Modeling GPU-CPU Workloads and Systems

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

ABSTRACT

Heterogeneous systems, systems with multiple processors tailored for specialized tasks, are challenging programming environments. While it may be possible for domain experts to optimize a high performance application for a very specific and well documented system, it may not perform as well or even function on a different system. Developers who have less experience with either the application domain or the system architecture may devote a significant effort to writing a program that merely functions correctly. We believe that a comprehensive analysis and modeling framework is necessary to ease application development and automate program optimization on heterogeneous platforms.

This paper reports on an empirical evaluation of 25 CUDA applications on four GPUs and three CPUs, leveraging the Ocelot dynamic compiler infrastructure which can execute and instrument the same CUDA applications on either target. Using a combination of instrumentation and statistical analysis, we record 37 different metrics for each application and use them to derive relationships between program behavior and performance on heterogeneous processors. These relationships are then fed into a modeling framework that attempts to predict the performance of similar classes of applications on different processors. Most significantly, this study identifies several non-intuitive relationships between program characteristics and demonstrates that it is possible to accurately model CUDA kernel performance using only metrics that are available before a kernel is executed.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Heterogeneous (hybrid) systems; D.3.4 [**Programming Languages**]: Retargetable compilers; D.2.4 [**Software Engineering**]: Statistical methods

General Terms

Performance, Design, Measurement

Keywords

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

GPGPU-3 March 14, 2010 Pittsburgh, PA, USA

1. INTRODUCTION

As programming models such as NVIDIA's CUDA [17] and the industry-wide standard OpenCL [8] gain wider acceptance, efficiently executing the expression of a data-parallel application on parallel architectures with dissimilar performance characteristics such as multi-core superscalar processors or massively parallel graphics processing units (GPUs) becomes increasingly critical. While considerable efforts have been spent optimizing and benchmarking applications intended for processors with several cores, comparatively less effort has been spent evaluating the characteristics and performance of applications capable of executing efficiently on both GPUs and CPUs.

As more applications are written from the ground up to perform efficiently on systems with a diverse set of available processors, choosing the architecture capable of executing each kernel of the application most efficiently becomes more important in maximizing overall throughput and power efficiency. At the time of this writing, we are not aware of any study that correlates detailed application characteristics with actual performance measured on real-world architectures.

Consequently, this paper makes the following contributions:

- We identify several PTX application characteristics that indicate relative performance on GPUs and CPUs.
- We use a statistical data analysis methodology based on principal component analysis to identify critical program characteristics.
- We introduce a model for predicting relative performance when the application is run on a CPU or a GPU.

The analysis and models presented in this paper leverage the Ocelot framework for instrumenting data-parallel applications and executing them on heterogeneous platforms. The Ocelot framework [13] is an emulation and compilation infrastructure that implements the CUDA Runtime API and either (1) emulates executing kernels, (2) translates kernels to the CPU ISA, or (3) emits the kernel representation to the CUDA driver for execution on attached GPUs. Ocelot is uniquely leveraged to gather instruction and memory traces from emulated kernels in unmodified CUDA applications, analyze control and data dependencies, and execute the kernel efficiently on both CUDA-capable GPUs and multicore CPUs. Consequently, in addition to enabling detailed and comparative workload characterizations, the infrastructures enables transparently portability of PTX kernels across CPUs and NVIDIA GPUs.

2. RELATED WORK

Analytical GPU models. Hong et. al. [10] propose a predictive analytical performance model for GPUs. The main components of their model are memory parallelism among concurrent warps and computational parallelism. By tuning their model to machine parameters, static characteristics of applications, and regressions, their performance model predicts kernel runtimes with errors of 13% or less. Our approach, on the other hand, does not assume particular principal components and instead attempts to determine them based on measurable statistics that may change substantially with the evolution of GPU and CPU micro-architecture.

MCUDA. MCUDA [19] defines a high-level source-to-source translator from CUDA to the C programming language. Blocks delimited by synchronization points are identified in the CUDA source, and thread loops are placed around them spilling live values. Like Ocelot, a software runtime dispatches CTAs to host hardware threads and implements support for texture sampling and special functions. MCUDA, however, requires applications to be recompiled from source and is not flexible to alternative methods. Further, it is a source-to-source translation framework which has its advantages, but cannot address the detailed ISA level characterizations and insights supported by Ocelot.

GPU-Simulators. Like Ocelot, Barra [4] and GPGPU-Sim [1] provide micro-architecture simulation to CUDA programs. Similar to Ocelot, these simulators intercept GPU kernel invocations and execute them instead on a functional simulator. They are primarily concerned with reproducing the executions of kernels on actual GPUs, and characteristics derived from such simulations are influenced by it. Results from Ocelot are decoupled from particular GPU implementations excepting the memory efficiency metric and provide insights into application behavior. Further, Ocelot provides a high-performance execution path to multicore CPUs and NVIDIA GPUs thereby offering opportunities to speedup applications by selecting the appropriate execution target.

PLANG / NVIDIA PTX to LLVM. NVIDIA has presented internal work translating PTX to LLVM [9] at the LLVM Developer's Workshop. This largely matches the aims of Ocelot to provide a high-performance execution path for CUDA to multicore CPUs. However, details are not public making comparisons difficult. Further, Ocelot provides and exposes an integrated emulator enabling detailed analysis via tool interfaces.

3. BACKGROUND

3.1 PTX

NVIDIA's Parallel Thread eXecution (PTX) [16] is a virtual instruction set architecture with explicit data-parallel execution semantics that are well-suited to NVIDIA's GPUs. PTX is composed of a set of RISC-like instructions for explicitly-typed arithmetic computations, loads and stores to a set of address spaces, instructions for parallelism and synchronization, and built-in variables. Functions implemented in PTX, known as *kernels* are intended to be executed by a large grid of threads arranged hierarchically into a *grid* of *cooperative thread arrays* (CTAs). The PTX thread hierarchy is illustrated in Figure 1. CTAs in this grid may be executed concurrently or serially with an unspecified mapping to physical processors. Threads *within* a CTA are assumed to be executing on a single processor and may synchronize at programmer-inserted barriers within the kernel. Data may be exchanged between the threads of a CTA by reading and writing to a shared scratchpad memory or by loading and storing to locations in global memory off chip; in either case, a barrier synchronization must be used to force all threads to complete their store operations before loads to the same location may be issued or the outcome of the resulting race condition is undefined.

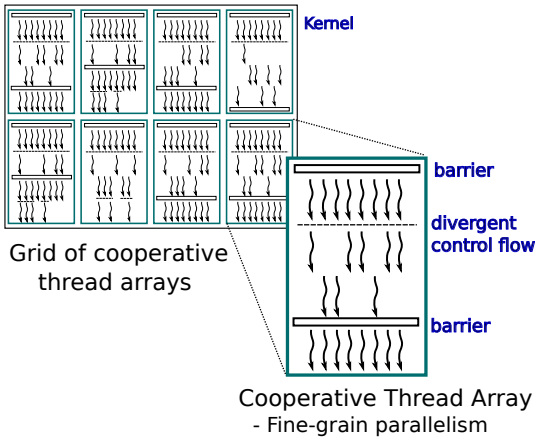

Figure 1: PTX thread hierarchy.

The PTX execution model permits the serialized execution of CTAs to avoid an overwhelmingly large explosion of state as, for example, a grid of thousands of CTAs are launched each with hundreds of threads resulting in possibly hundreds of megabytes of live state at any point in the kernel if all CTAs were executed concurrently. Coarse-grain synchronization among CTAs of a kernel is defined only at the end of a kernel.

Applications that efficiently target this execution model explicitly declare high levels of parallelism as well as localized synchronization and communication domains. Consequently, they may be efficiently scaled to architectures with varying numbers of processors.

3.2 Ocelot Infrastructure

The Ocelot compiler infrastructure strives to decouple CUDA applications from GPUs by wrapping CUDA Runtime API [17], parsing kernels stored as bytecode within the application into an internal representation, executing these kernels on devices present, and maintaining a complete list of CUDA memory allocations that store state on the GPU. By decoupling a CUDA application from the CUDA driver, Ocelot provides a framework for simulating GPUs, gathering performance metrics, translating kernels to architectures other than GPUs, instrumenting kernels, and optimizing kernels for execution on the GPU.

Ocelot's translation framework provides efficient execution of CUDA kernels on multi-core CPUs by first translating the kernel from PTX to the native instruction set of the Low-Level Virtual Machine (LLVM) [14] infrastructure then applying a series of transformations that implement the PTX execution model discussed in Section 3.1 with control and data structures available to typical scalar microprocessors. The complete translation process is beyond the scope of this paper; we only include a brief description of

the process as it pertains to the analysis methodology used in the following sections. A more detailed overview of the translation process is covered in our prior work[6].

4. TRANSLATION

4.1 LLVM

The Low Level Virtual Machine (LLVM) [14] is a maturing compiler infrastructure that maintains a strongly-typed program representation of that program throughout its lifetime. Multiple back-end code generators exist to translate LLVM IR to various popular instruction set architectures including x86 and x86-64. LLVM's IR itself includes explicit load and store instructions, integer and floating-point arithmetic, binary operators, and control flow operators. LLVM includes optimization passes that apply transformations to this intermediate form including well-known compiler optimizations such as common subexpression elimination, dead code removal, and constant propagation. By translating from one intermediate form to LLVM's IR and then leveraging LLVM's existing optimization and code generation components, a developer may construct a complete path to native execution on popular CPU architectures.

In Figure 2, an example kernel expressed in CUDA is first compiled by NVIDIA's CUDA compiler (nvcc) producing a PTX representation which is then translated by Ocelot's LLVM Translation framework yielding the kernel on the right side of the figure. This is an LLVM representation of the kernel that could be executed by one host thread for each thread in the CTA and correctly execute the kernel. Note the runtime support structure providing block and thread ID.

4.2 Execution Model Translation

Compiling a PTX kernel for execution on multicore CPUs requires first translating the kernel to the desired instruction set then transforming the kernel's execution semantics from PTX's thread hierarchy to a single host thread. Ocelot could conceivably execute a PTX kernel using the LLVM translation by launching one kernel-level host thread per thread in the CTA and relying on OS-level support for barriers and multi-threading to provide concurrency and context switching. This approach is similar to CUDA emulation mode except the underlying kernel representation is the same PTX representation that would be executed on the GPU. However, CUDA programs perform efficiently on GPUs when many light-weight threads are launched to hide latencies in the memory structure.

The strategy adopted here to map the thousands of logical CUDA threads onto a few host threads is a compile-time transformation that inserts procedures to perform light-weight context switching at synchronization barriers within the kernel, similar to thread fusion described in [19]. A scheduler basic block is inserted at the entry point of the kernel that selects branch targets depending on the currently selected thread and its progress through the kernel. When a thread reaches a synchronization point, control jumps back to the scheduler which stores live variables that have been written, updates the resume point, increments the thread ID, and jumps to the new current thread's previous instruction. Live variables are loaded as needed and execution continues.

This method does not require function calls or heavy-weight context switches. Thread creation at the start of the CTA incurs no incremental overhead other than the fixed costs of allocating shared and local memory which each worker thread completes once when a kernel is launched. Context switches require spilling live state and an indirect branch which is likely to be predicted correctly much of the time.

Some CUDA programs are written in a manner that assumes the warp size for the executing processor is at least a particular value, typically 32 threads for current CUDA GPU architectures. Whille this assumption is stronger than what the execution model guarantees, existing architectures execute such kernels correctly, and the additional clock cycles needed to execute synchronization instructions may be saved. The PTX emulator assumes the warp size is equal to the number of threads in the CTA and executes each instruction for as many threads as possible, so such applications work correctly without modification. The multicore execution model translation technique described here assumes warp size is equal to 1 thread, so applications must be recompiled with synchronization points following those statements expected to be executed simultaneously on architectures with a larger warp size.

4.3 CTA Runtime Support

When an application launches a kernel, a multi-threaded runtime layer launches as many worker threads as there are hardware threads available in addition to a context data structure per thread. This context consists of a block of shared memory, a block of local memory used for register spills, and special registers. Worker threads then iterate over the blocks of the kernel grid and each executes block as a CTA. The execution model permits any ordering of CTAs and any mapping to concurrent worker threads.

PTX defines several instructions which require special handling by the runtime. PTX compute capability 1.1 introduces atomic global memory operators which implement primitive transactional operations such as exchange, compare and swap, add, increment, and max, to name a few. Because global memory is inconsistent until a kernel terminates, we faced several options for implementing atomic accesses. The simplest option places a global lock around global memory. GPUs typically provide hardware support for texture sampling and filtering. PTX defines a texture sampling instruction which samples a bound texture and optionally performs interpolation depending on the GPU driver state. As CPUs do not provide hardware support for texture sampling, Ocelot performans nearest and bilinear interpolation in softwar by translating PTX instructions into function calls which examine internal Ocelot data structures to identify mapped textures, compute addresses of referenced samples, and interpolate accordingly.

Additionally, PTX includes several other instructions that do not have trivial mappings to LLVM instructions. These include transcendental operators such as *cos* and *sin* as well as parallel reduction. These too are implemented by calls into the Ocelot runtime which in turn calls C standard library functions in the case of the floating-point transcendentals. Reductions are defined for a particular warp size; in the translation to multicore, the warp size is a single thread, so they are reducible to a mov instruction.

5. CHARACTERIZATION METHODOLOGY

5.1 Metrics and Statistics

Ocelot's PTX emulator may be instrumented with a set of user-supplied event handlers to generate detailed traces

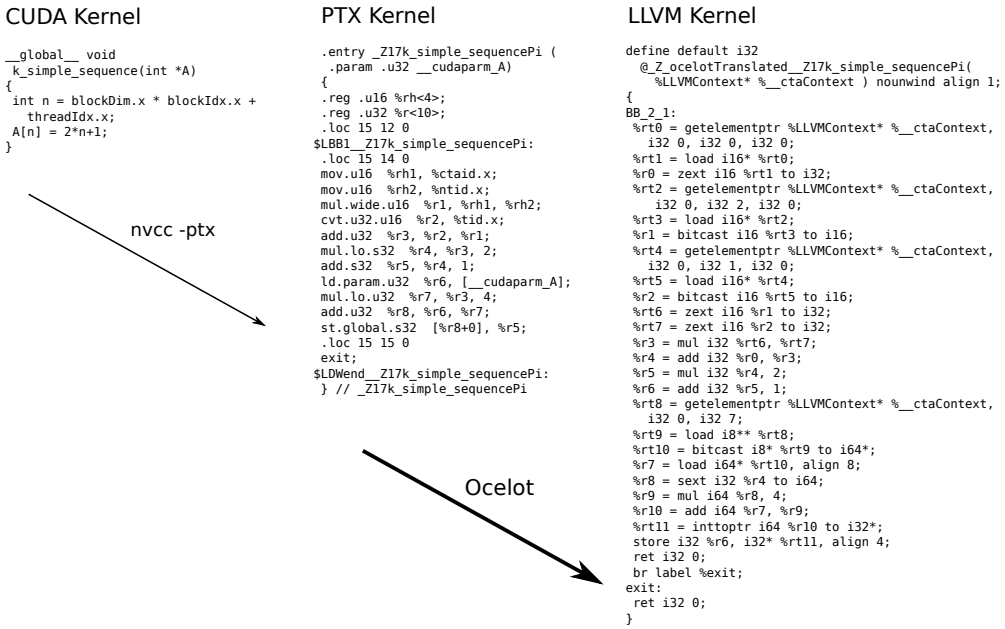

Figure 2: PTX to LLVM translation.

of instructions and memory references. After each dynamic PTX instruction is completed for a given program counter and set of active threads, an event object containing program counter, PTX instruction, activity mask, and referenced memory addresses is dispatched to each registered trace generator which handles the event according to the performance metric it implements. We present the following application metrics gathered in this manner building on the set of metrics defined in our previous work[13]:

Activity Factor. Any given instruction is executed by all threads in a warp. However, individual threads can be predicated off via explicit predicate registers or as a result of branch divergence. Activity factor is the fraction of threads active averaged over all dynamic instructions.

Branch Divergence. When a warp reaches a branch instruction, all threads may branch or fall through, or the warp may diverge in which the warp is split with some threads falling through and other threads branching. Branch Divergence is the fraction of branches that result in divergence averaged over all dynamic branch instructions.

Instruction Counts. These metric count the number of dynamic instructions binned according to the functional unit that would execute them on a hypothetical GPU. The functional units considered here include integer arithmetic, floating-point arithmetic, logical operations, control-flow, off-chip loads and stores, parallelism and synchronizations, special and transcendental, and data type conversions.

Inter-thread Data Flow. The PTX execution model includes synchronization instructions and shared data storage accessible by threads of the same CTA. Interthread data flow measures the fraction of loads from shared memory such that the data loaded was computed by another thread within the CTA. This is a measure of producer-consumer relationships among threads.

Memory Intensity. Memory intensity computes the fraction of instructions resulting in communication to off-chip memory. These may be explicit loads or stores to global or local memory, or they may be texture sampling instructions. This metric does not model the texture caches which are present in most GPUs and counts texture samples as loads to global memory.

Memory Efficiency. Loads and stores to global memory may reference arbitrary locations. However, if threads of the same warp access locations in the same block of memory, the operation may be completed in a single memory transaction; otherwise, transactions are serialized. This metric expresses the minimum number of transactions needed to satisfy every dynamic load or store divided by the actual number of transactions, computed according to the memory coalescing protocol defined in [17] §5.1.2.1. This is a measure of spatial locality.

Memory Extent. This metric uses pointer analysis to compute the working set of kernels as the number and layout of all reachable pages in all memory spaces. It represents the total amount of memory that is accessible to a kernel immediately before it is executed.

Context Switch Points. CTAs may synchronize threads at the start and end of kernels as well as within sections of code with uniform control flow, typically to ensure shared memory is consistent when sharing data. Each synchronization requires a context switch point inserted by Ocelot during translation for execution on multicore as described in[6].

Live Registers. Unlike CPUs, GPUs are equipped with large register files that may store tens of live values per thread. Consequently, executing CTAs on a multicore x86 CPU requires spilling values at context switches. This metric expresses the average number of spilled values.

Machine Parameters. GPUs and CPUs considered here are characterized by clock rate, number of concurrent threads, number of cores, off-chip bandwidth, number of memory controllers, instruction issue width, L2 cache capacity, whether they are capable of executing out-of-order, and the maximum number of threads within a warp.

Registers per Thread. The large register files of GPUs may be partitioned into threads at runtime according to the

Application	Full Name	Source
MRI-Q	Magnetic Resonance Imaging	Parboil
MRI-FHD	Magnetic Resonance Imaging	
CP	Coulombic Potential	
SAD	Sum of Absolute Differences	
TPACF	Two-Point Angular Correction	
PNS	Petri Net Simulation	
RPES	Rys Polynomial Equation Solver	
hotspot	Thermal simulation	Rodinia
lu	Dense LU Decomposition	
nbody	Particle simulation	CUDA SDK

Table 1: Benchmark Applications.

number of threads per CTA. Larger numbers of threads increases the ability to hide latencies but reduces the number of registers available per thread. On CPUs, these may be spilled to local memory. This metric expresses the average number of registers allocated per thread.

Kernel Count. The number of times an application launches a kernel indicates the number of global barriers across all CTAs required.

Parallelism Scalability. This metric determines the maximum amount of SIMD and MIMD parallelism[13] available in a particular application averaged across all kernels.

DMA Transfer Size. CUDA applications explicitly copy buffers of data to and from GPU memory before kernels may be called incurring a latency and bandwidth constrained transfer via the PCI Express bus of the given platform. We measure both the number of DMAs and the total amount of data transferred.

5.2 Benchmarks

For this study, we selected applications from existing benchmark suites. PARBOIL [11] consists of seven application-level benchmarks written in CUDA that perform a variety of computations including ray tracing, finite-difference time-domain simulation, sorting. Rodinia [3] is a separate collection of applications for benchmarking GPU systems. Finally, the CUDA SDK is distributed with over fifty applications showcasing CUDA features. A list of the applications we selected appears in Table 1.

Kernels from these applications were executed on processors whose parameters are summarized in Table 2. This selection consists of both CPUs and GPUs that together offer a wide range for each of the listed parameters. We expect these parameters that capture clock frequency, issue width, concurrency, memory bandwidth, and cache structure to sufficiently model the performance of kernels from the benchmark applications.

We chose to characterize PTX applications by the collection of statistics listed in Table 3. These may be classified according to the way they are gathered. Some quantities may be determined via *static analysis* before a kernel is executed such as static instruction counts of each kernel, the number/size of DMA operations initiated before a kernel launch, as well as upper bounds on working set size determined by conservative pointer analysis. Others may be determined at runtime by inserting *instrumentation* into kernels and recording averages as they execute; these include SIMD and MIMD parallelism metrics. Finally, some metrics – typically dynamic instruction counts – may only be determined by executing the kernel to completion via PTX *emulation* and analyzing the resulting instruction traces. Note that all of these metrics were collected via execution

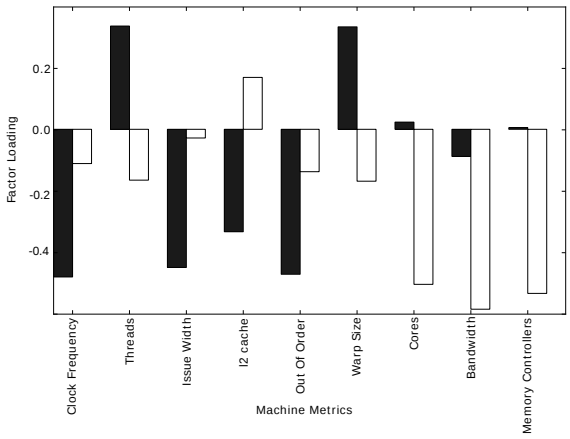

Figure 3: Factor loadings for two machine principal components. PC0 (black) corresponds to single core performance, while PC1 (white) corresponds to multi-core throughput.

on the Ocelot PTX Emulator, therefore, they are independent of the micro-architecture of a particular CPU or GPU.

Table 4 lists the quantitative results from each Parboil application collected for the statistics listed in Table 3. Our analysis also covered the Rodinia benchmarks and the CUDA SDK. While we include some of that analysis in the discussions the detailed results are omitted due to space constraints.

6. RESULTS

Our methodology for modeling the interaction between machine and program characteristics uses principal component analysis to identify independent parameters, similar to [7], cluster analysis to discovers sets of related applications, and multivariate regression combined with projections onto convex sets[2] to build predictive models from principal components.

6.1 Principal Component Analysis

Principal Component Analysis (PCA) is predicated on the assumption that several variables used in an analysis are correlated, and therefore measure the same property of an application or processor. PCA derives a set of new variables, called principal components, from linear combinations of the original variables such that there is no correlation between any of the new variables. PCA identifies the new variables with the most information about the original data thereby reducing the total number of variables needed to represent a data set. In our analysis, we use a normalized PCA (zero mean, unit variance) because each of our original metrics are expressed using different units. We choose enough principal components to account for at least 85% of the variance in the original data.

Once PCA has identified a set of principal components, we apply a varimax rotation[15] to the principal components. This distributes the contribution of each original variable to each principal component, such that each original variable either strongly impacts a principal component or it very weakly impacts it. In other words, it causes each original metric to influence a single principal component, easing analysis of the data.

	Nehalem	Atom	Phenom	8600 GS	8800 GTX	GTX280	C1060
Type	Out-of-order CPU	In-order CPU	Out-of-order CPU	In-order GPU	In-order GPU	In-order GPU	In-order GPU
Issue Width	4	2	3	1	1	1	1
Clock Frequency (GHz)	2.6	1.6	2.2	1.2	1.5	1.3	1.3
Hardware Threads per Core	2	2	1	24	24	24	24
Cores	4	1	4	2	16	30	30
Warp Size	1	1	1	32	32	32	32
Memory Controllers	3	1	2	2	6	8	8
Bandwidth per Controller (GB/s)	8.53	3.54	8.53	5.6	14.3	17.62	12.75
L2 Cache (kB)	512	512	512	0	0	0	0

Table 2: Machine parameters.

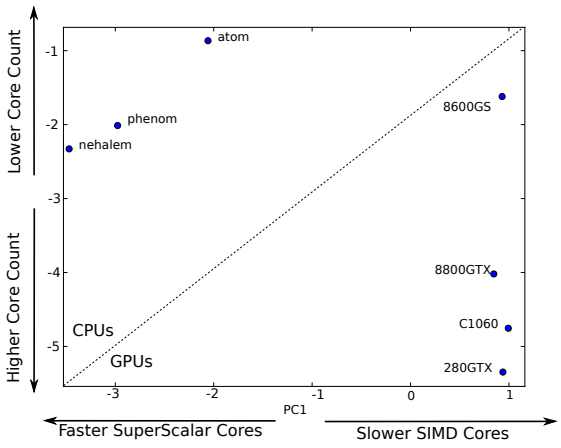

Figure 4: The machine principal components. GPUs have high core counts and slow SIMD cores while CPUs have fewer, but faster, cores.

For the statistics gathered in the previous section, we perform two separate PCAs: one which only includes application statistics and another which only includes machine statistics. This is valid because the metrics were collected via the Ocelot PTX emulator, which is architecture agnostic.

6.2 Machine Principal Components

From the set of machine statistics, PCA yielded two principal components that are shown in terms of factor loadings in Figure 3 and plotted in Figure 4. Clusters reiterate the few number of high-speed cores among the CPUs and a much larger number of lower-speed cores among the GPUs.

PC0: Single Core Performance. The variables that contribute strongly to the first principal component are shown in the left of Figure 3. Note that all of these metrics, clock frequency, issue width, cache size, etc correspond to the performance of a single processor core. Additionally, note that threads-per-core and warp size are negatively correlated with clock frequency, issue width, and out of order, highlighting the differences between GPU and CPU design philosophies.

PC1: Core and Memory Controller Count. The second PC illustrates that the core count is correlated with the memory controller count and memory bandwidth per channel, indicating that multi-core CPUs and GPUs are designed such that the off-chip bandwidth scales with the number of cores.

Discussion. Though the intent of this paper is to derive relationships between these metrics and the performance of machine-application combinations, this analysis also exposes trends in the way that CPUs and GPUs are designed. The division of the machine metrics into these two principal components can be explained as follows: the design of a single core typically does not influence the synthesis of many single cores and memory controllers into a multi-core processor. The performance of a single core in a processor is either characterized by clock frequency, cache size, superscalar width, etc or a high degree of hardware multithreading and large SIMD units. Figure 4 shows a clear distinction between CPU and GPU architectures.

This classification holds even for processors not included in this study. For example, recently released GPUs by Intel[12] and AMD can both be characterized by a large number of threads per core and wide SIMD units; even embedded CPUs such as ARM Cortex A9 [5] have begun to move towards out of order execution.

6.3 Application Components

The PCA of the application statistics yielded five principal components, the factor loadings of which are shown in Figure 5. We would like to note that PCA reveals relationships that hold only for a given set of data, in this case, the applications that were chosen. Given a different set of applications, PCA may reveal a different set of relationships. However, the fact that these trends are valid across application from both Parboil and Rodinia, which are designed to be representative of CUDA applications, indicates that they may represent fundamental similarities in the way that developers write CUDA programs.

PC0: MIMD Parallelism. The first principal component is composed of metrics that are related to the MIMD parallelism of a program. Recall that MIMD parallelism measures the speedup of a kernel on an idealized GPU with an infinite number of cores and zero memory latency. It is bound by the number of CTAs in each kernel. The correlation between MIMD parallelism and DMA Size indicates that applications that copy a larger amount of memory to the GPU will also launch a large number of CTAs. It is interesting to note that during our preliminary evaluation which only included the Parboil benchmarks shown in Table 4, this component also included the majority of dynamic instruction counts. Adding the Rodinia *hotspot* application and the SDK *nbody* application broke the relationship between MIMD parallelism and problem size, indicating that not all CUDA applications are weakly scalable. This distinction motivates the need for the cluster analysis in the next section, where applications with similar characteristics can be identified and modeled separately. As a final point, notice that several static instruction counts are highly cor-

Metric	Units	Description	Collection method
Extent_of_Memory	bytes	Size of working set	static analysis
Context_switches	switch points	Number of thread context switch points	static analysis
Live_Registers	registers	Number registers spilled at context switch points	static analysis
Registers_Per_Thread	registers	Number of registers per thread	static analysis
DMAs	transfers	Number of transfers between GPU memory	static analysis
Static_Integer_arithmetic	instructions	Number of integer arithmetic instructions	static analysis
Static_Integer_logical	instructions	Number of logical instructions	static analysis
Static_Integer_comparison	instructions	Number of integer compare instructions	static analysis
Static_Memory_offchip	instructions	Number of off-chip memory transfer instructions	static analysis
Static_Memory_onchip	instructions	Number of on-chip memory transfer instructions	static analysis
Static_Control	instructions	Number of control-flow instructions	static analysis
Static_Parallelism	instructions	Number of parallelism instructions	static analysis
Dynamic_Integer_arithmetic	instructions	Number of executed integer arithmetic instructions	emulation
Dynamic_Integer_logical	instructions	Number of executed integer logical instructions	emulation
Dynamic_Memory_offchip	instructions	Number executed off-chip memory transfer instructions	emulation
Dynamic_Memory_onchip	instructions	Number of executed on-chip memory transer instructions	emulation
Dynamic_Integer_comparison	instructions	Number of executed integer comparison instructions	emulation
Static_Float_single	instructions	Single-precision floating point arithmetic	static analysis
Static_Float_comparison	instructions	Single-precision floating point compare	static analysis
Static_Special	instructions	Special function instructions	static analysis
Memory_Efficiency	percentage	Memory efficiency metric	instrumentation
Memory_Sharing	percentage	Inter-thread data flow metric	instrumentation
Activity_Factor	percentage	Activity factor metric	instrumentation
MIMD	speedup	MIMD Parallelism metric	instrumentation
SIMD	speedup	SIMD Parallelism metric	instrumentation
Dynamic_Float_single	instructions	Number of single-precision arithmetic instructions	emulation
Dynamic_Float_comparison	instructions	Number of singl-precision comparison instructions	emulation
DMA_Size	bytes	Avg DMA transfer size	static analysis
Dynamic_Control	instructions	Number of executed control-flow instructions	emulation
Dynamic_Parallelism	instructions	Number of executed parallelism instructions	emulation
Dynamic_Special	instructions	Number of executed spcial instructions	emulation
Static_Float_double	instructions	Number of double precision floating point instructions	static analysis
Memory_Intensity	instructions	Memory Intensity metric	instrumentation
Dynamic_Float_double	instructions	Number of executed double-precision floating point instructions	emulation
Dynamic_Other	instructions	Other instructions	emulation

Table 3: List of metrics.

related with the problem size. This relationship is difficult to explain intuitively, and it would be relatively simple to craft a synthetic application that breaks this relationship. However, it is significant that none of these applications do. Results like this motivate the use of a technique like PCA, which is able to discover relationships that defy intuition.

PC1: Problem Size. The second component is composed most significantly of average dynamic integer, floating point, and memory instruction counts which collectively describe the number of instructions executed in each kernel. As described in the analytical model developed by Hong et. al.[10], these dynamic instruction counts are strong determinants of the total execution time of a program, and therefore the high degree of correlation is expected. What is not obvious is the relationship between the number of DMA calls executed before a kernel is launched and these instruction counts. We find that across all principal components, there is at least one metric that is available before launching a kernel that is highly correlated with the dynamic metrics in that component. We exploit this property in Section 6.5 to build a predictive model for application execution time using only static metrics.

PC2: Data Dependencies. We believe that the second principal component exposed the most significant and non-obvious relationship in this study. It indicates that data dependencies are likely to be propagated throughout all levels of the programming model; if there is a large degree of data sharing between instructions, then there is likely to be a large degree of data sharing among threads in each CTA and among all CTAs in a program. Notice that this component shows that registers that are alive at context switch points, Memory Sharing, and total kernel count are highly correlated. Data is typically passed from one thread to another at context switch points. More context switch points imply more opportunities for sharing data between threads and more registers alive at these context switch points imply more data that can be transferred to another thread. Memory sharing measures exactly, the amount of memory that is passed from one thread to another through shared memory. Finally, CTAs cannot reliably exchange data within the same kernel, but kernels have implicit barriers between launches that allow data to be exchanged between CTAs in different kernels. The correlation between memory sharing and kernel count seems to indicate that programmers will break computations that are required to share data among CTAs into multiple kernels. Furthermore, it seems to indicate that programs are either embarrassingly parallel at all levels, from the instruction level up to the task level, or have dependencies at all levels.

PC3: Memory Intensity. The next principal component is composed almost entirely of metrics that are associated with the memory behavior of a program. It should be clear that kernels that are given access to a large pool of memory via pointers are likely to access a significant amount of it. It is also interesting that applications that access a large amount of memory are likely to access it relatively efficiently, possibly because it can be accessed in a streaming rather than a random pattern. This component reveals that the memory intensive nature of applications is reflected in all levels of the memory hierarchy, from the

Metric	CP	MRI_FHD	MRI-Q	PNS	RPES	SAD	TPACF
Static							
Extent_of_Memory	1112592	582630	517229	720002676	59883768	8948776	5009948
Context_switches	0	0	0	19	5	2	4
Live_Registers	0.00000	0.00000	0.00000	23.15000	4.60000	5.50000	14.50000
Total_Registers	20.000	18.500	17.000	35.000	27.000	21.000	22.000
DMAs	11	17	11	224	6	3	3
DMA_Size	1.5351e+05	4.6501e+04	6.7397e+04	7.1420e+01	1.1537e+07	2.9996e+06	1.6602e+06
Integer_arithmetic	3410	31080	5388	13102152	200196	1620	448
Integer_logical	0	12516	2136	2300592	111150	280	72
Integer_comparison	220	8582	1400	2244480	25560	142	68
Float_single	5280	17290	2908	617232	1279008	186	18
Float_double	0	840	0	0	0	0	0
Float_comparison	0	1050	180	0	38448	0	6
Memory_offchip	1760	2478	468	1094184	10530	66	14
Memory_onchip	770	1862	332	1178352	142434	198	64
Control_instr	660	11466	1968	3226440	186300	182	134
Parallelism_instr	0	0	0	533064	15030	12	8
Special_instr	880	0	0	0	43254	0	0
Othe_instrr	0	0	0	0	0	0	0
Instrumented							
Activity_Factor	100.000	100.000	100.000	97.200	63.850	95.400	80.510
Memory_Intensity	0.010000	0.060000	0.040000	4.640000	2.740000	5.880000	0.010000
Memory_Efficiency	49.200	49.600	49.600	65.600	73.100	47.700	48.100
Memory_Sharing	0.00000	0.00000	0.00000	51.50000	76.60000	2.90000	12.40000
SIMD Parallelism	128.000	292.570	320.000	248.880	40.580	70.280	206.110
MIMD Parallelism	2.5600e+02	1.1057e+02	9.7500e+01	1.7990e+01	6.4757e+04	5.9400e+02	1.5663e+02
Emulated							
Integer_arithmetic	2.2596e+08	3.7218e+07	2.2805e+07	5.2469e+10	2.1425e+10	1.5161e+07	1.3488e+09
Integer_logical	0	1.3738e+07	7.8520e+06	2.4229e+10	8.9300e+09	5.9380e+05	2.2153e+08
Integer_comparison	1.1267e+08	2.6135e+07	1.2572e+07	5.6280e+09	5.0089e+09	6.0093e+05	2.0857e+08
Float_single	2.9293e+09	2.1333e+08	1.1878e+08	2.1047e+10	4.1966e+10	6.0445e+06	6.0892e+07
Float_double	0	11010048	0	0	0	0	0
Float_comparison	0	1.3738e+07	7.8505e+06	0	1.2119e+09	0	1.4098e+08
Memory_offchip	4.5056e+05	3.8136e+04	1.0896e+04	5.3392e+09	6.8786e+08	1.9127e+05	2.5571e+05
Memory_onchip	4.5064e+08	1.3801e+07	6.3024e+06	4.8278e+08	1.4313e+10	4.4984e+06	3.2223e+08
Control_instr	1.1278e+08	4.5258e+07	2.6641e+07	5.7445e+09	1.5469e+10	8.2229e+05	8.4251e+08
Parallelism_instr	0	0	0	4.1073e+07	2.6514e+09	1.9008e+04	2.0161e+07
Special_instr	9.0112e+08	0	0	0	9.9957e+08	0	0
Other_instr	0	0	0	0	0	0	0

Table 4: Metrics for each of the Parboil benchmark applications using the default input size.

register pressure to the ratio of memory to compute instructions to the amount of memory accessible by a kernel; for this analysis, a program with high register pressure can be predicted to be very memory intensive. Finally, this component is negatively correlated with dynamic floating point instruction count, indicating that applications either stress the memory hierarchy or the floating point units in a given processor, but not both. This information could be used in the design of highly heterogeneous architectures where some processors are given low latency and high bandwidth memory links, others are given extra floating point units, and workloads are characterized and directed to one or the other accordingly.

PC4: Control Flow Uniformity/SIMD Parallelism. The final component exposes several very interesting relationships involving the Activity Factor of an application. Recall that Activity Factor refers to the average ratio of threads that are active during the execution of a given dynamic instruction. First, Activity Factor is directly correlated with special instructions, indicating that it is unlikely that texture or transcendental operations will be placed immediately after divergent branches; if a special instruction is executed by one thread, it is likely to be executed by all other threads. This relationship is reversed for double precision floating point instructions; programs that execute a significant number of double precision instructions are likely to be highly divergent.

Discussion. Though the intent of this analysis was to identify uncorrelated metrics that could be used as inputs to the regression model and cluster analysis in the following sections, PCA also exposed several key relationships between program characteristics that may inform the design of CUDA applications, data parallel compilers, or even new processors optimized for different classes of applications. For example, we expected the dynamic single precision floating point instruction count to be negatively correlated with the dynamic double precision count. However, as can be seen in Figure 5, they are not correlated all, indicating that many applications perform mixed precision computation. After examining several applications, we realized that some used floating point constants in expressions involving single precision numbers. The compiler interprets all floating point constants as double precision unless they are explicitly specified to be single precision, and any operations involving these constants would be cast up to double precision, performed at full precision, and then truncated and stored in single precision variables. This is probably not the intention of the developer, and in processors where there are limited double precision floating point units, such as the C1060, this may incur a significant performance overhead.

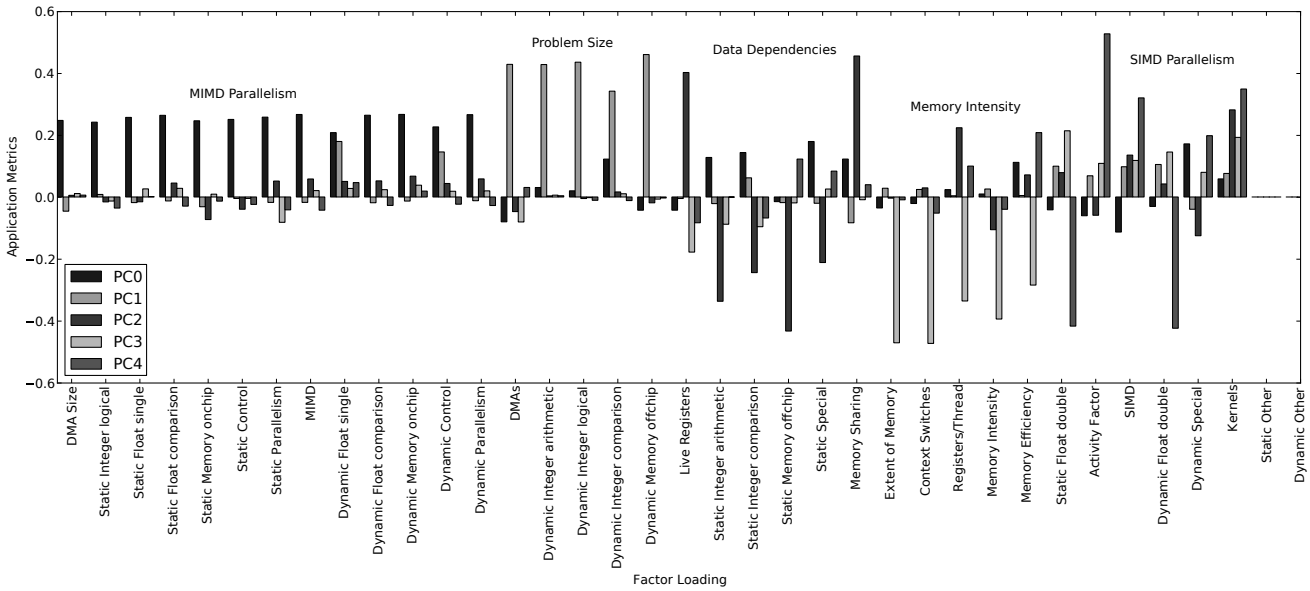

Figure 5: Factor loadings for the five application principal components. A factor loading closer to +-1 indicates a higher influence on the principal component.

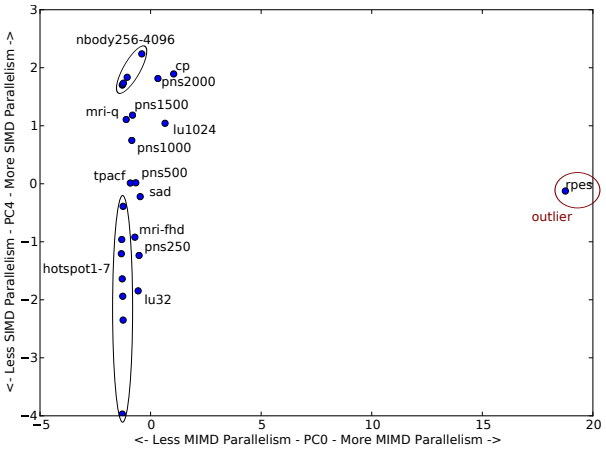

Figure 6: This plot compares MIMD to SIMD parallelism. It should be clear that these metrics are completely independent for this set of applications; the fact that an application can be easily mapped onto a SIMD processor says nothing about its suitability for a multi-core system. A complementary strategy may be necessary that considers both styles of parallelism when designing new applications.

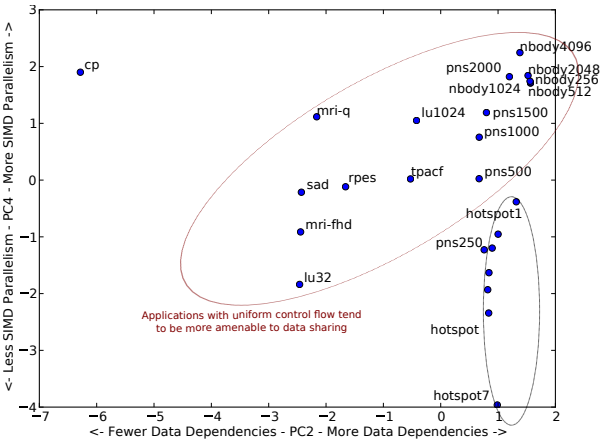

Figure 7: A comparison between Control Flow Divergence and Data Dependencies/Sharing. Excluding the hotspot applications, applications with more uniform control flow exhibit a greater degree of data sharing among threads. Well structured algorithms may be more scalable on GPU-like architectures that benefit from low control flow divergence and include mechanisms for fine grained inter-thread communication.

6.4 Cluster Analysis

Cluster Analysis is intended to identify groups applications with similar characteristics. It is useful in the development of benchmarks suites that are representative of a larger class of applications, visualizing application behavior, and in the context of this study, simplifying the development of accurate regression models via regression trees. For this analysis, we project the original application data onto the principal components. This allows the individual applications to be compared in terms of the principal components, for example, we can say that *cp* has the least data dependencies and *nbody4096* has the most. Though there are 10 possible projections of the five principal components, we present only three interesting examples due to space constraints. Figure 8 shows that MIMD and SIMD parallelism are not correlated, Figure 7 highlights a non-intuitive relationship between control flow uniformity and inter-thread data sharing, and Figure 6 illustrates how the *nbody* and *hotspot* applications scale with problem size. These individual cases are discussed in depth in the captions.

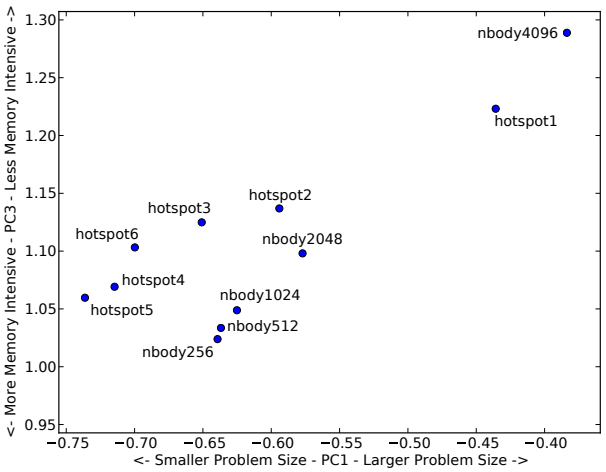

Figure 8: This figure shows the effect of increased problem size on the Memory Intensity of the Nbody and Hotspot applications. While this relationship probably will not hold in general, it demonstrates the usefulness of our methodology for characterizing the behavior of individual applications. We had originally expected these applications to become more memory intensive with an increased problem size; they actually become more compute intensive. This figure is also useful as a sanity check for our analysis, it correctly identifies the Nbody examples with higher body counts as having a larger problem size.

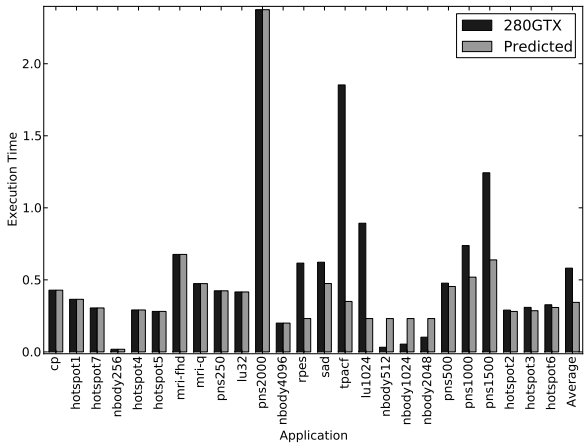

Figure 9: Predicted execution times for the 280GTX using only static data. The left 12 applications are used to train the model and the predictions are made for the rightmost 13 applications.

6.5 Regression Modeling

The goal of this study is to derive accurate models for predicting the execution time of CUDA applications on heterogeneous processors. If possible we would also like to be able to make these predictions using only metrics that are available before a kernel is executed. As shown in Section 6.3, 85% of the variance across the set of metrics can be explained by five principal components, each of which is composed of at least one static metric that is available before the execution of a kernel. For example, according to the PCA, the size of DMA transfers, the number of DMA transfers, the number of live registers at context switches, the extent of memory accessible by each kernel, and the number of double precision instructions are all statically available metrics that should be good predictors of kernel performance.

In this example, we use the polynomial form of linear regression to determine a relationship between static program metrics and the total execution time of an application. Modeling M variables, each with an N-th order polynomial, requires at least $N * M$ samples for an exact solution using the least squares method for linear regression. This limits the degree of our polynomial model in cases where only a few samples are available, which may be a concern for models that are built at runtime as a program is executing.

Though linear regression will generate a model for predicting the execution time of a given application on a particular CPU or GPU, it can generate predictions that are obviously not valid. For example, it is common for the model to predict short running kernels to have negative execution times, or predict that the execution time of a relatively more power processor is significantly slower than another processor that is invariantly slower in all other cases. In order to account for these cases, we applied a technique typically used in image processing, Projections Onto Convex Sets(POCS)[2], to each prediction. POCS works by applying a series of transformations to data, each of which enforces some a priori constraint on the data that must be true in all cases. If each successive transformation causes another invariant property to be violated, the iterative application of each transformation is necessary to find a result that satisfies all of the a priori constraints. In this case, we impose the constraints that all execution times must be greater than 0, and that all predictions for a given application on a given architecture should be within two standard deviations of the mean of all execution times of the same application on other architectures.

In this study, we recorded the metrics described in Section 5.1 and execution times of 25 applications on seven different processors. We used this data to predict the execution times of new applications on the same processor and the same application on different processors. We found that the models are most accurate when predicting the same application on a different, but similar style of architecture. For example, predicting the execution time of an application on an 8800GTX GPU that has previously run on an 8600GS and a 280GTX. Predicting the execution time of a new application on the same processor is also relatively accurate. However, models that attempt to, for example, predict GPU performance given CPU training data are wildly inaccurate. In these cases we believe that it will be necessary to develop separate models for each cluster of applications using a technique akin to regression trees on the data presented in Section 6.4.

Application Modeling. Our first experiment attempts to build a model for the execution time of a the remaining 13 applications on an NVIDIA 280GTX GPU using 12 randomly selected applications for training. We chose to only include results from the 280GTX in this paper because the models for the other architectures yielded similar results. Figure 9 compares the predicted execution time to the actual execution time for each of the 25 applications. Note that this model is intended to be used at runtime in a system that launches a large number of kernels, there-

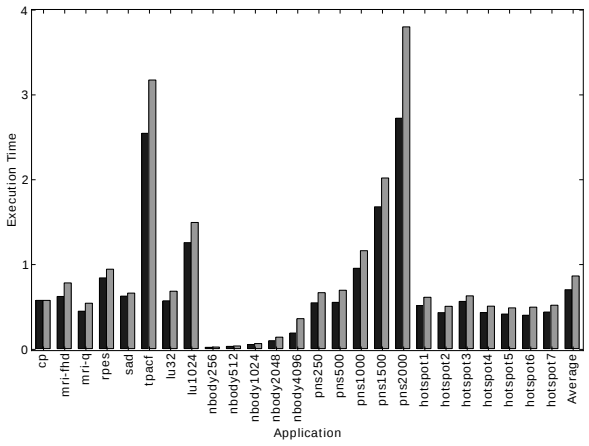

Figure 10: Predicted execution times for the 8800GTX using only static data and all other GPUs to train the model. Black indicates the measured time, and gray is the predicted time.

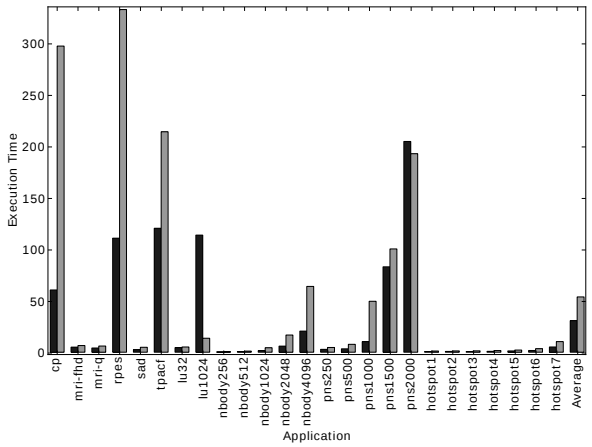

Figure 11: Predicted execution times for the AMD Phenom processor using the Atom and Nehalem chips for training.

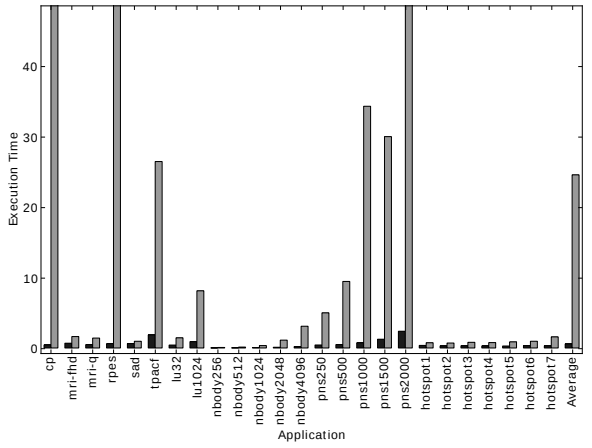

Figure 12: Predicted execution times for the 280GTX using all of the other processors for training. This is the least accurate model; it demonstrates the need for separate models for GPU and CPU architectures.

fore it should be perfectly accurate for kernels that it has already encountered. This model is the most accurate for the hotspot, sad, and smaller sized pns applications where all predictions are within 80% of the actual execution time. It is relatively inaccurate for the *nbody*, *tpacf*, *rpes*, *lu*, and larger *pns* applications. In the worst case, the execution time of tpacf is predicted to be only 22% of the recorded time.

GPU Modeling. The next experiment uses the actual execution times of each application on the Geforce 280GTX, Geforce 8600GS, and Tesla C1060 to predict the execution time of the same applications on the Gefore 8800GTX. Figure 10 shows the predicted execution time as well as the measured execution time for each application. This model was the most accurate that we evaluated, the worst case being the *pns2000* application, for which to total execution time is predicted to be 3.9s and the actual execution time was 2.8s. In all other cases, the model underestimates the performance of the 8800GTX by between 16% and 1%. It is worthwhile to note that this model is able to predict the impact of increased problem size on the same application. For example, the execution time of each run of the *nbody* application is predicted to increase as the number of bodies simulated increases. The Parboil benchmark *tpacf* is relatively difficult to be predict by this model, and, in fact, the CPU and application models as well. This model always places the performance of the 8800GTX between that of the 8600GS and the Tesla C1060, which is also true for the actual execution times of all applications.

CPU Modeling. The third experiment uses training results from the Intel Nehalem and Intel Atom processors to predict the performance of the AMD Phenom processor. It should be immediately clear that moving to CPU platforms changes the relative speed of each application. For example, on all of the GPU processors, the performance of *sad* and *rpes* are relatively similar. However, on the CPUs, *sad* is nearly 25x faster than *rpes*. This reinforces the point that GPUs and CPUs are more efficient for certain classes of applications than others even when they are both starting with the same implementation of the program.

Compared to the GPU model, the CPU model is slightly less accurate as can be seen by comparing Figure 10 and Figure 11. The CPU model in Figure 11 is the most accurate for *hotspot*, the larger *pns* benchmarks, the *mri* benchmarks, and *nbody*. For those applications the predicted execution times fall with 80% of the measured execution times. The model is the least accurate for the *cp* application, which is predicted to take 62s and actually takes 294s to execute. It is interesting to note that this application only takes 54s on the Intel Nehalem processor, which is typically competitive with the AMD Phenom. Whatever machine characteristic causes this large discrepancy in performance is not captured in our set of machine metrics. It is possible that a more detailed model including more machine metrics would be able to capture that relationship.

GPU-CPU Modeling. The final experiment demonstrates a case in which our methodology fails to generate an accurate model. Figure 12 presents the predictions for a model for the 280GTX using results from all other processors for training. This model excessively overestimates the execution time of each application with only *cp*, *hotspot*, and *sad* being within 50% of the total execution time. As

our cluster analysis shows, GPU and CPU style architectures have very different machine parameters and combining them in the same model significantly reduces the accuracy of the model. It motivates the need for a two stage modeling approach in which applications and processors are first classified into related categories with similar characteristics and then modeled separately.

6.6 Discussion

A significant result of this paper is that the methodology of principal components analysis, cluster analysis, and regression modeling is able to generate predictive models for CPUs and GPUs, suggesting that there are certain characteristics that make an application more or less suitable for a given style of architecture. Unfortunately, while the regression method used in this study can generate an accurate model, it usually includes complex non-linear relationships that are difficult to draw any fundamental insights from. Additional analysis is needed to discover these relationships, perhaps by determining correlations between application metrics and relative performance on a particular system in future work.

Though the relationships described in the preceding sections are applicable only for the applications machine configurations used in this study, the methodology is a universally valid tool that can be applied to any set of applications. An extension of this work would be to increase the set of possible benchmark applications and input sizes then use PCA and cluster analysis to select the most representative applications and inputs in a manner described in detail in [18].

Finally, this study exposed several non-intuitive relationships between application characteristics, for example, that applications with highly uniform control flow are more amenable to fine-grained synchronization and inter-thread communication. Discovering these relationships is becoming increasingly important as they can expose opportunities for architecture optimizations and influence the selection of well structured algorithms during application development. Clearly, there is a pressing need for further analysis and additional data.

7. CONCLUDING REMARKS

This paper presents an emulation and translation infrastructure and its use in the characterization of GPU workloads. In particular, standard data analysis techniques are employed to characterize benchmarks, their relationships to machine and application parameters, and construct predictive models for choosing between CPU or GPU implementations of kernel based on Ocelot's translation infrastructure. Accompanying the insights this approach provides is a clear need for deeper and more refined characterizations and prediction models - a subject of ongoing and future work.

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.

8. REFERENCES

[1] 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.

[2] J. Boyle and R. Dykstra. A method of finding projections onto the intersection of convex sets in hilbert spaces. 37:28–47, 1986.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. 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.

[4] S. Collange, D. Defour, and D. Parello. Barra, a modular functional gpu simulator for gpgpu. Technical Report hal-00359342, 2009.

[5] A. Corporation. The arm cortex-a9 processors. white paper, ARM, September 2009.

[6] G. Diamos. The design and implementation ocelot's dynamic binary translator from ptx to multi-core x86. Technical report, CERCS, 2009.

[7] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing computer architecture research workloads. *Computer*, 36(2):65–71, Feb 2003.

[8] K. O. W. Group. *The OpenCL Specification*, December 2008.

[9] V. Grover, S. Lee, and A. Kerr. Plang: Translating nvidia ptx language to llvm ir machine, 2009.

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

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

[12] Intel. Intel graphics media accelerator x3000. Technical report, 2009.

[13] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[14] 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.

[15] B. F. Manly. *Multivariate statistical methods: a primer*. Chapman & Hall, Ltd., London, UK, UK, 1986.

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

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

[18] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(2):412–423, 2007.

[19] J. Stratton, S. Stone, and W. mei Hwu. Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.