CUDA Execution Model - II

Objectives

- Understand the sequencing of instructions in a kernel through a set of scalar pipelines (width = warp)
  - Thread blocks and warps
  - Synchronization
  - Mapping to hardware execution units
  - Scheduling and resource management
  - Programmer’s view
- Have a basic, high level understanding of instruction fetch, decode, issue, and memory access
Reading Assignment

• Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 6.3

• CUDA Programming Guide

• Get the CUDA Occupancy Calculator
  - Find at nvidia.com
  - Use web version at http://lxkarthi.github.io/cuda-calculator/

Recap

```c
__global__ void vecAddKernel(float *A_d, float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n) C_d[i] = A_d[i] + B_d[i];
}
```
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) (more later)
  - Multiple streams mapped to each queue → serializes some kernels
- Kernel launch distributes thread blocks to SMs

CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Also referred to as cooperative thread arrays (CTAs)
- Programmer declares block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work

© David Kirk / NVIDIA and Wen-mei Hwu, 2007-2012 · ECE408 / CS483 / ECE498al, University of Illinois, Urbana-Champaign
Execution Semantics

- Execution of a thread block is partitioned into warps

NVIDIA GK110 (Kepler)

Image from http://mandetech.com/2012/05/20/nvidia-new-gpu-and-visualization/
SMX Organization

Multiple Warp Schedulers

64K 32-bit registers

192 cores – 6 clusters of 32 cores each

Warp size

Each thread block is allocated a slice

Image from http://mandetech.com/2012/05/20/nvidia-new-gpu-and-visualization/

Execution Sequencing in a Single SM

Single instruction stream

Cycle level view of the execution of a thread block

#lanes = warp size

(9)
TBs, Warps, & Scheduling

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

**Thread Blocks**

```
......
```

**SMXs**

```
SMX0  SMX1  SMX12
```

TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- Microarchitecture structures to track thread blocks and warps

**Thread Blocks**

```
......
```

**SMXs**

```
SMX0  SMX1  SMX12
```
• One TB has 64 threads or 2 warps

• Limits on #thread blocks/SM

Thread Blocks

SMXs

Remaining TBs are queued

SMXs
Example: VectorAdd on GPU

CUDA:

```
__global__ vector_add(
    float *a, float *b, float *c,
    int N)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < N)
        c[index] = a[index] + b[index];
}
```

PTX (Assembly):

```
setp.lt.s32 %p, %r5, %rd4; //r5 = index, rd4 = N
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6]; //r6 = &a[index]
ld.global.f32 %f2, [%r7]; //r7 = &b[index]
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3; //r8 = &c[index]
L2:
ret;
```

Example: Vector Code

PTX (Assembly):

```
setp.lt.s32 %p, %r5, %rd4; //r5 = index, rd4 = N
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6]; //r6 = &a[index]
ld.global.f32 %f2, [%r7]; //r7 = &b[index]
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3; //r8 = &c[index]
L2:
ret;
```
Example: VectorAdd on GPU

- N=8, 8 Threads, 1 block, warp size = 4
- 1 SM, 4 Cores
- Pipeline:
  - Fetch:
    - One instruction from each warp
    - Round-robin through all warps
  - Execution:
    - In-order execution within warps
    - With proper data forwarding
    - 1 Cycle each stage
- How many warps?

Execution Sequence

Note: Assume idealized execution – issues of dependencies/hazards will be addressed later. The point here is to understand the basic sequencing of instructions in SIMT execution.
Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
```

L1:
```
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
```

L2:
```
ret;
```

Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
```

L1:
```
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
```

L2:
```
ret;
```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra W1
bra W0;
@p bra W0
setp W1
setp W0
setp W0
setp W0
setp W1
setp W1
setp W1
setp W1
setp W0
setp W0
setp W0
setp W0
WB
WB
WB
WB

Warp0
Warp1

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra W1
bra W0;
@p bra W0
setp W1
setp W0
setp W0
setp W0
setp W1
setp W1
setp W1
setp W1
setp W0
setp W0
setp W0
setp W0
WB
WB
WB
WB

Warp0
Warp1

Execution Sequence (cont.)
Execution Sequence (cont.)

```assembly
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
@p bra W1
bra W0 bra W0 bra W0 bra W0
st.global.f32 [%r8], [%f3];
L2:
ret;
```

Execution Sequence (cont.)

```assembly
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
bra W1 bra W1 bra W1 bra W1
st.global.f32 [%r8], [%f3];
L2:
ret;
```
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Warp0  Warp1

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Warp0  Warp1
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

add W0
add W1
ld W1

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Warp0
Warp1

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

add W1
add W0

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Warp0
Warp1

(29)
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Warp0  Warp1

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Warp0  Warp1
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;

Warp0
Warp1

Execution Sequence (cont.)
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
%p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Warp0  Warp1
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;

---

How thread blocks are partitioned

- Partitioning a thread block
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation

- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must
    \texttt{syncthreads}() to get correct results

---

Note: Idealized execution without memory or special function delays

---
Execution Sequence (Cont.)

- SM implements zero-overhead warp scheduling
  - At any time, 1 or 2 of the warps is executed by SM
  - 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

```
Instruction: [TB1 W1] [TB2 W1] [TB3 W1] [TB2 W2] [TB3 W2] [TB1 W1] [TB1 W2] [TB1 W3] [TB3 W2] [TB3 W3]

TB1, W1 stall
TB2, W1 stall
TB3, W2 stall

Time
TB = Thread Block, W = Warp
```

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2012
ECE408/CS483/ECE498al, University of Illinois, Urbana-Champaign

---

Fine Grained Multithreading

- First introduced in the Denelcor HEP (1980’s)
- Can eliminate data bypassing in an instruction stream
  - Maintain separation between successive instructions in a warp
- Simplifies dependency between successive instructions
  - E.g., have only one instruction in the pipeline at a time
- What happens when warp size > #lanes?
  - Why have such a relationship?
Potential Use Cases: Multicycle Dispatch

- **Reliability**: Dynamically reduced the #lanes due to faults
  - E.g., mask off 4 lanes in an 8 lane warp.
  - Compiled code can still execute correctly with appropriate microarchitecture support

- **Power**: Dynamically reduce the number of lanes to reduce power consumption
  - Power gate 4 lanes in an 8 lane warp

- Make warp level compiler/algorithm optimizations portable across machines with different warp sizes
  - Analogy with dynamically scheduled ILP machines
  - No known machines that implement this at the moment
Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in current GPUs
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - More on control divergence later

Barrier Synchronization

- Impact on scalability
  - No constraints on execution order of thread blocks
- Impact on portability
  - Resource allocation on a per thread block basis
  - Launch thread blocks
Synchronization

- All threads must reach the barrier, otherwise indefinite wait
  - Note challenges with conditional statements
  - Note each barrier statement represents a distinct barrier

- Barriers only within a thread block

- No synchronization across thread blocks
  - Can be enforced via atomics

- Thread blocks can execute in any order
  - Loose synchronization can enhance scalability

- CUDA 9.0 introduces more flexible synchronization model
  - Define and synchronize groups of threads

Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
SIMD vs. SIMT

**Flynn Taxonomy**

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

- **Loosely synchronized threads**: Multiple threads can execute independently, with less synchronization overhead.
- **Single Scalar Thread**: Represents the single-threaded execution model.
- **Register File (RF)**: Stores data and instructions for execution.

**Comparison**

<table>
<thead>
<tr>
<th>MIMD</th>
<th>SIMD</th>
<th>SIMT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiple, independent threads and explicit synchronization</strong></td>
<td><strong>Single thread + vector operations</strong></td>
<td><strong>Multiple, synchronous threads</strong></td>
</tr>
<tr>
<td><strong>TLP, DLP, ILP, SPMD</strong></td>
<td><strong>DLP</strong></td>
<td><strong>TLP, DLP, ILP (more recent)</strong></td>
</tr>
<tr>
<td><strong>Easily embedded in sequential code</strong></td>
<td><strong>Easily embedded in sequential code</strong></td>
<td><strong>Independent control flow per thread</strong></td>
</tr>
</tbody>
</table>

- **E.g., pthreads**: Example of loosely synchronized threads.
- **E.g., SSE/AVX**: Example of SIMD execution.
- **E.g., PTX, HSA**: Example of SIMT execution.
Performance Metrics: **Occupancy**

- Performance implications of programming model properties
  - Warps, thread blocks, register usage

- Capture which resources can be dynamically shared, and how to reason about resource demands of a CUDA kernel
  - Enable device-specific online tuning of kernel parameters

**CUDA Occupancy**

- \( \text{Occupancy} = \frac{\text{(#Active Warps)}}{\text{(#MaximumActive Warps)}} \)
  - Measure of how well you are using max capacity

- Limits on the numerator?
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: thread blocks or warps

- Limits on the denominator?
  - Memory bandwidth
  - Scheduler slots

- What is the performance impact of varying kernel resource demands?
Performance Goals

What are the limiting Factors?

Resource Limits on Occupancy

SM – Stream Multiprocessor
SP – Stream Processor
### Programming Model Attributes

- #SMs
- Max Threads/Block
- Max Threads/Dim
- Warp size

What do we need to know about target processor?

How do these map to kernel configuration parameters that we can control?

### Impact of Thread Block Size

- Consider Fermi with 1536 threads/SM
  - With 512 threads/block, can only up to 3 thread blocks executing at an instant
  - With 128 threads/block \(\rightarrow\) 12 thread blocks per SM
  - Consider how many instructions can be in flight?

- Consider limit of 8 thread blocks/SM?
  - Only 1024 active threads at a time
  - Occupancy = 0.666

To maximize utilization, thread block size should balance demand for thread blocks vs. thread slots
Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, there are 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM. Also, this works for CUDA 3.0 and beyond but too large for some early CUDA versions.

Impact of #Registers Per Thread

- Assume 10 registers/thread and a thread block size of 256
- Number of registers per SM = 16K
- A thread block requires 2560 registers for a maximum of 6 thread blocks per SM
  - Uses all 1536 thread slots
- What is the impact of increasing number of registers by 2?
  - Granularity of management is a thread block!
  - Loss of concurrency of 256 threads!
Impact of Shared Memory

- Shared memory is allocated per thread block
  - Can limit the number of thread blocks executing concurrently per SM
- As we change the gridDim and blockDim parameters, how does demand change for shared memory, number of thread slots, or number of thread block slots?

Thread Granularity

- How fine grained should threads be?
  - Coarser grain threads →
    - Increased register pressure, shared memory pressure
    - Lower pressure on thread block slots and thread slots
- Merge threads blocks
- Share row values
  - Reduce memory BW

Merge these two threads
Balance

- Navigate the tradeoffs to maximize core utilization and memory bandwidth utilization for the target device
- **Goal**: Increase occupancy until one or the other is saturated

Performance Tuning

- Auto-tuning to maximize performance for a device
- Query device properties to tune kernel parameters prior to launch
- Tune kernel properties to maximize efficiency of execution
Querying the Device

- Tune kernel properties to maximize efficiency of execution

Get device properties
- #SMs
- Max Threads/Block
- Max Threads/Dim
- Warp size

Device Properties

```c
cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp *prop, int device)
```

Returns in *prop the properties of device *device*. The `cudaDeviceProp` structure is defined as:

```c
c struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int RegsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGdSize[3];
    int clockRate;
    size_t totalConstMem;
    int MajProd;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
}
```
Summary

• Sequence warps through the scalar pipelines
• Overlap execution from multiple warps to hide memory latency (or special functions)
• Out of order execution not shown → need scoreboard for correctness
• Occupancy calculator as a first order estimate of correctness

Questions?