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 ready-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 of a block when the hardware manages the scheduling.
   - 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.

**Core limitations of current GPGPU programming**

---

**GPU Programming Hierarchy**

---

**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 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 kernel finishes, it checks the queue for more work and continues doing so until no work is left, at which point the block retires.

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 dimension 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.

---

**Use Cases - Results**

---

**Use Case #1: CPU-GPU Synchronization**

- **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
  - PT implementations build an efficient queue to allow a single kernel to produce a variable amount of output per thread and balance those outputs onto threads for further processing.

---

**Use Case #2: Load Balancing**

- **Load Balancing**
  - Traversing an irregularly structured, hierarchical data structure
  - 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.

---

**Use Case #3: Active State**

- **Active State**
  - A kernel manages a list of ready-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.

---

**Use Case #4: Global Synchronization**

- **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 all blocks are resident and thus allows global synchronization.

---

**Minor modifications for native PT support**

---

**1. Hardware**
- Work Queues
- Fast Atomics
- Multi-Block Synchronization

**2. Software**
- Scheduler
- Language & API
- Pattern, Comm. Pattern
- API

---

**Kshitij Gupta, Jeff A. Stuart, John D. Owens**

University of California, Davis