A Study of Persistent Threads Style GPU Programming for GPGPU Workloads

Kshitij Gupta
Department of Electrical & Computer Engineering
University of California, Davis
kshgupta@ucdavis.edu

Jeff A. Stuart
Department of Computer Science
University of California, Davis
jastuart@ucdavis.edu

John D. Owens
Department of Electrical & Computer Engineering
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

In this paper, we characterize and analyze an increasingly popular style of programming for the GPU called Persistent Threads (PT). We present a concise formal definition for this programming style, and discuss the difference between the traditional GPU programming style (nonPT) and PT, why PT is attractive for some high-performance usage scenarios, and when using PT may or may not be appropriate. We identify limitations of the nonPT style and identify four primary use cases it could be useful in addressing—CPU-GPU synchronization, load balancing/irregular parallelism, producer-consumer locality, and global synchronization. Through micro-kernel benchmarks we show the PT approach can achieve up to an order-of-magnitude speedup over nonPT kernels, but can also result in performance loss in many cases. We conclude by discussing the hardware and software fundamentals that will influence the development of Persistent Threads as a programming style in future systems.

1. INTRODUCTION

GPGPU programming has spawned a new era in high performance computing by enabling massively parallel commodity graphics processors to be utilized for non-graphics applications. This widespread adoption has been possible due to architectural innovations of transforming the GPU from fixed-function hardware blocks to a programmable unified shader model, and programming languages like CUDA [11] and OpenCL [9] that present an easy-to-program coding style by heavily virtualizing processor hardware, and shifting the onus of extracting parallelism from the programmer (explicit SIMD) to the compiler (implicit SIMD) for instruction generation.

There are numerous examples of the current GPGPU programming environment facilitating good speed-up over sequential implementations [12]. This trend has been especially true for core compute portions from a broad variety of application domains which quite often tend to be brute force, or generally embarrassingly parallel in nature. However, from an application standpoint, the overall benefit of employing a GPU is governed by Amdahl's Law, necessitating the port of larger portions of an application to the GPU for achieving better overall application acceleration. As a wider variety of workloads are implemented on the GPU, many of which are highly irregular, limitations of GPU programming become apparent. We believe that the key source for these limitations stems from the programming environment that was originally created in 2006 [10] lacking native

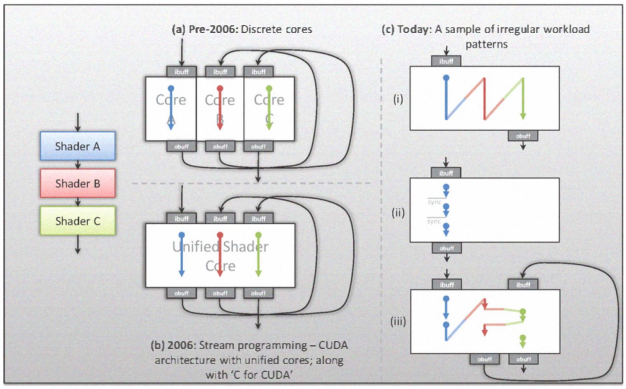

Figure 1: An illustration of GPU architecture and workload evolution through an example of a three-shader application—A (blue), B (red) and C (green)—(a) dedicated processor cores for each shader in the graphics pipeline; (b) unified processor architecture supporting all three shaders, well suited for data-parallel stream workloads (*kernels*); (c) data-flow patterns of modern workloads: (i) multiple kernels within an application to be executed successively (without having to write intermediate values to global memory), (ii) processing of a single kernel with synchronization requirement before the next iteration can be started, and (iii) strongly inter-dependent kernels B and C producing variable work for the next stage to process.

support for handling the kinds of communication patterns shown in Figure 1(c).

Many application-specific solutions have been proposed in literature recently to address these limitations — ranging from fundamental parallel programming primitives like scan and sort [8] and core algorithms like FFT [18] to programmable graphics pipelines like ray tracing [1] and Reyes-style rendering [16] — to maximize performance. These solutions while seemingly disjoint in usage are all centered around a common programming principle, called Persistent Threads (PT). However, showing that PT is useful in a specific instance does not address the larger questions: broadly, *when* is the PT style applicable and *why* is it the right choice. We go a step further by providing insights into this successfully used GPU-compute programming style in this paper.

Our first major contribution is to formally introduce/define

978-1-4673-2633-9/12/$31.00 ©2012 IEEE

the Persistent Threads model in a domain-independent way that is understandable to anyone with reasonable knowledge of GPU computing. We hope it contributes to a deeper understanding of this programming style. Our second major contribution is to concisely categorize the possible usage scenarios based on our needs and those encountered in literature into four broad use cases, each of which have been clearly described and benchmarked in the paper. These use cases encompass a wide body of issues that are encountered with the irregular workloads addressed by PT and therefore this paper provides a good starting point for anyone trying to look at ways to improve their performance. As this paper is neither intended as a summary nor validation of results seen in past literature, our third and most important contribution is a deep distillation and analysis of the use cases via un-biased benchmarks developed by us to provide insight into the key questions of using PT—*when* and *why* is PT appropriate. We believe this understanding is sorely lacking today. In this paper we take a systematic approach that would be useful to a broad audience from a variety of application domains.

In specific cases, the reason that makes PT successful is because of inefficiencies in current hardware and programming systems. We hope that over time, many of these inefficiencies that we identify in this paper will be resolved as hardware and software continue to move forward. However, we believe that the use cases we describe in this paper are relevant today and will continue to be relevant in the future. So another contribution of this paper is the creation of a set of benchmarks (comparison points) to help evaluate implementations using PT against not only the traditional programming style but also (and most importantly) against future software and hardware advances. As these use cases will continue to exist despite hardware and software evolution, the broader discussion we hope to initiate through this paper is the relative merits of hardware scheduling (as in the traditional GPU programming style) vs. software scheduling (as in PT).

We begin by describing the present programming style and identifying key performance bottlenecks in GPGPU programming today, followed by a description of the Persistent Threads style of programming in Section 2. In Section 3 we briefly describe each of the four use cases where we find PT to be applicable, and discuss them in detail subsequently. Through synthetic micro-kernel benchmarks, we present guidelines for when a PT formulation could be beneficial to programmers. We propose modifications to hardware and software programming based on the experience gained while running our benchmarks for native-PT-style programming in Section 4. We conclude in Section 5 by observing that the changes required for native support for PT style does not require a significant re-working of the processor hardware, making it a potentially attractive option for inclusion in near-term hardware and software programming extensions of CUDA and OpenCL.

2. PROGRAMMING STYLE BACKGROUND

2.1 Traditional Programming Style

GPU's hardware and programming style are designed such that they rely heavily on Single Instruction Multiple Thread (SIMT) and Single Program Multiple Data (SPMD) programming paradigms. Both these paradigms virtualize the

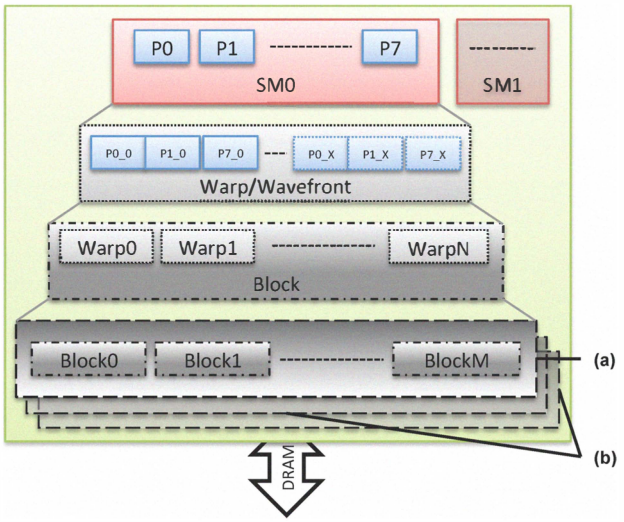

Figure 2: An illustration of the various levels of hierarchy in GPGPU programming. Solid rectangular boxes indicate a physical processor while dotted boxes indicate virtual entities. (a) Multiple blocks can simultaneously execute on a single SM; (b) Further virtualization of blocks on the SM scheduled over time.

underlying hardware at multiple levels, abstracting the view for the software programmer from actual hardware operation. Every lane of the physical SIMD Streaming Multiprocessor (SM) is virtualized into larger batches of threads, which are some multiple of the SIMD-lane width, called *warps* or *wavefronts* (SIMT processing). Just like in SIMD processing, each warp operates in lock-step and executes the same instruction, time-multiplexed on the hardware processor. Multiple warps are combined to form a higher abstraction called *thread blocks*, with threads within each block allowed to communicate and share data at runtime via L1 cache/shared memory or registers. The process is further virtualized with multiple thread blocks being scheduled onto each SM simultaneously, and each block operating independently on different instances of the program on different data (SPMD processing). A hierarchy of these virtualizations are shown in Figure 2.

This programming style (which we shall refer to as 'nonPT') forces the developer to abstract units of work to virtual threads. As the number of blocks is dependent on the number of work units, in most scenarios there are several hundreds or thousands more blocks to run on the hardware than can be initiated at kernel launch. In the traditional programming style, these extra blocks are scheduled at runtime. The switching of blocks is managed entirely by a hardware scheduler, with the programmer having no means of influencing how blocks are scheduled onto the SM. So while these abstractions provide an easy-to-program model by presenting a low barrier to entry for developers from a wide variety of application domains, it gets in the way of seasoned programmers working on highly irregular workloads that are already hard to parallelize. This exposes a significant limitation of the current SPMD programming style that neither guarantees order, location and timing, nor does it explicitly allow developers to influence the above three parameters without

CUDA	OpenCL
thread	work item
warp	—
thread *block*	work group
grid	index space
local memory	private memory
shared memory	local memory
global memory	global memory
scalar core	processing element
multi-processor (SM)	compute unit

Table 1: Equivalent terminologies between CUDA and OpenCL are listed here. Although we make use of NVIDIA hardware and 'C' for CUDA terminologies in this paper, the same discussion can be extended to GPUs from other vendors using OpenCL.

bypassing the hardware scheduler altogether. To circumvent these limitations, developers have used a PT style and its lower level of abstraction to gain performance by directly controlling scheduling.

2.1.1 Core Limitations of GPGPU Programming

A brief summary of other major characteristics of the current GPGPU programming style, many of which impact performance and are directly targeted by the Persistent Threads programming style, are outlined below:

1. **Host-Device Interface:**

   Master-slave processing: Only the host (master) processor has the ability to issue commands for data movement, synchronization, and execution on the device outside of a kernel.

   Kernel size: The dimensions of a block, and the number of blocks per kernel invocation, are passed as launch configuration parameters to the kernel invocation API.

2. **Device-side Properties:**

   Lifetime of a block: Every block is assumed to perform its function independent of other blocks, and retire upon completion of its task.

   Hardware scheduler: The hardware manages a list of yet-to-be executed blocks and automatically schedules them onto a multi-processor (SM) at runtime. As scheduling is a runtime decision, the programming style offers no guarantees of when or where a block will be scheduled.

   Block state: When a new block is mapped onto a particular SM, the old state (register and shared memory) on that SM is considered stale, disallowing any communication between blocks, even when run on the same SM.

3. **Memory Consistency:**

   Intra-block: Threads within a block communicate data via either local (per-block) or global (DRAM) memory. Memory consistency is guaranteed between two sections within a block if they are separated by an appropriate intrinsic function (typically a block-wide barrier).

   Inter-block: The only mechanism for inter-block communication is global memory. Because blocks are independent and their execution order is undefined, the most common method for communicating between blocks is to cause a global synchronization by ending a kernel and starting a new one. Inter-block communication through atomic memory operations is also an option, but may not be suitable or deliver sufficient performance for some application scenarios.

4. **Kernel Invocations:**

   Producer-consumer: Due to the restrictions imposed on inter-block data sharing, kernels can only produce data as they run to completion. Consuming data on the GPU produced by this kernel requires another kernel.

   Spawning kernels: A kernel cannot invoke another copy of itself (recursion), spawn other kernels, or dynamically add more blocks. This is especially costly in cases where data reuse exists between invocations.

2.2 Persistent Threads Programming Style

The requirement imposed by the nonPT programming style of dividing a workload into several blocks, more than can physically reside simultaneously at kernel launch time, is a design choice of the programming style, and not due to some constraint imposed by the SM. From a hardware perspective, threads are active throughout the execution of a kernel. This differs from a developer's perspective using nonPT style coding guidelines by implying that as blocks run to completion, threads corresponding to these blocks are "retired", while a batch of threads are "launched" as new blocks are scheduled onto the SM.

The Persistent Threads style of programming alters the notion of the **lifetime** of *virtual* software threads, bringing them closer to execution lifetime of *physical* hardware threads, i.e. the developer's view is that threads are active for the entire duration of a kernel. This is achieved by two simple modifications to kernel code: First, the virtualization of hardware SMs is limited to the level shown in Figure 2(a), also referred to as *maximal launch*. Second, the lack of additional blocks shown in Figure 2(b) is compensated by employing *work queues*. Both these are described in greater detail below. The PT style of coding is much closer to how one would program CPUs or Intel's Larrabee processor [14], which exposes the scheduler for the programmer to influence, if desired.

1. **Maximal Launch: A kernel uses at most as many blocks as can be concurrently scheduled on the SM:**

   Since each thread remains persistent throughout the execution of a kernel, and is active across traditional block boundaries until no work remains, the programmer schedules only as many threads as the GPU SMs can concurrently run. This represents the upper bound on the number of threads with which a kernel can launch. The lower bound can be as small as the number of threads required to launch a single block. In order to distinguish nonPT and PT thread blocks, we will refer to blocks in PT style programming as *thread*

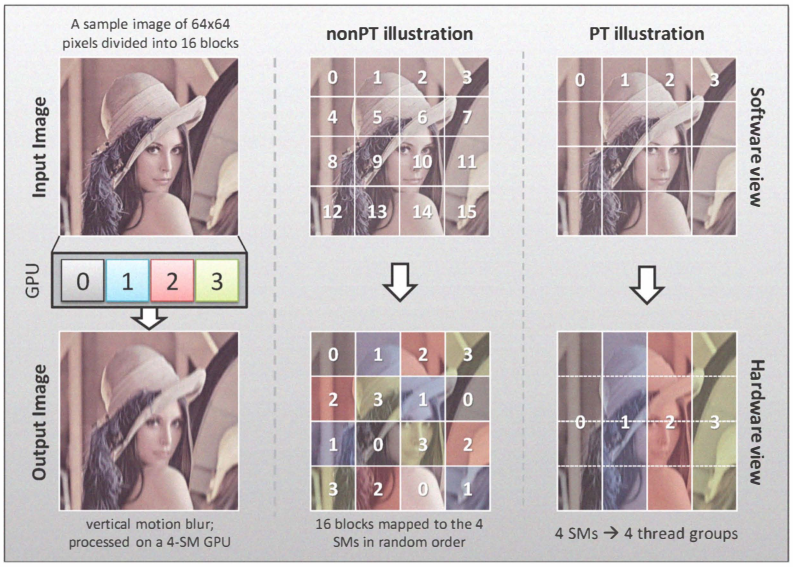

Figure 3: An illustration of nonPT and PT programming styles through an example image of 64x64 pixels undergoing vertical motion blur — In this example, 16x16 threads combine to form a block and process the image in a set of 4x4 blocks. The GPU has four SMs labeled 0-3. We assume a load-balanced system where each SM runs one block at a time and four blocks each. For nonPT, the kernel launches with 16 blocks. At run time the hardware non-deterministically schedules new blocks to SMs as other blocks complete. A PT kernel launches at-most the number of blocks corresponding to 'maximal launch', which in this example correspond to four thread groups, along with a 16-entry work queue; with each entry in this queue representing a block index. Through the work queue, developers can control scheduling of blocks onto SMs. In this example, each SM processes all blocks within a vertical sub-section of the image, thus helping data-reuse as pixels are shared across vertical block boundaries.

groups for the remainder of this paper. A thread group has the same dimensions as a thread block, but is formed by combining persistent threads launched at kernel invocation, processing work equivalent to zero, one, or many thread blocks, and remaining active until no more work is left for the kernel to process.

2. **Software schedules work through work queues, not hardware:**

   The traditional programming environment does not expose the hardware scheduler to the programmer, thus limiting the ability to exploit workload communication patterns. In contrast, the PT style bypasses the hardware scheduler by relying on a work queue of all blocks that are to be processed for kernel execution to complete. When a block finishes, it checks the queue for more work and continues doing so until no work is left, at which point the block retires.
   Depending on the communication pattern exhibited by the algorithm, the queue can either be static (known at compile time) or dynamic (generated at runtime) and can be used to control the order, location, and timing of the execution of each block. There are several kinds of optimizations one could incorporate in how work is submitted and fetched from these queues – one example would be to incorporate distributed queues instead of a global queue that could lead to optimal load balancing via work stealing/donating or a hybrid of the two approach. In this paper we focus on the base case of a single queue which primarily as a FIFO.

3. USE CASES

Over the past two years, several researchers have used PT techniques in their applications to improve performance. We have categorized these into four use cases, each of which are described in detail in the following four subsections.

PT-style programming lacks native hardware and programming API support. While past studies have shown PT to be advantageous, the benefits and limitations of hacking the nonPT model to fit into a PT style in software is not well understood and documented, and therefore cannot be universally applied to all scenarios. Although a handful benchmark suites are available for GPU computing [5, 6], these workloads primarily deal with performance at a higher level than we wanted for our analysis. Hence, apart from distinctly categorizing use cases in this paper, another contribution of this paper is the creation of an un-biased micro-kernel benchmark suite that future studies can use/compare against, and that vendors can use to evaluate future designs.

For each use case, we give a brief introduction and cite relevant studies that have utilized the use case. We provide implementation details of our synthetic workloads followed by results and conclusions. We wrote our tests in both OpenCL and CUDA. Specifically, since OpenCL provides the ability for fine-grain profiling, we wrote the first use case in OpenCL, and the remaining in CUDA. We ran all our use cases on an NVIDIA GeForce GTX295[1]. Un-

[1]We also tested on a GTX580 (Fermi) and found the general trends were similar, even though we made heavy use of atomics and the Fermi architecture supposedly has better atomic support (atomics in Fermi happen in the L2 cache/shared

Use Case	Scenario	Advantage of Persistent Threads
CPU-GPU Synchronization	Kernel A produces a variable amount of data that must be consumed by Kernel B	nonPT implementations require a round-trip communication to the host to launch Kernel B with the exact number of blocks corresponding to work items produced by Kernel A.
Load Balancing	Traversing an irregularly-structured, hierarchical data structure	PT implementations build an efficient queue to allow a single kernel to produce a variable amount of output per thread and load balance those outputs onto threads for further processing.
Maintaining Active State	A kernel accumulates a single value across a large number of threads, or Kernel A wants to pass data to Kernel B through shared memory or registers	Because a PT kernel processes many more items per block than a nonPT kernel, it can effectively leverage shared memory across a larger block size for an application like a global reduction.
Global Synchronization	Global synchronization within a kernel across thread blocks	In a nonPT kernel, synchronizing across blocks within a kernel is not possible because blocks run to completion and cannot wait for blocks that have not yet been scheduled. The PT model ensures that all blocks are resident and thus allows global synchronization.

Table 2: Summary of Persistent Threads use cases

less otherwise stated, every block consists of 256 threads in our experiments, and as a proxy for work per thread, we perform a varying number of fused multiply-add (FMA) operations on single-precision floating-point values. In the following subsections, "more FMAs per item" is equivalent to "more work per item". For each experiment, we implemented two versions of our synthetic benchmarks, nonPT and PT (both with equivalent functionality), to evaluate performance and identify tradeoffs. The four use cases are CPU-GPU synchronization, load balancing/irregular parallelism, producer-consumer locality, and global synchronization, which are summarized in Table 2.

3.1 CPU-GPU Synchronization

Since the CPU (*host*) and GPU (*device*) are coupled together as a master and slave, the device lacks the ability to submit work to itself. Outside of the functionality built into a kernel, the device is dependent on the host for issuing all data movement, execution and synchronization commands. In cases where a *producer kernel* generates a variable number of items for the *consumer kernel* to process at run time, the host must issue a readback, determine the number of blocks needed to consume the intermediate data, and then launch a new kernel. The readbacks have significant overhead, especially in systems where the host and device do not share the same memory space or are not on the same chip.

Boyer et al. present a systematic approach for accelerating the process of detecting and tracking in-vivo blood vessels of leukocytes [3]. By modifying memory access patterns, reformulating the computations within the kernel, and fusing kernels to minimize the CPU-GPU synchronization overhead, they report a 60× speedup over the baseline naive implementation. Further, upon restructuring the computations that originally required 50,000 kernel invocations to just one PT invocation, they achieved a 211× speedup over the baseline. Tzeng et al. utilized a work-queue based task-donation strategy for efficiently handling split and dice stages in the Reyes pipeline [16]. Their work splits micropolygons of a scene into a variable number of patches at each step. With the nonPT traditional programming style, this would require a host readback of the number of patches to be split in the next iteration. But by combining these two stages into a single uberkernel and wrapping it in a PT kernel, they eliminated the need for this readback and created a self-sustaining Reyes engine that efficiently handles their irregular workload.

memory, not in DRAM).

3.1.1 Implementation

Overhead Characterization: In order to characterize the overhead of host-device synchronization, we performed a timing analysis of the three steps involved, shown in red as the critical path in Figure 4-I: a blocking readback by the host to determine the subsequent launch bounds, the host overhead of generating configuration parameters for a kernel with the appropriate number of items, and the kernel launch time from the moment of issue to the moment the device starts the kernel.

To obtain an accurate assessment of the synchronization overhead, we used the low-level profiling API, *clGetEventProfilingInfo()*, provided in OpenCL (CUDA does not provide the fine-grained control we need, hence why we did not implement this use case in CUDA). This call provides four instances pertaining to when a call in the command queue is *issued* by the host, *dispatched* to the device execution queue, *begins* execution on the device, and the *end* time. For this experiment, we computed the time it takes from the moment the producer kernel (*GPU-kP*) ends to when the consumer kernel (*GPU-kC*) starts.

Persistent Threads Alternative: The PT alternative requires two modifications. First, *GPU'-kC* is reformulated to encapsulate the original kernel in a *while* loop, along with an *atomic* operation to either increment or decrement the number of items remaining. Second, the modified kernel is launched with a fixed number of blocks by the host, regardless of how many items are to be processed. When the work queue counter exceeds the item count in the case of an incremental queue or goes below zero in the case of a decremental queue, the corresponding block exits. At runtime, the last block of the *GPU-kP* writes the number of items *GPU'-kP* must process to a predefined GPU-accessible lo-

cation. *GPU'-kP* then uses this value as its terminal count. Thread groups retire once they reach this count.

Both modifications outlined here in the PT formulation are sources of overhead. In order to further understand this overhead w.r.t. nonPT kernels, we created two synthetic workloads, one that was compute-intensive (CI), and the other a combination of compute and memory (CMI) operations. For the CI kernel we read data from memory, perform a variable number of FMAs, and then write the result to global memory. For the CMI kernel we perform strided access to memory for every iteration of FMA computation, committing the result to global memory after the pre-determined number of operations has been performed. In both cases, we unroll the FMA loop completely in order to minimize loop overhead. For the nonPT case, the time of execution is the sum of performing a synchronous data transfer from GPU to CPU, and kernel execution.

3.1.2 *Results & Conclusions*

Using empty kernels we measured the round-trip cost of a synchronized copy to the CPU and the launch on the GPU to be around 400 μs. In addition to running this use case on a NVIDIA GTX295, we also ran it on a GeForce 9400M processor. A more detailed analysis of PT vs. nonPT performance for a varying number of FMAs and blocks for our workloads on both the processors is shown in Figure 5.

From Figure 5(a) we see that non-compute-intensive kernels benefit from a PT formulation. As the number of blocks to process increases, the atomic pressure increases considerably, resulting in a slowdown. As the arithmetic intensity of the amount of work processed per block increases, the PT formulation generally results in a speedup, eventually dipping below the nonPT baseline. In Figure 5(b) we see a different trend than for the CI case. While having fewer blocks results in significant speedup with a small amount of work per thread, as this work increases, we transition to a considerable slowdown. We attribute this, in part, to the drop in *occupancy*, the ratio of number of thread groups started at launched time to the theoretical maximum supported by the hardware. As the number of iterations to be unrolled increases, the compiler requires more registers, and is more restricted in making optimizations within the while-loop for the PT case.

In stark contrast, we see a very distinct set of results for these workloads on the GTX295. On both CI and CMI workloads, the PT version is almost always slower than its nonPT counterpart. The slowdown further increases as the number of blocks to process increase. The only reasonable explanation we hypothesize for this is the cost of contention for atomics. The 9400M has only 2 SMs while the GTX295 has 30.

From these workloads we conclude that using PT to avoid CPU-GPU synchronization is beneficial for small workloads with few memory accesses and small arithmetic intensity. Further, PT through software is better for kernels launching fewer thread groups, and where the cost of synchronization is a significant fraction of the overall cost of the application.

3.2 Load Balancing / Irregular Parallelism

Efficiently processing poorly-structured algorithms or irregular data-structures is a challenge on any compute device, but especially the GPU. Extracting parallelism on the GPU

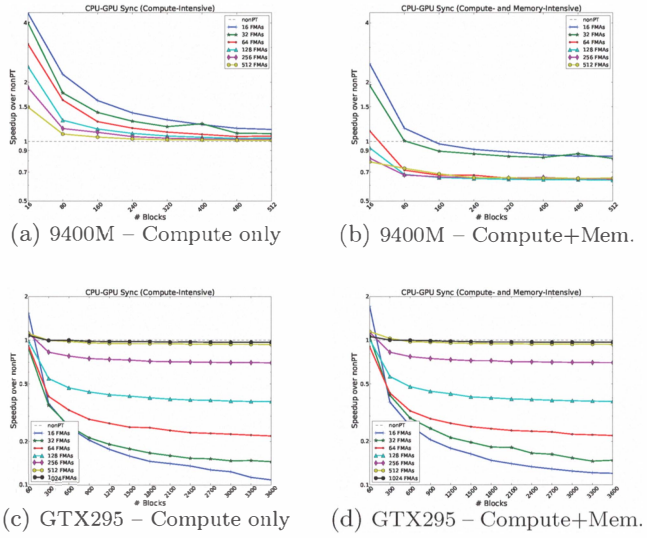

Figure 5: Results from CPU-GPU Synchronization on NVIDIA's GeForce 9400M and GTX295.

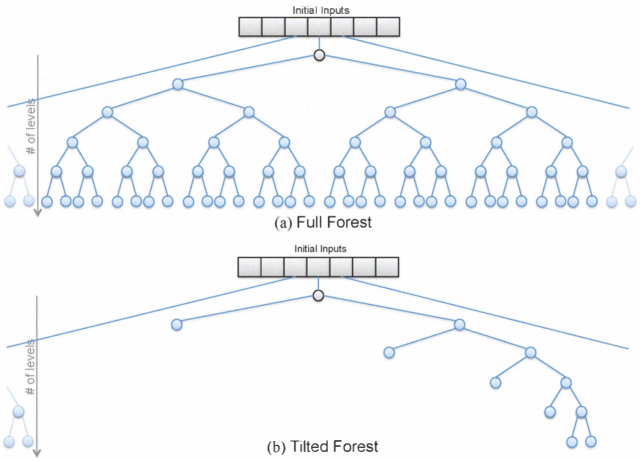

Figure 6: An illustration of our complete and tilted forest.

is left to the programmer, often requiring low-level control routines to be written in software. These workloads present two important issues—vector processor efficiency and handling variable work items being consumed or produced, Aila et al. present a thorough study of improving the efficiency of processing the irregular the *trace()* phase in ray tracing on vector processors through PT [1]. Their solution for handling the varying depths of traversal for each ray was by using warp-wide blocks for avoiding a single ray holding several warps within a block hostage[2],and bypassing the hardware scheduler by the use of PT. Tzeng et al. [16] addressed the

[2]We surmise that the large speedups seen by Aila et al. and other researchers in this area were in large part due to warp retirement serialization issues with pre-Fermi NVIDIA hardware, and that newer hardware does not suffer from the same issues. However, we note that this presumed hardware issue is only relevant to this particular use case; the other three use cases do not retire warps out of order.

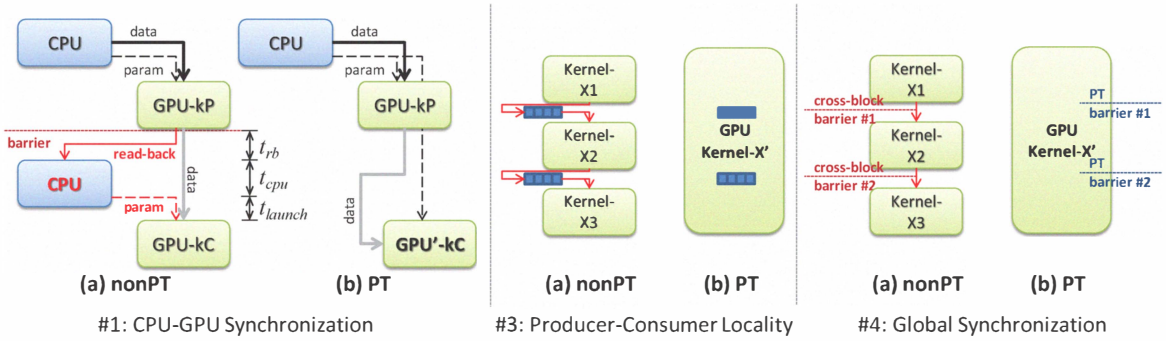

Figure 4: nonPT and PT illustrations of the use cases.

issue of variable producer/consumer work items using distributed work queues coupled with a task-donation strategy to mitigate load-balancing issues of the split and dice phases in the Reyes pipeline.

3.2.1 *Implementation*

In order to explore Load Balancing and Irregular Parallelism (LBIP), we design a test (a *forest expansion*) that emits a variable amount of work per thread and requires multiple stages to complete. The algorithm starts with an initial number of input items. Each thread performs a predefined amount of FMAs in register space, after which the owning block generates zero, one, or two new items (we tried using a larger maximum branching factor, but it had little effect on the resulting performance trends). The result is a forest, where items that generate another item(s) are either roots or intermediate nodes, and items that do not generate work are leaves. Every block processes one item at a time using all available threads.

We experimented with two different types of forests. We call the first a *complete forest*, since every input item generates a complete binary tree of a shallow, pre-specified depth. The second we call a *tilted forest*. In it, the "first half" of the items on any given level generate no new items, and every item in the "second half" generates two new items, shown in Figure 6. Work generation halts at a pre-specified depth. The tilted-tree model has the same number of nodes on every level (the first level might differ by one item). Instead of halting the forest growth at a shallow level, we let the tree grow to a pre-specified depth of at most one thousand.

nonPT Implementation: Each nonPT kernel invocation processes one or more items comprising a complete level of the forest. Each node generates zero or two new items. We can safely store only the current level and the next level of the forest since we work on a level at a time. Each level of the tree has a counter dictating how many items are in that level. For a given block, we check the global index of the block and terminate immediately if there are fewer items than this index, otherwise the block processes the input and helps to populate the next level of the forest (until the forest reaches a certain depth). To achieve perfect load balancing (a 1-to-1 matching of number of blocks and items at the current level), we could issue a readback before each kernel invocation to get a tight bound on the number of blocks we need, or since we know implicitly the new number of items in the next level (we know the initial number and that ev-

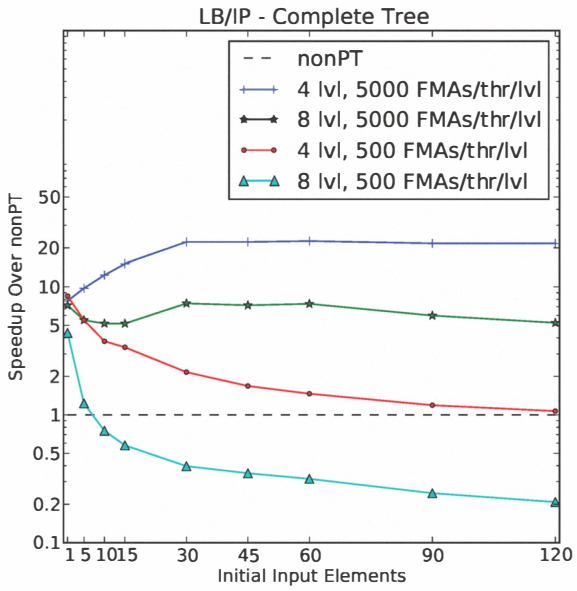

Figure 7: Load Balancing/Irregular Parallelism Results (Complete Forest): We run with a large variety of MADs per thread per level (0–50,000), maximum depth (1–12), and the number of PT blocks (1–240). The graph shows the general trends we notice from all runs. We see a general loss of speedup as the number of elements increases as the NonPT kernel has more work to do. We see PT speedup greater than 1.0 when the number of MADs is high, starting somewhere between 1000 and 5000 per thread per level.

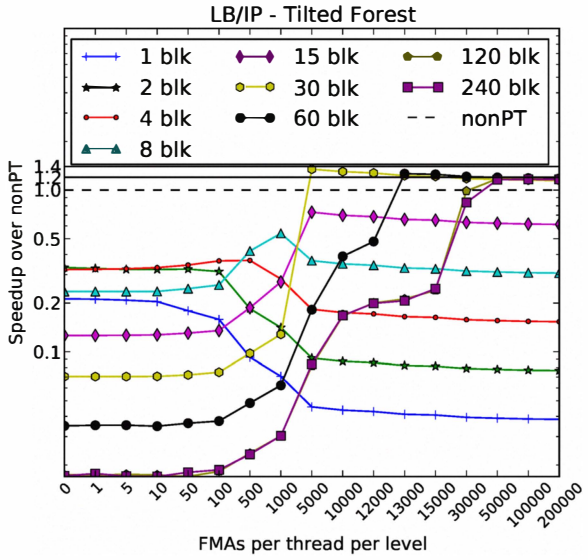

Figure 8: Load Balancing/Irregular Parallelism Results (Tilted Forest): We again run with a large variation in our configuration. However, we found that as we varied the maximum depth and the number of inputs, the trends stayed consistent. So we show our results from a maximum depth of 50 with 128 initial inputs. We found similar behavior as we varied the number of initial elements. This graph shows the general trends we notice from all runs. As we saturate the SMs with at least 30 blocks, we achieve better performance than the NonPT kernel when we have a high amount of work per thread per level.

ery item creates two new items at each level), we could just use that knowledge. But we felt doing either of these went against the spirit of the use case, since real application probably would not have full expansion, and it might be slower to perform the readback than to just schedule the empty blocks.

When processing an item, we assume that a block knows how many new items its current item will generate before actually processing the item. This enables us to cheat slightly by maximizing the distance between the atomic increment and the use of the calculated value. The complete forest application knows ahead of time how many levels of items to process, so it launches the correct number of kernels all at once in a single stream, thus lowering kernel-launch overhead. For the tilted forest, we do not allow the application to presume knowledge of the final level of work. Thus, the application schedules X levels of work at a time and then performs a readback to determine if any work remains. We did not want to assume that either application knew how many items were on any level on either test, so we scheduled the maximum number of blocks necessary to process a full level. This caused a performance hit for our nonPT kernel because we allowed each level to grow quite large (on the order of millions of elements for the complete forest and tens of thousands of element for the tilted forest).

PT Implementation: Our PT kernel is different from our nonPT kernel in that it lacks implicit knowledge of the level of an item. We use a single work queue to store work items instead of two queues and a ping pong strategy (as with nonPT). Our queue grants access via an atomically guarded spin lock. Each block loops until it detects program termination via an atomically guarded variable. As a block enters an iteration of the loop, it locks the queue, reads an item, and releases the lock. If the block reads a valid item, it processes that item and determines how many new items to generate. The block locks the queue to add new items and then unlocks the queue. Locking is important both for adding and removing elements since it alone guarantees coherency of the queue pointers. When no more items are available and all blocks are idle, the kernel exits. In theory the implementation of our PT kernel is quite simple, but correctly manipulating the locks and counters for the queue is tricky.

Our PT implementation uses a shared, global queue. There are other queuing implementations such as many global queues, or a global queue with local queues combined with task stealing and/or donation. We used a single queue based on prior tests, implementation time, performance, and the desire to keep low the number of variables in our experiments. We also noted that with the complete-tree implementation, using small local queues offered no significant performance improvement because the queues fill up rapidly (often even at the end of every level). For the tilted tree, one or more blocks must constantly steal work from other queues, again mitigating any performance improvements.

3.2.2 *Results & Conclusions*

LBIP has a large number of variables, both for the nonPT and the PT kernel. The common variables between them are the initial number of input items, the amount of work per item, and the depth of the forest. We also varied the total number of blocks to use per SM for PT and the number of invoked kernels per read back (to check for termination in the nonPT kernel).

We conclude that the more items per level, the poorer performance for PT, because the hardware scheduler will always best an ad hoc software scheduler. The stable and downward trends in Figure 7 illustrate this point. Concurrent atomic pressure is a major factor in our PT kernel. We use only round-trip atomics, meaning we need to wait for the result of every atomic operation, and thus each block must wait for the atomic pipeline to flush. The atomic pipeline in NVIDIA GPUs (both on Tesla and Fermi) does not scale linearly with the number of concurrent atomics (in fact the behavior seems non-deterministic and suffers from starvation in cases where atomics are used as locks). However, as each block spaces out queue-access requests with significant amounts of work, the number of concurrent accesses decreases, the overall time to access the queue decreases rapidly, and the ratio between the two (and thus queue access time in general) quickly becomes negligible. Figure 8 shows this trend very well: once the number of MADs per thread per level increases beyond a certain threshold (dictated by the number of active threads on the GPU), performance stabilizes in relation to nonPT.

Our experiments show two different trends. First, regardless of the number of FMAs per item, PT tends to to outperform nonPT with a small number of initial inputs. However,

as the number of inputs grows, PT only does well when the kernel executes a lot of work per item. This is due to concurrent atomic pressure. With many items (regardless of work per item), nonPT blocks spend most of their time doing actual work, instead of simply coming alive, seeing no work, and dying. With many items, but with little work per item though, PT blocks spend a significant fraction of their time accessing the high-contention queue. As the amount of work per item increases, access to the queue spreads out, thus lowering contention and significantly speeding up queue-access time.

Based on our workloads on the GPU under study, we conclude that for the general case, PT is better on specific combinations of variables. PT outperforms nonPT on small, irregular work and regular deeply-recursive work, and in either of our forest implementations PT tends to outperform nonPT when there are not many initial input elements and when the growth in elements is fairly constrained.

3.3 Producer-Consumer Locality

Many algorithms require data sharing between different threads within a kernel. *Producer-consumer locality* refers to the ability of one piece of work to pass its results to another piece of work with minimal cost. Current hardware allows data sharing by threads in the same block via on-chip memory, but threads in different blocks must communicate through DRAM and use expensive intrinsics to guarantee coherency. This limitation is an artifact of the lifetime of a block (threads retire after finishing a small amount of work), and the inability to express explicit communication patterns that aid in guaranteeing coherence without using DRAM. PT addresses this limitation.

In previous work in this area, Bell and Garland [2] use persistent warps for passing the "carry-out" of one iteration as the "carry-in" of the next in their COO format for SpMV computations. Breitbart [4] details a prefix-sum implementation that requires a single kernel launch by using static workgroups as opposed to many kernel launches. Recent sorting and prefix-sum scan work also uses PT to minimize global memory traffic by exploiting producer-consumer locality [8, 13].

3.3.1 Implementation

To test how PT deals with producer-consumer locality, we use two different workloads. The first is a large reduce function; given a set of N floats, compute their sum. The second is a dummy benchmark that executes a synthetic workload of FMAs in two stages. We begin with a set of N inputs. In the first step, we perform a certain amount of arithmetic on each item, then in the second step, perform additional arithmetic on only a certain percentage of those items.

nonPT Reduce: We use a multi-kernel approach for our nonPT reduce. Given N floats, we schedule $\lceil N/256 \rceil$ blocks to reduce the initial input set. Each block reduces 256 floats down to 1 float and stores it back to DRAM. This reduces the input set size from N to $\lceil N/256 \rceil$. We continue to schedule kernels in the same manner until we reduce our set to a single item, the final value. This implies that we must schedule $\lceil \log_{256} N \rceil$ kernels to reduce any input set.

PT Reduce: We use a single kernel with at most 256 blocks to complete the two-stage process. The kernel statically divides the input. Each thread reads a value from the array, performs a reduction, and then when it has read all its values, performs a blockwise reduction. Once all blocks finish their reductions, they write their single value to DRAM and the kernel executes a global barrier. After the barrier, block 0 reads all values from DRAM and performs one final reduction.

nonPT Synthetic Workload: For our nonPT synthetic workload, we have two implementations; one for the case that *most/all* threads in the second stage would work on items passed from the first stage, and one for the case that *relatively few* threads in the second stage would work on items passed from the first stage.

The first implementation uses a single kernel. We parameterize the number of items to pass per block. Each of the 256 threads in a block reads a single float from DRAM, performs a certain amount of FMAs, conditionally progresses to the second stage (or dies), and carries with it a single resultant float stored in a register. Once in the second stage, the thread performs the same amount of arithmetic as the first stage and stores a single float out to DRAM. This implementation is most efficient when all input items produce an output item, meaning every thread in the first stage remains active during the second stage and also passes with it a value to compute in the second stage.

Our second implementation is similar to the above in terms of compute characteristics but differs by storing intermediate values to DRAM and using two kernels (one producer, one consumer). As each block finishes stage one, the threads conditionally store their intermediate data into a tightly packed array in DRAM (those who do not pass the condition simply die). Next, we launch the consumer with as many threads as input items; each thread reads an element from the array, performs the same amount of arithmetic as in the producer kernel, and writes the result to DRAM.

Our second approach is most efficient when relatively few input items produce an output item. In this case, since the consumer kernel can be configured for the appropriate number of items, it would better utilize hardware resources (in the first, many threads per block do not progress to the consumer stage, thus holding hostage many hardware threads per SM).

PT Synthetic Workload: We again use a single kernel written as an uberkernel, statically partition the entire input among blocks, and, as with the nonPT implementation, parameterize the number of items to keep per block. Each block has a shared-memory storage arena. Just as in the nonPT case, each thread again reads in one float, does a pre-determined number of FMAs, and conditionally stores the result into the shared memory queue. When the queue fills up, the entire block processes the queue until it is empty by moving an item from the queue into a register, performing the same amount of arithmetic per item as in the producer stage, and then storing the result to DRAM. The block then resumes processing more items from the producer stage.

3.3.2 Results & Conclusions

Reduce: Our PT implementation of reduce outperforms an optimized nonPT reduce at every size of input we tested, with speedup eventually leveling off around 1.1× as the amount of work to process increases. At small input sizes, the largest bottleneck in nonPT reduce is the global barrier accomplished via implicit kernel boundaries. By using a

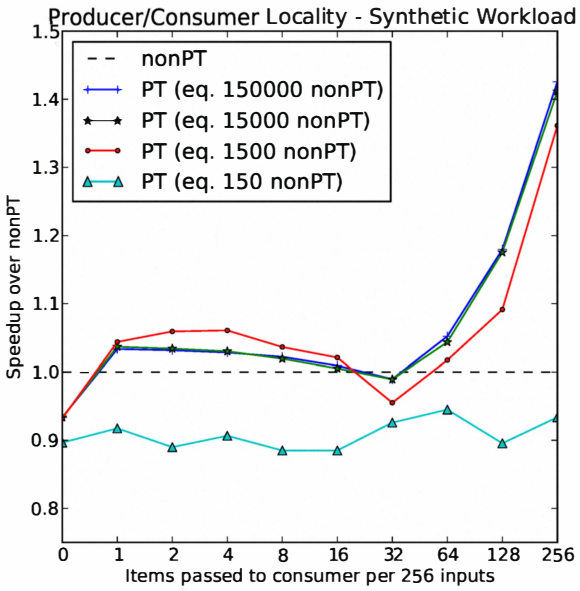

Figure 9: Producer-Consumer Results: We run our synthetic workload passing items from the producer stage to the consumer stage. We vary the number of items to pass, as well as the full number of initial items to process. As we vary the number of items to pass from 0 to 32 (per 256), we maintain performance between 0.95–1.05× that of a nonPT kernel. However, when we pass a large percentage (at least 25%) of the items to the consumer stage, we get up to 1.40× speedup, in large part because we have producer-consumer data locality and can pass items via registers and shared memory.

cheaper global barrier in our PT implementation we achieve a significant reduction in execution time, between 1.50× and 1.85×. As input size increases, the time taken to execute a barrier is small compared to DRAM access time. Thus, the performance improvements we see come not so much from using an explicit PT barrier, but from passing items via shared memory.

Synthetic Workload: We draw the following conclusions from the three major data points in Figure 9: When we pass no items from producer to consumer, we achieve slowdown with our PT implementation because the hardware scheduler is much better at load-balancing the SMs than our PT kernel. As the number of passed input items ranges between one and less than thirty-two (per block), we achieve only moderate speedup. As the number of input items to pass grows beyond thirty-two, we begin to achieve noticeable speedup, finishing off at approximately 1.45× for large input sizes where we pass every produced item to the consumer stage.

3.4 Global Synchronization

The GPU provides hardware support for synchronizing threads in a block, but not for synchronizing between active blocks on an the same SM, or the entire GPU. Current hardware guidelines dictate that the only way to globally synchronize is to launch separate kernels at would-be synchronization points. Kernel-launch overhead costs between 3-7 μs [17], which can lead to significant overheads depending on the number of kernel invocations [3]. Stuart and Owens used PT-based global synchronization in their implementation of a message-passing API for GPUs [15] to efficiently implement collective communications such as barriers. Luo et al. [7] present a hierarchical kernel arrangement for building a BFS through the use a global barrier to synchronize between blocks to mitigate the cost of kernel launch overhead.

We, and others, specifically propose the use of Global Synchronization (GS) through PT to minimize kernel-launch overhead. The GS construct is a barrier implemented in software for inter-block communication. The GS barrier acts on every block concurrently on the GPU. It can be used for separating different instances of the same kernel being processed recursively, or multiple sections of code within the same kernel in the case of uberkernel formulations. The GS construct has been shown to consume between 1.3–2.0 μs [18]. These results, however, only cover one data point, and in order to better understand the performance tradeoffs, we varied a wide number of parameters for our micro-benchmarks.

3.4.1 Implementation

nonPT: The nonPT implementation of our synthetic GS use case works as a two-step process, one on the GPU and one on the CPU. The GPU stage simply performs a predetermined number of FMAs in register space, then stores a value in DRAM. The CPU stage then implictly synchronizes by launching another kernel. We do not explicitly synchronize, nor do we need to, since the hardware guarantees that if kernels are launched asynchronously in a single stream, all blocks of the preceding kernel will run to completion before the next kernel begins. The point of our synthetic workload is to determine roughly how much work on average a block must perform to mitigate the cost of GS.

PT: The PT implementation of GS use case works as a single-step process consisting of two stages, with both stages executing on the GPU. Like the nonPT version, each block in the first PT stage performs its pre-defined set of operations. However, because blocks are persistent, each must perform the work of a variable number of blocks from the nonPT implementation. As each block completes an iteration, it begins processing the second stage. This stage is where blocks synchronize globally. Our implementation of GS is based on the lock-free version described by Xiao and Feng [18].

3.4.2 Results & Conclusions

We varied the number of blocks, the number of FMAs to be performed per input item, and the total number of blocks to process. The overall trend remains the same until 2000 FMAs per thread, but falls off dramatically as the compute intensity is increased. After inspecting the PTX, this seems to be due to the compiler no longer unrolling for loops. Figure 10 shows the graph obtained on running GS on a workload performing 1000 FMAs, with a varying number of items to process per block and a varying number of syncs. We see a consistent speedup of 2–2.5×, which is in line with the previously-reported results[3]. From our tests we conclude two things. First, the amount of arithmetic to

[3]We note one exception: we achieve less speedup when run-

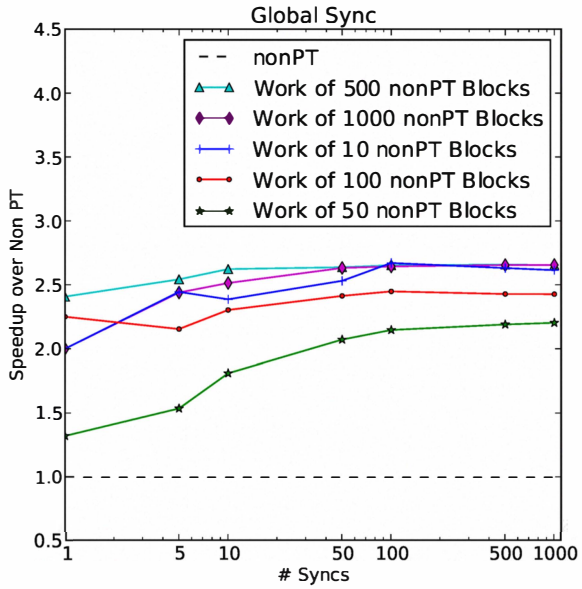

Figure 10: Global Synchronization Results: The runs for this graph used 1000 FMAs per input item per sync.

be performed has little bearing on the performance of global synchronization. Second, the benefit of syncing on the GPU increases asymptotically with the number of syncs: when each PT block executes a significant amount of work, the time taken to launch a new kernel is small in relation to the total execution time. Thus the application provides little room for a PT-based speedup.

From these results we conclude that the amount of arithmetic to be performed has little bearing on the performance of global synchronization, and the benefit of syncing on the GPU increases asymptotically with the number of syncs.

4. DISCUSSION

Based on previous work, and the micro-benchmarks discussed in Section 3, we have shown that the PT programming style can be used to address a range of computational patterns. PT addresses many of the restrictions mentioned in Section 2.1.1 pertaining to the current hardware and programming style. Taking a long-term view, we discuss the potential impact use cases outlined in this paper are likely to have with eminent changes in future, and possible changes hardware vendors and Khronos partners might consider useful to look into for future iterations of the programming environment.

1. **CPU-GPU Synchronization:** The cost of synchronization across the CPU and GPU is largely dependent on the round-trip time for data to be copied from GPU to CPU memory-space, and kernel launch overhead. For integrated processors with CPU and GPU on the same die, like AMD's Llano and Intel's Sandy Bridge processors, that will share dedicated on-chip memory for inter-processor communication in future revisions of the architecture, we see comparitively lesser benefit in using PT on such platforms when compared to discrete systems where the CPU and GPU are connected through a system interconnect bus like PCI-Express.

2. **Load Balancing:** Effectively utilizing all cores on any parallel processor is a difficult problem. Quite often the best way to load balance is by running task-parallel workloads, an area that is emerging as an important *primitive*. Hence, for this use case to be handled efficiently, it is imperative that there be dedicated support for queues in hardware with API extensions that provide tracking of head and tail pointers, queue wrap-around logic, overflow/underflow guard flags, nearly-full/empty flags, distributed queues, etc. Currently the developer is required to implement one in software, taking care of guaranteeing both data consistency and deadlock avoidance, which is non-trivial. In addition to this, the ability for kernels to generate and launch their own work would greatly ease the implementation of load-balanced and task-parallel systems.

3. **Producer-Consumer:** Extracting producer-consumer locality is the most challenging of all the use cases discussed in this paper. The nonPT style provides no way of expressing data locality patterns, which is especially important on massively data-parallel machines. Using cache-coherence could be one way of addressing this, but would be expensive to build and does not scale very well, and hence not popular in most GPUs today. The PT style helps address these issues by giving more control over scheduling and exploiting locality exhibited by the underlying algorithm. We see this as the biggest benefit of using PT style programming, with use even in future processor generations.

4. **Global Synchronization:** As the number of processors on GPU grows, so will the number of blocks that can be resident on it. This implies that beyond a point the cost of synchronizing on the GPU would be greater than the kernel launch overhead, and therefore for future systems might not be very useful at a global level. However, there are several patterns that require only a subset of blocks to communicate and synchronize, and hence it could still be quite useful to synchronize on the GPU. Ultimately, research into fast and good higher-level synchronization primitives than a barrier is required.

4.1 Implementation Aspects

As discussed in Section 2.2, the primary purpose of PT style programming is to bypass the default (traditional) programming style offered by CUDA and OpenCL by altering the lifetime of thread blocks and how they are scheduled. In providing a greater degree of flexibility to optimally deal with algorithms with complex compute, data movement and synchronization patterns, a much greater burden is put upon the developer as aspects that would otherwise have been handled transparently by the driver/hardware must now be done *manually*.

Even minor modifications in code can lead to enough change in register and shared memory usage that ultimately affects

ning the workload of 50 nonPT blocks and have yet to find a reasonable answer as to why.

multi-processor occupancy (the number of thread groups that can be simultaneously scheduled on the SM). In the absence of formal support for PT kernels, the programmer is required to manually account for changes in occupancy as it directly influences maximal launch parameters (especially relevant to the global synchronization use case), and also impacts the implementation of the underlying algorithm using complex queue-control structures. A list of portability aspects with reference to each use case are shown in Table 3. One possible means of extending support for PT is through a dedicated API. A sample of how simple this modification could be is shown in API 1.

4.2 Return-on-Investment

Our results clearly show that there is no guarantee of success for kernels designed via PT *today*. With limited native support either in the form of dedicated APIs or lower-level GPU primitives, developer time and effort involved in PT style programming, in a wide majority of cases, is significant. Further, due to the complexities involved in altering kernels to fit into a PT framework, debugging can also prove to be an extremely challenging task.

Based on our experiences and results, we suggest that any decision to use this programming style should be made after careful consideration as a significant investment in all aspects of design, implementation, and debug are involved, that require considerably more effort to program successfully than any nonPT implementation.

4.3 Power

Although we have not done an analysis regards impact of PT on power consumption, we believe this to be an interesting aspect for future work to consider, particularly relevant to low-power GPUs in consumer electronic devices. On the one hand as atomic operations are inherently serial in nature, they are likely to exhibit a poor power profile (the power consumption of PT with a high degree of contention that gets resolved in L2 cache or global memory is likely to be quite high), it can also lead to bypassing the hardware scheduler (that is one of the most complex parts of the modern GPU) thereby providing dynamic power saving opportunities by putting these transistors into power-save mode for the duration of the PT kernel.

5. CONCLUSION

GPU computing is still a nascent field, with its programming models and styles, and hardware that supports them, still in a rapid state of change. While GPU architectures have made steady strides since the adoption of *unified shaders* for supporting GPU computing workloads, there is a long list of potential advancements from the CPU world, from the supercomputing world, and that have yet to be developed that may be required to unlock its full potential. Support for recursion, memory management, and preemption, just to name a few, may eventually be in the GPU, but only after potentially significant redesigns of the architecture.

Rather than only inheriting techniques from the CPU world, one of the most exciting aspects of GPU computing is how promising software techniques developed first on the GPU quickly become system and hardware features. Our discussion in Section 4 shows that extending native support for PT would not be a particularly expensive exercise, and we certainly expect that some of them are already on the radar of future system architects. The most important use case in the broad sense is that of producer-consumer locality: locality is poorly supported by the traditional programming style, but PT shows a definite win over the traditional style in a realm that is absolutely critical for future system efficiency. Further investigation into how to expose locality in future programming models is an essential research direction for the community at large.

More broadly, the GPU is moving from an engine that supports only regular workloads to one that increasingly targets irregular ones with complex parallelism and dependencies. These modifications, while certainly helping PT implementations, promise to also allow the GPU the broader potential to support a wider variety of irregular workloads. Our tests clearly show that PT implementations are not necessarily an out-right winner, and do not in fact lead to significant gains for some use cases on our micro-benchmark workloads, *today*. However, these results could be markedly different in future as GPUs continue to evolve, and some of the suggestions discussed in this paper are incorporated by hardware vendors for PT style programming, natively. Our ultimate goal of this paper is to formally introduce this important programming style to the broader GPU computing community, and through an initial set of micro-kernel benchmarks to initiate a broader discussion on this topic. We hope that our work toward characterizing and understanding persistent threads will play an important role in simplifying and enhancing general-purpose applications on the GPU.

Acknowledgments

Thanks to our reviewers for their constructive and valuable feedback, and to Mark Harris, David Luebke, Erik Lindholm, Aaron Lefohn and Mike Houston for thoughtful discussions and suggestions on this work. We appreciate the support of an Intel Microprocessor Technology Lab grant, the SciDAC Institute for Ultrascale Visualization, and the National Science Foundation (grants OCI-1032859 and CCF-1017399).

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009*, pages 145–149, August 2009.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, pages 18:1–18:11, November 2009.

[3] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, May 2009.

[4] Jens Breitbart. Static GPU threads and an improved scan algorithm. In *Euro-Par 2010 Workshops: Proceedings of the Third Workshop on UnConventional High Performance Computing (UCHPC 2010)*, volume 6586

Use Case #	Occ.	Sch.	Comments
1. CPU-GPU Sync	—	—	indirectly impacts portability in the form of CPU-GPU *workload partitioning* based on application profiling analyses
2. LBIP	—	✓	for the simple case (of a single queue) we see minimal design and implementation impact, but when more complex queuing structures (local + global) and work stealing/donation optimizations are used, portability becomes non-trivial
3. Prod-Cons	✓	✓	requires different *kernel organization* and *partitioning* strategies for parallel-friendly algorithm modification on processors with varying compute capabilities
4. Global Sync	✓	—	changes in occupancy makes PT-based kernels extremely difficult to debug as the number of thread groups that must synchronize must be entered manually by the programmer

Table 3: Fundamental aspects that affect PT portability and usability based on occupancy and scheduling issues for our four use cases.

API 1 Example modifications to CUDA/OpenCL APIs for Persistent Threads.

CUDA API:
current : `<<< grid, block, shmem >>>`
proposed : `<<< grid, block, shmem, pt >>>`

OpenCL API:
current : `clEnqueueNDRangeKernel(cmdQ, kern, wkDim, gOff, *WrkGrp, *WrkItm, numEve, *EveList, *Eve)`
proposed : `clEnqueueNDRangeKernel(cmdQ, kern, pt, wkDim, gOff, *WrkGrp, *WrkItm, numEve, *EveList, *Eve)`

Description: *pt* can take the following parameters – (a) *NONPT*: nonPT kernel launch; (b) *MAX_TGROUPS*: PT kernel is invocated with maximal launch; (c) *USER_TGROUPS*: PT kernel is invocated with user-supplied *grid* or *wkDim*. To avoid global synchronization deadlock issues, if the user-supplied thread group size is greater than that possible for maximal launch, the kernel would exit with a pre-set error code.

of *Lecture Notes in Computer Science*, pages 373–380. Springer, August 2010.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, pages 44–54, October 2009.

[6] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing SHOC benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, March 2010.

[7] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, pages 52–55, June 2010.

[8] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.

[9] Aaftab Munshi. *The OpenCL Specification (Version 1.1, Document Revision 48)*, June 2010.

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, March/April 2008.

[11] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture, programming guide, 2011. `http://developer.nvidia.com/`.

[12] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[13] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[14] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, August 2008.

[15] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[16] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of High Performance Graphics 2010*, pages 29–37, June 2010.

[17] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 31:1–31:11, November 2008.

[18] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*, April 2010.