Execution Scheduling w/ CUDA
The NVIDIA GPU Memory Ecosystem

October 2, 2015
“You will become way less concerned with what other people think of you when you realize how seldom they do.”

— David Foster Wallace (1962-2008)
Please terminate your interactive jobs on Euler when finished
[a message from your friendly Cluster Police team]

- The issue:
  - Eight GPUs reserved specifically for this class

- Executing `srun` with `--gres=gpu:1` will hold a GPU for you
  - Nobody else can use it

- People forget to release the GPU

- If you are not sure that you released your job type: “`squeue -u $USER`”

- Then type “`scancel JOBID`” where `JOBID` is the job number shown by `squeue`

[Hammad]→
Better Solution: Use `sbatch` instead of `srun`

- **Step 1:**
  - Create a “slurm.sh” file in your homework directory with the following contents

```bash
#!/bin/sh
#SBATCH --partition=slurm_me759
#SBATCH --time=00:05:00 # maximum run time in days-hh:mm:ss
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --error=/home/USER/cuda.err
#SBATCH --output=/home/USER/cuda.out
#SBATCH --gres=gpu:1
./your_homework_executable
```

- **Step 2:**
  - Execute “slurm.sh” by typing “`sbatch slurm.sh`”
Before We Get Started

- Issues covered last time:
  - CUDA Execution Configuration: threads, blocks, grids
  - CUDA API

- Today’s topics
  - Scheduling for execution on the NVIDIA GPUs
  - The CUDA memory ecosystem

- Assignment:
  - HW03 – due tonight at 11:59 PM
  - HW04 –due on Oct. Oct. 7 at 11:59 PM

- Midterm Exam: 10/09 (Friday)
  - Review on Th 10/08, at 7:15 PM, room TBA
We Are Already There...

- All that’s needed to get going has been discussed
  - Basic examples highlighting use of GPU computing w/ CUDA
  - Execution configuration
    - Grids, Blocks, Threads
    - Going from the thread index to the thread id
    - Figuring out a global index from a thread/block index combo
  - Timing a CUDA kernel
  - Discussed CUDA API
GPU Computing: Why go beyond this point?

- Hone our “computational thinking” skills

- “computational thinking” gained by
  - Programming
  - and more importantly,
  - Gaining a good understanding of how the hardware supports the execution of your code (the hardware/software interplay)

- Good programming skills ensures we get correct results
- Computational thinking allows us to get correct results fast
Execution Scheduling Issues
[NVIDIA cards specific]
Thread Execution Scheduling

- **Topic we are about to discuss:**
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM. How is this possible?
There are two schedulers at work in GPU computing

- A device-level scheduler: assigns blocks to SM that indicate at a given time “excess capacity”

- An SM-level scheduler, which schedules the execution of the threads in a block onto the functional units available to an SM

- The more interesting is the SM-level scheduler
Device-Level Scheduler

- Grid is launched on the device
- Thread Blocks are distributed to the SMs
  - Potentially more than one block per SM
  - There is a limit on the number of blocks an SM can take.
- As Thread Blocks complete kernel execution, resources are freed
  - Device-level scheduler can launch next Block[s] in line
- This is the first levels of scheduling:
  - For running [desirably] a large number of blocks on a relatively small number of SMs (16/14/etc.)
- Limits for resident blocks on one SM:
  - 32 blocks on Maxwell SMM
  - 16 blocks can be resident on a Kepler SMX
  - 8 blocks can be resident on a Fermi & Tesla SM
**SM-Level Scheduler[s]**

- Each Thread Block divided in 32-thread “warp”
  - “32”: selected by NVIDIA, programmer has no say

- Warps are the basic scheduling unit on the SM

- Limits, number of resident warps on an SM:
  - 64: on Kepler & Maxwell (i.e., 2048 resident threads)
  - 48: on Fermi (i.e., 1536 resident threads)
  - 32: on Tesla (i.e., 1024 resident threads)

- EXAMPLE: If 3 blocks are processed by an SM and each Block has 256 threads, how many warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- Cycles needed to dispatch the same instruction for all threads in a warp
  - On Tesla: 4 cycles
  - On Fermi: 1 cycle

- How is this relevant?
  - Suppose you use a Fermi card AND your code has 1 global memory access every 10 simple instructions
  - Then, a minimum of 40 Warps are needed to fully tolerate 400-cycle memory latency:

\[
\frac{400}{10} = 40 \rightarrow 40 \text{ Warps are needed}
\]
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads
- One warp issued at each clock cycle by each scheduler
- During no cycle can more than 2 warps be dispatched for execution on the four functional units
- Scoreboarding is used to figure out which warp is ready to go
Example: Fermi Related (GTX480)

- Scheduler works at 607 MHz
- Functional units work at 1215MHz

Question:
- What is the peak flop rate of GTX480?
  - 15 SMs * 32 SPs * 1215 * 2 (Fused Multiplied Add) = 1166400 Mflops
  - That is, 1.166 Tflops, single precision
Fermi Specifics

- As illustrated in the picture, at no time can we see more than two warps being dispatched for execution during a cycle.
- Note that at any given time we might have more than two functional units working though (which is actually very good, device kept busy).
# NVIDIA GPUs: Architecture Specifications

<table>
<thead>
<tr>
<th>Architecture specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Number of ALU lanes for integer and floating-point arithmetic operations</td>
<td>8\textsuperscript{[19]}</td>
</tr>
<tr>
<td>Number of special function units for single-precision floating-point transcendental functions</td>
<td>2</td>
</tr>
<tr>
<td>Number of texture filtering units for every texture address unit or render output unit (ROP)</td>
<td>2</td>
</tr>
<tr>
<td>Number of warp schedulers</td>
<td>1</td>
</tr>
<tr>
<td>Number of instructions issued at once by scheduler</td>
<td>1</td>
</tr>
</tbody>
</table>
The Kepler SM
(called SMX, since it’s eXtreme)

Compute capability 3.0
The Maxwell SM
(called SMM)

Compute capability 5.0
## Technical Specifications and Features

<table>
<thead>
<tr>
<th>Technical specification</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum y-, or z-dimension of a grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threads are Executed in Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently
Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing
  - Warp scheduling is not something the user can control in CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead.
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling.
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion.
Revisiting the Concept of Execution Configuration

- Prefer thread block sizes that result in mostly full warps

**Bad:** \( \text{kernel}^{<<<N, 1>>>}( \ldots ) \)

**Okay:** \( \text{kernel}^{<<<(N+31)/32, 32>>>( \ldots )} \)

**Better:** \( \text{kernel}^{<<<(N+127)/128, 128>>>( \ldots )} \)

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency

- Resource like `__shared__` may constrain number of threads per block

- Algorithm and decomposition of problem will reveal the preferred amount of shared data and `__shared__` allocation
  - We often have to take a step back and come up with a new algorithm that exposes parallelism
Scheduling: Summing It Up…

- When host invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity

- Up to 8 blocks (on Fermi) can be executed at the same time by an SM
  - Up to 16 on Kepler, 32 on Maxwell

- When a block of threads is executed on an SM, its threads are grouped in warps. The SM manages several warps at the same time
  - Up to 64 warps can be managed on Kepler and Maxwell. Up to 48 warps on Fermi

- When a thread block finishes, a new block is launched on the vacated SM
Thread Divergence

[1/4]

- Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads (even i) in the warp execute the `if` clause, the other half (odd i) the `else` clause
Thread Divergence

[2/4]

- The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

- Often, this requires that the hardware execute multiple paths through a kernel for a warp.
  - For example, both the if clause and the corresponding else clause.
Thread Divergence

[3/4]

```c
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
```
Thread Divergence

[4/4]

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one does need to consider divergence when reasoning about the performance of a program

- NVIDIA calls execution model SIMT (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here’s an extreme example...

```c
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
    case 0 : x[i] = a(x[i]);
             break;
    case 1 : x[i] = b(x[i]);
             break;
    ...  
    case 31: x[i] = v(x[i]);
             break;
    }
}
```
Performance of Divergent Code

Compiler and hardware can detect when all threads in a warp branch in the same direction
  - Example: all take the if clause, or all take the else clause
  - The hardware is optimized to handle these cases without loss of performance

```c
if (threadIdx.x / WARP_SIZE >= 2) { }
```

- Creates two different control paths for threads in a block
- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses