

Lynx: A Dynamic Instrumentation System for Data-Parallel Applications on GPGPU Architectures

Naila Farooqui*, Andrew Kerr†, Greg Eisenhauer*, Karsten Schwan* and Sudhakar Yalamanchili†

*College of Computing, Georgia Institute of Technology

†School of Electrical and Computer Engineering, Georgia Institute of Technology

Abstract—As parallel execution platforms continue to proliferate, there is a growing need for real-time introspection tools to provide insight into platform behavior for performance debugging, correctness checks, and to drive effective resource management schemes. To address this need, we present the *Lynx* dynamic instrumentation system. *Lynx* provides the capability to write instrumentation routines that are (1) *selective*, instrumenting only what is needed, (2) *transparent*, without changes to the applications’ source code, (3) *customizable*, and (4) *efficient*. *Lynx* is embedded into the broader GPU Ocelot system, which provides run-time code generation of CUDA programs for heterogeneous architectures. This paper describes (1) the *Lynx* framework and implementation, (2) its language constructs geared to the Single Instruction Multiple Data (SIMD) model of data-parallel programming used in current general-purpose GPU (GPGPU) based systems, and (3) useful performance metrics described via *Lynx*’s instrumentation language that provide insights into the design of effective instrumentation routines for GPGPU systems. The paper concludes with a comparative analysis of *Lynx* with existing GPU profiling tools and a quantitative assessment of *Lynx*’s instrumentation performance, providing insights into optimization opportunities for running instrumented GPU kernels.

I. INTRODUCTION

Instrumentation is a technique of inserting additional procedures into an application to observe and/or improve its behavior. Dynamic binary instrumentation involves inserting code at the instruction level of an application while the application is executing. Such instrumentation offers opportunities for detailed low-level inspection such as register spilling, instruction layout, and code generated downstream from compiler optimization passes. It also offers the capability to attach/detach instrumentation at runtime, thus selectively incurring potential overheads only when and where instrumentation is required.

Although dynamic instrumentation has been proven to be a useful program analysis technique for traditional architectures [1], it has not been fully exploited for GPU-based systems. This is in part due to the vendor-specific instruction set architectures (ISAs) and limited tool-chain support for direct modification of native binaries of GPUs.

Lynx is a dynamic binary instrumentation infrastructure for constructing customizable program analysis tools for GPU-based, parallel architectures. It provides an extensible set of C-based language constructs to build program analysis tools that target the data-parallel programming paradigm used in GPUs. The highlights of *Lynx*’s design, therefore, are *transparency*, *selectivity*, *customization*, and *efficiency*.

Transparency is achieved by inserting instrumentation at the Parallel Thread eXecution, or PTX level [2], ensuring that

the original CUDA applications remain unchanged. PTX is the virtual instruction set targeted by NVIDIA’s CUDA [3] and OpenCL compilers, which implements the Single-Instruction Multiple-Thread (SIMT) execution model. Furthermore, the instrumentation generated with *Lynx* does not alter the original behavior of the application.

Selective instrumentation implies specification of the instrumentation that is needed and where it should be inserted. With *Lynx*, instrumentation can be inserted at the beginning or end of kernels and basic blocks, and/or at the instruction level. It is also possible to instrument only particular classes of instructions.

Lynx makes such **customization** of instrumentation easy to do because the instrumentation for data-parallel programs is specified and written with a subset of the C programming language, termed C-on-Demand (COD). Further, the *Lynx* instrumentation APIs capture major constructs of the data-parallel programming model used by GPUs, such as threads, thread blocks, and grids. This results in a low learning curve for developers, particularly those already familiar with CUDA and/or OpenCL.

Lynx provides **efficient** instrumentation by using a JIT compiler to translate, insert and optimize instrumentation code. Additionally, the *Lynx* API supports the creation of instrumentation routines specifically optimized for GPUs. An example is warp-level instrumentation, which captures the behavior of a group of threads as opposed to individual threads. By taking advantage of the lock-step execution of threads within a warp, it becomes possible to reduce the memory bandwidth costs incurred by instrumentation.

Lynx also provides portability by extending an existing dynamic heterogeneous compilation framework, GPU Ocelot [4], which supports several processor backends, such as NVIDIA GPUs, x86 multi-core CPUs [5], PTX emulation, and AMD Radeon GPUs [6]. This paper, however, focuses on instrumentations and optimization opportunities for GPUs.

In summary, the technical contributions of *Lynx* described in this paper are as follows:

- A dynamic instrumentation system for GPGPU-based, parallel execution platforms that supports transparency, selectivity, customization, and efficiency.
- Implementation and design of a C-based language specification for defining instrumentation for data-parallel programs, which supports the construction of custom instrumentation routines.
- Demonstration of the versatility and usefulness of the *Lynx*

instrumentation system with implementations of performance metrics applicable to GPGPU-based architectures.

- Evaluation of Lynx’s performance and instrumentation overheads for a variety of GPU workloads, specifically highlighting optimization opportunities on GPUs.

The design and implementation of Lynx are reviewed in Section II, followed by a discussion of its instrumentation capability in Section III. Section IV discusses GPU-specific instrumentation techniques and performance metrics implemented via the Lynx instrumentation system. A comparison of Lynx with existing GPGPU profiling tools as well as an assessment of the performance and overheads of kernels instrumented with Lynx for several metrics are discussed in Section V. Related research appears in Section VI, followed by conclusions in Section VII.

All of the experiments were performed on a system with an Intel Core i7 running Ubuntu 10.04 x86-64, equipped with an NVIDIA GeForce GTX 480, except for the experiment comparing Lynx’s performance with Ocelot’s emulator, which was performed on an Intel Xeon X5660 CPU machine equipped with an NVIDIA Tesla C2050. Benchmark applications for experiments are chosen from the NVIDIA CUDA SDK [3], the Parboil Benchmark Suite [7], and the Rodinia Benchmark Suite [8].

II. DESIGN AND IMPLEMENTATION

In this section, we present a system overview of Lynx, followed with a detailed discussion on Lynx’s execution and/or run-time flow within GPU Ocelot.

A. System Overview

The Lynx instrumentation system is implemented as a core abstraction of GPU Ocelot and provides the following new capabilities:

- A C-based language specification for constructing customized instrumentation routines
- A C-based JIT compilation framework, which translates instrumentation routines to PTX
- A PTX instrumentation pass to modify existing kernels with PTX-generated instrumentation code
- Run-time management of instrumentation-related data structures

Figure 1 illustrates Lynx’s software architecture.

The Lynx system is comprised of an *Instrumentation API*, an *Instrumentor*, a *C-on-Demand (COD) JIT Compiler*, a *C-to-PTX Translator*, and a *C-to-PTX Instrumentation Pass*. The system’s implementation is embedded into GPU Ocelot, which provides the additional following components: a re-implementation of the *CUDA Runtime API*, a *PTX Parser*, a *PTX-PTX Transformation Pass Manager*, and a *PTX Translator/Code Generator* with support for multiple backed targets. When a CUDA application is linked with GPU Ocelot, API calls to CUDA pass through Ocelot’s CUDA Runtime, which provides a layer of compilation support, resource management, and execution. The CUDA application is parsed into PTX modules, consisting of one or more PTX kernels.

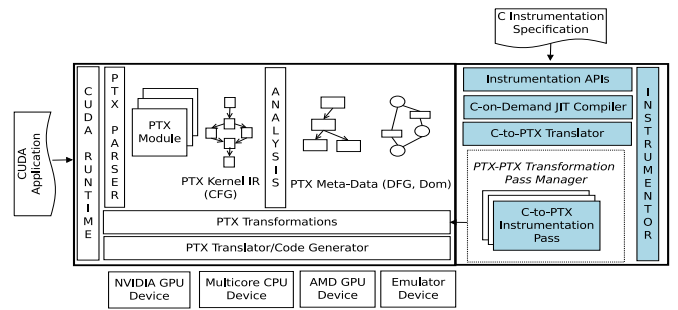


Fig. 1. Lynx software architecture embedded within GPU Ocelot. The highlighted Lynx components are the contributions of this paper.

If the application is being instrumented, a C-based instrumentation specification is provided to the framework. The Instrumentor serves as the run-time engine for generating the instrumented PTX kernel from the C specification and the original PTX kernel, by enlisting the COD JIT Compiler, the C-to-PTX Translator, and the PTX-PTX Transformation Pass Manager. The COD JIT Compiler produces a RISC-based IR for the C specification and the C-to-PTX Translator generates equivalent PTX instructions. The specification defines where and what to instrument, using the instrumentation target specifiers discussed in the next section. The C-to-PTX Instrumentation Pass, invoked by the Pass Manager, inspects these specifiers and inserts the translated instrumentation PTX accordingly into the original PTX kernel(s).

B. Execution Flow

Figure 2 illustrates Lynx’s execution/run-time flow within GPU Ocelot. CUDA applications compiled by `nvcc` are converted into C++ programs, with PTX kernels embedded as string literals. When such a program links with our framework, the CUDA Runtime API function, `cudaRegisterFatBinary`, parses these PTX kernels into an internal representation. The original PTX kernel is provided as input to GPU Ocelot’s Pass Manager, together with the instrumentation PTX generated from the C code specification via the COD JIT Compiler and the C-to-PTX Translator. The Pass Manager applies a sequence of PTX kernel transformations to the original PTX kernel. A detailed discussion of GPU Ocelot’s Pass Manager and PTX transformation passes can be found in our earlier work [9].

A specific pass, C-to-PTX Instrumentation Pass, is implemented as part of the Lynx framework to insert the generated PTX into the original PTX kernel, according to Lynx’s language specification. The final output, the instrumented kernel, is prepared for native execution on the selected device by the PTX Translator/Code Generator.

Since GPU Ocelot implements the CUDA Runtime API, it enables the insertion of hooks into the runtime system for managing resources and data structures needed to support instrumentation. The Lynx framework utilizes this capability via the Instrumentor component. Its general approach for managing instrumentation-related data is to allocate memory on

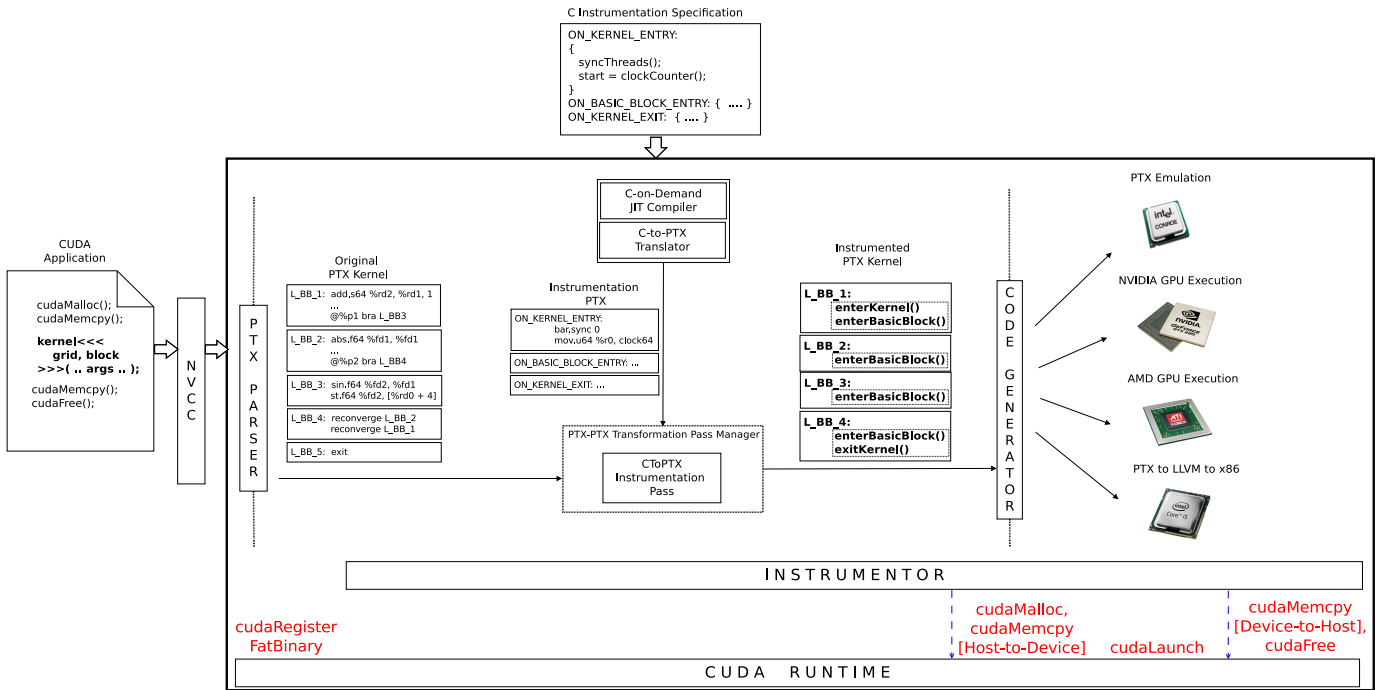


Fig. 2. Overview of Lynx’s run-time/execution flow. The **Original PTX Kernel** is augmented with the **Instrumentation PTX** generated by the C-to-PTX translation layer to produce the **Instrumented PTX Kernel**, which is finally executed on the selected device.

the device, populate the instrumentation-related data structures during kernel execution, and then move the data back to the host, freeing up allocated resources on the device.

Specifically, the Instrumentor invokes Ocelot’s CUDA Runtime `cudaMalloc` and `cudaMemcpy` APIs to allocate and initialize data structures on the device. After the kernel is executed on the device, the Instrumentor invokes `cudaMemcpy` and `cudaFree` to move the instrumentation data stored on the device onto the host and free up allocated resources on the device.

III. INSTRUMENTATION WITH LYNX

The Lynx instrumentation API provides both relevant functions to perform instrumentation in a data-parallel programming paradigm and *instrumentation specifiers*, which designate where the instrumentation needs to be inserted. We first present an overview of the Lynx Instrumentation API, followed by a description of three example instrumentation specifications. We end this section with a discussion on the C-to-PTX translator’s role in generating PTX from the instrumentation specifications.

A. Instrumentation API

Data-parallel kernels are executed by a tiered hierarchy of threads, where a collection of threads, also known as a thread block, is executed concurrently on a single *stream multiprocessor*, or SM. Typically, kernels are launched with tens or hundreds of thread blocks onto a limited set of available SMs. Threads within a thread block execute in a SIMT manner in groups called warps. A warp is defined as the maximal subset of

threads within a thread block that execute the same instructions concurrently. Therefore, the Lynx API at a minimum must capture the notions of thread, thread blocks, warps, and SMs. Table 1 describes a subset of the Lynx API. The table also includes functions such as `basicBlockExecutedInstructionCount()`, which represent attributes that are obtained via static analysis of data-parallel kernels.

The instrumentation specifiers are defined as C labels in the language. They can be categorized into four types: instrumentation targets, instruction classes, address spaces and data types. Table 2 captures all of the currently available specifiers in the Lynx instrumentation language.

Specifier Type	Available Specifiers
Instrumentation Target	ON_KERNEL_ENTRY, ON_KERNEL_EXIT, ON_BASIC_BLOCK_ENTRY, ON_BASIC_BLOCK_EXIT, ON_INSTRUCTION
Instruction Class	MEM_READ, MEM_WRITE, CALL, BRANCH, BARRIER, ATOMIC, ARITHMETIC
Address Space	GLOBAL, LOCAL, SHARED, CONST, PARAM, TEXTURE
Data Types	INTEGER, FLOATING_POINT

TABLE 2
AVAILABLE SPECIFIERS IN THE LYNX API

Instrumentation target specifiers have a dual purpose. First, they describe where the instrumentation needs to be inserted. For instance, instrumentation can be inserted at the beginning

Function Name	Description
globalThreadId	Global thread identifier for the current thread.
blockThreadId	Thread identifier for the current thread within its thread block.
blockId	Thread block identifier for the current thread block within its grid.
blockDim	Number of threads per thread block.
syncThreads	Barrier synchronization within a thread block.
leastActiveThreadInWarp	Determines the least active thread in the current warp.
uniqueElementCount	Total number of unique elements in a given buffer of warp or half-warp size.
basicBlockId	Returns the index of the current basic block.
basicBlockExecutedInstructionCount	Total number of executed instructions in the current basic block.

TABLE 1
A SUBSET OF THE LYNX API

or end of kernels, beginning or end of basic blocks, and/or on every instruction. Instrumentation is inserted just before the last control-flow statement when inserting at the end of basic blocks or kernels. Second, the instrumentation target specifiers serve as loop constructs. In other words, the instrumentation routine following the target specifier is applied to *each and every* kernel, basic block, or instruction, depending on the target specifier. Note that if no instrumentation target is specified, the instrumentation is inserted at the beginning of the kernel by default.

A user may only want to instrument certain classes of instructions, or only instrument memory operations for certain address spaces. For this reason, the Lynx API includes instrumentation class and address space specifiers. The multiple address space notion is part of the memory hierarchy model of data-parallel architectures. Threads may access data from various memory spaces, including on-device and off-device memory units. Finally, data type specifiers are provided to categorize arithmetic operations by integer and floating-point functional units.

B. Example Instrumentation Specifications

Figure 3 shows three example instrumentation specifications defined using Lynx’s instrumentation language: a basic-block level instrumentation, an instruction level instrumentation and a kernel level instrumentation. The instrumentation target specifiers are noted in each of the examples.

Dynamic instruction count is a **basic-block** level instrumentation that computes the total number of instructions executed, omitting predicated instructions that do not get executed. The instrumentation results are stored in the global address space of the device in a dynamically allocated array, on a per-basic-block, per-thread-block, per-thread basis. The *globalMem* variable is globally available to store instrumentation data. The actual allocation of this array takes place in the specific instrumentor class that is defined for this metric.

The memory efficiency metric is an example of an **instruction** level instrumentation. For every global load or store instruction, each thread within a thread block computes the base memory address and stores it in shared memory via the *sharedMem* global variable. For NVIDIA GPUs, a half-warp of 16 threads can coordinate global memory accesses into a single transaction. The least active thread in a warp maintains a count

of unique memory addresses for each half-warp, to determine the total number of memory transactions required to satisfy a particular global memory request, and increments the dynamic warp count.

Finally, the kernel runtime specification is a **kernel** level instrumentation that uses barrier synchronization for threads, *syncThreads*, within a thread block and the clock cycle counter API, *clockCounter*, to obtain the kernel run-time in clock cycles for each thread block. Since each thread block gets executed on a single SM, the current SM identifier for each thread block, obtained via *smId*, is also stored in global memory.

C. C-to-PTX Translator

The C-to-PTX translator is responsible for parsing the C code specification and translating the RISC-based C IR to PTX. Toward this end, the C-to-PTX translator walks through each C IR instruction and emits equivalent PTX instructions. Special handling occurs for each of the instrumentation specifiers. Since the specifiers are C-style labels, each label is converted to a basic-block label as a place-holder in the generated PTX. All of the PTX instructions that belong to the specifier become a part of a new translation basic-block. The C-to-PTX instrumentation pass then looks for these labels and inserts the corresponding instructions of that translation basic-block into the designated location of the application PTX kernel.

Although multiple instrumentation specifiers are allowed, each resulting in its own translation basic-block, the current implementation does not prevent conflicts between the instrumentation of different translation blocks. A possible solution is to enforce that each instrumentation is applied individually to the application’s PTX kernel as a separate instrumentation pass, effectively resulting in a chain of instrumentation passes executed independently. We have left this implementation as a future enhancement to our framework.

IV. GPU-SPECIFIC INSTRUMENTATION

In this section, we discuss GPU-specific instrumentation and present common metrics for GPGPU-based systems. We also present multiple methods for implementing the dynamic instruction count metric.

Dynamic Instruction Count

```

unsigned long currentBlockDim = blockDim();
unsigned long currentTID = blockThreadId();
unsigned long currentBlockId = blockIdx();

ON_BASIC_BLOCK_EXIT: ← insert instrumentation at the
                        end of every basic block
{
    unsigned long offset =
        basicBlockCount() * basicBlockId()
        + currentBlockDim * currentBlockId + currentTID;

    globalMem[offset] +=
        basicBlockExecutedInstructionCount();
}

```

Memory Efficiency

```

unsigned long threadIdx = blockDim();
unsigned long warpId = (blockDim() * blockDim()
    + threadIdx) >> 5;

ON_INSTRUCTION: ← insert instrumentation on every instruction
MEM_READ:
MEM_WRITE: ← only apply instrumentation to
             global memory instructions
GLOBAL:
{
    sharedMem[threadIdx] = computeBaseAddress();

    if(leastActiveThreadInWarp())
    {
        globalMem[warpId * 2] +=
            uniqueElementCount(sharedMem, true);
        globalMem[warpId * 2 + 1] += 1;
    }
}

```

Kernel Runtimes and Thread Block-SM Mapping

```

unsigned long start, stop;

ON_KERNEL_ENTRY: ← insert instrumentation at the
                  beginning of every kernel
{
    start = clockCounter();
    syncThreads();
}

ON_KERNEL_EXIT: ← insert instrumentation at the
                 end of every kernel
{
    syncThreads();
    stop = clockCounter();

    if(threadIndexX() == 0)
    {
        globalMem[blockId() * 2] = stop - start;
        globalMem[blockId() * 2 + 1] = smId();
    }
}

```

Fig. 3. Example Instrumentation Specifications

A. Warp-Level Instrumentation

A warp-level instrumentation is one that focuses on the behavior of a *warp*, the collection of threads physically issuing instructions on an SM during the same cycle, rather than the independent behaviors of each thread. Such instrumentation is sensitive to the way SIMD processors are utilized as well as how memory requests are coalesced before issuing to DRAM. GPUs implement thread divergence through implicit hardware predication; a bit mask stores the set of active threads within a particular warp. Warp-level instrumentation typically involves a reduction query across particular values computed by each thread in a warp. These could include conditional branch predicates, memory addresses, or function call targets.

B. Performance Metrics for GPUs

1) *Memory Efficiency*: Memory efficiency is an example of a GPU-specific **warp-level** metric that characterizes the spatial locality of memory operations to global memory. Global memory is the largest block of memory in the PTX memory hierarchy and also has the highest latency. To alleviate this latency cost, the PTX memory model enables coalescing of global memory accesses for threads of a half-warp into one or two transactions, depending on the width of the address bus. However, scatter operations, in which threads in a half-warp access memory that is not sequentially aligned, result in a separate transaction for each element requested, greatly reducing memory bandwidth utilization. The goal of the memory efficiency metric, therefore, is to characterize memory bandwidth utilization by determining the ratio of dynamic warps executing each global memory dynamic instruction to the number of memory transactions needed to complete these instructions. Memory efficiency for several applications, obtained via Lynx’s instrumentation system, are presented in Figure 4.

The Lynx instrumentation system provides useful APIs to enable the creation of intricate warp-level instrumentations. For example, for the memory efficiency metric, the base address of each global memory operation is computed and stored for all

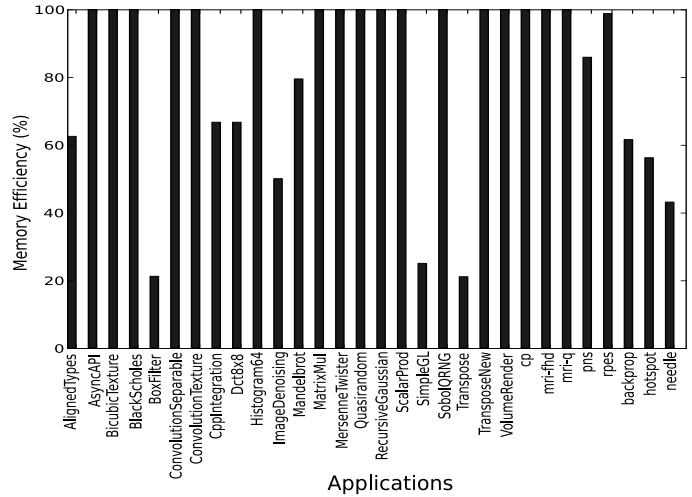


Fig. 4. Memory Efficiency

threads in a thread block. If the base address is the same for all threads belonging to a half-warp, then the memory accesses will be coalesced. A single thread within a warp, determined by the `leastActiveThreadInWarp` API, performs an online reduction of the base addresses written to a shared buffer by all threads in the warp. The `uniqueElementCount` API, which is used to keep a count of unique base addresses within a half-warp, determines the total number of memory transactions required for a particular memory operation.

2) *Branch Divergence*: Branch divergence is another example of a warp-level metric. It provides insight into a fundamental aspect of GPU performance, namely its SIMT execution model. In this model, all threads within a warp execute each instruction in lock-step fashion until they reach a data-dependent conditional branch. If such a condition causes the threads to diverge, the warp serially executes each branch path taken, disabling threads that are not on that path. This imposes a large penalty on

kernels with heavy control-flow. The branch divergence metric measures the ratio of divergent branches to the total number of branches present in the kernel for all SMs in the GPU, characterizing kernels with control-flow.

We have implemented the branch divergence metric, using the Lynx instrumentation framework, and present our findings in Figure 5.

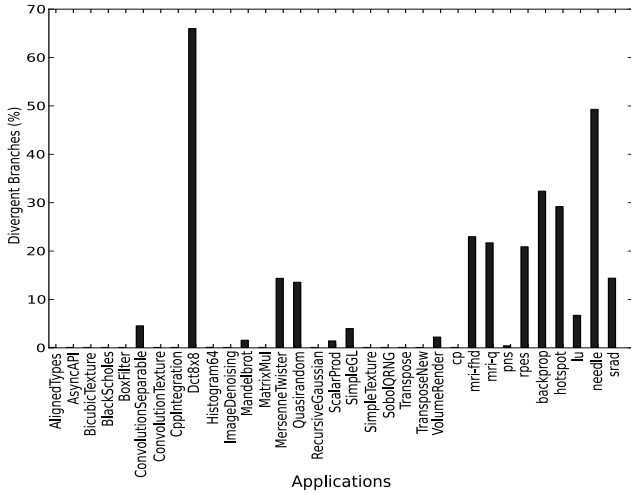


Fig. 5. Branch Divergence

Although one might expect to observe an inverse relationship between branch divergence and memory efficiency, less correlation is apparent in practice. Regions with frequent branches and short conditional blocks may have a high incidence of branch divergence but quick re-convergence. Warps executing in these regions consequently exhibit high SIMD utilization, as is the case with *Dct8x8*. Memory instructions located in blocks of the program in which all threads have re-converged therefore have high memory efficiency. Thus, SIMD utilization is more strongly correlated with memory efficiency than the fraction of branches which diverge.

3) *Kernel Runtime*: The kernel runtime metric is useful in precisely measuring the time it takes for a kernel to execute on a GPU. Although there are other methods for measuring kernel runtimes, such as via source-level assertions, these require participation from the host-side, such as synchronization after the kernel launch, for the measurements to be meaningful [10]. The use of instrumentation enables polling hardware counters on the device to obtain kernel runtimes, which capture precise measurements of multiple events within the execution of a single kernel without including latencies of PCI bus, driver stack, and system memory.

Our methodology for implementing kernel runtime involves capturing the runtime, in clock cycles, for each thread block and its corresponding SM executing the kernel. This enables us to determine whether the number of thread-blocks and corresponding workloads result in SM load imbalance due to unequal distribution of work among all the thread-blocks for a given kernel. Such fine-grained instrumentation provides useful

insights into the GPU thread scheduler’s performance via the degree of load balancing it is able to achieve. These insights in turn can provide useful feedback for performance tuning by re-structuring applications, such as the number and size of thread-blocks.

4) *Dynamic Instruction Count*: The dynamic instruction count metric captures the total number of instructions executed on a GPU. We provide two distinct implementations for this metric. Our first implementation, *counter-per-thread instruction count*, maintains a matrix of counters in global memory with one row per basic block in the executed kernel and one column per dynamic PTX thread. As each thread reaches the end of a particular basic block, it increments its counter index by the number of executed instructions in that basic block. Counters of the same basic-block for consecutive threads are arranged in sequential order to ensure that global memory accesses are coalesced.

Our second implementation is a warp-level instrumentation for the instruction count metric. Since the counter-per-thread instruction count maintains a counter in global memory for every dynamic PTX thread, it contributes a significant overhead in terms of memory bandwidth. To reduce this overhead, we implement a *counter-per-warp instruction count*, where a counter in global memory is maintained for every warp instead of every thread. As each warp reaches the end of a basic block, the least active thread in each warp multiplies the active thread count for that warp with the number of executed instructions in that basic block. This product is added to the specific warp’s counter.

We use our kernel runtime metric to measure runtimes for the “most representative”, or the longest-running, kernel with and without our two instruction count instrumentations for selected applications in the CUDA SDK. Normalized runtimes for these kernels are presented in Figure 6.

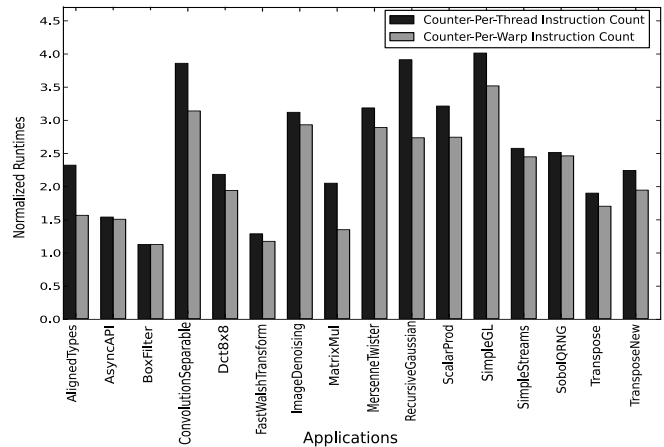


Fig. 6. Normalized runtimes of selected applications due to dynamic instruction count instrumentation.

As depicted in Figure 6, the counter-per-warp implementation consistently outperforms the counter-per-thread instruction count. We attribute this to the lower memory bandwidth over-

FEATURES	Compute Profiler /CUPTI	GPU Ocelot Emulator	Lynx
Transparency (No Source Code Modifications)	YES	YES	YES
Support for Selective Online Profiling	NO	YES	YES
Customization (User-Defined Profiling)	NO	YES	YES
Ability to Attach/Detach Profiling at Run-Time	NO	YES	YES
Support for Comprehensive Online Profiling	NO	YES	YES
Support for Simultaneous Profiling of Multiple Metrics	NO	YES	YES
Native Device Execution	YES	NO	YES

TABLE 3
COMPARISON OF LYNX WITH EXISTING GPU PROFILING TOOLS

head resulting from a warp-level instrumentation. However, in some cases, the difference between the two implementations is negligible, such as with *BoxFilter* and *AsyncAPI*, whereas in other cases, the difference is significant, such as with *ConvolutionSeparable* and *RecursiveGaussian*.

We used NVIDIA’s Compute Profiler [11] to gain more insight into the L1 cache behavior of these applications. Our findings indicated that the number of global load/store misses in L1 cache, when compared to the number of cache misses with no instrumentation, was between 1.6-1.8x greater for the counter-per-warp instruction count, and 3-4.5x greater for the counter-per-thread instruction count, for the *AsyncAPI* and *BoxFilter* kernels. However, in the case of *ConvolutionSeparable* and *RecursiveGaussian*, the number of L1 cache misses for global operations was around 3x greater for the counter-per-warp instruction count and almost 9x greater for the counter-per-thread instruction count. This lends us to believe that certain kernels are more bandwidth-sensitive than others, and for such kernels, the performance gain from the counter-per-warp instrumentation versus the counter-per-thread approach is likely to be more significant.

V. EVALUATION

We first provide a comparative analysis of Lynx with existing GPU profiling tools, such as NVIDIA’s Compute Profiler/CUPTI [11], [12], and GPU Ocelot’s emulator. We then look at both the impact of dynamic compilation and the increases in execution times for instrumented kernels, to evaluate the performance of Lynx.

A. Comparison of Lynx with Existing GPU Profiling Tools

As noted earlier, the core highlights of Lynx’s design include transparency, selectivity, and customization. We look at these as well as other features that distinguish Lynx from NVIDIA’s profiling tools and GPU Ocelot’s emulator. A comparative summary is presented in Table 3.

Lynx provides online instrumentation of applications transparently, i.e. without source code modifications. Although

NVIDIA’s Compute Profiler also provides transparency, the performance counters of interest need to be configured prior to the application run and cannot be modified during the execution of the application. Lynx, however, does not require pre-configuration of metrics. As a result, integrating Lynx with online optimizers, such as kernel schedulers or resource managers, is transparently feasible. Additionally, Lynx can attach/detach instrumentation at run-time, selectively incurring overheads only when and where instrumentation is required. This capability is not supported by NVIDIA tools.

Lynx also provides the complementary capabilities of selective and comprehensive online profiling. Users can profile applications at different granularities, such as on a per-thread, per-warp, per-thread-block, or per-SM basis, or can profile all threads for all SMs to capture the complete kernel execution state. NVIDIA tools, however, only provide data for a restrictive set of SMs [11], [12]. This limitation results in possible inaccuracy and inconsistency of extrapolated counts, since the data depends on the run-time mapping of thread blocks to SMs, which can differ across multiple runs [11]. It also restricts what one can do with these tools. For example, the thread-block runtimes to SM mappings, implemented via Lynx, informs programmers of potential SM load imbalance. Such fine-grained analysis of workload characterization cannot be obtained by existing NVIDIA tools.

Another distinguishing feature of Lynx is the support for customized, user-defined instrumentation routines, using a C-based instrumentation language. NVIDIA’s Compute Profiler/CUPTI provide a specific pre-defined set of metrics to choose from for specific generations of GPUs. User defined instrumentation is not supported. As a result, the memory efficiency metric cannot be obtained by existing NVIDIA tools for NVIDIA GPUs with compute capability 2.0 or higher [11], [12]. NVIDIA tools also do not allow certain counter values to be obtained simultaneously in a single run, such as the `divergent_branch` and `branch` counters. This can skew results as multiple runs with likely different counter values may be required to obtain a single metric, such as branch divergence. Lynx, on the other hand, supports simultaneous profiling of multiple metrics in a single run.

Although the Ocelot emulator has most of the capabilities Lynx provides, a core limitation of the emulator is native device execution. Since applications in emulation mode do not run on the GPU, the emulator is unable to capture hardware-specific behaviors and results in being orders of magnitude slower. Figure 7 presents slowdowns of a subset of applications’ execution on an Intel Xeon X5660 CPU with hyper-threading enabled (via Ocelot’s emulator) versus execution on NVIDIA’s Tesla C2050 GPU (via Lynx) for the memory efficiency metric. Applications with short-running kernels were purposely chosen for this experiment since longer-running kernels were prohibitively slow on the emulator.

B. Performance Analysis of Lynx

1) *JIT Compilation Overheads*: When using a JIT compilation framework, the compilation time is directly reflected in the

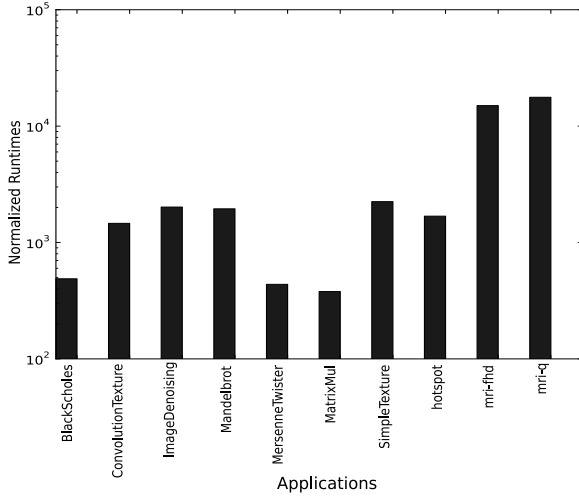


Fig. 7. Slowdowns of selected applications’ execution on Intel Xeon X5660 CPU via Ocelot’s emulator vs execution on NVIDIA Tesla C2050 GPU via Lynx for the memory efficiency metric

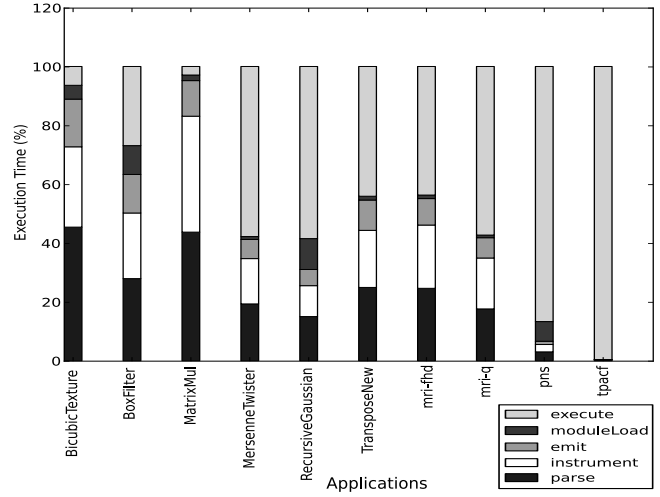


Fig. 8. Compilation overheads for selected applications from the CUDA SDK and Parboil benchmark suites, instrumented with dynamic instruction count

application’s runtime. To characterize overheads in each step of the compilation pipeline, we divide the total runtime into the following categories: *parse*, *instrument*, *emit*, *moduleLoad*, and *execute*. *Parse* is the time taken to parse the PTX module. *Instrument* measures all the various tasks necessary to insert the instrumentation into the application, namely, parsing the C specification, lowering it to the COD IR, translating the IR to PTX, and invoking the pass manager to apply the instrumentation pass over the original kernels. *Emit* is the time spent in GPU Ocelot’s PTX emitter, while *moduleLoad* measures the time spent loading the PTX modules into the CUDA driver. Finally, *execute* refers to the execution time of the kernels on the GPU.

Figure 8 shows the compilation overheads for several applications instrumented with the **counter-per-thread instruction count** metric (see Section IV-B4). This metric was chosen because the extent of static code expansion is proportional to the kernel size. The selected benchmark applications show a considerable range of static kernel sizes, ranging from applications with a single, small kernel, such as *MatrixMul*, to those with many, small and medium-sized kernels, such as *TransposeNew* and *pns*, to a single, large kernel, such as *tpacf*.

The results indicate that the *instrument* overhead is consistently less than the overhead associated with parsing the module but is generally more than the overheads associated with emitting and loading the module. Additionally, for applications that either have many small or medium-sized kernels, such as *RecursiveGaussian* and *pns*, or have a single, large kernel, such as *tpacf*, most of the time is spent in executing the kernels on the GPU. This indicates that for longer-running applications, Lynx’s JIT compilation overheads are either amortized or hidden entirely.

2) *Instrumentation Dynamic Overhead*: As shown in the previous evaluation, most of the overhead of instrumentation

for longer-running applications is due to executing the instrumented kernels. In this section, we evaluate the overhead of our instrumentation routines on kernel execution times.

Figure 9 presents slowdowns from three instrumentations: kernel runtime, dynamic instruction count, and branch divergence. The kernel runtime metric presents an average slowdown of 1.1x, indicating a minimal impact on performance. This is expected since only a few instructions are added to the beginning and end of kernels to perform barrier synchronization and obtain the clock cycle count for all threads in a thread block.

The dynamic instruction count instrumentation incurs both memory bandwidth and computation overheads for every basic block. Therefore, applications with few compute-intensive but potentially large basic blocks, such as *BicubicTexture*, *BlackSholes*, and *Transpose*, experience the least slowdown since memory access costs are amortized. Applications with several short basic blocks, such as *mri-fhd* and *mri-q*, exhibit a much larger slowdown (over 4x). Although *Mandelbrot* and *QuasirandomGenerator* have a mixture of large and small basic blocks, these two applications exhibit the largest slowdown due to the large number of predicated instructions in their kernels. The dynamic instruction count only takes into account instructions that are actually *executed*. As a result, for every predicated instruction in the kernel, this instrumentation checks whether the predicate is set for each thread to determine if the instruction count needs to be incremented. Consequently, applications that have a large overhead due to branch divergence instrumentation also exhibit a significant slowdown from the dynamic instruction count instrumentation since both of these instrumentations are impacted by the existence of predicated instructions.

Slowdown due to branch divergence varies from as low as 1.05x (*ConvolutionTexture*) for some applications to as high as 15x (*QuasirandomGenerator*) for other applications. This

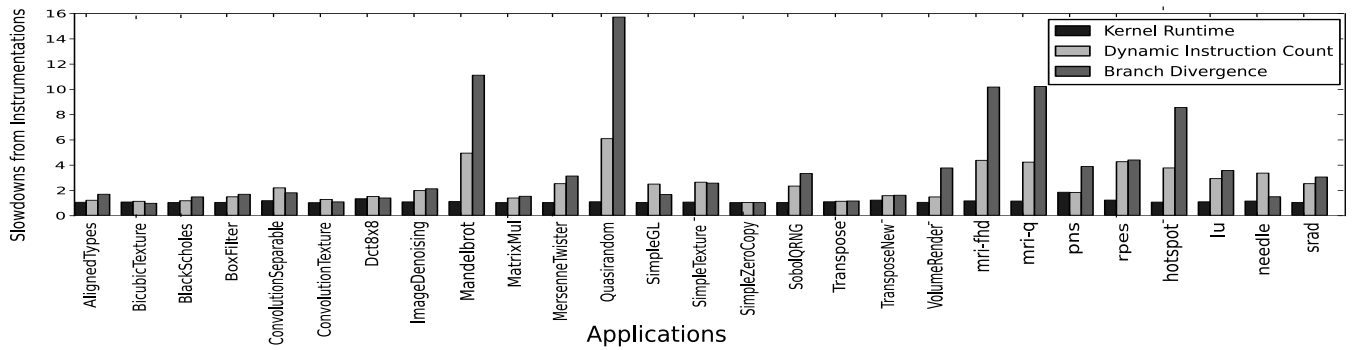


Fig. 9. Slowdowns of selected applications due to kernel runtime, dynamic instruction count, and branch divergence instrumentations.

instrumentation depends on the number of branch instructions present in the kernel. Hence, applications that exhibit significant control-flow, such as *Mandelbrot* and *QuasirandomGenerator*, incur the largest performance penalty from this instrumentation while those with little to no control-flow, such as *BicubicTexture* and *ConvolutionTexture*, incur the least penalty.

Figure 10 shows slowdowns of various applications due to the memory efficiency instrumentation.

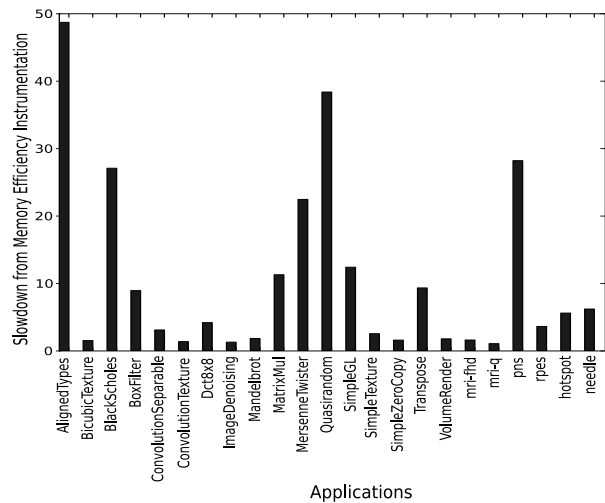


Fig. 10. Slowdowns of selected applications due to memory efficiency instrumentation.

Once again, we see that certain applications achieve minimal slowdown from this instrumentation while others exhibit slowdowns as high as 30-48x. Since this instrumentation only checks for global memory operations, applications with few global memory operations, such as *BicubicTexture* and *ConvolutionTexture*, exhibit the least slowdown. However, applications with many global memory operations, such as *AlignedTypes*, *QuasirandomGenerator* and *pns*, pay a large performance penalty. Memory efficiency is the most compute-intensive instrumentation we have implemented, with a notable memory bandwidth demand as well. These factors contribute to its high overhead. However, this overhead is to be weighed against the value of information provided by metrics enabled by

this type of instrumentation, which is not available via vendor tools. Further, as shown in Figure 7, evaluating such metrics can take several orders of magnitude more time using an instruction set emulator.

C. Optimizing Instrumentation Performance

Certain optimizations can lower the overhead costs of instrumentation. For example, we could use registers for counter variables to reduce the memory bandwidth cost of our instrumentations. Additionally, currently we have manually applied basic optimizations, such as constant propagation, dead-code elimination and removal of duplicate instructions, directly to the instrumentation code. Applying these and other optimizations automatically via subsequent optimization passes on to instrumented kernels can reduce the overhead costs further. First, since Ocelot’s Pass Manager coalesces passes by type, extracting the optimizations as separate passes would allow them to be applied in parallel on multiple kernels. This would improve the performance of the *instrument* phase in the JIT compilation. Second, the optimizations would be applied holistically on to the instrumented kernel, enabling re-use of PTX instructions in the original kernel for instrumentation. For example, the original kernel may be using special PTX registers that are also needed, and therefore generated, by the instrumentation code. Such duplicate PTX instructions would be removed by the optimization passes, thus improving overall kernel execution time.

VI. RELATED WORK

Pin [13] is a dynamic instrumentation system for CPU application binaries that supports multiple CPU architectures, such as x86, Itanium, and ARM. Just like Pin, Lynx also supports the creation of portable, customized program analysis tools. However, unlike Pin, Lynx targets data-parallel applications and enables the execution of such programs on heterogeneous backend targets, such as GPUs and multi-core CPUs.

NVIDIA’s Compute Visual Profiler [11] and *CUDA Profiling Tools Interface*, CUPTI [12], were released to address the profiling needs of developers of GPU compute applications. Each provides a selection of metrics to choose from by reading performance counters available in NVIDIA GPUs after applications have run. CUPTI enables the creation of profiling

tools for CUDA applications via APIs that perform source code interjection. However, as discussed earlier, these utilities do not offer the opportunity to insert customized, user-defined instrumentation procedures, and provide data for only a limited number of SMs.

The TAU Performance System [10] provides profiling and trace analysis for high-performance parallel applications. TAU, however, relies on source instrumentation and library wrapping of the CUDA Runtime/Driver APIs to obtain measurement before and after CUDA API calls via callbacks and/or events. TAU also does not offer the flexibility to end-users to define their own customized instrumentation routines and relies heavily on the device manufacturer to provide support for events and callbacks. Furthermore, unlike TAU, by obtaining kernel execution measurements via Lynx's dynamic instrumentation system, there is no participation needed from the host to perform the GPU-related measurement.

GPU simulators and emulators [14], [15], [16], visualization tools built on top of such simulators [17], and performance modeling tools [18], [19] are generally intended for offline program analyses to identify bottlenecks and predict performance of GPU applications. Many metrics of interest typically obtained through simulation may also be obtained through instrumentation, particularly those metrics reflecting application behavior. Thus, Lynx is one viable tool to drive research efforts such as the aforementioned. Hardware interactions may also be observed through instrumentation, provided care is taken to avoid perturbing the very effect being measured. With Lynx's selective and efficient profiling tools, the same levels of detail can be captured while natively running applications on the GPU, achieving notable speedups over CPU execution.

VII. CONCLUSION

This paper presents the design and implementation of Lynx, a dynamic instrumentation tool-chain that provides *selective*, *transparent*, *customizable* and *efficient* instrumentation for GPU computing platforms. Lynx's goal is to facilitate the creation of binary instrumentation tools for data-parallel execution environments, a novel capability that no existing tool available to the research community provides. Auto-tuning and optimization tools can be built on top of Lynx's dynamic instrumentation engine to improve program performance at runtime. The online feedback capability provided by Lynx also makes it a viable candidate to drive run-time resource management schemes for high-performance GPGPU-based clusters. Monitoring and execution analysis tools can be integrated with Lynx to run natively rather than on a simulator, achieving several orders of magnitude speedup. Furthermore, Lynx enables the creation of correctness checking tools that provide insights into both functional and performance bugs to improve programmer productivity. Our future work looks at several of the above-mentioned use cases. We are currently building a *GPU-top*, resource monitoring utility on top of Lynx, and investigating the design of effective optimization passes for instrumented kernels. Lynx is freely available at <http://code.google.com/p/gpuocelot/>, under the `lynx` branch.

ACKNOWLEDGEMENTS

This research was supported by NSF under grants CCF-0905459, OCI-0910735, and IIP-1032032.

REFERENCES

- [1] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R.B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer, IEEE*, 28(11):37–46, 1995.
- [2] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, 1.3 edition, October 2008.
- [3] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 edition, October 2008.
- [4] Gregory Damos, Andrew Kerr, and Sudhakar Yalamanchili. Gpuocelot: A binary translation framework for ptx., June 2009. <http://code.google.com/p/gpuocelot/>.
- [5] Gregory Damos, 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*, Newport Beach, CA, USA, March 2011. ACM.
- [7] IMPACT. The parboil benchmark suite, 2007.
- [8] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct. 2009.
- [9] Naila Farooqui, Andrew Kerr, Greg Damos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, Newport Beach, CA, USA, March 2011. ACM.
- [10] S. Shende and A.D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20:287–311, 2006.
- [11] NVIDIA. *NVIDIA Compute Visual Profiler*. NVIDIA Corporation, Santa Clara, California, 4.0 edition, May 2011.
- [12] NVIDIA. *NVIDIA CUDA Tools SDK CUPTI*. NVIDIA Corporation, Santa Clara, California, 1.0 edition, February 2011.
- [13] 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.
- [14] 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)*, pages 163–174, Boston, MA, USA, April 2009.
- [15] Sylvain Collange, David Defour, and David Parelo. Barra, a modular functional gpu simulator for gpgpu. Technical Report hal-00359342, 2009.
- [16] Andrew Kerr, Gregory Damos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.
- [17] Aaron Ariel, Wilson W. L. Fung, Andrew E. Turner, and Tor M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–174, White Plains, NY, USA, March 2010. IEEE Computer Society.
- [18] Yao Zhang and John D. Owens. A quantitative performance analysis model for gpu architectures. In *17th International Conference on High-Performance Computer Architecture (HPCA-17)*, pages 382–393, San Antonio, TX, USA, February 2011. IEEE Computer Society.
- [19] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '10*, pages 105–114, New York, NY, USA, 2010. ACM.