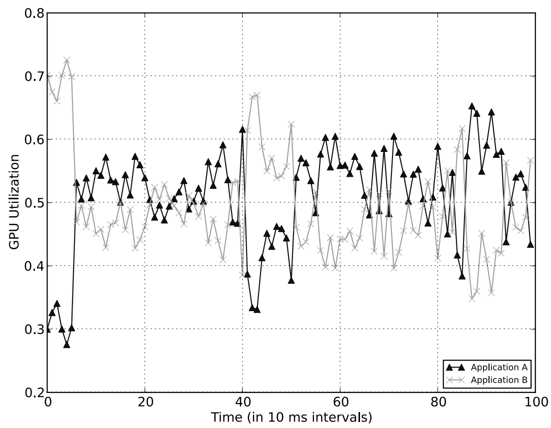
Figure 8 (a) plots GPU utilizations for the two applications running without any scheduling policy. As noted earlier, Application B launches kernels with considerably larger workloads than Application A. Figure 8 (b) plots the same applications with kernel launch rates specified by the process scheduler with a policy that tries to achieve the 50% utilization target for the two applications. The scheduler uses online information about kernel runtimes for gathering profiling results and adjusts kernel launch rates so that Application B launches kernels less frequently than A. The startup transient visible during the first few kernel launches results as runtime histories are constructed for both applications, and then utilization converges toward the 50% target.

Experiment 5 - Characterization of JIT Compilation Overheads. Dynamic binary instrumentation invokes a compilation step as the program is running. Application runtime is impacted both by the overheads associated with executing instrumentation code when it is encountered and also by the process of inserting the instrumentation itself. Dynamically instrumented CUDA programs require an additional just-in-time compilation step to translate from PTX to the native GPU instruction set, but applications are typically written with long-running kernels in mind. In this experiment, we attempt to characterize overheads in each step of Ocelot's compilation pipeline from parsing large PTX modules, performing static analysis, executing PTX-to-PTX transformations, JIT compiling via the CUDA Driver API, and executing on the GPU.

Figure 9 presents the dynamic compilation overheads in compiling and executing the Parboil application *mri-fhd* and instrumenting it with the basic block counters described in Section 3.3. This application consists of a single PTX module of moderate size (2,916 lines of PTX). The figure shows the relative time spent performing the instrumentation passes 14.6% is less than both the times to parse the



(a) A set of applications with imbalanced workloads and no rate limiting scheduler.



(b) Process scheduler targeting 50% utilization for both processes by adjusting the kernel launch rate.

Figure 8: Instrumentation used for process scheduling to achieve load balance in a simulated virtualized environment.

PTX module and to re-emit it for loading by the CUDA Driver API, steps that would be needed without adding instrumentation. Online use of instrumentation would not need to perform the parse step more than once. Results indicate there would be less than a 2x slowdown if kernels were instrumented and re-emitted with each invocation, a slowdown which would decrease for longer running kernels.

5. FUTURE WORK

With our current work, we have demonstrated the capability and usefulness of online profiling for GPU compute applications. However, we have only touched the surface with what we can achieve with this capability. For future work, we would like to develop a much more comprehensive suite of metrics. We would also like to investigate customized instrumentations for capturing application-specific behavior. One of the major benefits of our approach is the ability to do selective instrumentation for specific cases. From a design perspective, our instrumentation framework provides APIs that ease the design and implementation

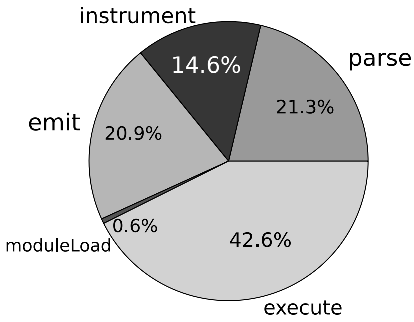


Figure 9: Overheads in compiling and executing an application from the Parboil benchmark suite. Instrumentation occupies 14.6% of total kernel runtime including compilation.

of custom, user-defined instrumentations. However, these APIs still require instrumentation to be specified at the PTX level. We are currently investigating higher-level, C-like constructs to specify instrumentation instead of requiring users of our tool to generate PTX instructions via the Ocelot IR interface. Finally, we demonstrated the ability for online scheduling as an example of how our framework can be used to enforce system policies. This capability opens new avenues for automatic run-time resource allocation and decision-making, potentially leading to higher system efficiency. There are several real-world use cases that can benefit from this capability. GPU-accelerated Virtual Machines that provide system management capabilities for heterogeneous manycore systems with specialized accelerators [13] is one such example. Another interesting use-case for online instrumentation is optimizing GPUs for power consumption. The GPU Power and Performance Model work by Hong and Kim predicts the optimal number of active processors for a given application [14]. We would like to explore opportunities to integrate with such analytical models.

6. RELATED WORK

Pin [15] is a mature dynamic instrumentation tool for inserting probes into CPU application binaries. It strives to be architecture independent, supporting CPU architectures such as x86, Itanium, ARM, and others. Our approach to instrumenting CUDA kernels was largely inspired by Pin's instrumentation model by facilitating the creation of user-supplied instrumentation tools and inserting them into existing applications. Pin does not target data-parallel architectures or execution models such as GPUs and PTX nor does it identify embedded PTX kernels as executable code. Moreover, Ocelot manages resources such as device memory allocations and textures and presents these to the instrumentation tools when they are inserted.

NVIDIA's Compute Visual Profiler [12] was released to address the profiling needs of developers of GPU compute applications and provides a selection of metrics to choose from. This utility is implemented by reading hardware performance counters available in NVIDIA GPUs after applications have run. However, it does not offer the opportunity to insert user-supplied instrumentation procedures and the results are not conveniently available for making online scheduling decisions.

Boyer, et. al. [16] propose statically analyzing CUDA programs and inserting instrumentation at the source level. This approach was used to detect race conditions among threads within a CTA and bank conflicts during accesses to shared memory. In their tool, a front end parser was written to construct an intermediate representation of the high-level program, to annotate it with instrumentation procedures, and ultimately to re-emit CUDA for compilation using the existing toolchain. This is a heavy-weight approach that must be performed before the program is run and does not enable removing instrumentation once a program has reached a steady phase of operation. Moreover, this approach misses the opportunity to observe behaviors only visible after CUDA has been compiled to PTX. These phenomena include PTX-level register spills if the program has been compiled with register count constraints or requires features available on narrow classes of hardware. Finally, instrumentation procedures added at the source level exist upstream of lower-level optimization transformations and may not reflect the true behavior of the program. For example, a loop unrolling transformation may duplicate a basic block counter in the loop body even though the resulting binary has fused several loop iterations into a single block. Ocelot enables instrumentation to be added at arbitrary points as PTX-level optimizations are applied.

Many metrics of interest may be available through simulation using tools such as GPGPU-Sim [17], Barra [18], or Ocelot's own PTX emulator [8]. Additionally, these may drive timing models such as those proposed by Zhang [19] and by Hong [20]. While these provide a high level of detail, compute-intensive GPU applications are intended to achieve notable speedups over CPUs *when run natively on GPU hardware.* CPUs executing an emulator executing a GPU compute application exhibit slowdowns on the order of 100x to 1000x which can be prohibitively slow when characterizing long-running benchmark applications. Such simulators implement GPU architectural features with varying levels of fidelity and range from cycle-accurate simulation of a hypothetical GPU architecture in the case of GPGPU-Sim to complete decoupling of the abstract PTX execution model in the case of Ocelot. Consequently, they do not necessarily capture the non-deterministic but significant interactions encountered by executing the GPU kernels on actual hardware. Instrumenting PTX kernels and executing them natively renders large GPU compute applications tractable and captures hardware-specific behaviors.

7. CONCLUSION

GPU compute toolchains have greatly facilitated the explosion of research into GPUs as accelerators in heterogeneous compute systems. However, toolchains have lacked the ability to perform dynamic binary instrumentation, and if such were to be added, instrumentation would have to be written for each processor architecture. By inserting instrumentation at the PTX level via a dynamic compilation framework, we are able to bridge the gap in transparent dynamic binary instrumentation capability for CUDA applications targeting both NVIDIA GPUs and x86 CPUs. This platform provides an open-ended and flexible environment for adding instrumentation probes not present in closed tools such as the Compute Visual Profiler. We envision our tool improving the state of workload characterization by profiling native executions rather than simulations, facilitating profile-directed optimization of data-parallel kernels, and enabling application-level scheduling and resource management decisions to be made online.

Acknowledgements

This research was supported by NSF under grants CCF-0905459 and OCI-0910735, IBM through an OCR Innovation award, LogicBlox Corporation, and an NVIDIA Graduate Fellowship. We gratefully acknowledge the insights of Vishakha Gupta and Alexander Merritt for their suggestion of the online credit-based scheduler.

8. REFERENCES

[1] KHRONOS OpenCL Working Group. *The OpenCL Specification*, December 2008.

[2] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 edition, October 2008.

[3] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, 1.3 edition, October 2008.

[4] Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. Gpuocelot: A binary translation framework for ptx., June 2009. `http://code.google.com/p/gpuocelot/`.

[5] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[6] Rodrigo Dominguez, Dana Schaa, and David Kaeli. Caracal: Dynamic translation of runtime environments for gpus. In *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, 2011. To appear.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

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

[9] Chris Lattner and Vikram 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.

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

[11] NVIDIA Corporation. Nvidia's next generation compute architecture: Fermi. white paper, NVIDIA, November 2009.

[12] NVIDIA. *NVIDIA Compute Visual Profiler*. NVIDIA Corporation, Santa Clara, California, 1.0 edition, October 2010.

[13] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[14] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. *Computer Architecture. IEEE International Symposium on*, 2010.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[16] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. Third Workshop on Software Tools for MultiCore Systems (STMCS), 2008.

[17] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor 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.

[18] Sylvain Collange, David Defour, and David Parello. Barra, a modular functional gpu simulator for gpgpu. Technical Report hal-00359342, 2009.

[19] Yao Zhang and John D. Owens. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, February 2011.

[20] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.