Kernel Execution

Objectives

- Understand the operation of major microarchitectural stages in launching and executing a kernel on a GPU device
- Enumerate generic information that must be maintained to track and sequence the execution of a large number of fine grained threads
  - What types of information is maintained at each step
  - Where can we augment functionality, e.g., scheduling
Reading

- CUDA Programming Guide


Execution Model Overview

- GPU: Many-Core Architecture
- CUDA: NVIDIA’s GPU Programming Model

![Diagram of GPU architecture](image)
Baseline Microarchitecture

- Modeled after the Kepler GK 110 (best guess)
- 14 stream multiprocessors (SMX)
- 32 threads/warp
- 2048 maximum number of threads/SMX
- 16 maximum number of TBs
- Remaining stats
  - 4 Warp Scheduler
  - 65536 Registers
  - 64K L1/Share Memory

From Kepler GK110 Whitepaper

Kernel Launch

- Commands by host issued through streams
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams are mapped to hardware queues in the device in the kernel management unit (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
CUDA Context

- Analogous to a CPU process
- Encapsulate all CUDA resources and actions
  - Streams
  - Memory objects
  - Kernels
- Distinct address space
- Created by runtime during initialization and shared by all host threads in the same CPU process
- Different CUDA context for different processes
  - E.g. MPI rank

Baseline Microarchitecture: KMU

- Kernels are dispatched from the head of the HWQs
- Next from the same HWQ cannot be dispatched before the prior kernel from the same queue completes execution *(today)*
• Same #entries as #HWQs → maximum number of independent kernels that can be dispatched by the KMU

• Kernel distributor provides one entry on a FIFO basis to the SMX scheduler
Kernel Parameters

- Managed by runtime and driver
- Dynamically allocated in device memory and mapped to constant parameter address space
- 4KB per kernel launching

```c
.entry foo( .param .u32 len ) {
    .reg .u32 %n;
    ld.param.u32 %n, [%len];  // Load from parameter address space
    ...
}
```

SMX Scheduler

- Initialize control registers
- Distribute thread blocks (TB) to SMXs
  - Limited by #registers, shared memory, #TBs, #threads
- Update KDE and SMX scheduler entries as kernels complete execution
**SMX Scheduler: TB Distribution**

- Distribute TBs in every cycle
  - Get the next TB from control registers
  - Determine the destination SMX
    - Round-robin (today)
  - Distribute if enough available resource, update control registers
  - Otherwise wait till next cycle

**SMX Scheduler Control Registers**

- KDEI
- NextBL
- KDE Index
- Next TB to be scheduled

---

**SMX Scheduler: TB Control**

- Each SMX has a set of TBCRs, each for one TB executed on the SMX (max. 16)
  - Record KDEI and TB ID for this TB
  - After TB execution finishes, notify SMX scheduler
- Support for multiple kernels
Streaming Multiprocessor (SMX)

- Thread blocks organized as warps (here 32 threads)
- Select from a pool of ready warps and issue instruction to the 32-lane SMX
- Interleave warps for fine grain multithreading
- Control flow divergence and memory divergence

Warp Management

- Warp Context
  - PC
  - Active masks for each thread
  - Control divergence stack
  - Barrier information

- Warp context are stored in a set of hardware registers per SMX for fast warp context switch
  - Max. 64 warp context registers (one for each warp)
Warp Scheduling

• Ready warps
  - There is no unresolved dependency for the next instruction
  - E.g. memory operand is ready

• Round-robin
  - Switch to next ready warp after executing one instruction in each warp

• Greedy-Then-Oldest (GTO)
  - Execute current warp until there is an unresolved dependency
  - Then move to the oldest warp in the ready warp pool

Concurrent Kernel Execution

• If a kernel does not occupy all SMXs, distribute TBs from the next kernel

• When SMX scheduler issues next TB
  - Can issue from the kernel in the KDE FIFO if current kernel has all TBs issued

• On each SMX there can be TBs from different kernel
  - Recall SMX TBCR
  - Thread Block Control Registers (TBCR) per SMX
    - KDE Index
    - Scheduled TB ID (in execution)
Executing Kernels

Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels
### Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>Kernel 1</th>
<th>SMX1</th>
<th>Kernel 3</th>
<th>SMX12</th>
<th>Kernel 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td>Kernel 1</td>
<td>SMX1</td>
<td>Kernel 3</td>
<td>SMX12</td>
<td>Kernel 100</td>
</tr>
</tbody>
</table>

---

### Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

<table>
<thead>
<tr>
<th>SMXs</th>
<th>Kernel 1</th>
<th>SMX1</th>
<th>Kernel 3</th>
<th>SMX12</th>
<th>Kernel 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMX0</td>
<td>Kernel 1</td>
<td>SMX1</td>
<td>Kernel 3</td>
<td>SMX12</td>
<td>Kernel 100</td>
</tr>
</tbody>
</table>

---
Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

![Diagram of SMXs and kernels]

---

Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels

![Diagram of SMXs and kernels]
Occupancy, TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels
  
  Wait till previous kernel finishes

![Diagram showing SMXs and kernels]

Achieved SMX Occupancy: 32 kernels * 2 warps/kernel / (64 warps/SMX * 13 SMXs) = 0.077
  
  Wait till previous kernel finishes

![Diagram showing SMXs and kernels]
Balancing Resources

- Limit: 32 for concurrent kernels

Performance: Kernel Launch

- For synchronous host kernel launch
  - Launching time is ~10us (measured on K20c)

- Launching time breakdown
  - Software:
    - Runtime: resource allocation, e.g. parameters, streams, kernel information
    - Driver: SW-HW interactive data structure allocation
  - Hardware:
    - Kernel scheduling