1 Author Information

Andrew Kerr arkerr@gatech.edu	Gregory Diamos gregory.diamos@gatech.edu	Sudhakar Yalamanchili sudha@gatech.edu
Graduate Research Assistant	Graduate Research Assistant	Professor
Georgia Institute of Technology (GIT)	GIT	GIT

Mailing addresses:

Andrew Kerr
923 Peachtree St. #1331
Atlanta, GA 30309.

Gregory Diamos
1911 Dry Creek Rd.
Campbell, California. 95008.

Sudhakar Yalamanchili
266 Ferst Drive, KACB 2316
Atlanta, GA 30332

2 Article Name Proposal

GPU Application Development, Debugging, and Performance Tuning with GPU Ocelot

3 Target Audience

The primary audience of this gem consists of developers of GPU computing applications who benefit from tools to debug, performance tune, and characterize applications. The secondary audience consists of researchers investigating compilation techniques, processor architectures, and workload characterization. We will focus on the primary audience in this paper.

4 Problem Statement

Developers of highly-parallel applications that achieve large speedups on GPUs face new and unique challenges in terms of developer productivity and the portability of applications. These challenges can be addressed by gaining insights into GPU compute application behaviors, utilizing effective tools to detect errors and identify performance bottlenecks, and finally to adapt GPU applications according to the particular compute platform on which it is executed.

GPU Ocelot [1] is a dynamic compilation framework that addresses these challenges by providing a single infrastructure capable of: (1) instrumenting and profiling CUDA applications, (2) observing and analyzing complex program behaviors, (3) identifying errors and performance bottlenecks, and (4) executing and dynamically optimizing CUDA applications running on CPUs and GPUs. We will discuss in greater detail how Ocelot's instrumentation and functional simulation capabilities can be leveraged by GPU application developers to identify program bugs

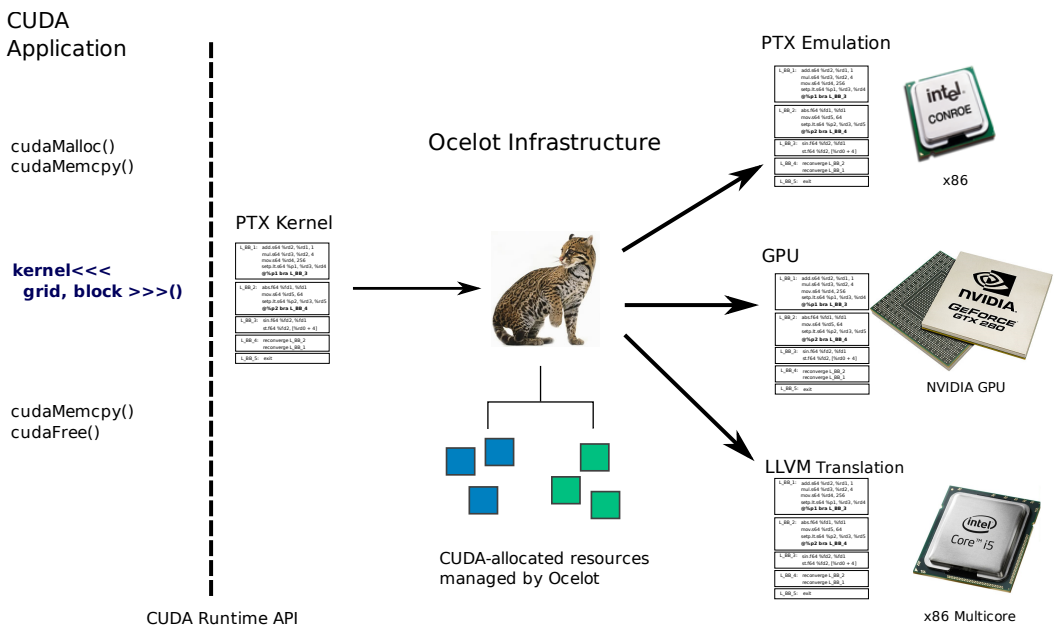

Figure 1: GPU Ocelot Dynamic Compilation Framework.

quickly, predict the efficiency of their application executing on entire class of GPU architectures, and understand the behaviors of their applications in detail.

5 Core Technology

GPU Ocelot's core capabilities consist of (1) an implementation of the CUDA Runtime API [2], (2) a complete internal representation of PTX kernels [3] coupled to control- and data-flow analysis procedures, (3) a functional emulator for PTX, (4) a translator to multicore x86-based CPUs for efficient execution, (5) and a backend to NVIDIA GPUs via the CUDA Driver API. Ocelot supports an extensible trace generation framework in which application behavior such as control-flow uniformity, memory access patterns, and data sharing may be observed at instruction-level granularity.

Ocelot's three backend execution targets - PTX emulator, multicore translator, and NVIDIA GPU - present a heterogeneous compute platform for data-parallel workloads. To use Ocelot, a developer links their compiled CUDA application against Ocelot's static library instead of NVIDIA's `libcudart` making integration with existing compiled applications seamless. GPU Ocelot is tested for correctness against all of the CUDA SDK [4], Parboil benchmark suite [5], and Thrust [6] unit tests and is currently part of the development toolchains of several GPU-computing related projects. Ocelot's support for efficient execution on multicore CPUs has enabled research in heterogeneous computing such as predictive performance modeling [7] and research in optimization techniques for data-parallel workloads [8].

6 Description of Work

GPU Ocelot is characterized by its front-end interface to existing CUDA applications, its capacity to analyze and transform PTX kernels using its IR, its complete representation of CUDA

kernels and memory resources, and its support of three backend execution targets. Ocelot is implemented in C++ with source code available under BSD license and distributed both through static releases as well as anonymous SVN checkout from the Ocelot Project Site [1] available at `http://code.google.com/p/gpuocelot/`.

GPU Ocelot implements the CUDA Runtime API. Compiled CUDA source files store modules as static blocks of text represented as PTX, NVIDIA's virtual instruction set architecture for GPUs, that are explicitly registered when the application is initialized. As illustrated in Figure 1, Ocelot is invoked when applications are intialized as they attempt to register modules and variables. Ocelot parses the PTX representation of the modules into a data structure representing the complete state of the CUDA-managed resources in an application.

CUDA Runtime API functions perform resource management procedures such as allocation of memory on devices, binding of textures and arrays, and copying memory between address spaces. Ocelot implements these functions and constructs a data structure representing each resource in addition to allocating the block of memory in the selected device's address space. This enables memory checking for memcpy functions such as *cudaMemcpy* be ensuring destination regions are contained by existing allocations. The PTX emulator and multicore translator backends include support for more fine-grain memory checking by testing each address used by a load or store against the set of valid memory allocations.

GPU Ocelot supports three backend execution targets: a functional PTX simulator which will be described in greater detail in the following sections, a translator to multicore x86 CPUs, and a backend to NVIDIA GPUs via the CUDA Driver API. The multicore translator is implemented as a set of PTX-to-PTX transformations including conversion to static single-assignment form followed by a translation from PTX to the native instruction set of Low-Level Virtual Machine (LLVM) [8]. Most PTX instructions have a one-to-one correspondence with LLVM instructions, and more complex instructions that typically have hardware support on GPUs, such as trigonometric functions and texture sampling, are implemented in software. When the LLVM kernel is executed, a runtime statically maps blocks of the grid onto hardware worker threads which execute each Cooperative Thread Array (CTA), serializing CUDA threads within the CTA and respecting barrier synchronizations.

The NVIDIA GPU backend is implemented by emitting PTX modules as text and compiling them to the selected GPU via the CUDA Driver API. Memory resources managed by Ocelot are allocated by calls to the CUDA Driver API, and global variables are made consistent before and after each kernel invocation. By using Ocelot as the implementation of the CUDA Runtime API, applications can benefit from memory bounds checking during copy operations. Though we have not fully explored the possibilities of instrumenting kernels for execution on GPUs, Ocelot offers the unique capability to modify every kernel launched by a CUDA application and insert additional correctness checks, watches on global variables, and instrumentation to monitor application behavior.

6.1 PTX Functional Simulation

Ocelot's PTX emulator models a virtual architecture illustrated in Figure 2 implementing the PTX execution model. Each PTX instruction is executed for as many threads as possible before moving on to the next instruction corresponding to an arbitrarily wide SIMD processor. Blocks of memory store values for the virtual register file as well as the addressable memory spaces. The emulator interprets each instruction according to opcode, data type, and modifiers such as

rounding or clamping modes, updating the architectural state of the processor with computed results.

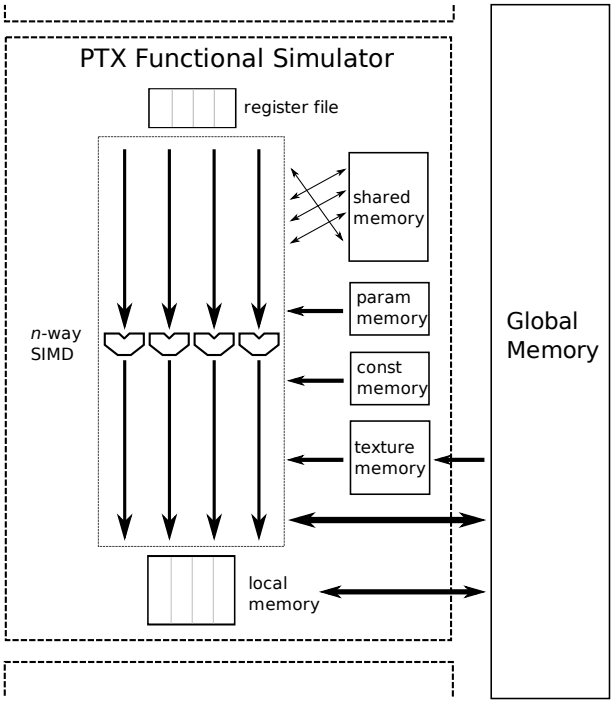

Figure 2: PTX emulator virtual architecture.

Kernels executed on the PTX emulator present the entire observable state of a virtual GPU to user-extensible instruction trace generators. These are objects implementing an interface that receives the complete internal representation of a PTX kernel at the time it is launched for initial analysis. Then, as the kernel is executed, a trace event object is dispatched to the collection of active trace generators after each instruction completes. This trace event object includes the instruction's internal representation and PC, set of memory addresses referenced, and thread ID. At this point, the instruction trace generator has the opportunity to inspect the register file and memory spaces accessible by the GPU such as shared and local memory. Practically any observable behavior may be measured using this approach. In the next section, we will discuss Ocelot's interfaces for user-extended trace generators that compute custom metrics.

6.2 Extensible Trace Generation Framework

GPU Ocelot's trace generation and analysis framework presents a clear and concise interface to user-extensible trace generators which are the preferred method to instrument and profile GPU applications. In this section, we will explain how trace generators are invoked by the PTX emulator and discuss the information that is available.

The execution of a PTX kernel can be partitioned into three phases: launch configuration, computation, and completion. Trace generators implement event handlers corresponding to each of the three phases. At *launch configuration*, the values of kernel parameters and grid dimensions are known. These are presented to each trace generator along with the internal representation of the PTX kernel to be executed as well as constant references to the entire structure of loaded modules and CUDA-managed resources. The entire state of the application is observable, and

static analysis of the kernel may be completed at this point. Trace generators may, for instance, count the number of static floating-point arithmetic instructions versus memory instructions. As another example, the number of live registers may be counted as well as the number of synchronization points.

During *computation*, the kernel is executed on one of the available Ocelot backends which include GPUs and CPU execution targets. For the purposes of trace generation, only Ocelot's PTX emulator may be configured to generate events during computation. In this case, each instruction is executed by one or more threads within the CTA, and then an object describing the instruction is passed to each active trace generator. This TraceEvent object includes an internal representation of the PTX instruction executed, vector describing the threads that executed, block ID, and memory addresses referenced by the instruction.

Trace generators are notified when the kernel's execution is *complete* presenting an opprotunity to process instruction traces and insert results into a database or some other output mechanism. Each of the existing trace generators distributed with GPU Ocelot uses the Boost serialization library [10] to compile a database of all kernel invocations over all runs of an application enabling results to be gathered over any number of runs of a set of applications. These are post-processed by a collection of analysis tools which load the serialized data issued by the trace generators, compute the metric or metrics of interest, and output results in a form suitable for visualization.

7 Improvements of Performance or Quality

GPU Ocelot is a powerful tool useful for debugging GPU computing applications to ensure correctness and performance as well as characterizing the behaviors of applications. Its functional PTX simulator in particular enables detailed observation of CUDA applications as they are executing and presents an opportunity to identify program errors such as memory faults, race detections, and deadlocks. Additionally, correct applications may be improved by identifying hot paths to focus optimization efforts as well as detect and avoid performance pitfalls such as bank conflicts and poor memory efficiency. In this section, we will discuss several of the facilities within Ocelot to perform this kind of correctness checking and profiling.

7.1 Trace Generation

GPU Ocelot is distributed with a selection of instruction trace generators intended to observe several important characteristics of program behavior. The list of trace generators is summarized in Table 1. These were implemented to ensure program correctness, detect faults at runtime, and compute several application metrics. A complete list of metrics with precise definitions appear in [11] which also contains results gathered from the CUDA SDK examples and Parboil.

7.2 Correctness

`MemoryChecker` compares the address of every load, store, and texture sampling instruction against the set of valid CUDA memory allocations. Addresses that are out of range result in throwing a runtime exception describing the thread ID, address, and program counter of the offending instruction. The following CUDA example illustrates Ocelot's PTX emulator detecting a bad memory reference and throwing a runtime exception identifying the faulting store

Table 1: Trace generators distributed with GPU Ocelot.

Trace Generator	Summary of Functionality
`Branch`	Measures control-flow uniformity and branch divergence
`Instruction`	Measures number of static and dynamic instructions
`KernelDimension`	Measures kernel grid and block dimensions
`MachineAttributes`	Observes and records machine characteristics
`Memory`	Measures working set size, memory intensity, and memory efficiency
`MemoryChecker`	Instruments memory accesses and ensures they map to allocated regions
`MemoryRaceDetector`	Identifies race conditions on shared memory
`Parallelism`	Measures limits of MIMD and SIMD parallelism
`SharedComputation`	Measures extent of data-flow among threads
`WarpSynchronous`	Measures hot paths and regions suitable for warp-synchronous execution
`Watch`	Observes and records reads and writes to regions of memory

instruction, the invalid address, the block and thread ID on which the instruction was executed, and a line number in the original CUDA source file producing the PTX store instruction. Note, the runtime exception also prints the existing allocations that constitute valid memory regions.

```
// file: memoryCheck.cu
__global__ void badMemoryReference(int *A) {
  A[threadIdx.x] = 0;          // line 3 - faulting store
}
int main() {
  int *invalidPtr = 0x0234;    // pointer arbitrarily chosen, not allocated via
                               //  cudaMalloc()
  int *validPtr = 0;
  cudaMalloc((void **)&validPtr, sizeof(int)*64);
  badMemoryReference<<< dim3(1,1), dim3(64, 1) >>>(invalidPtr);
  return 0;
}
```

```
====== Ocelot Runtime Exception ==============
[PC 6] [thread 0] [cta 0] st.global.s32 [%r5 + 0], %r0 -
  Global memory access 0x234 is not within any allocated or mapped range.

All allocations On device Ocelot PTX Emulator:
====== Ocelot global memory allocations ======
=== allocation ===
= 0x1740230 - 0x174032f
= pointer: 0x1740230
= 256 bytes
= device address space: 0
= linear structure, 1D
= pitch: 256
= width: 256
= height: 1
==============================================
Near memoryCheck.cu:3:0
==============================================
```

Cooperative Thread Arrays in CUDA consist of a collection of threads that may share data and synchronize at thread barriers. Threads may be executed serially or concurrently provided they all reach barrier instructions before any thread moves on to the next instruction. CTAs exchanging data among threads through shared memory necessarily must synchronize with barriers to ensure all threads have finished writing to their particular location before a

dependant thread reads the value. Failure to include synchronizations is a common source of transient correctness errors in applications, and the consequences are frequently subtle enough to avoid detection. To avoid such race conditions, Ocelot's `MemoryRaceDetector` optionally annotates each byte of shared memory with the ID of the last thread to write to it. Subsequent loads and stores check to determine whether multiple threads have shared data without an intervening barrier synchronization. If no barrier has been excuted, a runtime exception is thrown indicating a race condition. The following listing demonstrates Ocelot's detection of a race condition in a kernel from Thrust.

```
==Ocelot== Emulator failed to run kernel
"_ZN6thrust6detail6device4cuda6detail23radixSortBlocksKeysOnlyILj4ELj0ELb0ENS3_19modified_
   preprocessINS3_11encode_uintIiEEjEEEEvP5uint4SA_jjT2_"
with exception:
==Ocelot== [PC 114] [thread 0] [cta 0] ld.shared.u32 %r89, [%r63 + 124]
- Shared memory race condition, 0x8c was previously written by thread 31
without a memory barrier in between.
==Ocelot== In
/home/normal/checkout/thrust/thrust/detail/device/cuda/detail/stable_radix_sort.inl:101:0
==Ocelot==
```

Deadlocks in CUDA occur when two threads reach different thread barriers and both necessarily block waiting for the other to resume. Because the PTX emulator attempts to reconverge threads as early as possible, detecting whether a particular barrier synchronization instruction will result in a deadlock is as straightforward as ensuring all threads have reconverged and are active. GPU Ocelot's detailed runtime exception identifies the particular synchronization on which threads have deadlocked and could be used to identify the control paths taken by the diverged threads.

7.3 Performance Tuning

The largest factors affecting performance of GPU applications are related to memory access and control flow uniformity. Memory behavior is impacted by spatial locality and the efficiency in which off-chip bandwidth is utilized. Ocelot measures efficiency by implementing the memory coalescing protocol defined in the CUDA Programming Guide and determining the number of cycles needed to satisfy each memory request. Figure 3 illustrates the average efficiency of loads and stores to global memory relative to peak memory bandwidth. This data was recorded by `MemoryTraceGenerator` over a selection of CUDA applications and offers feedback to developers by identifying which kernels are the most memory intensive and which memory instructions are the least efficient.

Control flow uniformity refers to the fraction of threads that take the same control paths between synchronizations. If all threads of a warp execute the same path, the SIMD units in each multiprocessor execute them concurrently. If threads of the same warp diverge, they must be serialized. In previous work, we define activity factor as the average number of threads that would be executed concurrently on an infinitely wide SIMD machine and provide a trace generator to measure this in actual applications. Figure 4 illustrates activity factor gathered from a selection of CUDA applications by `BranchTraceGenerator` using two different warp reconvergence mechanisms in the PTX emulator. With GPU Ocelot, it is possible to identify which branches are the most divergent.

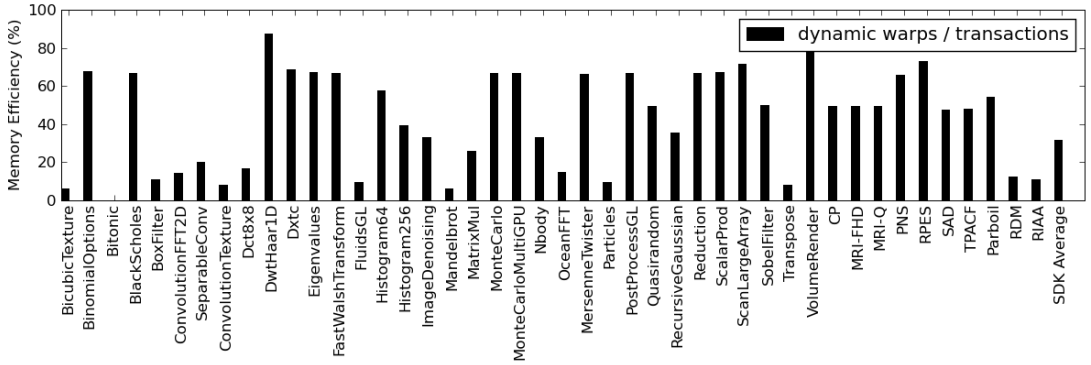

Figure 3: Memory efficiency of CUDA applications.

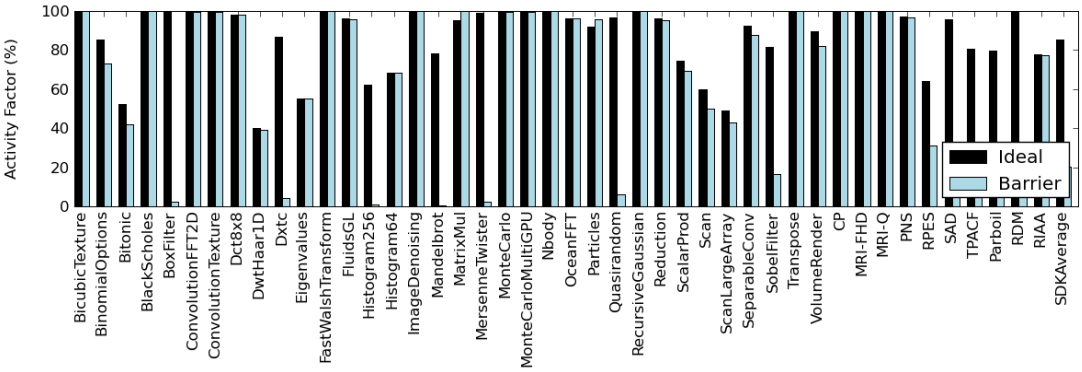

Figure 4: Activity factor for CUDA applications.

7.4 Custom Trace Generation

GPU Ocelot's trace generation framework provides a concise interface in which custom trace generators can be implemented and inserted into CUDA programs to measure application- or research-specific behaviors. `TraceEvent` objects encapsulate all structural information needed to interpret a PTX instruction, and trace generators can inspect the complete state of all CUDA-managed resources as well as control- and data-flow graphs of the kernel being executed. We have used this infrastructure to support our own research efforts [7],[11]. Additionally, Ocelot's trace generation framework has been used to construct instruction traces as inputs to timing models for GPU architecture simulation in [12]. Finally, we are aware of several additional projects using GPU Ocelot to understand memory behavior of CUDA applications and to model the effects of warp formation.

8 Additional Work

GPU Ocelot is actively developed by several core contributors to pursue research on runtimes and dynamic compilation techniques for heterogeneous compute platforms. By the end of May, 2010, we plan to release a version of GPU Ocelot with two additional capabilities: (1) device switching and (2) vectorization for multicore CPU execution targets.

Ocelot is conceived as the dynamic compilation framework in a larger infrastructure for man-

aging execution in a system with several different types of processors. To enable optimizations such as running each kernel on the device where it will execute the fastest, Ocelot will support the ability to migrate state to different execution targets as the application is executing on kernel boundaries. We envision this also being useful to expedite profiling, as all kernels could execute on their fastest processor except a particular kernel of interest which could execute on the PTX emulator to gather detailed results.

To improve utilization on execution targets lacking hardware support for branch divergence, Ocelot's multicore translator is being enhanced to support packing multiple logical threads into a single vectorized control path using techniques discussed in [13]. We expect performance to increase by factors of 2× and more due to both utilizations of vector units in addition to more efficient memory and control flow behavior. Multicore x86 CPUs include support for 4-wide SIMD units via SSE instructions, and together with other processor architectures such as Cell and Larrabbee constitute a collection of vectorizable execution targets. Traditional approaches to parallelizing and vectorizing regions of code have not started from explicitly data-parallel execution paths as available in PTX, and we believe vectorization within Ocelot will achieve superlinear speedup of kernels executing on multicore CPUs.

9 References

[1] G. Diamos, A. Kerr, and S. Yalamanchili. *GPU Ocelot: a Binary Translation Framework for PTX*. Available: `http://code.google.com/p/gpuocelot/`

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

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

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

[5] IMPACT, "The PARBOIL Benchmark Suite," 2007.
Available: `http://www.crhc.uiuc.edu/IMPACT/parboil.php`

[6] Thrust: C++ Template Library for CUDA. Available: `http://code.google.com/p/thrust/`

[7] G. Diamos, A. Kerr, and S. Yalamanchili. "Modeling GPU-CPU Workloads and Systems," *Third Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. Pittsburgh, PA. 2010.

[8] G. Diamos. "Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86." Tech. Rep. 0918.
Available: `http://www.cercs.gatech.edu/tech-reports/tr2009/git-cercs-09-18.pdf`

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

[10] Boost C++ Libraries. Available: `http://www.boost.org`

[11] G. Diamos, A. Kerr, and S. Yalamanchili. "A Characterization and Analysis of PTX Kernels," *International Symposium on Workload Characterization (IISWC'09)*. Austin, TX. 2009.

[12] N. Lakshminarayana and Hyesoon Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," `Workshop on Language, Compiler, and Architecture Support for GPGPU`, Bangalore, India. 2010.

[13] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU Binaries to Tiered SIMD Architectures with Ocelot," Tech. Rep. 0901, January 2009. Available: `http://www.cercs.gatech.edu/tech-reports/tr2009/git-cercs-09-01.pdf`