A Study of Persistent Threads Style Programming Model for GPU Computing

Kshitij Gupta  [/shi/ /tij/]
UC Davis
A Study of Persistent Threads Style GPU Programming for GPGPU Workloads

Kshitij Gupta, Jeff A. Stuart, John D. Owens
UC Davis
Outline

- GPGPU Programming ("nonPT")
  - Limitations

- Introduction to "PT"

- Use Cases

- Observations/Discussion
**NOTE:** (CUDA | OpenCL) Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Warp</td>
<td>--</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Grid</td>
<td>Index space</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Global memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Scalar core</td>
<td>Processing element</td>
</tr>
<tr>
<td>Multi-processor (SM)</td>
<td>Compute unit</td>
</tr>
</tbody>
</table>

* We will use CUDA terminology, but the same† discussion can be extended to OpenCL.
Preliminaries: GPGPU Programming Hierarchy

- SM_0
- SM_1
- SM_2
- SM_3

DRAM

GPU
Preliminaries: GPGPU Programming Hierarchy

- **SIMD**
- **SIMT**

**Hierarchy**
- **DRAM**
- **Warp**
- **SM0**
- **P0**, **P1**, **P7**
- **P0_0**, **P1_0**, **P7_0**
- **P0_X**, **P1_X**, **P7_X**
Preliminaries: GPGPU Programming Hierarchy
Preliminaries: GPGPU Programming Hierarchy

- SIMT
- SPMD
- Virtualize
- VirtualizE
- ViRtUaLiZe

Hierarchy:
- SM0
  - P0
  - P1
  - P7
- Warp
  - P0_0
  - P1_0
  - P7_0
  - P0_X
  - P1_X
  - P7_X
- Block
  - W0
  - W1
  - WN
- Block
  - B0
  - B1
  - BM
- DRAM

(a)
Preliminaries: GPGPU Programming Hierarchy

- **SIMT**
  - SM0
  - Warp: P0, P1, P7, P0_X, P1_X, P7_X
  - Block: W0, W1, WN

- **SPMD**
  - B0, B1, BM

- **Virtualize**

- **VirtualizE**

- **ViRtUaLiZe**

- **VIRTUALIZE!!**
Preliminaries: Workload Evolution

(a) Pre-2006: Discrete cores

(b) 2006: Stream programming – CUDA architecture with unified cores; along with ‘C for CUDA’
Preliminaries: Workload Evolution

(a) Pre-2006: Discrete cores

(b) 2006: Stream programming – CUDA architecture with unified cores; along with ‘C for CUDA’

(c) Today: A sample of irregular workload patterns
Core GPGPU Programming Characteristics (& Limitations)

1. Host-Device Interface
   - Master-slave processing
   - Kernel size

2. Device-side Properties
   - Lifetime of a Block
   - Hardware Scheduler
   - Block State

3. Memory Consistency
   - Intra-block
   - Inter-block

4. Kernel Invocations
   - Producer-consumer
   - Spawning kernels

Irregular workloads
Preliminaries:
Example – Image Processing (outside Pixar HQ)
A sample image of 128x128 pixels divided into 16 blocks.

vertical motion blur; processed on a 4-SM GPU.
A sample image of 128x128 pixels divided into 16 blocks; vertical motion blur; processed on a 4-SM GPU

nonPT illustration

16 blocks mapped to the 4 SMs in random order
Persistent Threads: Properties

1. Maximal Launch – A kernel uses only as many threads as can be concurrently scheduled on the SMs

2. Software, not hardware, schedules work
A sample image of 128x128 pixels divided into 16 blocks.

**Input Image**

Vertical motion blur; processed on a 4-SM GPU.

**GPU**

16 blocks mapped to the 4 SMs in random order.

**Output Image**

4 SMs → 4 thread groups.

**Software view**

**Hardware view**
Persistent Threads: Properties

1. Maximal Launch – *A kernel uses only as many threads as can be concurrently scheduled on the SMs*
   - **Thread-group** (v/s thread-block)
     - Upper-bound: maximal launch
     - Lower-bound: 1

2. Software, not hardware, schedules work
   - **Work-queues**
     - Several optimizations possible
     - In his paper – single global FIFO
Common Communication Patterns

- Linear
- Diagonal
- Zig-Zag
- Scanline
- Wavefront
- Pinwheel
- Checker
Use Cases

- μ-kernel benchmarks
- Workload comprises of FMAs
- Nvidia GeForce GTX295
# PT Use Cases

<table>
<thead>
<tr>
<th>#</th>
<th>Use Case</th>
<th>Scenario</th>
<th>Advantage of Persistent Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>CPU-GPU Synchronization</strong></td>
<td>Kernel A produces a variable amount of data that must be consumed by Kernel B</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Load Balancing</strong></td>
<td>Traversing an irregularly-structured, hierarchical data structure</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Maintaining Active State</strong></td>
<td>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</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Global Synchronization</strong></td>
<td>Global synchronization within a kernel across workgroups</td>
<td>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.</td>
</tr>
</tbody>
</table>


# Use Case #1: CPU-GPU Synchronization

## Scenario

Kernel A produces a variable amount of data that must be consumed by Kernel B.

## Advantage of Persistent Threads

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.

---

### Diagram

(a) nonPT

(b) PT
Use Case #1: CPU-GPU Synchronization
# Use Case #2: Load Balancing/Irregular Parallelism

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Advantage of Persistent Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traversing an irregularly-structured, hierarchical data structure</td>
<td>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.</td>
</tr>
</tbody>
</table>
Use Case #2: Workload Illustration – Tree(s)

(a) Full Tree

(b) Tilted Tree
Use Case #2: Workload – Complete Tree

![Diagram showing speedup over non-PT with different configurations of levels and FMAs/threads/levels.](image)
Use Case #2: Workload – Tilted Tree
# Use Case #3: Maintaining Active State

## Scenario

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Advantage of Persistent Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td>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.</td>
</tr>
</tbody>
</table>

## Diagram

- **(a) nonPT**
- **(b) PT**
Use Case #3: Workload Illustration – Reduction
Use Case #3: Workload – Reduction

![Graph showing speedup over nonPT for different workloads](image)
## Use Case #4: Global Synchronization

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Advantage of Persistent Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global synchronization within a kernel across workgroups</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Diagram:

(a) nonPT  
(b) PT
Use Case #4: Global Synchronization

![Graph showing speedup over non-PT with different workloads and number of syncs.](image)
## Portability & Usability

<table>
<thead>
<tr>
<th>#</th>
<th>Use Case</th>
<th>Occupancy</th>
<th>Scheduling</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU-GPU Synchronization</td>
<td>-----</td>
<td>-----</td>
<td>indirect; CPU-GPU workload partitioning</td>
</tr>
<tr>
<td>2</td>
<td>Load Balancing/Irregular Parallelism</td>
<td>-----</td>
<td>⬤</td>
<td>non-trivial when sophisticated queuing structures (local + global) and work stealing/donation optimizations are used</td>
</tr>
<tr>
<td>3</td>
<td>Maintaining Active State</td>
<td>⬤</td>
<td>⬤</td>
<td>different kernel organization and partitioning strategies</td>
</tr>
<tr>
<td>4</td>
<td>Global Synchronization</td>
<td>⬤</td>
<td>-----</td>
<td>hard to debug as occupancy changes</td>
</tr>
</tbody>
</table>
Looking Ahead...

<table>
<thead>
<tr>
<th>#</th>
<th>Use Case</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU-GPU Synchronization</td>
<td>• Less of an issue on future consumer systems; but HPC still a problem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• We expect this to be addressed in the future</td>
</tr>
<tr>
<td>2</td>
<td>Load Balancing/Irregular Parallelism</td>
<td>• Provide support for queues</td>
</tr>
<tr>
<td>3</td>
<td>Maintaining Active State</td>
<td>• Very hard to solve!</td>
</tr>
<tr>
<td>4</td>
<td>Global Synchronization</td>
<td>• Kernel launch might be cheaper than synchronizing across an entire chip in future-generation hardware</td>
</tr>
</tbody>
</table>
Looking Ahead...

- Return-on-Investment

- Power?

- Modifications?
  - Native support would not require a complete re-making of the underlying hardware
  - Small changes could lead to reasonable gains
  - Augment existing APIs?
Thank You!

Kshitij Gupta

www.kshitijgupta.com