BIOINFO Computer Engineering
ISSN: 2249-3980 & E-ISSN: 2249-3999, Volume 2, Issue 1, 2012, pp.-25-30.
Available online at http://www.bioinfo.in/contents.php?id=322

ANALYZING THE CUDA APPLICATIONS WITH ITS LATENCY AND BANDWIDTH TOLERANCE

PISTULKAR V.N. AND UTTARWAR C.A.

Department of Computer Science, Jawaharlal Darda Institute of Engineering & Technology, Yavatmal, MS, India.
*Corresponding Author: Email-vrushali118@gmail.com, chaitaliuttarwar@gmail.com

Received: February 21, 2012; Accepted: March 15, 2012

Abstract- The CUDA scalable parallel programming model provides readily-understood abstractions that free programmers to focus on efficient parallel algorithms. It uses a hierarchy of thread groups, shared memory, and barrier synchronization to express fine-grained and coarse-grained parallelism, using sequential C code for one thread. This paper explores the scala- bility of CUDA applications on systems with varying interconnect latencies, hiding a hardware detail from the programmer and making parallel programming more accessible to non-experts. We use a combination of the Ocelot PTX emulator [1] and a discrete event simulator to evaluate the UIUC Parboil benchmarks [2] on three distinct GPU configurations. We find that these applications are sensitive to neither interconnect latency nor bandwidth, and that integrated GPU-CPU systems are not likely to perform any better than discrete GPUs or GPU clusters.

Citation: Pistulkar V.N. and Uttarwar C.A.(2012) Analyzing the CUDA Applications with its Latency and Bandwidth Tolerance. BIOINFO Computer Engineering, ISSN: 2249-3980 & E-ISSN: 2249-3999, Volume 2, Issue 1, pp.-25-30.

Introduction

While single-thread performance of commercial super scale rmi-croprocessors is still increasing, a clear trend today is for com-puter manufacturers to provide multithreaded hardware hat strongly encourages software developers to provide explicit par-allelism when possible. One important class of parallel computer hardware is the modern graphics processing unit(GPU) [22,25].With contemporary GPUs recently crossing that flop barri-er [2,34] and specific efforts to make GPUs easier to program for non-graphics applications [1, 29, 33], there iswidespread interest in using GPU hardware to accelerate nongraphicsapplica-tions.Obviously this is undesirable. A developer for a parallel or distributed system is presented with a nearly unmanageable degree of complexity. Should data be redundantly computed in parallel or broadcast from a single node? Will CPU throughput or network bandwidth be the bottleneck? Are random mem- ory accesses significantly slower than sequential accesses? Is hard-ware acceleration available for common math functions? At some point, the complexity becomes significant enough that application development for parallel systems becomes intractable.

We argue that abstractions are needed to reduce the com-plexity of programing parallel and distributed systems. In this context, abstractions are programming language or hardware constructs that hide system complexities from users. For exam- ple, caches are abstractions that hide the latency and bandwidth gap be-tween SRAM and DRAM from the programmer.

The most useful abstractions hide complexity without significantly sacrificing performance.

This work evaluates the utility of abstractions in the CUDA pro-gramming model for hiding GPU-CPU interconnect latency and bandwidth. In Section II we cover typical GPU system configura-tions. In Section III, we present an overview of the CUDA pro-gramming model, and highlight the abstractions that hide GPU-CPU communication. In Section IV, we describe the infrastructure used to evaluate the latency sensitivity of

CUDA application. In Section V, we present results from several CUDA benchmarks. Section VI briefly covers related work and Section VIII concludes with the most significant implications of our findings.

CUDA Data Parallel Threading Model

Here, we'll describe the data parallelism model supported in CUDA and the GPU. Why should PGI users want to understand the CUDA threading model? Clearly, PGI CUDA Fortran users should want to learn enough to tune their kernels. Programmers using the directive-based PGI Accelerator programming model will also find it instructive in order to understand and use the compiler feedback (-Minfo messages) about which loops were run in parallel or vector mode on the GPU; it's also important to know how to tune performance using the loop mapping clauses. So let's start with an overview of the hardware in today's NVIDIA Tesla and Fermi GPUs. [17]

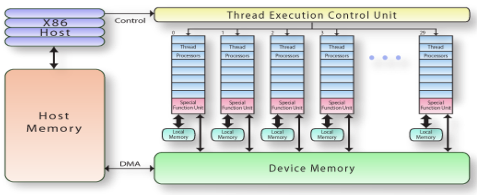

Fig. 1- NVIDIA Tesla Block Diagram

CUDA offers a data parallel programming model that is supported on NVIDIA GPUs. In this model, the host program launches a sequence of kernels. A kernel is organized as a hierarchy of threads. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. Kernels can use these indices to compute array subscripts, for instance. Performance tuning on the GPU requires optimizing all these architectural features:

Finding and exposing enough parallelism to populate all the multiprocessors.

Finding and exposing enough additional parallelism to allow multithreading to keep the cores busy.

Optimizing device memory accesses for contiguous data, essentially optimizing for stride-1 memory accesses.

Utilizing the software data cache to store intermediate results or to reorganize data that would otherwise require non-stride-1 device memory accesses.

GPU System Architecture

Though GPUs have typically been used as accelerator cards connected via a system interconnect like PCIe, there has been an increasingly popular migration towards tightly-integrated heterogeneous CPU-GPU processors in the embedded domain and distributed multi-GPU systems in the high performance domain. Each of these classes of systems is expected to run the same workloads. However, the latency and bandwidth of the CPU-GPU communication link changes significantly for each of these systems. In this paper we would like to explore the impact on currently existing applications exposed by these new system configurations. We begin by highlighting the differences among the three classes of systems.

Discrete GPU - PCIe

GPUs have traditionally been used as add-in accelerators for offloading graphics applications on desktop systems. Systems in this configuration are typically referred to as having a discrete GPU. As add in cards, discrete GPUs have historically been treated as slave devices where the CPU issues a series of commands and data transfers through the northbridge over an interconnect such as PCI. These devices contain their own locally managed DRAM that is not directly visible to the host CPU. As systems with discrete GPUs evolved, new interfaces such as AGP and PCIe increased the communication bandwidth between the CPU and the GPU and DMA engines were added to free the CPU from direct involvement in data transfers.

Discrete GPUs represent middle of the road interconnection latency and bandwidth. The theoretical upper-bound perfor- mance of the commonly used 16x PCIe 2.0 interconnect is

16GB/s for bidirectional communication. Real world perfor- mance is limited by driver and protocol overheads, and our own measurements show that it is possible to attain 4.2GB/s for a unidirectional DMA transfer. Other studies have shown that the best case PCIe latency is on the order of 1us for small transfers [3]. Compared to other machine configurations, discrete GPUs have lower bandwidth and higher latency than tightly integrated GPU-CPU systems, but higher bandwidth and lower latency than GPU clusters. As a final point, most CUDA applications were designed specifically for systems with discrete GPUs.

HPC - Bridged PCIe

NVIDIA has recently introduced a new class of GPUs system where a set of several GPUs are packaged into a standalone 1U blade. These GPUs are connected via bridged PCIe to a host system, and the entire unit is meant to be used as a node in a cluster or grid. In the best case, the bandwidth of these systems will be reduced to half of that of an equivalent discrete GPU due to the PCIe bridge servicing two GPUs on the same link. In the worst case, kernels may be launched on a GPU that is not directly connected to a node, forcing the kernel's data and code to be transferred over the node- to-node interconnect. The most popular cluster interconnects are currently Infiniband and Ethernet, with bandwidths ranging from 250MB/s to 12GB/s and minimum latencies ranging from

1us to 100us for a single hop. Though very high-end clusters can attain interconnect performance similar to that of a discrete GPU system, the average or worst cases increase latency by up to 100x, and reduce bandwidth by up to 32x.

Integrated GPU-CPU

Intel and AMD have driven research into heterogeneous integrated GPU-CPU processors where a number of GPU and CPU cores are integrated on the same die, sharing a last level cache and having direct access to the DRAM controllers. Pangea was a research implementation of such a processor that was designed in RTL and synthesized on an FPGA [4]. In their paper, Wong et al. state that the communication latency from the CPU to the GPU was only 12 cycles. Additionally, because the GPU and CPU share the same memory space, DMA copies that would be sent over PCIe in other systems can simply be copied in memory1 . From our experiments, GPU memory typically has 10x greater bandwidth than a large DMA operation over PCIe.

Taken together, these system configurations have latencies spanning four orders of magnitude and bandwidths varying by up to

320x. If CUDA applications are particularly latency or bandwidth sensitive, then it is probably that many applications would have to be significantly re-written to run on these different systems. It turns out that this is not the case; most A copy is still necessary for many applications due to the semantics of the CUDA programming model.
applications experience no performance degradation nor any improvement when moving from one system to another. The next section explores the characteristics of CUDA applications that might make them tolerant to the wildly variant character- istics of integrated, discrete, and clustered GPU systems.

The CUDA programming model

CUDA was introduced in 2006 as a programming language for NVIDIA GPUs with minimal extensions to the C program- ming language. What was not emphasized was that CUDA is based around the idea of a Bulk-Synchronous Parallel (BSP) program, an idea first introduced by Valiant [5] in 1990. As explained in the subsequent section, BSP programs are implic- itly designed to be latency tolerant in order to account for the rising cost of global synchronization. Coupled together with the fact that applications are composed of parallel streams of GPU kernels and CPU code that are periodically synchronized, there is significant evidence to suggest that CUDA applications can tolerate communication latency in the GPU-CPU link.

Bulk-Synchronous Parallel Programming

CUDA applications, and other BSP programs, are built around the idea that the number of cores per processor will continue to increase, as will the time needed to performance a global synchronization operation across all cores in the system. In order to ensure that applications are scalable on future processors, BSP programs (Kernels in CUDA) must be specified in terms of a large number of work units (referred to as CTAs in CUDA) that cannot communicate other than at periodic global barrier operations. In many cases, the large number of CTAs per Kernel represent enough work to hide the global synchronization overhead, which in CUDA represents a GPU-CPU communication operation.

CPU and GPU Streams

CUDA allows an application developer to partition a pro- gram into highly parallel, completely encapsulated, GPU ker- nels interleaved with C statements, where kernels are executed on the GPU and the C statements are executed on the host CPU. The explicitly partitioned design of CUDA programs allows them to be expressed conceptually as separate streams of operations, one which executes on a GPU device and the other which executes on a CPU core. This characteristic makes CUDA programs amenable to execution on systems with high communication latency between the CPU and GPU, as kernel execution on the GPU can be overlapped with C++ execution on the CPU.

Infrastructure

In order to evaluate the impact of interconnect latency and bandwidth on the performance of CUDA applications, We leveraged two existing simulation tools, Ocelot and NfinSim, coupled with new interconnect models designed specifically for this evaluation to simulate the execution of complete CUDA applications on systems with varying interconnect characteristics. For this evaluation, we used the UIUC Parboil

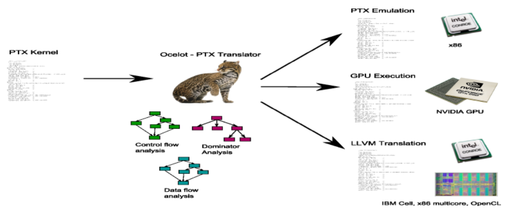

Fig. 1- High Level Overview of Ocelot

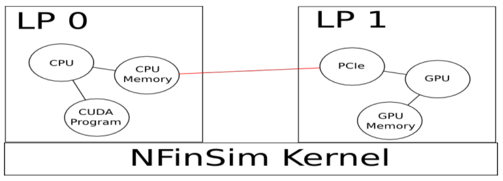

Fig. 2-An Example of A System Simulated in Parallel with NfinSim

benchmark suite, which is designed to be representative of compute workloads for GPUs.

Ocelot - A CUDA Emulator

Ocelot is a just-in-time (JIT) compiler and runtime for CUDA applications capable of running applications on mul- tiple processors, not only GPUs. Figure 1 shows the backend targets that are currently supported by Ocelot. CUDA appli- cations are composed of two complementary components: 1) binaries for each kernel and 2) a runtime component that sets up the environment in which a kernel is executed. The binaries for each kernel are stored in a virtual instruction set (referred to as PTX) [6] which is normally translated to the native instruction set of a particular GPU during execution. Ocelot replaces the NVIDIA JIT compiler which only supports NVIDIA GPUs with a custom compiler that includes back -end targets for multi-core x86 (and other LLVM targets), NVIDIA GPUs, and instruction by instruction emulation. Ocelot also replaces NVIDIA's implementation of the CUDA runtime with a custom implementation that makes CPU and Emulated devices appear to be CUDA-capable GPUs.
In the context of this study, we use Ocelot to instrument CUDA applications as they are running. We collect the fol- lowing information as a CUDA program is being executed: 1) the sequence of calls into the CUDA runtime, 2) the execution time of each call, 3) the code size and execution time of each call, 4) the size of all DMA operations, and 5) the time spent executing host code between successive CUDA calls. In order to account for the startup latency associated with executing a kernel, we measured the execution time of a series of no-op kernels and subtracted this average startup cost on our test system from the execution time of each kernel.
Assuming that the GPU used in each system configuration is the same, then the execution time of the kernel should be the same for each system. The only variance should be due to the latency of sending commands and data to the GPU. We express every CUDA runtime command as a packet that is processed by the GPU along with the measured time required to execute the call. To determine the total execution time of the application using different interconnect configurations, we treat the series of CUDA commands and host sections as independent streams of opera-

tions that are synchronized on DMA copies to or from the GPU. This is similar to the early Decoupled Access/Execute architectures where opera- tions from independent streams could be overlapped [7], albeit our approach works at a higher level.

NFinSim - A Full System Simulator

NFinSim is a distributed discrete event simulator designed to provide cycle-level simulation of large-scale parallel sys- tems. It uses a modular design where a large system can be composed of a collection of smaller models that are explicitly connected and communicate by exchanging events, an example of which is shown in Figure 2. The natural division of a large system into components eases the partitioning of the simulation into closely connected clusters than can be simulated relatively independently, subject only to infrequently exchanges of events among clusters. The goal is to design a simulator for parallel systems that can take advantage of multi- core processors and clusters to maintain a constant slowdown factor between native execution on a parallel system and its simulation on the same system.

For this study, we modeled each GPU system using the following components:

1. A CPU core,
2. a simple host network stack,
3. a point to point communication channel,
4. a GPU network stack, and
5. the GPU core.

CPU Core Model. For our CPU core model, we did not perform detailed instruction level simulation. Instead, we used the recorded execution of each host code segment from the trace captured by Ocelot combined with the clock frequency of the simulated processor to generate a cycle count. The CPU model implemented two different protocols for processing host code and CUDA calls, blocking and non-blocking. In the blocking protocol, the CPU would execute each host code section to completion before beginning the next CUDA call. Furthermore, all CUDA calls were acknowledged by the GPU such that only one call could be outstanding at any time. For the non-blocking protocol, the CPU would execute host code sections as they were encountered in the program like the blocking protocol. However, most CUDA calls would be launched asynchronously without waiting for
an acknowledgement before moving on to the next call or section of host code. Synchronization would only take place at DMA transfers which are required to complete before a new host section can be executed in-case it uses the data copied from the GPU or writes over the buffer being copied to the GPU.

Host Network Stack. The host network stack is responsible for establishing a connection between the CPU and GPU models when the program starts up. Once this has been accomplished, it receives packets with encapsulated CUDA calls from the CPU, marshals them into frames that can be transferred to the GPU and routes them to the correct GPU in a system with multiple devices, ensuring that calls are delivered to the GPU in the order in which they were sent. The overhead associated with each stage of the protocol is modeled using an analytical model that takes into account call packet size, marshaled data size, inter-packet delay, routing time, and user- to-OS buffer copy time.

The GPU Model. The final GPU model is used to determine the execution time of a particular CUDA call on the simu- lated GPU device. We evaluated the possibility of performing detailed cycle-level simulations for each CUDA kernel using either the Ocelot emulator as a front-end to drive timing models or analytical models as in [8], or using another PTX simulator such as GPGPU-SIM [9]. However, we eventually decided to use measured execution times from real hardware based on the idea that changing the communication latency between the GPU and CPU will change the time at which a kernel begins execution rather than its total execution time. In this case, our model is a very simple module that accepts packets with recorded GPU execution time and converts them into cycles based on the clock frequency of the simulation.

Parboil - A CUDA Benchmark Suite

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 [2]. 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. For this study, we assume that the Parboil benchmarks are representative of CUDA applications. This may or may not be a reasonable assumption.

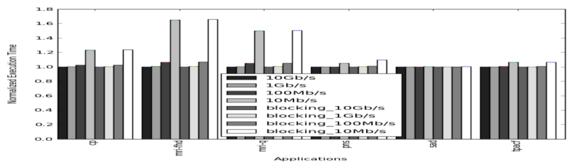

Fig. 3- Impact of Bandwidth on Total Execution Time

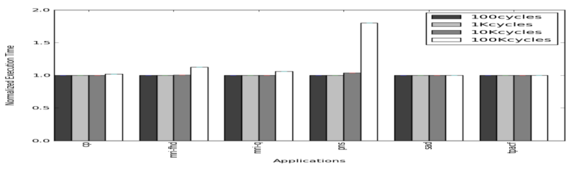

Fig. 4- Impact of Latency on Total Execution Time

We recommend that any conclusions that are drawn from the results of this study should not be applied directly to other applications without first verifying that the application is similar in structure to at least one of the Parboil benchmarks.

Results

In order to determine the latency tolerance of the Parboil benchmarks, we collected traces of each application running using Ocelot on the system in Table I. We acknowledge that

Table 1-Tenst System

CPU	Intel i920 Quad-Core 2.66Ghz
GPU	NVIDIA Tesla C1060
Memory	8GB DDR-1333 DRAM
CPU Compiler	GCC-4.4.1
CUDA Compiler	NVCC-2.3
CUDA Runtime/JIT	Ocelot 0.9.264
OS	64-bit Ubuntu 9.10

The GPU used in our experiments is a high end discrete GPU that would probably not be packaged together with a CPU on the same chip due to power constraints. However, the intent of these experiments is to highlight the latency sensitivity of CUDA applications that may impact the design of applications for a specific system configuration, rather than to model any given system. We begin by exploring the sensitivity of CUDA applications to link bandwidth before moving on to link latency.

Bandwidth

In this experiment, we started with the worst case bandwidth that an application could ever experience in a realistic envi- ronment and gradually scaled up the bandwidth until there was no further improvement in performance for any of the benchmarks. For this experiment, we assume that there is no communication startup latency between the GPU and the CPU to isolate the effect of bandwidth on the total execution time of the program. We began by simulating a 10Mb/s link similar to an older Ethernet standard or a high end Internet connection and move up to 10Gb/s (slightly slower than PCIe2.0) as shown in Figure 3. The simulation was run using both blocking and non-blocking communication.

As can be seen in the figure, most applications are not sensitive to the interconnect bandwidth under any of the configurations tested. Moving from blocking to non-blocking execution, does not significantly impact the execution time of any application, leading us to believe that overlapped GPU- CPU execution is not the source of the latency tolerance of CUDA applications. Even the most sensitive application, MRI- FHD, only experiences a 1.6x increase in execution time using the simulated 10Mb/s link. This is significantly slower than the GPU cluster configuration. In fact, these results suggest that it would even be possible to run these applications over an Internet class connection without significant performance degradation. Needless to say, none of these application are bandwidth sensitive.

Latency

For the second experiment, we started with the worst case latency reported for any of our three system configurations,

100cycles, and swept the latency down to that of the fastest tightly -integrated system, about 100k cycles. This simulation was run using the blocking and non-blocking protocols like the previous experiment, and like the previous experiment, the moving from blocking to non-blocking execution does not significantly change the results. Figure 4 only presents the results for the non-blocking protocol to improve the readability of the figure.

Like the bandwidth experiment in the previous section, these applications experience almost no performance degradation with increased interconnect latency. The slowest application is only 1.8 slower with an additional 100k cycles of latency for sending a new packet over the link. It is also worthwhile to note that the PNS application is the most latency sensitive application that we tested, whereas MRI-FHD was the most bandwidth sensitive application. Taken together, these results suggest that CUDA applications are neither latency nor band- width sensitive.

Implications

These results have potentially significant implications on the design of GPU applications. One of the primary motivations of the design of tightly-integrated GPU-CPU systems like Pangea is the reduced communication latency from the CPU to the GPU where the authors claim that "This can achieve a two-order of magnitude reduction in thread spawning latency" [4]. For at least the applications studied in this paper, this two-order of magnitude reduction in latency may not matter. Pangea does not directly explore the advantage of this reduc- tion in thread-spawn latency, instead showing that a kernel's execution time is very sensitive to DRAM memory latency, which is an entirely different system parameter. As a positive note, these results suggest that CUDA applications may be good candidates for execution on distributed or cluster sys - tems, or even future many-core architectures with significant global synchronization latency.

Open problem and future work

Even though the results presented in this paper strongly suggest that existing CUDA applications are not sensitive to the latency or bandwidth of the GPU-CPU link, it may be that applications developers are forced to spend extra efforts to achieve this property when designing CUDA applications. It is still the case that most CUDA applications are developed and deployed on discrete GPU systems due to their significant share of the total GPU market. This may artificially force developers to account for constrained latency and bandwidth resources in these systems such that the only existing CUDA applications are latency tolerant by design rather than by some inherent property in the programming model.

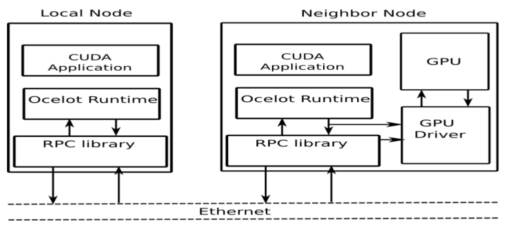

Fig. 5-CUDA RPC Libra

Conclusion

This paper explores the latency and bandwidth sensitivity of CUDA applications leveraging the Ocelot CUDA emulator and the NFinSim system simulator. For the applications evaluated in this study, the most sensitive application only experiences a 1.6x slowdown in response to a 1000x reduction in inter- connect bandwidth and only a 1.8x slowdown in response to a 1000x increase in interconnect latency. Though determining the exact cause of this insensitivity is beyond the scope of this paper, these results suggest that current GPU applications will see no benefit from moving to tightly integrated GPU-CPU systems. Instead, current applications designed for discrete GPUs have great potential to be deployed without modification on GPU clusters.

References

[1] Kerr A., Diamos G. and Yalamanchili S. (2009) *IEEE International Symposium on Workload Characterization*, 3-12.
[2] IMPACT (2007) *The parboil benchmark suite*.
[3] Holden B. (2006) *Latency comparison between hypertransportT M and pci- expressT M in communications systems, in HyperTransportT M Con- sortium*.
[4] Wong H., Bracy A., Schuchman E., Aamodt T.M., Collins J.D.

Wang P.H., Chinya G., A. K., Groen A.K., Jiang H. and Wang H. (2008) 17*th international conference on Parallel architectures and compilation techniques*, 52-61.

[5] Valiant L.G. (1990) *Commun. ACM*, 33(8), 103-111.

[6] *NVIDIA Corporation* (2008) *NVIDIA Compute PTX- Parallel Thread Execution,* 1*st ed.*

[7] Smith J.E. (1982) 9*th annual symposium on Computer Architecture* ,112-119.

[8] Hong S.and Kim H. (2009) *An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness*, 152-163.

[9] Bakhoda A., Yuan G.L., Fung W.W.L., Wong H. and Aamodt T.M. (2009) *IEEE International Symposium on Performance Analysis of Systems and Software* ,163-174.

[10]Gelado I., Kelm J.H., Ryoo S., Lumetta S.S., Navarro N. and Hwu W. (2008) *International Conference on Supercomputing* 299-308.

[11]Shi L., Chen H. and Sun J. (2009) *IEEE International Symposium on Parallel and Distributed Processing* ,1-11.

[12]Gupta V., Gavrilovska A., Schwan K., Kharche H., Tolia N., Talwar V. and Ranganathan P. (2009) *The* 3*rd ACM Workshop on System-level Virtualization for High Performance Computing*, 17-24.

[13]Stuart J.A. and Owens J.D. (2009) *IEEE International Symposium on Parallel & Distributed Processing*, 1-12.

[14]Diamos G. and Yalamanchili S. (2008) *HPDC'*08.

[15]24*th IEEE International Parallel & Distributed Processing Symposium* (2010).

[16]Sudnya Padalikar NFinTes Marietta, Georgia 30067 Gpu System Architecture

[17]Michael Wolfe, *PGI Compiler Engineer Understanding the CUDA Data Parallel Threading Model.*