A Characterization and Analysis of PTX Kernels

Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
arkerr@gatech.edu gregory.diamos@gatech.edu sudha@ece.gatech.edu

***Abstract*—General purpose application development for GPUs (GPGPU) has recently gained momentum as a cost-effective approach for accelerating data- and compute-intensive applications. It has been driven by the introduction of C-based programming environments such as NVIDIA's CUDA [1], OpenCL [2], and Intel's Ct [3]. While significant effort has been focused on developing and evaluating applications and software tools, comparatively little has been devoted to the analysis and characterization of applications to assist future work in compiler optimizations, application re-structuring, and micro-architecture design.**

This paper proposes a set of metrics for GPU workloads and uses these metrics to analyze the behavior of GPU programs. We report on an analysis of over 50 kernels and applications including the full NVIDIA CUDA SDK and UIUC's Parboil Benchmark Suite covering control flow, data flow, parallelism, and memory behavior. The analysis was performed using a full function emulator we developed that implements the NVIDIA virtual machine referred to as PTX (Parallel Thread eXecution architecture) - a machine model and low level virtual ISA that is representative of ISAs for data parallel execution. The emulator can execute compiled kernels from the CUDA compiler, currently supports the full PTX 1.4 specification [4], and has been validated against the full CUDA SDK. The results quantify the importance of optimizations such as those for branch reconvergence, the prevalance of sharing between threads, and highlights opportunities for additional parallelism.

I. Introduction

General purpose application development for GPUs (GPGPU) has recently gained momentum as a cost-effective approach for accelerating data- and compute-intensive applications. It has been pushed to the forefront by the introduction of C-based programming environments for highly data parallel architectures. CUDA for NVIDIA GPUs, OpenCL as an emerging standard [2], and Intel's Ct [3] are at the forefront of these environments. As perhaps the earliest widely available C-based programming environment for commodity GPUs, NVIDIA's CUDA programming model has dominated the GPGPU application landscape having been successfully employed to exploit data parallelism from diverse domains such as signal and image processing [5], database acceleration [6], and financial analysis [7]. Library developers have followed closely behind, with implementations of generic algorithms and data structures on GPUs [8]. Although targeted to NVIDIA's products, the CUDA programming abstractions are representative of the broader range of data parallel applications with single instruction stream multiple data (SIMD) stream execution models.

It is significant that this landscape of applications has been driven by the ready availability of highly parallel SIMD architectures. The NVIDIA GT200 architecture is reported to have up to 30 processors, each supporting 512-way multithreading on 32 SIMD units each with dual issue data paths while the AMD r700 architecture is comprised of up to 60 processors, each with 16 SIMD units of 5 wide VLIW data paths. Figure 1 shows an abstraction of the NVIDIA architecture. Future generations of these processors can be expected to scale the number of processors as well as SIMD width. Central to the efficiency and effectiveness of these future architectures is an understanding of the impact of workload features such as control flow, data sharing patterns, memory referencing behavior, etc., and the use of this information to inform compiler optimizations, application level restructuring, and micro-architecture support.

This paper proposes a set of metrics for data parallel workloads and uses these metrics to analyze the behavior of CUDA programs. We report on a comprehensive analysis of over 50 kernels and applications including the full CUDA SDK covering control flow, data flow, parallelism and memory behavior. We specifically study the impact of some key known optimizations such as branch re-convergence mechanisms and memory read coalescing to assess their importance while also attempting to identify important targets for future optimizations. The goal of this paper is the analysis of the behavior of data parallel application kernels and consequently the emphasis is different than micro-architecture centric studies such as [9], [10] (see Section V). The analysis was performed using a full function emulator we developed that implements the NVIDIA virtual machine referred to as PTX (Parallel Thread eXecution architecture) - a machine model and low level virtual ISA that is translated by the NVIDIA driver to the native GPU ISA at load time. The emulator is based on the development of a set of backend code analysis and binary translation tools for PTX, currently supports the full PTX 1.4 specification [4], and has been validated against the full CUDA SDK. The environment can execute CUDA applications compiled with NVIDIA compilation flow and in fact can transparently replace the GPU device. While we use PTX to leverage the available NVIDIA compilation tool chains, we argue that from the perspective of the application centric characterization performed here, the PTX machine model is general enough to be representative of other SIMD architectures including x86 multicore, Intel GMA, AMD Radeon GPUs, IBM Cell, and others.

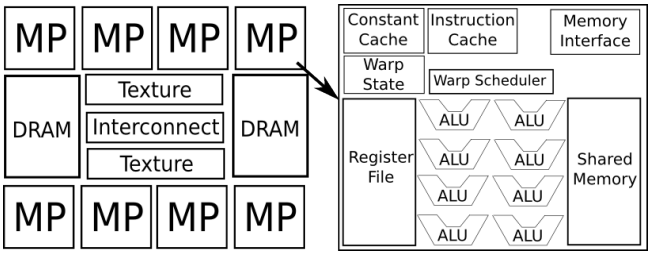

Fig. 1. Abstract CUDA GPU Architecture

The following section provides some important background information on the CUDA programming environment, the PTX ISA, and the design of our emulator. Section III describes the workloads that are analyzed followed by the definition of metrics and the key results of our analysis. The paper concludes with a brief comparison with related work and a summary of anticipated and ongoing work.

II. Analysis Infrastructure

Our analysis infrastructure is comprised of a i) software emulator, ii) run-time, and iii) analysis modules. This infrastructure is part of a larger dynamic translation infrastructure called Ocelot. The simulator and analysis currently supports the parallel thread execution (PTX) virtual ISA of NVIDIA's CUDA programming model and target architectures. This section describes the CUDA environment and architecture, our analysis infrastructure, and the feasible extension to more generic architectures.

A. The CUDA Programming Model

CUDA can be viewed as a set of extensions to the C/C++ programming language that distinguishes between highly multithreaded functions (henceforth *kernels*), and single threaded host functions. Kernels are expressed as SIMD programs and explicitly manage the GPU memory hierarchy. Kernels are managed by a series of API calls defined by the CUDA runtime that allocate GPU memory, copy data between the host and GPU memories, launch kernels, etc. The CUDA compiler compiles kernels to PTX (and possibly several native representations), which are packed into a fat binary structure containing separate entries for each target ISA; the fat binary is stored as a static array along with the native C/C++ code to be executed on the host. As of CUDA 2.2, the PTX textual representation is inlined directly in unicode format in the fat binary structure. The resulting source file is completely C++, which is then passed to a native compiler such as GCC. The generated binary contains references to the CUDA API functions, and must be linked against an implementation of the CUDA runtime to be executed.

B. The PTX Machine Model

Thread Hierarchy. PTX defines a virtual machine model that explicitly expresses single instruction stream multiple data stream (SIMD) and multiple instruction stream multiple data stream (MIMD) style parallelism within an application. The lowest level of the thread hierarchy implements the SIMD model of computation where SIMD threads are grouped together into Cooperative Thread Arrays (CTAs). Each CTA executes on a single processor that is referred to as a multiprocessor (labeled MP in Figure 1). Threads within a CTA can communicate via shared memory and synchronize via barriers. Distinct CTAs can execute on distinct MPs representing a MIMD model of execution. The execution model precludes any ordering constraints between CTAs. At the highest level, a kernel is composed of a set of CTAs that may be executed concurrently and in an unspecified order. As there is no hardware support for synchronization between CTAs within the same kernel invocation, communication must be orchestrated by the application software through global memory.

The memory hierarchy consists of the register file, shared memory, local memory, and global memory in order of increasing access latency. In addition there are read-only caches for constant and texture memory accessible in a MP. Threads within a CTA can communicate via shared memory with barrier synchronization support while communication between CTAs must take place through global memory.

Extending The Model. Though the PTX machine model was developed by NVIDIA for their specific GPUs, it is generic enough to target other parallel architectures. We base this assertion on the following observations:

- The base PTX instruction set is a RISC instruction set closely resembling LLVM [11]. In fact, PTX has equivalents of all of the LLVM instructions except for *switch, unwind, malloc, free, alloca, getelementptr, phi*, and *va_arg*.
- Stratton et al. [12] show how SIMD groups of threads can be compressed into single threaded-loops, enabling an architecture with any SIMD size to execute PTX programs efficiently.
- CTAs are allowed to execute in any order without any synchronization. This allows them to be run in parallel on multicore processors, or serialized on sequential processors without requiring context switching. It also precludes the need for any cache coherence mechanism between cores.

All three major commercial GPU architectures (NVIDIA GT200, AMD r700, and Intel GMA) include SIMD units and support arbitrary control flow, making them natural targets for this model. These architectures are based around large register files and hardware multithreading to hide memory latency. Other architectures such as AMD K10, Intel Nehalem, and Intel Larabee take a different approach. They support a relatively small number of threads in hardware and use caches and prefetching to hide memory latency. Previous work has shown that the PTX machine model can be mapped to this type of architecture as well [12], [13]. Therefore, by analyzing the memory, control flow, and parallel execution behavior of PTX programs, we hope to gain insights into the performance behavior of GPGPU kernels as represented by PTX implementations.

C. Ocelot

Ocelot is an open source infrastructure developed for the purpose of understanding data parallel (GPU) applications by i) driving workload characterization studies, ii) supporting

Building Blocks	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bitonic Sort	1	128	1	1091	176	11
ConvolutionFFT2D	15	85.31	152.6	960129	14681	3
Separable Convolution	4	141.58	304	204480	10112	5
Texture Convolution	2	192	1376	520192	2752	3
Histogram64	2	191.77	119.5	1169535	56969	4
Histogram256	5	189.45	140.8	3309550	293753	5
Matrix Multiply	1	256	40	9760	200	3
Reduction	2	127.89	32.5	41521	4354	3
Scalar Product	1	256	256	148224	16128	6
Scan	3	309.31	1	1455	102	5
Scan Large	10	249.26	8.2	28500	1782	5
Transpose	4	256	4096	802816	24576	2

TABLE I
SDK BUILDING BLOCK STATISTICS

Applications	Kernels	CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bicubic Texture	27	256	1024	222208	5120	3
Binomial Options	1	256	4	725280	68160	8
Black-Scholes Options	1	128	480	3735550	94230	4
Box Filter	3	32	16	1273808	17568	4
DCT	9	70.01	2446	1898752	25600	3
Haar wavelets	2	479.99	2.5	1912	84	5
DXT Compression	1	64	64	673676	28800	8
Eigen Values	3	256	4.33	9163154	834084	13
Fast Walsh Transform	11	389.94	36.8	32752	1216	4
Fluids	4	36.79	32.6	151654	3380	5
Image Denoising	8	64	25	4632200	149400	6
Mandelbrot	2	256	40	6136566	614210	26
Mersenne twister	2	128	32	1552704	47072	7
Monte Carlo Options	2	243.54	96	1173898	76512	8
Threaded Monte Carlo	4	243.54	96	1173898	76512	8
Nbody	1	256	4	82784	1064	5
Ocean	4	64	488.25	390786	17061	7
Particles	16	86.79	29.75	277234	26832	16
Quasirandom	2	278.11	128	3219609	391637	8
Recursive Gaussian	2	78.18	516	3436672	41088	8
Sobel Filter	12	153.68	426.66	2157884	101140	6
Volume Render	1	256	1024	2874424	139061	5

TABLE II
SDK APPLICATION STATISTICS

kernel debugging tools, and iii) driving micro-architectural performance simulators to support compiler-micro-architecture co-design studies [14]. This section describes an addition to the Ocelot code base that we developed to enable PTX emulation.

We have developed a software emulator for PTX that can execute compiled PTX kernels. A PTX assembly file is parsed and analyzed to produce an internal representation of the program stored as a control flow graph. The parser determines the memory requirements of statically declared variables and the requested sizes of GPU specific memory structures before allocating equivalent structures in host memory. A register allocation phase converts from the infinite virtual register set used by PTX to a more manageable size to improve cacheability on the host processor. Our register allocator simply performs linear-scan allocation, creating as many virtual registers as required to eliminate all spills.

Execution of a kernel is handled by issuing CTAs one at a time to the emulator which executes the CTA to completion. The PTX execution model defines a *warp* as a set of threads within a CTA to execute in SIMD fashion, with explicit barrier instructions to synchronize all warps within a CTA. While the number of threads per warp is an architecture-dependent constant, Ocelot is decoupled from particular implementations by setting warp size equal to the CTA size for each kernel. Divergent control flow in which some threads of a warp branch while others fall through is handled using a context stack and predication as described by [15].

Most CUDA applications interleave native code segments with PTX kernel calls. Interoperability with these applications is provided by a runtime component of Ocelot that implements the CUDA runtime API but makes calls into the Ocelot emulation framework rather than dispatching kernel invocations to the GPU driver.

III. WORKLOADS

Using the preceding infrastructure we analyze four major categories of workloads - the CUDA SDK, the UIUC Parboil Benchmark Suite [16], a GPU implementation of the VSIPL API [5], and a retail Risk Inference and Analysis Application that we refer to as RIAA. All workloads were compiled with the NVCC 2.1 CUDA frontend with the GCC 4.3.2 compiler

Parboil Benchmarks	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
CP	10	128	256	430261760	10245120	3
MRI-FHD	7	256	110.571	9272268	198150	5
MRI-Q	4	256	97.5	7289604	393990	5
PNS	112	256	17.85	683056349	33253961	11
RPES	71	64	64768.7	1395694886	95217761	13
SAD	3	61.42	594	4690521	87813	7
TPACF	1	256	201	1582900869	230942677	18

TABLE III
PARBOIL STATISTICS

Workloads	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
SDK	145	217.64	457.25	55884066	3504904	26
RIAA	10	64	16	322952484	23413125	16
RDM	2237	174.558	63.0595	46448530	4082425	6
Parboil	208	177.238	9435.09	4113166257	370339472	11

TABLE IV
WORKLOAD STATISTICS

backend, and the emulated execution results were validated with outputs from the execution of the same binaries on a 280GTX NVDIA processor. In the following tables of results, we present the dynamic SIMD instruction counts for each application as well as the average number of kernels, CTAs, threads, and the maximum divergent context stack size (branch depth).

A. CUDA SDK

The CUDA SDK is comprised of the following groups of applications.

Regression Tests. The regression tests in the CUDA SDK evaluate the correct functionality of the unique features of CUDA. For example, there is a test for alignment of data structures in memory, asynchronous kernel launches and memory operations, GPU performance counters, atomic operations, reduction instructions, OpenGL interoperability, multithreaded host applications, multi-GPU support, and texture interpolation. We support these tests and have validated their execution in Ocelot but omit their results due to space constraints and in deference to application suites.

Building Blocks. These represent efficient implementations of low level primitives such as parallel sorting, reductions, convolution, FFTs, histograms, matrix multiplication, parallel prefix sum, data parallel scalar product, and matrix transpose. These applications represent 12 out of 50 SDK and their basic characteristics are shown in Table I.

Full Applications. The remaining 22 SDK applications represent complete applications and their basic characteristics are shown in Table II. They cover the three prominent stock option pricing algorithms: binomial expansion, Monte Carlo, and Black- Scholes. For image processing, there are examples of bicubic interpolation, DirectX texture compression, box filters, knn denoising, and a Sobel filter. In signal processing, there are implementations of discrete cosine transform, Haar wavelet decomposition, and fast walsh transform. There are several physics simulations for particle dynamics, n-body simulation, turbulent fluid flow, and ocean wave flow. Finally, there are implementations of Mersenne Twister and Niederreiter random number generators.

B. Parboil

Parboil is a GPU benchmark suite written entirely in CUDA with the intent to provide a means for characterizing the performance of GPUs for compute intensive applications [16]. It includes two magnetic resonance imaging applications, a coulombic grid potential application, a sum of absolute difference kernel taken from an H.264 application, a two point angular correlation function kernel, a petri net simulator, and a polynomial equation solver. The characteristics of these applications are shown in Table III.

C. VSIPL

The Vector Signal Image Processing Library (VSIPL) [5] is a standardized object-based signal processing API developed by the DARPA High-Performance Embedded Computing Software Initiative (HPEC-SI) [17]. GPU VSIPL [18] is an implementation of VSIPL targeting NVIDIA GPUs with numeric functions implemented as CUDA kernels. The implementation of this library is distributed with several sample applications including a range-Doppler map implementation, a common front-end processing algorithm for synthetic aperture radar (SAR) systems. This application uses GPU VSIPL's FFT functionality, itself provided by the CUDA FFT library, to process raw SAR data into an image. Here we analyze the Range Doppler Map (RDM) application.

D. RIAA

RIAA is a statistical forecasting application used to predict the default probabilities of loan portfolios owned by specific companies. It uses a Monte Carlo approach where a statistical distribution is sampled to produce inputs to the model subject to the constraints of an individual portfolio. The model conditionally applies a series of analyses to the initial random samples, resulting in code that is both highly data parallel and branch intensive. After the results have been generated for a given set of samples, they are aggregated together to form a statistically significant result.

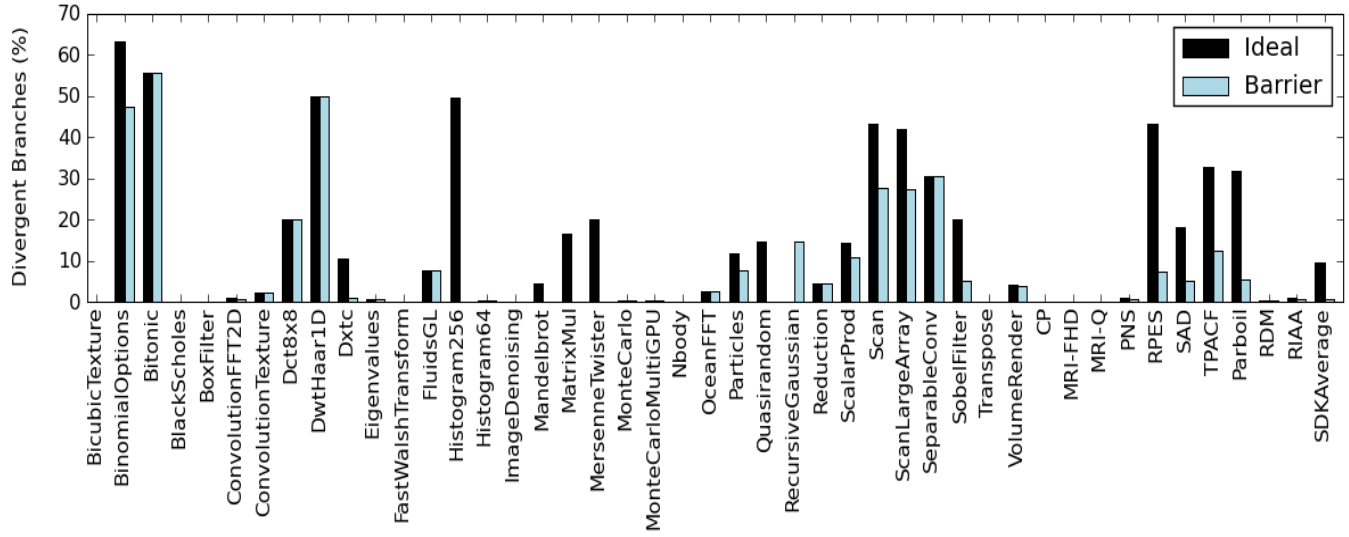

Fig. 2. Ratio of Dynamic Divergent Branches to Total Dynamic Branches

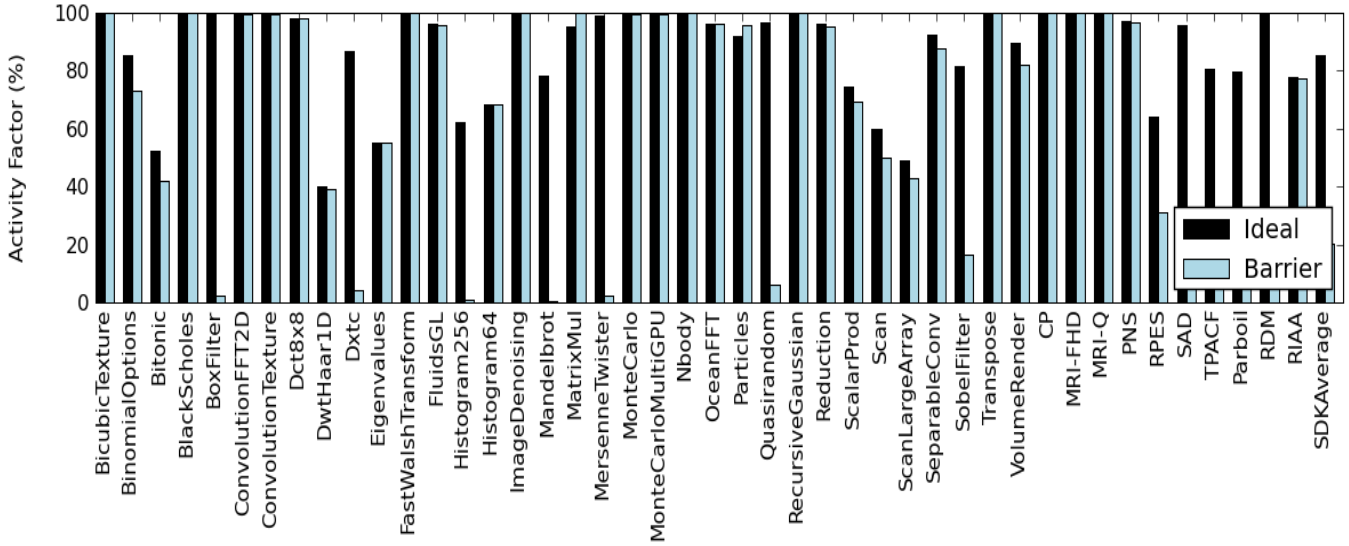

Fig. 3. Activity Factor

IV. Analysis

A. Control Flow

Activity Factor. Instructions are executed in SIMD manner across all threads of a CTA. However, control flow instructions may cause threads of a CTA to diverge, as some threads take a branch while others fall through. We use the term *activity factor* to refer to the average fraction of threads that are active at a given time. It is defined as

$$AF = \frac{1}{N}\sum_{i=1}^{N} active(i) \quad (1)$$

where $active(i)$ is the number of threads executing dynamic instruction i, for N total dynamic instructions. For a kernel with no control flow or control flow in which all threads branch or fall through a branch instruction uniformly, thread activity is 100%.

Divergent Branches. The PTX machine model defines a divergent branch as a branch instruction where some threads within the same warp branch while others fall through. In this study, we report the branch divergence of an application as the ratio of divergent branches to total branches. As control flow divergence results in two separate sets of threads whose execution must be serialized, the location of the synchronization point has a profound impact on thread activity and the number of dynamic instructions executed. In [19], Fung et al. show that the earliest (ideal) location of thread reconvergence is the immediate post dominator of the branch instruction - that is, the nearest successor basic block that *must* be executed if the branch instruction is executed regardless of control path taken. We also investigated an alternative method in which divergent threads are not reconverged until explicit synchronization barriers are encountered by ignoring compiler inserted reconverge points. We denote the former method *ideal*

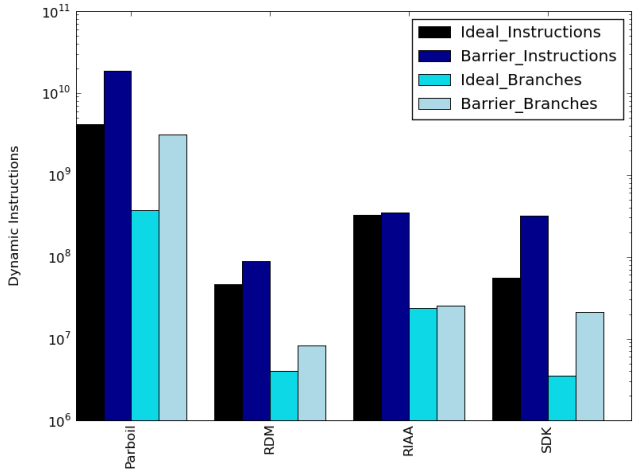

Fig. 4. Dynamic Instruction Count

reconvergence and the latter method *barrier reconvergence*.

Results. Over the entire CUDA SDK, there are about 55 million dynamic instructions executed by each CTA. As each dynamic instruction is executed in SIMD fashion by multiple threads, this count would be much greater if threads were serialized on a single-threaded architecture. Of these dynamic instructions, roughly 6.5% are branches, and of those branches, 9.5% are divergent. Parboil is comparatively a much larger benchmark with over 4 billion dynamic SIMD instructions across the entire suite. The ratio of branch instructions is slightly higher at 9.01%, and the ratio of divergent branchs to branches is significantly higher at 31.7%.

Moving from ideal reconvergence at the immediate post dominator to reconvergence at the next barrier instruction increases the average number of dynamic instructions significantly by a factor of 5.72 for the SDK as shown in Figure 4, which presents dynamic instruction counts using ideal reconvergence as well as barrier reconvergence for each workload. Total branch instructions using both reconvergence algorithms is also presented. The RDM and Parboil workloads have a similar response, increasing by a factor of 1.91 and 3.84 respectively. The RIAA application, on the other hand, only increases slightly by 1.06x. The activity factor is also dramatically impacted as can be seen in Figure 3, dropping from 85.15% to 20.35% across the SDK. When looking at the SDK in detail, some applications like Histogram256 are affected dramatically by the warp convergence mechanism, while others like Bitonic are not affected at all. One explanation is that some applications have very few to no divergent branches and consequently are unaffected by the divergence mechanism. A second explanation is that the code is structured such that the placement of the barrier largely coincides with the behavior of the post dominator scheme. This is the case for RIAA which exhibits substantive control flow divergence but whose performance is minimally impacted by the reconvergence mechanism.

Figure 2 shows that the ratio of divergent branches to total branches decreases by 15x using barrier reconvergence in the CUDA SDK Average suggesting that consecutive branches are correlated, and the ideal reconvergence scheme leads to situations where warps are recombined and then immediately split again. If the programmer has intuition as to how long threads will remain divergent, it would be beneficial to allow them to specify the convergence points if the execution cost of splitting and recombining warps is high. However, as long as the hardware cost of splitting and recombining warps is low, it would seem preferable to converge as soon as possible at the immediate post dominator, since this scheme performs at least as well and often significantly better in terms of activity factor and dynamic instructions as the barrier scheme in all applications that we have examined.

Recommendations. Several applications such as DwtHaar1D, Mandelbrot, and RIAA exhibit control flow behavior that includes handling of special cases that necessarily result in lower thread activity. Similarly, applications such as Histogram256 exhibit correlated branches. In these cases, grouping threads by such affinity (user or compiler optimizations) would be beneficial from an activity factor perspective. For example, the Particles example performs different computations depending on the scene being simulated. Additional support via profiling or prediction coupled with dynamic recompilation or hardware remapping of threads to CTAs (and/or warps) as in [19] would enable efficient use of such SIMD architectures.

B. Memory Behavior

To characterize the memory demand of PTX programs, we define the metric **memory intensity** (I_M) as the ratio of the total number of operations to global memory (M_i) to the number of dynamic instructions (D_i) multiplied by the activity factor (A_f) as in Equation 2. For the purposes of this metric, texture sampling is counted as a load of four 32-bit words. Memory intensity for all workloads is presented in Figure 5.

$$I_M = A_f \times \frac{\sum_{i=0}^{kernels} M_i}{\sum_{i=0}^{kernels} D_i} \tag{2}$$

PTX defines six address spaces: constant, global, local, param, shared, and texture. Constant, param, and texture memory are read-only and backed by a cache [20]. Global memory is the largest block of memory in the memory hierarchy and also that with the largest latency. The PTX memory model enables multiple threads accessing words in the same 64- or 128-byte segment of global memory in the same load or store instruction to coalesce these accesses into a single transaction. This may result in one or two memory operations depending on width of the address bus and size of the transaction. Scatter operations in which each thread accesses a word in a unique segment result in one transaction per segment greatly reducing useful memory bandwidth on real-world platforms.

To characterize spatial locality of operations to global memory, the CUDA memory coalescing protocol for compute capability 1.2 was implemented [20]. This protocol groups operations made by a SIMD load or store instruction into transactions that each access contiguous segments of memory. While Ocelot largely ignores the concept of a warp - a set of consecutive threads that are executed on the hardware's SIMD

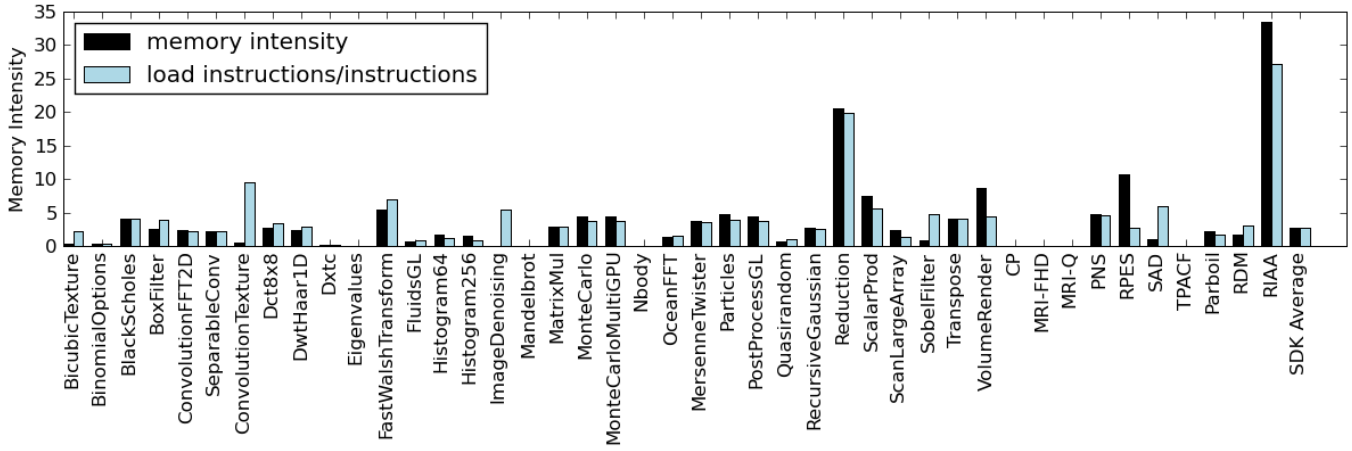

Fig. 5. Memory Intensity

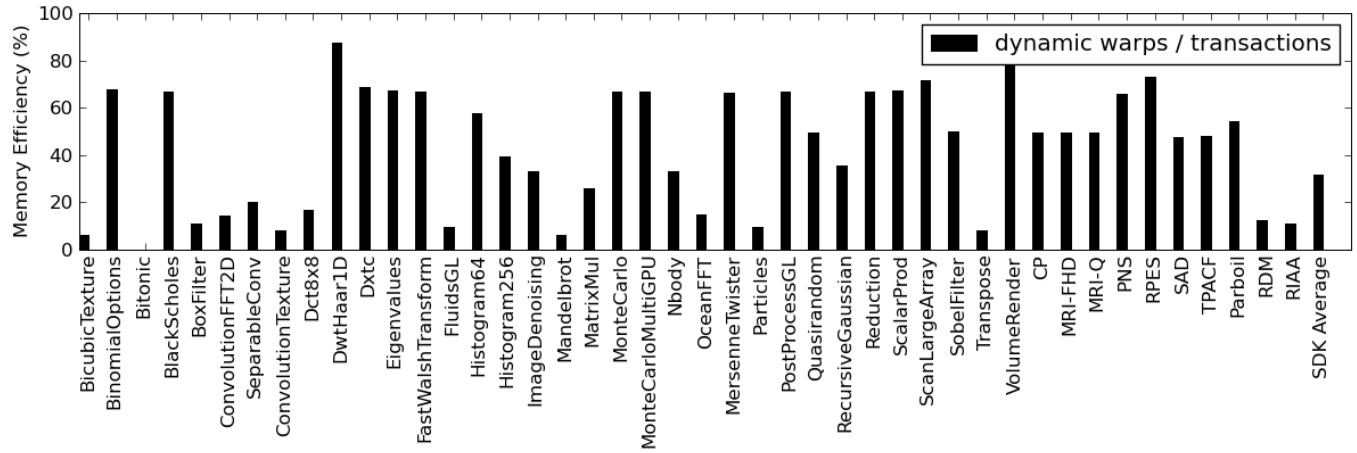

Fig. 6. Memory Efficiency

units concurrently - this metric may only be computed assuming a particular warp size. In this case, we choose 32 threads per warp corresponding to NVIDIA's GT200 architecture. Each memory instructions produces at least two transactions, one for each half-warp. Memory efficiency (E_M) is therefore defined in terms of the number of dynamic warps executing each global memory dynamic instruction (W_i) and the number of memory transactions needed to complete these instructions (T_i) for kernel i. A dynamic warp is a warp containing at least one active thread.

$$E_M = \sum_{i=0}^{kernels} \frac{2W_i}{T_i} \tag{3}$$

Memory bandwidth utilization approaches theoretical peaks as **memory efficiency** approaches 100%. As the number of transactions required to satisfy a warp's loads or stores increases due to scattered access patterns, this ratio decreases. Memory efficiency for the CUDA SDK, Parboil, RDM, and RIAA applications are presented in Figure 6.

Results. Applications in the evaluated workloads exhibit low memory intensity. The CUDA SDK average memory intensity is 2.70%, Parboil's is 1.71%, and the RDM application's is 3.02%. This indicates the possibility they are compute bound. Applications with particularly high memory intensity include several regression tests from the CUDA SDK that intentionally stress the memory system as well as the outlying RIAA application in which memory intensity is 27.1%. The CUDA SDK average memory efficiency of 31.0% indicates, on average, three memory transactions are required to satisfy each memory instruction. Among these workloads, applications with the least memory efficiency tend to be those that rely heavily on texture sampling.

Recommendations. Several applications exhibit low memory efficiency, but high activity factor. In other words, even though threads are executing the same instructions, they are not accessing spatially local data. Previous solutions that focus on hardware memory coalescing can only do so much in these cases as threads within a warp that each access different DRAM pages must issue separate transactions. We are exploring the use of memory profiling of threads and automatically grouping them into warps based on the locality of their memory accesses. Such warp formation must be contrasted with the impact on branch divergence behavior.

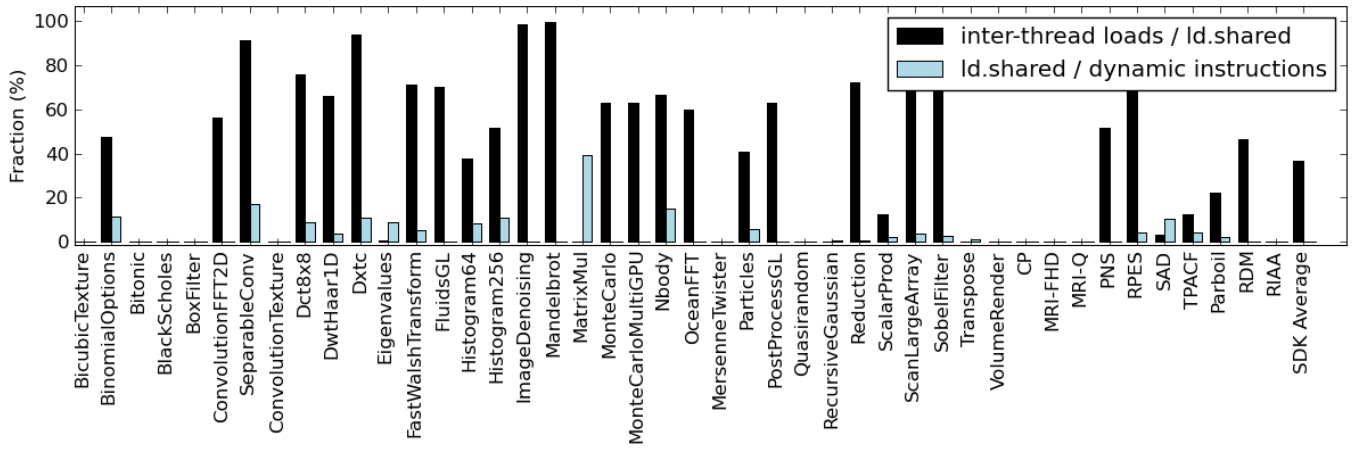

Fig. 7. Inter-thread Data Flow within shared memory.

C. Data Flow

Data Sharing. CTAs as described in II-B have access to a scratchpad memory block known as **shared memory**. Shared memory provides a mechanism through which data may be exchanged among the threads of a CTA. To characterize workloads with producer-consumer behavior among threads of the same CTA, stores to shared memory are tracked to corresponding loads from other threads within the CTA. Each byte of data may be annotated by the ID of the thread that last stored data to that address. Loads from shared memory examine the producing thread ID for that byte, and the load is marked as an inter-thread load if the producing ID is different from the ID of the thread loading the data. The metric we report, **inter-thread data flow**, is the number of inter-thread loads from shared memory relative to the total number of words loaded from shared memory (within the CTA).

To distinguish between true producer-consumer relationships among threads from uses of shared memory as an effective software-managed read-only cache, stores to shared memory do not update the producing thread ID annotation if the register containing the data to store was last written by a load from global memory within the same basic block. This avoids counting uses of shared memory intended to accommodate global memory coalescing and focuses on cases in which the computed results of one thread are consumed by another. This is presented for all workloads in Figure 7. To illustrate the impact of shared loads on the entire kernel, the fraction of shared loads to total dynamic instructions also appears in the figure.

Results. Several applications such as the Monte Carlo simulations (Black-Scholes, MonteCarlo, and RIAA) do not make use of shared memory at all and exhibit no interdependencies among the threads. This suggests that threads may be arbitrarily grouped (into warps) within CTAs for example to improve the activity factor. Other applications such as matrix multiply use shared memory to broadcast data streamed in from global memory and could be accommodated by architectures with data caches. Many applications, however, exhibit a high fraction of inter-thread loads. The CUDA SDK averages 36.7% of inter-thread load instructions reading words produced by other threads. The RDM application is dominated by FFTs with over 45% of shared loads constituting inter-thread data dependencies. The Parboil benchmarks share relatively less data between threads (22.1%), but have a significant fraction of shared loads to total instructions (2.3%).

Recommendations. Many applications in this selection of workloads exchange data between threads requiring synchronization and a conduit for transferring data. Synchronization in the form of PTX barriers introduces overhead for NVIDIA architectures by forcing warps to be removed from the scheduling pool until all other warps in a CTA reach the barrier. In fact, some high performance applications including CUDPP [8] try to reduce these overheads by relying on an implicit barrier between instructions across all threads within a warp – a characteristic of current NVIDIA GPUs, but not of the PTX machine model. This approach breaks compatibility with any architecture whose warp size does not equal that of NVIDIA's (32 threads), but it provides a performance boost on current NVIDIA GPUs.

Providing support for smaller synchronization domains will enable more efficient inter-thread (producer-consumer) data flow - only threads within a domain would be required to wait on a barrier and other threads in a CTA could proceed. Compiler analysis or a local barrier primitive for PTX could allow programmers to express the semantics in a portable way.

D. Parallelism

Unlike most machine models, PTX programs explicitly declare the number of threads and CTAs in the program, statically exposing parallelism in an application. This parallelism allows programs to be scaled to future GPU architectures simply by adding additional cores and SIMD units. For a given set of applications, it is useful to discover the limits of this scalability in order to determine how many MPs (Figure 1) can be used. Towards this end, we define two metrics that capture parallelism in an application, *MIMD parallelism* and *SIMD parallelism*.

MIMD parallelism is computed as the speed up of a GPU with an infinite number of multiprocessors over a GPU with a single multiprocessor, ignoring memory bandwidth and latency

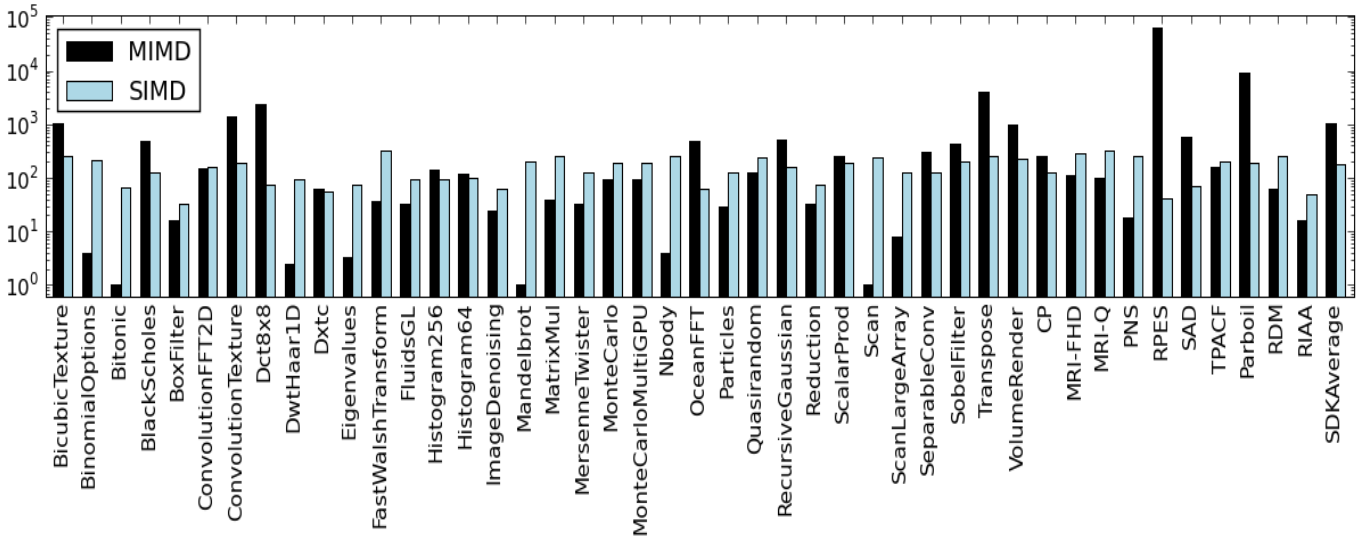

Fig. 8. SIMD and MIMD Parallelism

constraints. It is computed by assuming that each instruction takes a single cycle to complete and dividing the total number of dynamic instructions by the dynamic instruction count (D_i) of the longest running CTA in a kernel as shown in Equation 4. It is averaged over all kernels as in Equation 5. **SIMD parallelism**, on the other hand, is computed as the average activity factor (A_f) of a CTA, multiplied by the number of threads in the CTA, and weighted by the number of dynamic instructions in the CTA as shown in Equation 6. It is averaged over all CTAs in a kernel as in Equation 7.

$$MIMD_{kernel} = \frac{\sum_{i=0}^{ctas} D_i}{\max_{i=0}^{ctas}(D_i)} \tag{4}$$

$$MIMD_{application} = \frac{\sum_{i=0}^{kernels} D_i * MIMD_{kernel\_i}}{\sum_{i=0}^{kernels} D_i} \tag{5}$$

$$SIMD_{kernel} = \frac{\sum_{i=0}^{ctas} A_f * D_i}{\sum_{i=0}^{ctas} D_i} \tag{6}$$

$$SIMD_{application} = \frac{\sum_{i=0}^{kernels} D_i * SIMD_{kernel\_i}}{\sum_{i=0}^{kernels} D_i} \tag{7}$$

Results. The average MIMD parallelism across the SDK is 1076.46, while the average SIMD parallelism is 180.92. This is particularly interesting because it represents a reduced data size for the SDK where many applications are weakly scalable. The Parboil benchmarks, which are significantly larger than the SDK, have a similar SIMD parallelism of 186.96, but an even greater MIMD parallelism of 9427.08. The RDM and RIAA applications have less MIMD parallelism, 62.6 and 15.83 respectively. These applications are also weakly scalable and the lower MIMD parallelism may be an artifact of their higher computational density than many of the SDK examples, several of which simply apply several instruction transformations to each data point. The SIMD parallelism of the SDK, Parboil, and RDM workloads are relatively similar, 180.92, 186.96, and 258.32 respectively, yet strikingly larger than the 49.72 for the RIAA workload. The RIAA workload is limited by a combination of low activity factor and low number of threads per CTA driven by high register usage per thread; the large working set of the application can be seen in the abnormally high memory intensity in Figure 5. Of the 16 CTAs that we did run for this application, the execution time of each CTA was relatively constant resulting in a MIMD parallelism metric of 15.836.

Recommendations. The most important observation that can be drawn from these results is that developers should express PTX programs using as many threads and CTAs as possible to maximize the future scalability of the application. While it is relatively simple for the PTX JIT compiler to serialize CTAs and threads on architectures without many hardware threads [12], [13], it is very difficult to go the other way. A characteristic example can be observed in the Mandelbrot application: in this example, the developer manually compressed the calculation of several pixels into a single thread. This was most likely done to reduce overheads of starting additional CTAs and using more registers on NVIDIA GPUs, but it ends up limiting the amount of MIMD parallelism in the application by 46.875x.

This practice of artificially reducing the parallelism within an application in order to reduce scheduling and synchronization overheads has been observed in other contexts as well. In a study of Intel's threading building blocks library [21], it was observed that adding too many parallel tasks in a black scholes application caused synchronization costs to overwhelm any performance gained from parallelism. The fact that CTAs in PTX can be serialized by the compiler alleviates this problem to some extent, but this requires intelligent compiler heuristics that balance between parallelism and overheads. In general there is unresolved contention between optimizations that reduce overheads and optimizations that improve parallelism; we would like to see this contention addressed in future work.

V. Related Work

GPU Simulation. The closed nature of native GPU instruction sets has resulted in a dearth of architecture simulators in the academic community, compared to simulators for general purpose processors, which are plentiful. In 2007, Fung et al. developed a timing accurate simulator (GPGPU-Sim) for an NVIDIA-like GPU built around the SimpleScalar back end [19]. The core of SimpleScalar was modified to include different modes for simulating native code and GPU kernels and to switch between modes on kernel invocations. Fung et al. used this simulator to evaluate several different schemes for warp formation in NVIDIA-like GPUs.

Recently, this simulator has been extended by Bakhoda et al. to support PTX applications as well including detailed interconnect, cache, and dram models [10]. In their analysis, the authors of GPGPU-Sim compare the performance of 12 benchmark applications on an NVIDIA-like GPU architecture where each GPU core is augmented with a data cache and connected via various network topologies. Consequently, their analysis focuses on the impact of these modifications, for example, network latency sensitivity, dram utilization, and cache effects. In constrast, Ocelot does not include micro-architecture models and focuses on machine independent characteristics that can drive machine dependent optimizations.

Barra. Recently, Collange et al. have developed Barra, a functional simulator for the native NVIDIA G8x and G9x instruction sets [9]. With a reimplementation of the low level CUDA runtime API, they produced a functional simulator capable of running CUDA applications.

Compared to Ocelot, Barra simulates the native GPU instruction set, while Ocelot executes the PTX virtual machine directly. Similarly GPGPU-Sim focuses on the architecture evaluation of NVIDIA-like GPUs, while Ocelot is more concerned with program and compiler analysis; we believe that this distinction makes Barra and GPGPU-Sim more suitable for the evaluation of specific architectural modifications to existing GPUs via the addition of detailed timing models, while Ocelot provides more insight into the high level characteristics of a CUDA application and offers avenues for evaluating applications on architectures other than NVIDIA-specific GPUs.

VI. Conclusions

This paper reports on an infrastructure for the analysis of data parallel applications. The infrastructure currently supports NVIDIA's PTX virtual ISA. It is validated against the full CUDA SDK, a DoD standard signal and image processing library and a computationally intensive risk assessment application. We propose several metrics for characterizing these kernels and report on the analysis of the proposed workloads.

The machine model assumed by PTX presents a unique perspective on parallel programming. Constraints within the programming model actually simplify compiler analysis and reduce runtime overheads. Looking forward, we hope that the insights gained from this analysis of PTX applications combined with similar studies that focus on other parallel systems such as Intel TBB [21] will inform the design of future applications, compilers, and systems.

Acknowledgements

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc., IBM Corp., and NVIDIA Corp. both through research grants, fellowships, as well as technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp. We also thank David Kaeli, Hyesoon Kim, and Nagesh Lakshminarayana for their insightful comments on this paper.

References

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

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

[3] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou, "Ct: A flexible parallel programming model for tera-scale architectures," *Intel Technology Journal*, vol. 11, no. 4, October 2007.

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

[5] D. Schwartz, R. Judd, W. Harrod, and D. Manley, "Vsipl 1.3 api," VSIPL Forum, Tech. Rep., 2008.

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

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

[8] M. Harris, S. Sengupta, and J. Owens, *Parallel Prefix Sum (Scan) in CUDA*. Addison Wesley, 2007.

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

[10] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Boston, MA, USA, April 2009.

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

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

[13] G. Diamos, A. Kerr, and M. Kesavan, "Translating gpu binaries to tiered simd architectures with ocelot," Tech. Rep. 0901, January 2009. [Online]. Available: http://www.cercs.gatech.edu/tech-reports/tr2009/abstracts/01.html

[14] G. Diamos, A. Kerr, and S. Yalamanchili, "Gpuocelot: A binary translation framework for ptx." June 2009, `http://code.google.com/p/gpuocelot/`.

[15] J. S. Sven Woop and P. Slusallek, "Rpu: A programmable ray processing unit for realtime ray tracing," in *Proceedings of ACM SIGGRAPH 2005*, July 2005. [Online]. Available: http://www.saarcor.de/

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

[17] D. Campbell, "The high performance embedded computing software initiative: C++ and parallelism extensions to the vector, signal, and image processing library standard," 2005, pp. 278–283.

[18] A. Kerr, D. Campbell, and M. Richards, "Gpu vsipl: High-performance vsipl implementation for gpus," in *HPEC'08: High Performance Embedded Computing Workshop*, Lexington, MA, USA, 2008.

[19] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420.

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

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