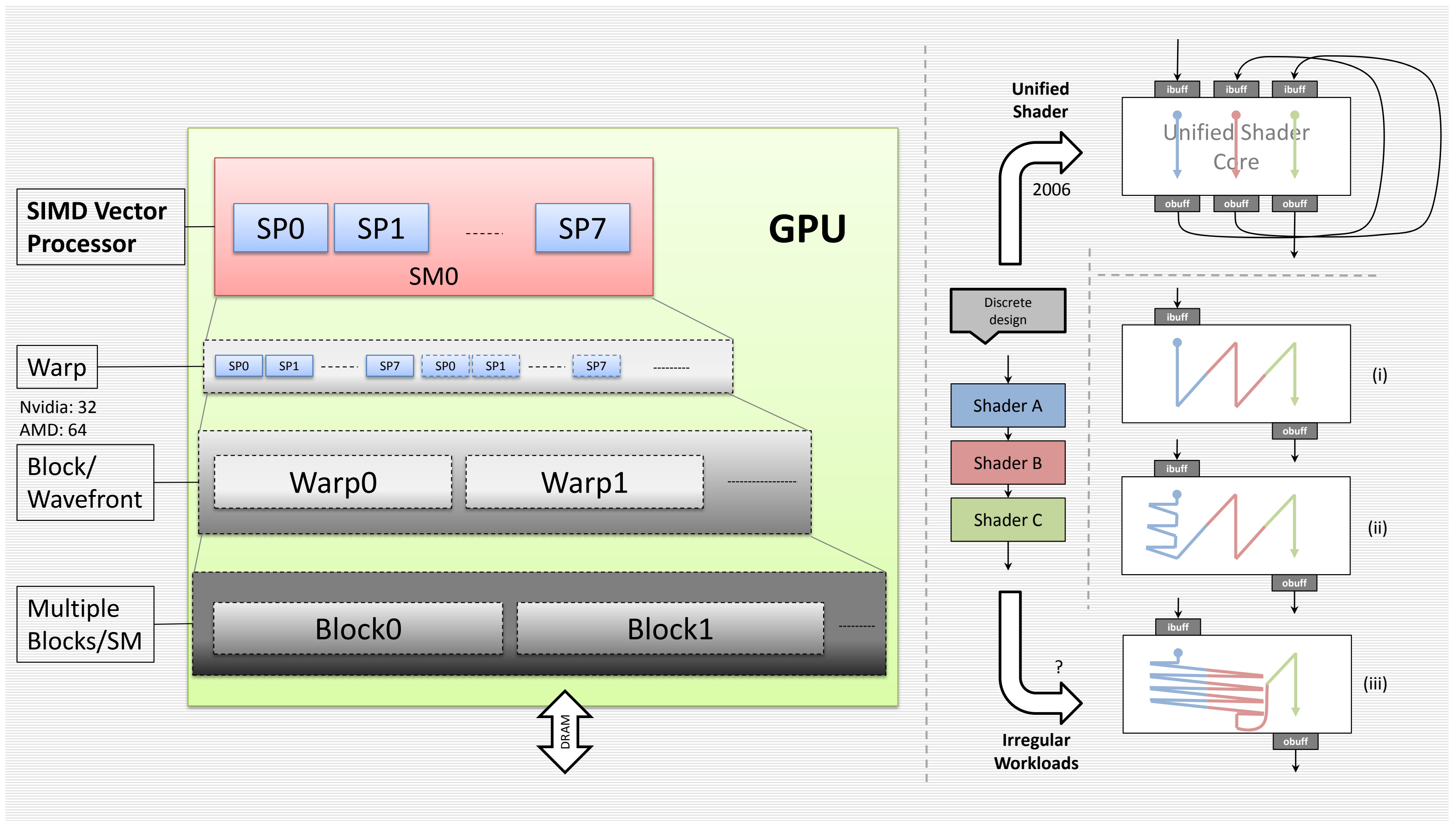
A Study of Persistent Threads Style Processing on the GPU

Kshitij Gupta, Jeff A. Stuart, John D. Owens
University of California, Davis

GPU Programming Hierarchy



Core limitations of current GPGPU programming*

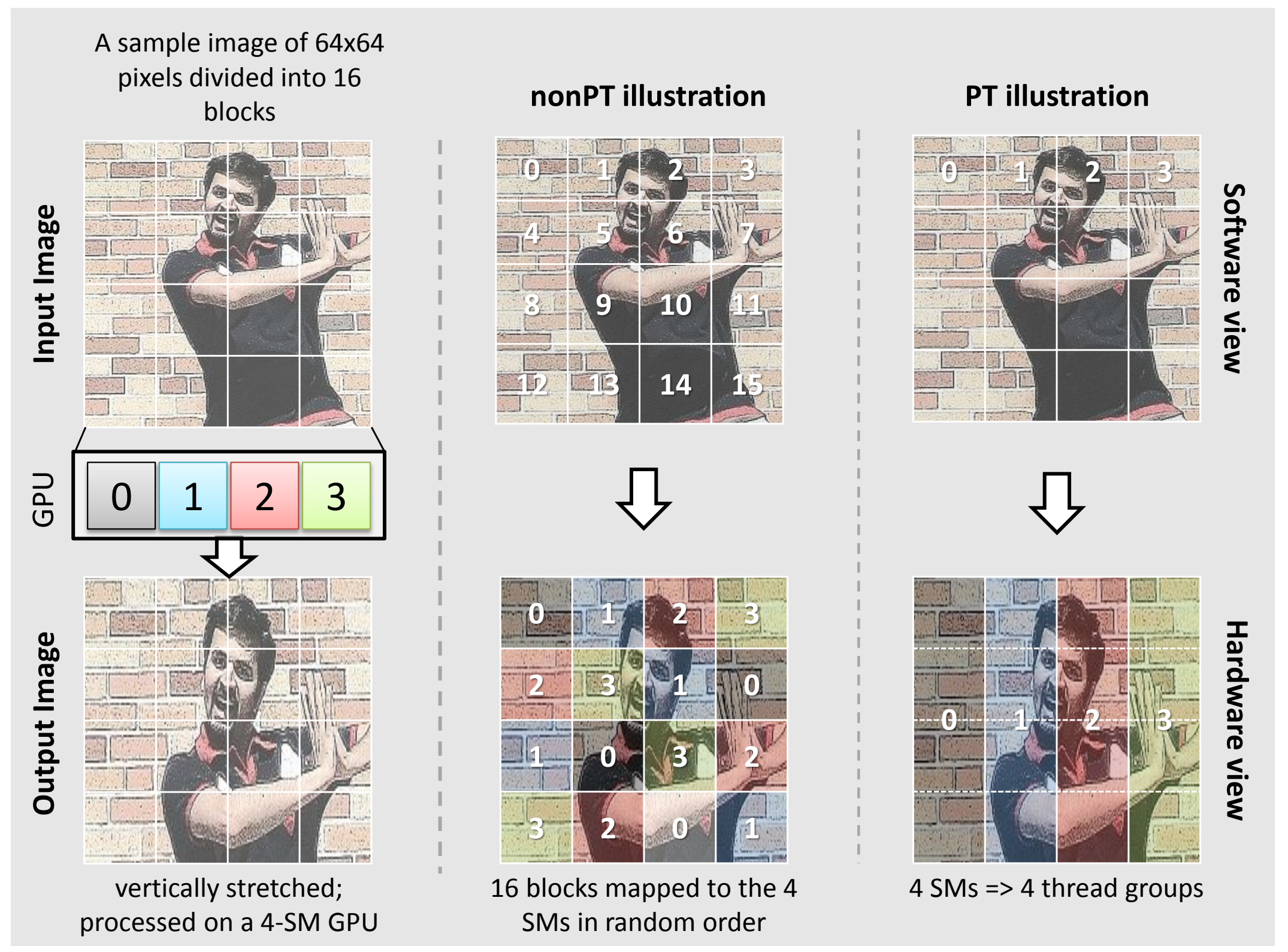
1. **Host-Device Interface:**
   - **Master-slave processing:** Only the host (master) processor has the ability to issue commands for data movement, synchronization, and execution on the device outside of a kernel.
   - **Kernel size:** The dimensions of a block, and the number of blocks per kernel 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 PM 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-kernel 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.

Persistent Threads is useful in addressing most of these limitations!

*All limitations listed here are applicable to OpenCL, while the newest version of CUDA coupled with the latest generation NVIDIA architecture has addressed some of these.

Salient characteristics of Persistent Threads

1. **Software, not hardware, schedules work:** The current 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.
2. **Maximal Launch – *A kernel uses only as many threads as can be concurrently scheduled on the SMs*:** 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. The lower bound can be as small as the number of threads required to launch a single block. Since a hardware thread and software thread do not have a direct relation in PT style of programming, we will distinguish software blocks from ***thread groups***. *A thread group has the same dimensions as a block, but forms by combining hardware threads launched at kernel invocation from one or more software blocks, and remains active until no more work is left.*

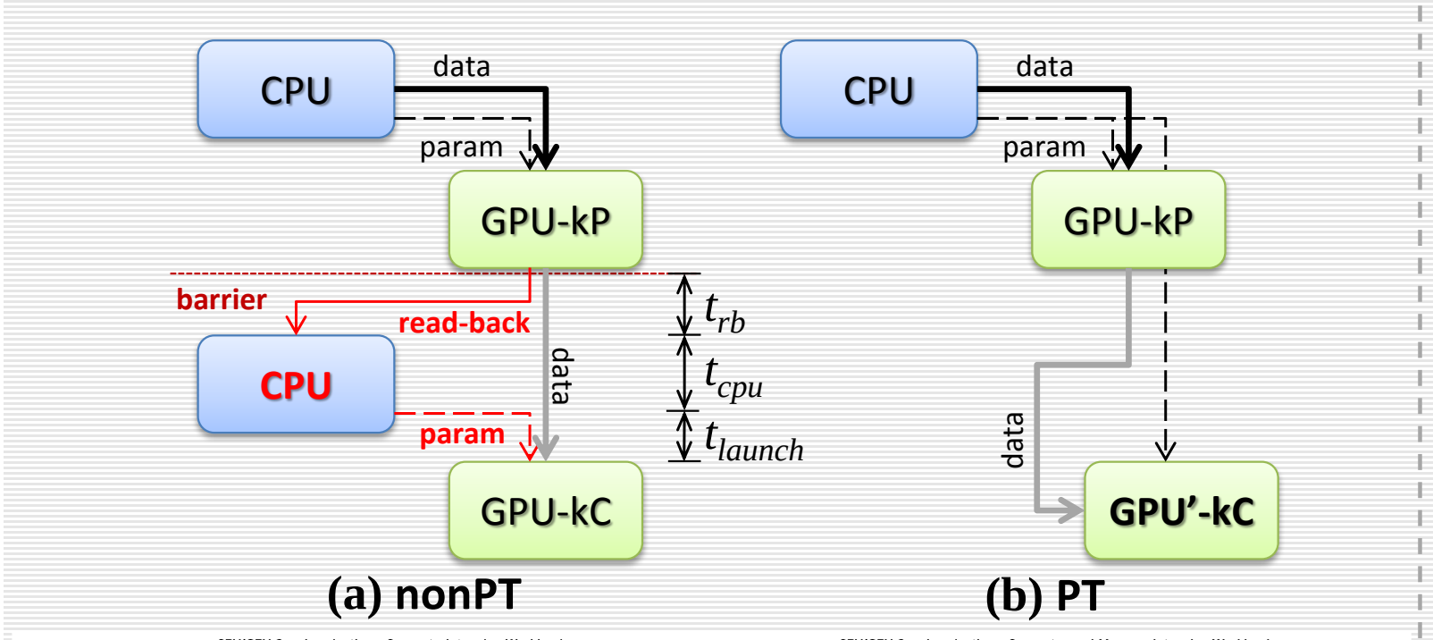

A sample image of 64x64 pixels divided into 16 blocks
nonPT illustration
PT illustration
Input Image
Software view
GPU
0
1
2
3
Output Image
Hardware view
vertically stretched; processed on a 4-SM GPU
16 blocks mapped to the 4 SMs in random order
4 SMs => 4 thread groups

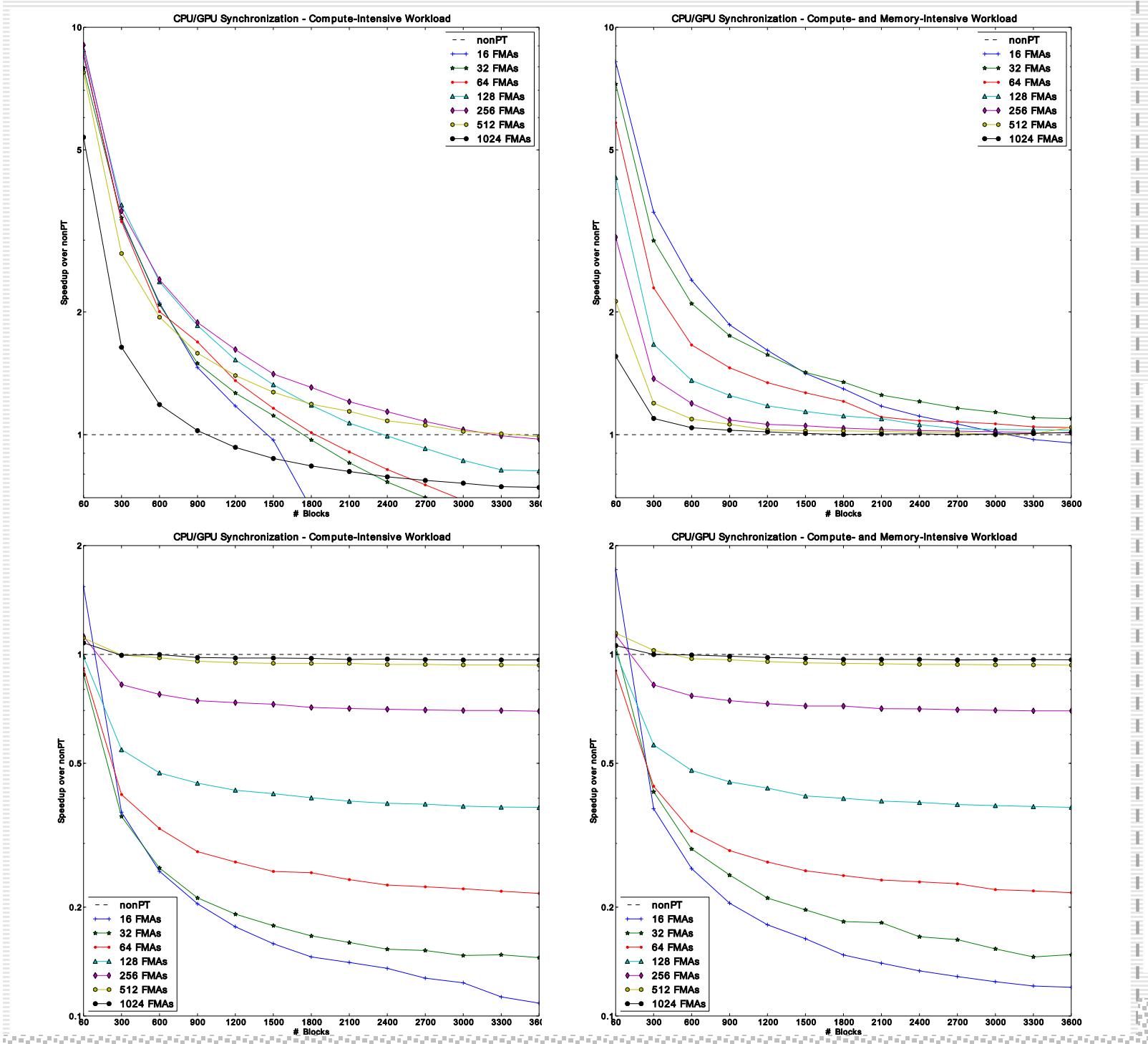
An illustration of nonPT and PT style programming through an example image of 64x64 pixels which needs to be vertically stretched - In this example, 16x16 threads combine to form a block and process the image in a set of 4x4 blocks. The GPU has four SMs labeled 0--3. We assume a load-balanced system 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 four thread groups, along with a 16-entry queue; each entry represents a block index. Through the work queue, the programmer controls the scheduling of blocks to SMs. In this example, each SM processes all blocks within a vertical sub-section of the image. This helps performance when the algorithm requires sharing neighboring pixels between vertical blocks.

#	Use Case	Scenario	Advantage of Persistent Threads
1	**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.
2	**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.
3	**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.
4	**Global Synchronization**	Global synchronization within a kernel across workgroups	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.

Use Cases - Results

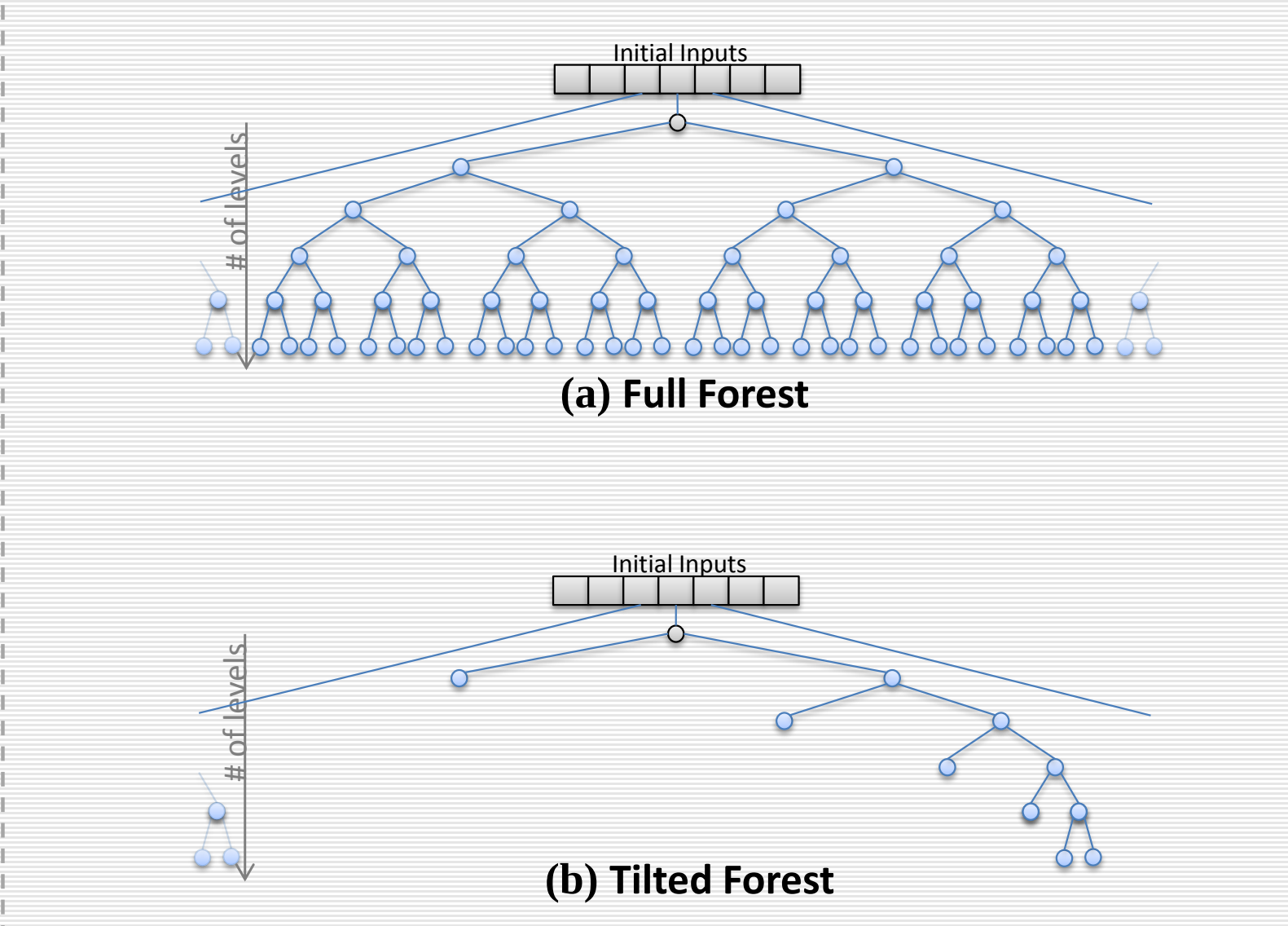
Use Case #1: CPU-GPU Synchronization

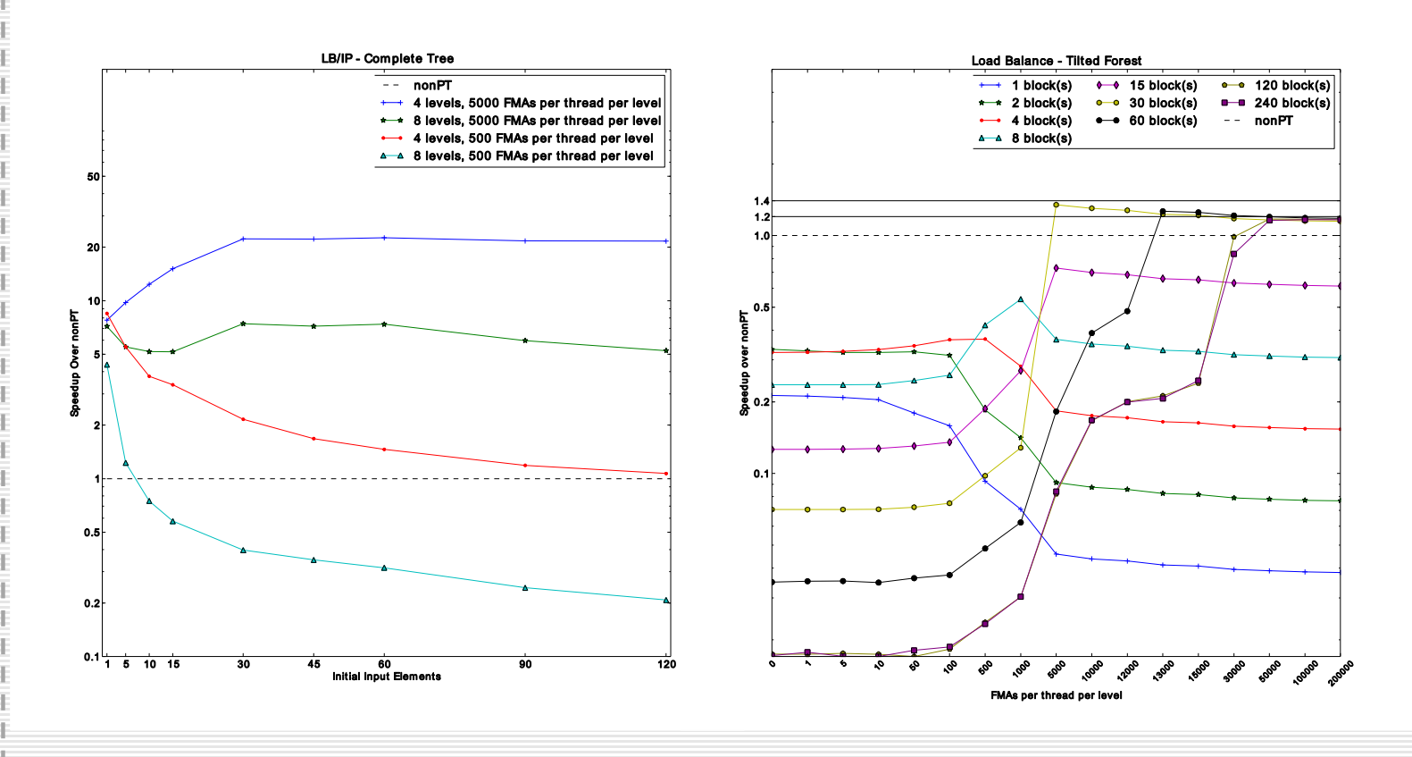




Conclusion: We see two different trends on two different processors. As the number of blocks to be processed increases, on the 9400M regardless of whether the workload is CI or CMI, we see a declining gain in speedup; while on the 295GTX we see a slow-down.

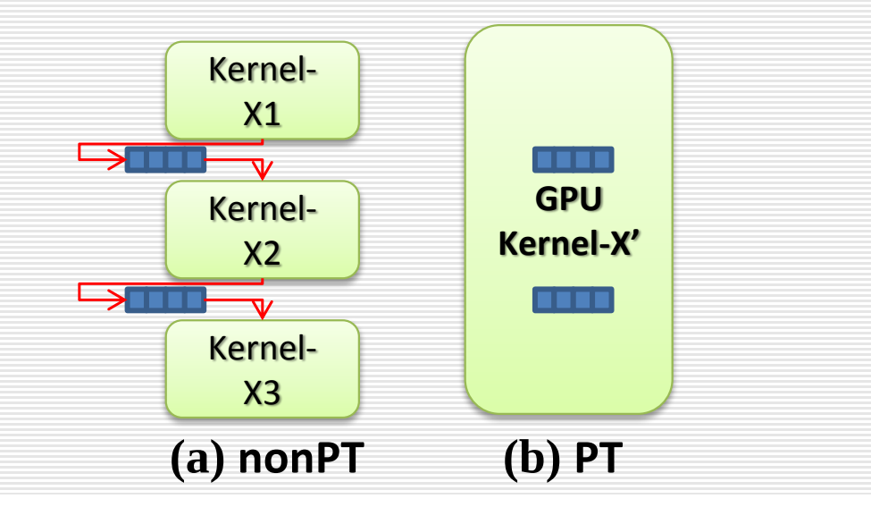
Use Case #2: Load Balancing

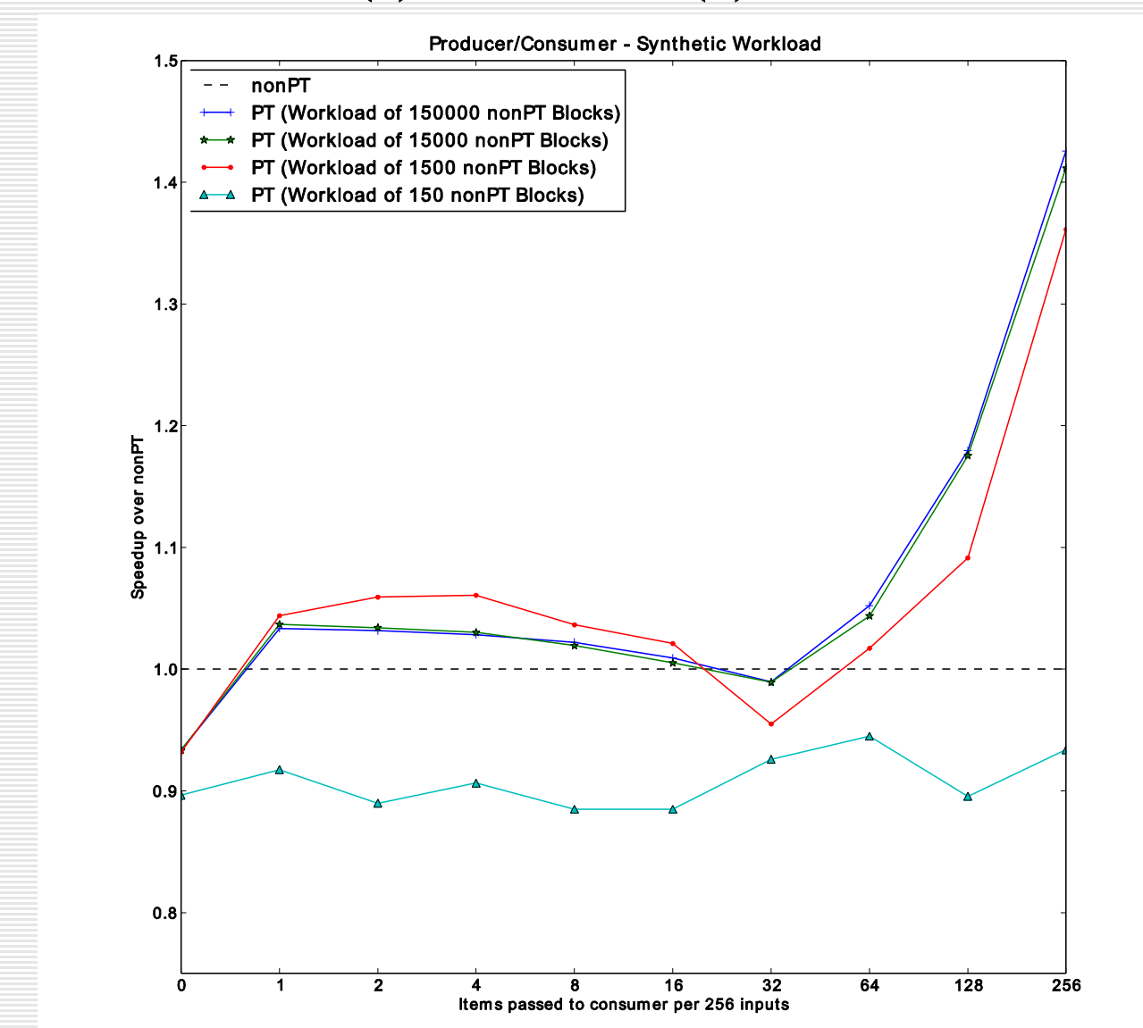




Conclusion: PT is better on specific combinations of variables. PT outperforms nonPT on small, irregular work and regular deeply-recursive work, and in either of our forest implementations PT tends to outperform nonPT when there are not many initial input elements and when the growth in elements is fairly constrained.

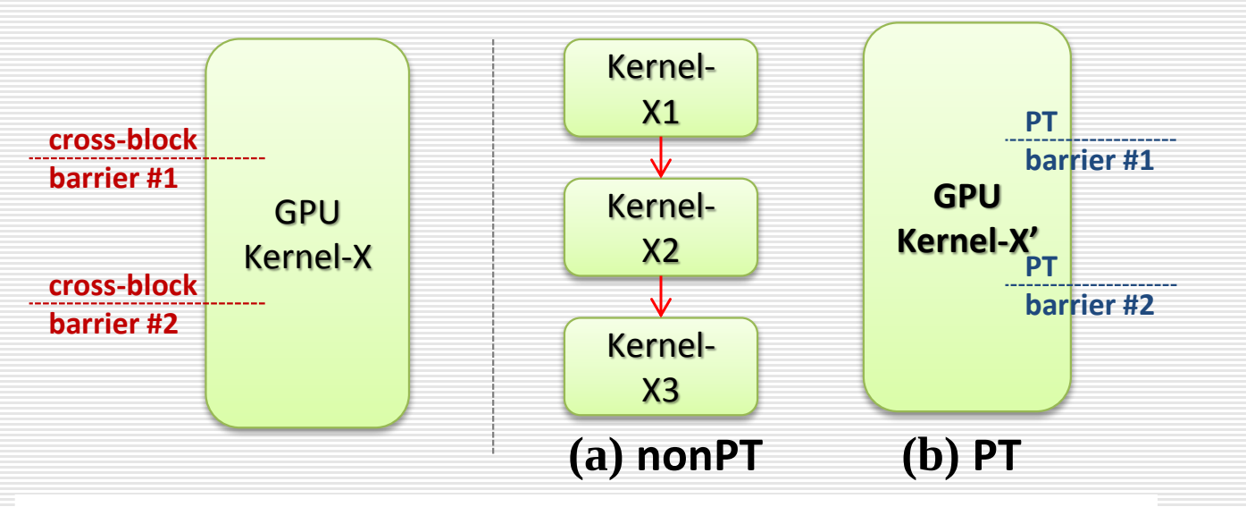
Use Case #3: Active State

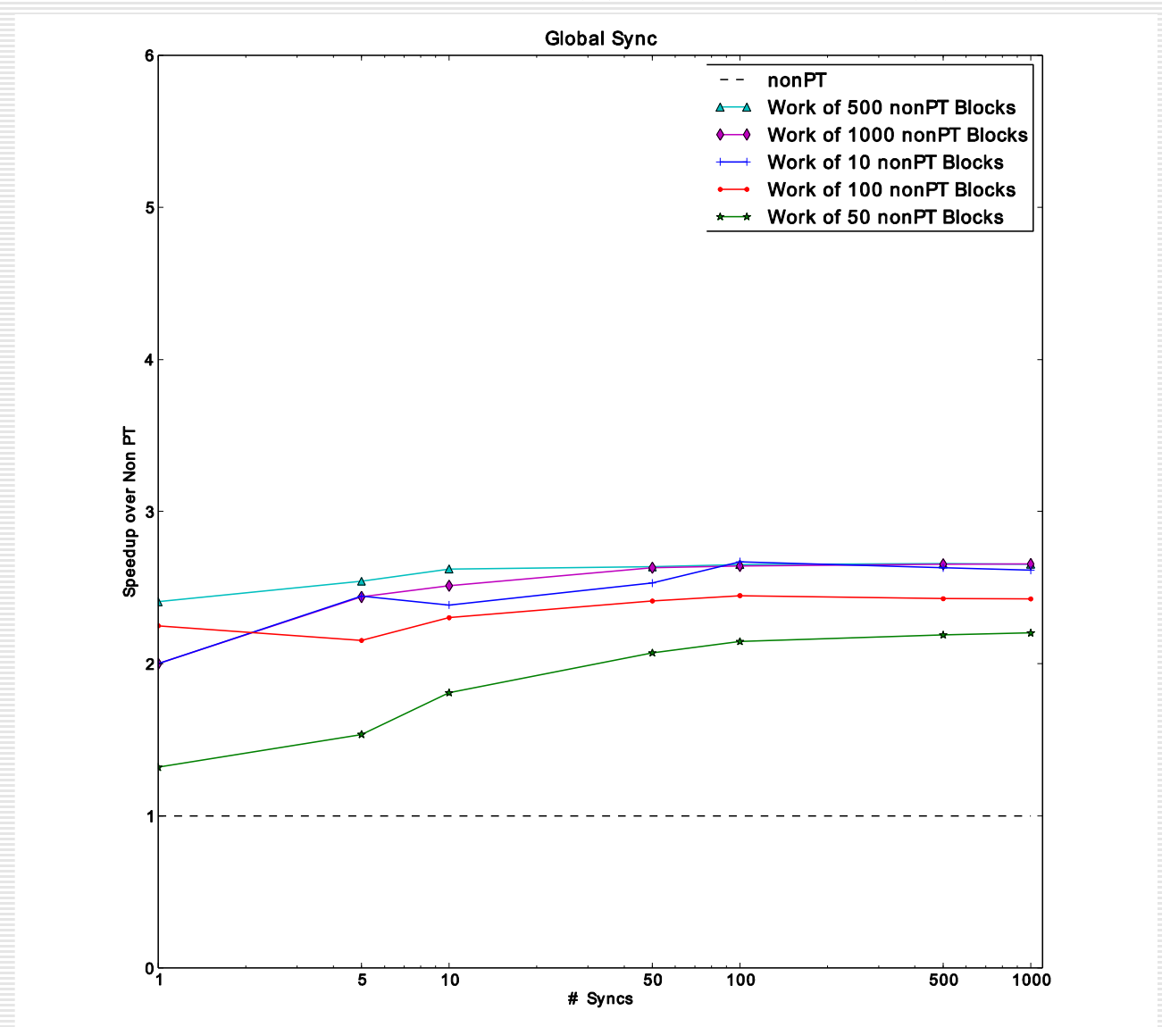




Conclusion: As the number of input items to be processed increases, a PT approach results in greater speed-up, noticeably beyond 32 on the GTX295 (which has 30 SMs).

Use Case #4: Global Synchronization





Conclusion: 1: The amount of arithmetic to be performed has little bearing on the performance of global synchronization. 2: the benefit of syncing on the GPU increases asymptotically with the number of syncs

Minor modifications for *native* PT support

1. **Hardware**
   - Work Queues
   - Fast Atomics
   - Multi-Block Synchronization
2. **Software**
   - Scheduler
3. **Language & API**
   - Comm. Pattern
   - API

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 GS 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.

PT brings order to the otherwise undefined behavior of (CUDA/OpenCL) nonPT model(s)