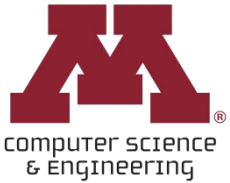
CSCI 5451: Introduction to Parallel Computing

Lecture 19: CUDA Compute Architecture



UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcements (11/10)

- HW3 out today
  - Due date moved (2 weeks from yesterday 11/23)
- HW4
  - Release → 11/17
  - Due date → 11/30
  - Small CUDA problem. Will overlap HW3 for one week.
- Project PDF released today
  - By Nov 16 → Schedule a time with me to meet for 10 minutes to discuss your project idea

Lecture Overview

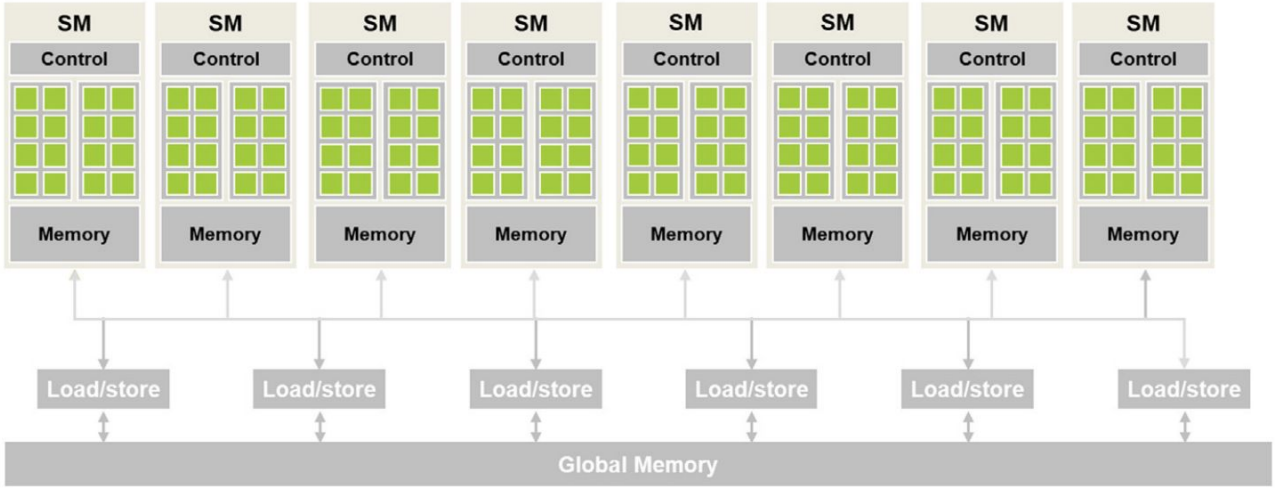
- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ Examples

Lecture Overview

- ❑ **Overview**
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ Examples

GPU Architecture (Full GPU)

- GPUs are organized hierarchically into smaller subunits called Streaming Multiprocessors (SMs)
- Each SM has a number of cores responsible for independent computation
- A100s have 108SMs & 64 Int32 cores/SM (6912 integer operation cores)

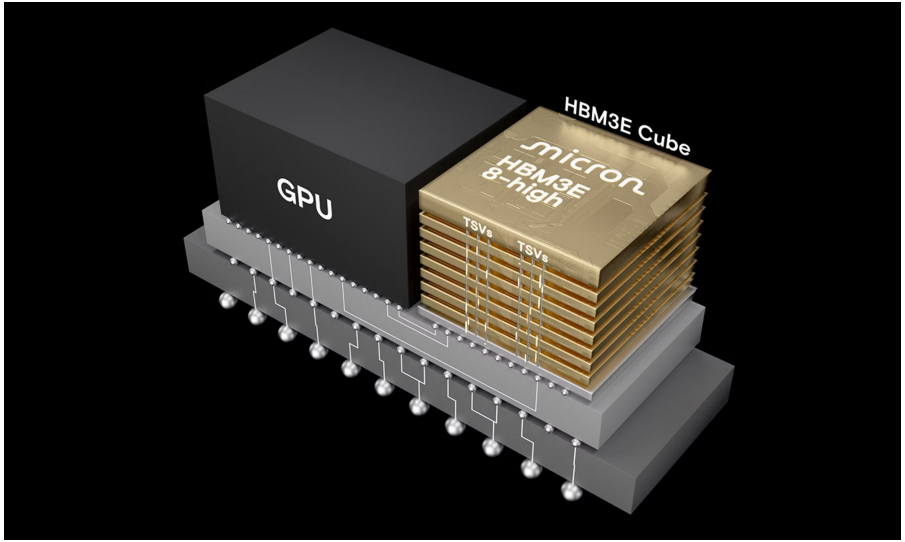


GPU Architecture (SM)

- Each SM has a number of different components (including cores)
- There are different kinds of ‘cores’ depending on the kind of execution desired
  - More modern gpus has support for other kinds of cores (more complex tensor cores, integer types, floating point types)
- We will discuss most of the components at right in today’s lecture

GPU Memory

- Older GPUs relied upon externally connected DRAM - similar to how CPUs are connected to DRAM devices
- Newer devices use High Bandwidth Memory (HBM)
  - HBM is packaged together with the GPU
  - Higher bandwidth (as the name suggests)
  - Less latency, more parallelism



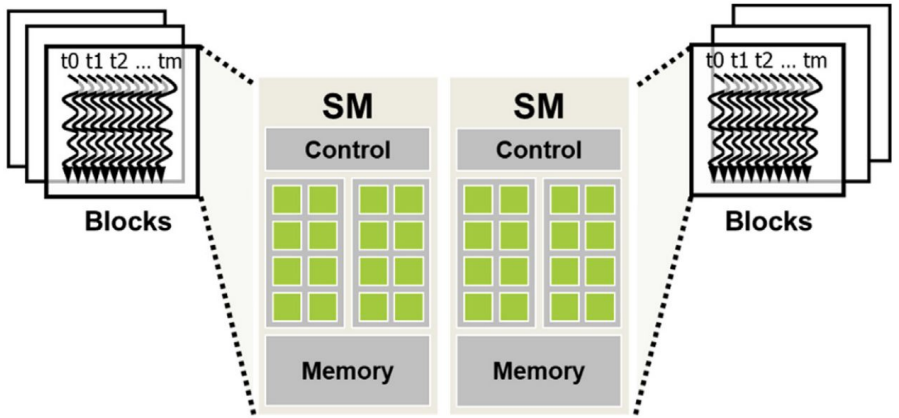
Lecture Overview

- ❑ Overview
- ❑ **Thread Blocks in Detail**
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ Examples

Thread Block Scheduling

- Recall that we launch a grid of threads as separate blocks of threads (in the example at right, each block has 256 threads)
- Each thread block is assigned to one and only one streaming multiprocessor
- Multiple blocks can be assigned to an SM at the same time (the limit is system dependent)
- If the number of blocks is greater than the total number of blocks which can be assigned at one type across all SMs, then some thread blocks will be queued for later execution (i.e. they execute sequentially after other blocks complete)

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```



Thread Synchronization

- To synchronize all threads *within* a block, we use a function called __syncthreads()
- This ***does not*** mean that threads between blocks will synchronize
- Only threads within a block will be synchronized

```cpp
#include <stdio.h>

__global__ void syncExampleNoShared() {
    int tid = threadIdx.x;

    // Each thread prints before synchronization
    printf("Thread %d: before sync\n", tid);

    __syncthreads();  // Wait here until all threads reach this point

    // Each thread prints after synchronization
    printf("Thread %d: after sync\n", tid);
}

int main() {
    syncExampleNoShared<<<1, 4>>>(); // one block, 4 threads
    cudaDeviceSynchronize();
    return 0;
}
```

Thread Synchronization

- All threads within a threadblock must execute the ***same*** *__syncthreads* call
- If they do not, the program will hang indefinitely

```cpp
#include <stdio.h>

__global__ void deadlockExample() {
    int tid = threadIdx.x;

    if (tid % 2 == 0) {
        // even threads go here
        printf("Thread %d: even path before sync\n", tid);
        __syncthreads();  // ⚠ only even threads reach this barrier
        printf("Thread %d: even path after sync\n", tid);
    } else {
        // odd threads go here
        printf("Thread %d: odd path before sync\n", tid);
        __syncthreads();  // ⚠ only odd threads reach this barrier
        printf("Thread %d: odd path after sync\n", tid);
    }
}

int main() {
    deadlockExample<<<1, 4>>>();  // 1 block, 4 threads
    cudaDeviceSynchronize();      // will hang indefinitely
    return 0;
}
```

Synchronizing across blocks

- Once a kernel is invoked, there is no way to synchronize ***between*** threadblocks
- If you need to synchronize ***between all threads***, then you should split your kernel into two separate kernels, then synchronize in between them

```
kernelA<<<numBlocks, blockSize>>>();
cudaDeviceSynchronize(); // wait for kernel A to finish
kernelB<<<numBlocks, blockSize>>>();
```

Synchronizing across blocks

- Once a kernel is invoked, there is no way to synchronize ***between*** threadblocks
- If you need to synchronize ***between all threads***, then you should split your kernel into two separate kernels, then synchronize in between them

```
kernelA<<<numBlocks, blockSize>>>();
cudaDeviceSynchronize(); // wait for kernel A to finish
kernelB<<<numBlocks, blockSize>>>();
```

Why do we have to synchronize after kernel A?

Synchronizing across blocks

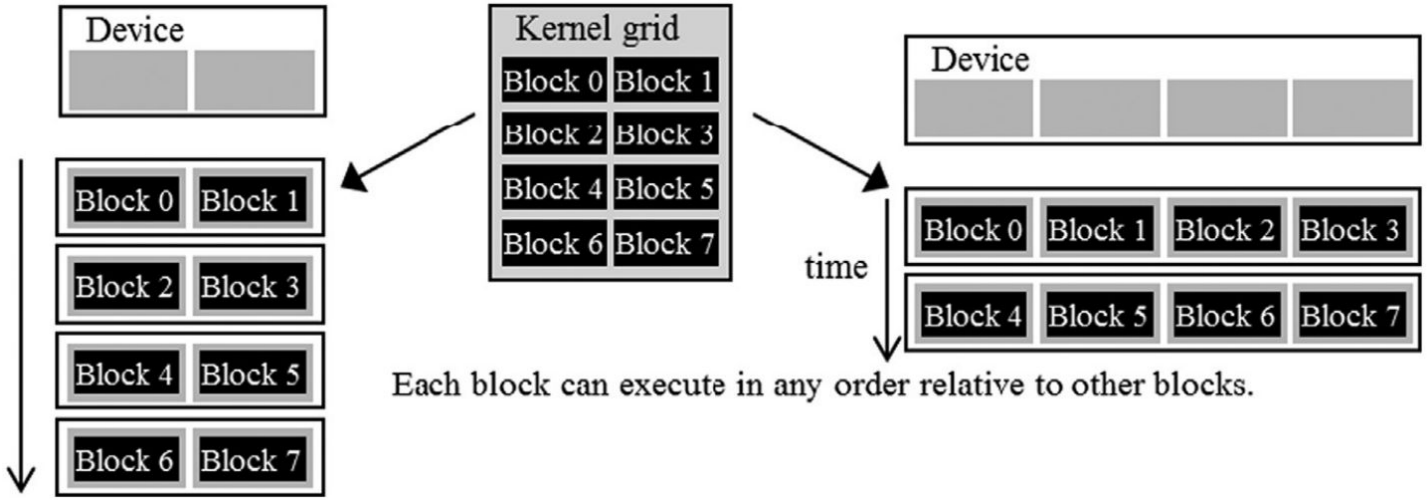
- Once a kernel is invoked, there is no way to synchronize ***between*** threadblocks
- If you need to synchronize ***between all threads***, then you should split your kernel into two separate kernels, then synchronize in between them

```
kernelA<<<numBlocks, blockSize>>>();
cudaDeviceSynchronize(); // wait for kernel A to finish
kernelB<<<numBlocks, blockSize>>>();
```

Why do we have to synchronize after kernel A?
Kernel invocations are non-blocking

Why Use Threadblocks?

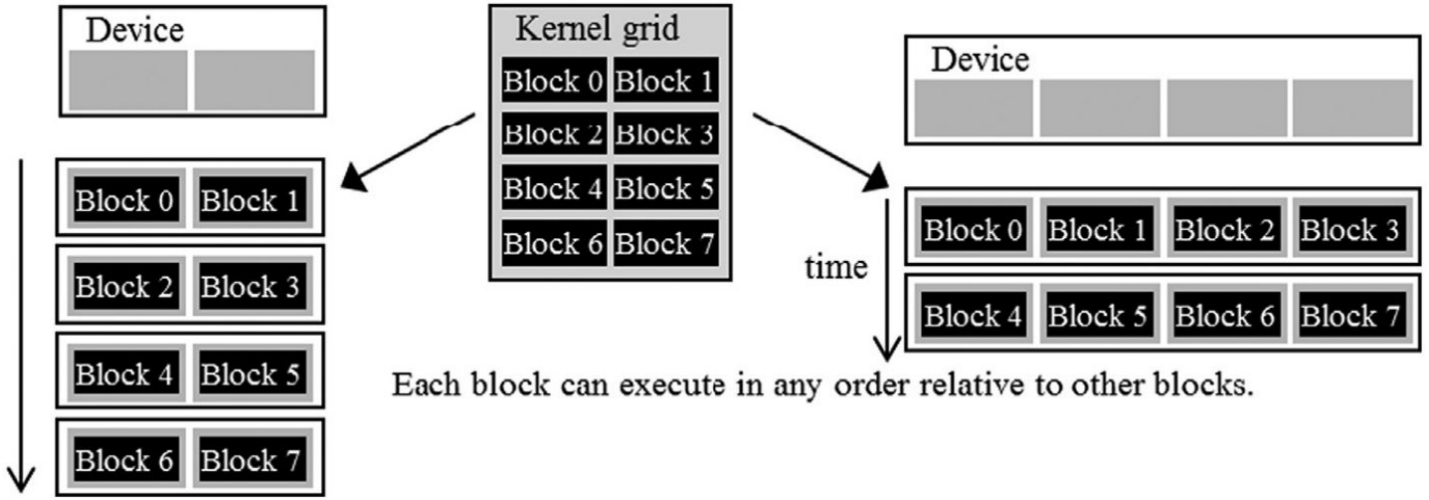
Using threadblocks allows for scalable programs which work regardless of the number of cores on any given cuda device → If we change the GPU, we can still use our original program



Why Use Threadblocks?

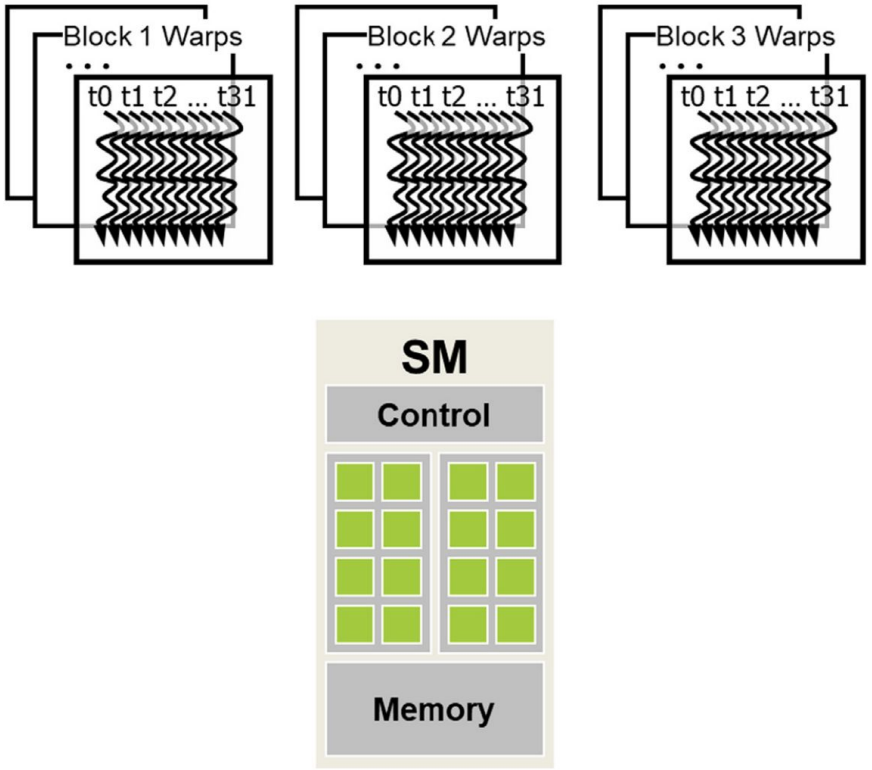
Using threadblocks allows for scalable programs which work regardless of the number of cores on any given cuda device → If we change the GPU, we can still use our original program

However - this means that we do not know the exact order of execution across threads/blocks → We need to design our kernels with this in mind.



Warps

- Threadblocks do not all execute at ***exactly the same*** time
- Instead they execute in ***warps***
- Every warp has exactly 32 threads in it
- Every threadblock is split into groups of 32 threads
- When a block is not divisible by 32, then the last warp in the block is *padded*, with some number of inactive threads



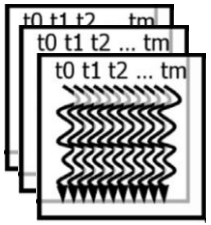
Lecture Overview

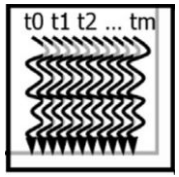
- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ **Hardware, Software & their intersection**
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ Examples

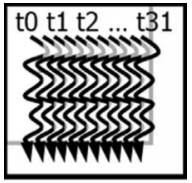
Software Recap

CUDA software exposes us to each of the following in a hierarchical fashion

- Grid of threads
- Threadblocks
- Warps
- Individual Threads



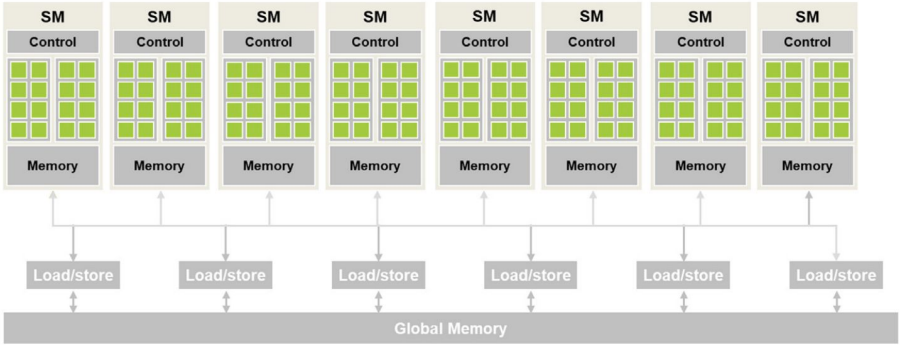




Hardware In-depth

GPU execution hardware is organized in the following hierarchy (starting from the bottom)

- Processing Block (Subpartition)
- Streaming Multiprocessor
- GPU-Wide



Hardware In-depth

GPU execution hardware is organized in the following hierarchy (starting from the bottom

- **Processing Block (Subpartition)**
- Streaming Multiprocessor
- GPU-Wide

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - Load/store units
  - Warp Scheduler
  - Dispatch Unit
  - Cores (int, fp, tensor cores, sfu)
  - Registers

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - **Load/store units**
  - Warp Scheduler
  - Dispatch Unit
  - Cores (int, fp, tensor cores, sfu)
  - Registers

Handling loading & storing operations to either send or receive from memory

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - Load/store units
  - **Warp Scheduler**
  - Dispatch Unit
  - Cores (int, fp, tensor cores, sfu)
  - Registers

Determines which warps should be running at a given time

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - Load/store units
  - Warp Scheduler
  - **Dispatch Unit**
  - Cores (int, fp, tensor cores, sfu)
  - Registers

Sends a decoded instruction to the appropriate core

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - Load/store units
  - Warp Scheduler
  - Dispatch Unit
  - **Cores (int, fp, tensor cores, sfu)**
  - Registers

The actual execution hardware. Performs computations in parallel. The type of core dictates what will execute.

Processing Block

- Each SM is composed of 4 processing blocks
- A Processing Block is also called a **Subpartition**
- A Processing Block consist of:
  - Load/store units
  - Warp Scheduler
  - Dispatch Unit
  - Cores (int, fp, tensor cores, sfu)
  - **Registers**

Holds all register information
(note that these are quite large)

Hardware In-depth

GPU execution hardware is organized in the following hierarchy (starting from the bottom

- Processing Block (Subpartition)
- **Streaming Multiprocessor**
- GPU-Wide

SM

- An SM holds each of the processing blocks, plus some additional logic
- The additional components for the SM are
  - L1 cache
  - Barrier unit
  - Warp State table

SM

- An SM holds each of the processing blocks, plus some additional logic
- The additional components for the SM are
  - **L1 cache**
  - Barrier unit
  - Warp State table

Shared cache across all threads in a given block

SM

- An SM holds each of the processing blocks, plus some additional logic
- The additional components for the SM are
  - L1 cache
  - **Barrier unit**
  - Warp State table

Keeps track of threadblock barriers across all threads

SM

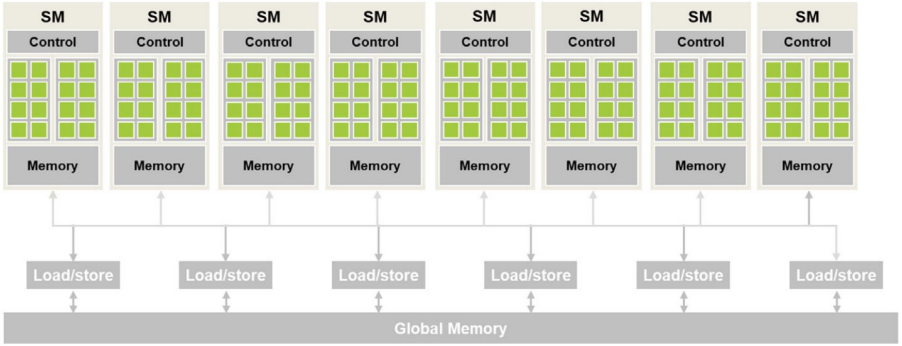
- An SM holds each of the processing blocks, plus some additional logic
- The additional components for the SM are
  - L1 cache
  - Barrier unit
  - **Warp State table**

Keeps track of the current state of each warp in the threadblock

Hardware In-depth

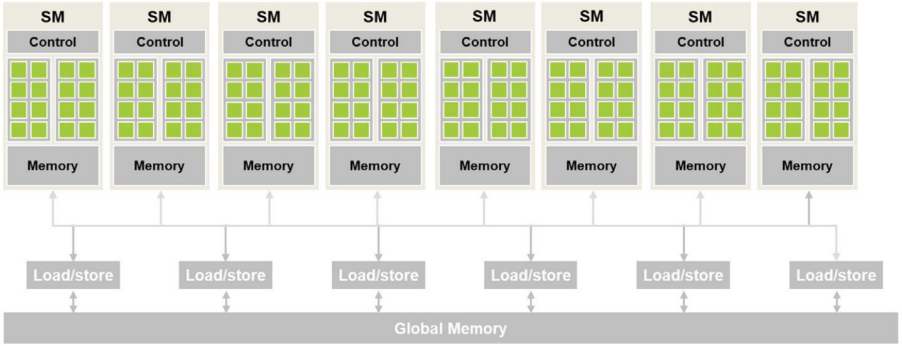
GPU execution hardware is organized in the following hierarchy (starting from the bottom

- Processing Block (Subpartition)
- Streaming Multiprocessor
- **GPU-Wide**



Full GPU

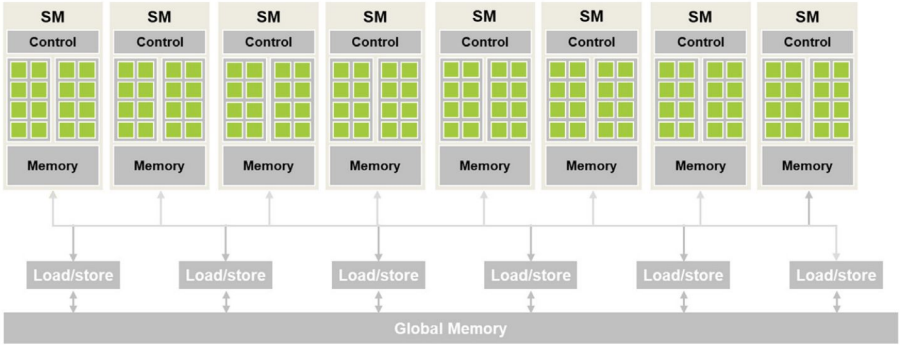
- Each GPU contains many SMs
- Each GPU additionally has
  - L2 cache
  - HBM/Memory controllers/interconnect fabric
  - GTE (threadblock provisioning)



Full GPU

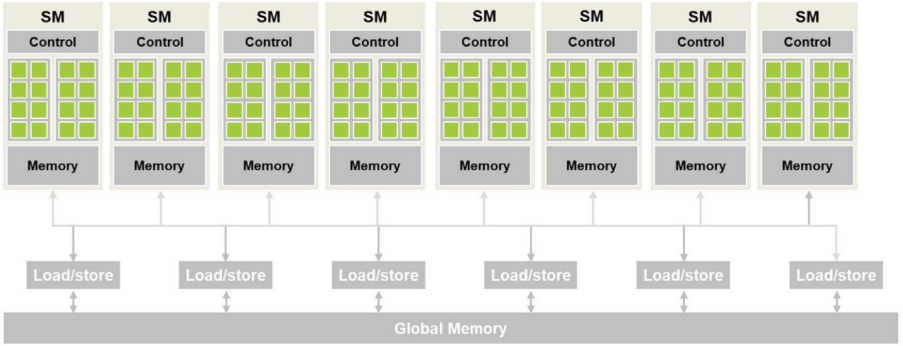
- Each GPU contains many SMs
- Each GPU additionally has
  - **L2 cache**
  - HBM/Memory controllers/interconnect fabric
  - GTE (threadblock provisioning)

Cache shared across all SMs



Full GPU

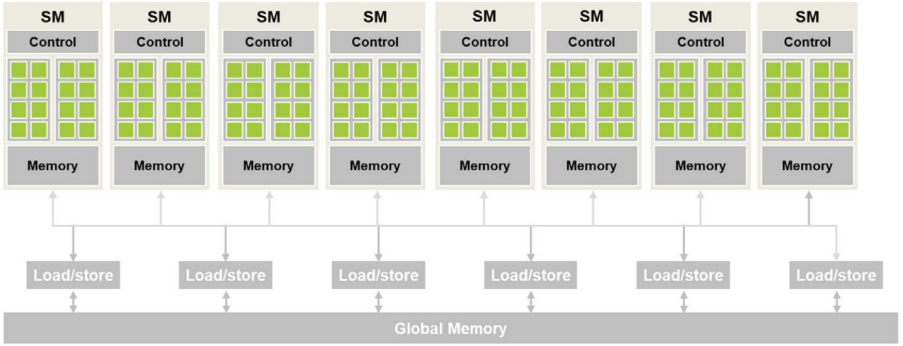
- Each GPU contains many SMs
- Each GPU additionally has
  - L2 cache
  - **HBM/Memory controllers/interconnect fabric**
  - GTE (threadblock provisioning)



Memory + logic to connect and transfer memory to the SMs

Full GPU

- Each GPU contains many SMs
- Each GPU additionally has
  - L2 cache
  - HBM/Memory controllers/interconnect fabric
  - **GTE (threadblock provisioning)**

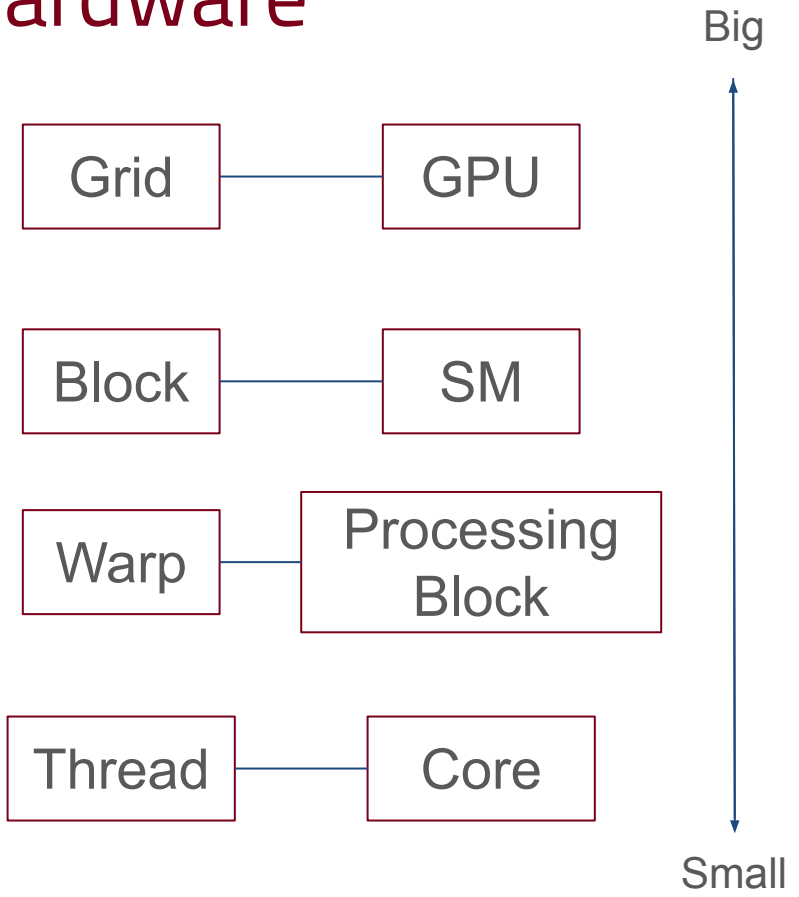


The GigaThread Engine (GTE) schedules threadblocks & distributes them to each SM

Connecting the Software & Hardware

We can connect the software & hardware hierarchies as follows

- Objects in CPU are sent to the GPU & a grid of threads is prepared at the full GPU
- The grid is split into blocks. Each block is sent to an SM. Multiple blocks may be provisioned on each SM at a given time
- At each SM, the blocks are split into warps. Each warp is sent to one processing block
- Computation occurs at each individual core



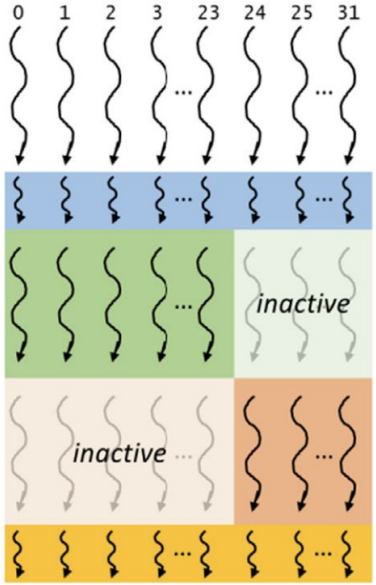
Lecture Overview

- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ **Warp Divergence**
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ Examples

Warp Divergence

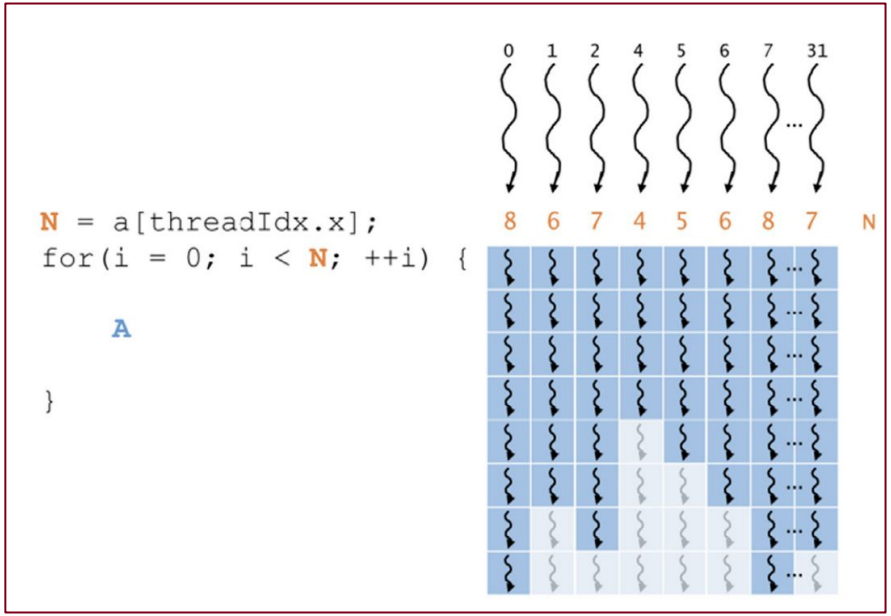
- At the warp level, things operate most closely to the Single Instruction Multiple Data (SIMD) paradigm
- When threads follow different execution paths, this can cause a good deal of overhead for inactive regions
- In other words, both paths may be taken sequentially

```
if(threadIdx.x < 24) {
    A
} else {
    B
}
C
```



Warp Divergence

- Modern GPUs (V100+) introduce the ability to interleave the execution to reduce some of the overhead
- However, this does not eliminate overhead - it just reduces it



Lecture Overview

- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ **Oversubscription & Occupancy**
- ❑ Device Information
- ❑ Examples

Warp Divergence

- Notably, warp divergence can occur when we are performing the checking the edge conditions for threads in the vector addition + image examples from the previous lecture
- Examples
  - Vector Addition
  - Grayscale Convergence example

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

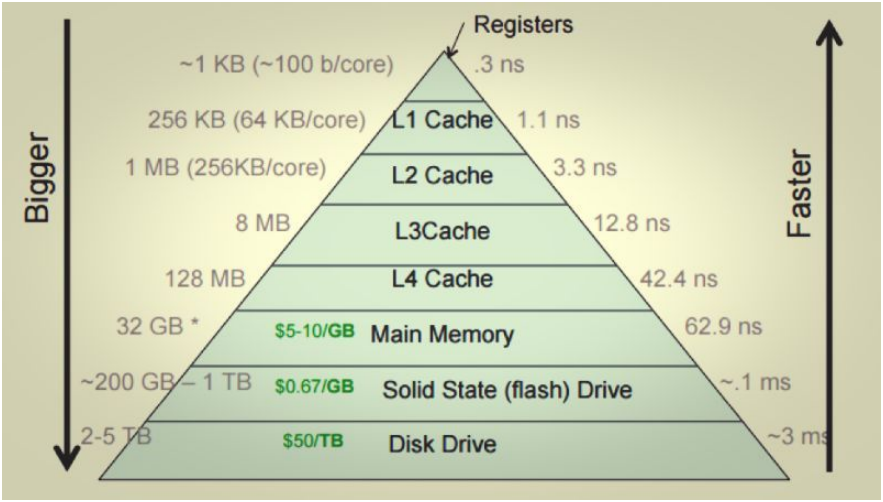
```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
dim3 dimBlock(16, 16, 1);
colorToGrayscaleConversion<<<dimGrid,dimBlock>>>
                                (Pin_d, Pout_d, m, n);
```

Oversubscription

- As discussed blocks can have at most 1024 threads
- However, most SMs only have 64 cores in total (16 per processing block)
- This is called oversubscription
- Why is oversubscription useful?
  -

Oversubscription

- As discussed blocks can have at most 1024 threads
- However, most SMs only have 64 cores in total (16 per processing block)
- This is called oversubscription
- Why is oversubscription useful?
  - For ***Latency Hiding*** (also called ***Latency Tolerance***)
  - This allows threads to always be executing, even if they are waiting for long-latency operations



Oversubscription

- Latency can occur in...
  - Memory Operations
  - Pipelined core operations (particularly floating point)
  - Branching instructions
- Warp scheduler tracks which threads are waiting on latency ops → schedules those that are ready
- This is how many blocks/warps can be assigned to the same SM, even when there are many more threads assigned than available cores
- Example (A100)

Oversubscription

- Latency can occur in...
  - Memory Operations
  - Pipelined core operations (particularly floating point)
  - Branching instructions
- Warp scheduler tracks which threads are waiting on latency ops → schedules those that are ready
- This is how many blocks/warps can be assigned to the same SM, even when there are many more threads assigned than available cores
- Example (A100)

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

Occupancy

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

```
Kernel<<<2, 1024>>>()
Kernel<<<4, 512>>>()
Kernel<<<8, 256>>>()
Kernel<<<16, 128>>>()
Kernel<<<32, 64>>>()
```

Occupancy

- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

```
Kernel<<<2, 1024>>>()
Kernel<<<4, 512>>>()
Kernel<<<8, 256>>>()
Kernel<<<16, 128>>>()
Kernel<<<32, 64>>>()
```

Full occupancy (100%)

Occupancy

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

```
KernelA<<<64, 32>>>()
KernelB<<<128, 16>>>()
```

Occupancy

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

```
KernelA<<<64, 32>>>()
KernelB<<<128, 16>>>()
```

KernelA → (50% occupancy)
KernelB → (25% occupancy)

Occupancy

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

```
KernelA<<<4, 768>>>()
KernelB<<<12, 256>>>()
```

Occupancy

A100 SM Limits:
1024 threads per block
32 blocks per SM
64 warps per SM (2048 threads)

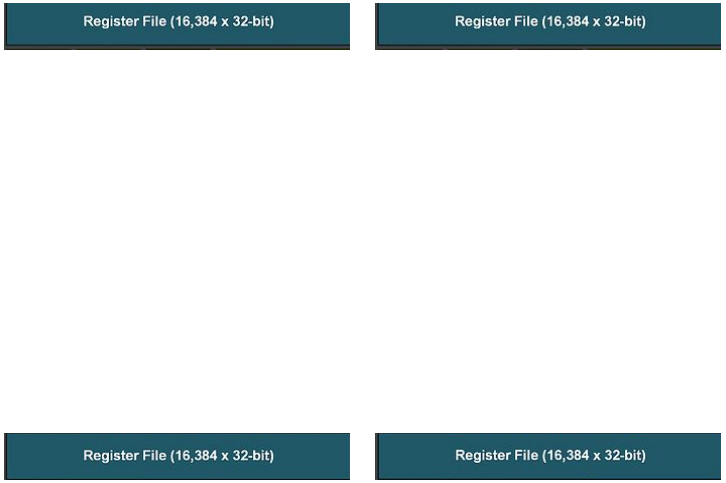
- The ratio of the number of warps assigned to a given SM versus the maximum number of warps supported is called the ***occupancy***
- Proper oversubscription maximizes occupancy
- Examples

```
KernelA<<<4, 768>>>()
KernelB<<<12, 256>>>()
```

KernelA → (75% occupancy)
KernelB → (100% occupancy)

Register Occupancy

- Other resource limitations (beyond block size, warp limitations) can impact resource utilization
- Registers add to this
- In each SM on the A100, there are 65,536 registers
- Given that each SM can run 2048 threads, that means we have 65,536/2048 = 32 registers per thread



Register Occupancy (Example)

- 32 registers per thread
- If we have a kernel which uses 64 registers per thread, then the occupancy will drop down to 50% → We will not be utilizing as much of the resources as we are able

Lecture Overview

- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ **Device Information**
- ❑ Examples

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - warpSize

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- **cudaGetDeviceCount**
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - warpSize

Number of GPUs

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    int deviceCount = 0;
    cudaError_t error = cudaGetDeviceCount(&deviceCount);

    if (error != cudaSuccess) {
        printf("Error: %s\n", cudaGetErrorString(error));
        return 1;
    }

    printf("Number of CUDA devices: %d\n", deviceCount);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - **maxThreadsPerBlock**
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - warpSize

Threads allowed in each block (typically 1024)

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d\n", prop.maxThreadsPerBlock);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - **multiProcessorCount**
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - warpSize

Total number of Streaming Multiprocessors (SMs)

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d\n", prop.multiProcessorCount);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - **clockRate**
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - warpSize

Clock rate of the GPU

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d\n", prop.clockRate);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - **maxThreadsDim**
  - maxGridSize
  - regsPerBlock
  - warpSize

Maximum allowable threads along each dimension of a block

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d %d %d\n",
           prop.maxThreadsDim[0],
           prop.maxThreadsDim[1],
           prop.maxThreadsDim[2]);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - **maxGridSize**
  - regsPerBlock
  - warpSize

Maximum allowable blocks along each dimension of a grid

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d %d %d\n",
           prop.maxGridSize[0],
           prop.maxGridSize[1],
           prop.maxGridSize[2]);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - **regsPerBlock**
  - warpSize

Number of available registers in each SM

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d\n", prop.regsPerBlock);
    return 0;
}
```

Getting Device Information

The following functions allow the user to query the GPU to get information regarding its properties

- cudaGetDeviceCount
- DevProperties
  - maxThreadsPerBlock
  - multiProcessorCount
  - clockRate
  - maxThreadsDim
  - maxGridSize
  - regsPerBlock
  - **warpSize**

The number of threads in each warp. On every CUDA platform up to now (Nov 2025), this has been 32

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("%d\n", prop.warpSize);
    return 0;
}
```

Lecture Overview

- ❑ Overview
- ❑ Thread Blocks in Detail
- ❑ Hardware, Software & their intersection
- ❑ Warp Divergence
- ❑ Oversubscription & Occupancy
- ❑ Device Information
- ❑ **Examples**

Example Problems

What is the number of warps per block?
What is the number of warps in the grid?
For the statement on line 04:

i. How many warps in the grid are active?
ii. How many warps in the grid are divergent?
iii. What is the SIMD efficiency (in %) of warp 0 of block 0?
iv. What is the SIMD efficiency (in %) of warp 1 of block 0?
v. What is the SIMD efficiency (in %) of warp 3 of block 0?

For the statement on line 07:

i. How many warps in the grid are active?
ii. How many warps in the grid are divergent?
iii. What is the SIMD efficiency (in %) of warp 0 of block 0?

For the loop on line 09:

i. How many iterations have no divergence?
ii. How many iterations have divergence?

```
01    __global__ void foo_kernel(int* a, int* b) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(threadIdx.x < 40 || threadIdx.x >= 104) {
04            b[i] = a[i] + 1;
05        }
06        if(i%2 == 0) {
07            a[i] = b[i]*2;
08        }
09        for(unsigned int j = 0; j < 5 - (i%3); ++j) {
10            b[i] += j;
11        }
12    }
13    void foo(int* a_d, int* b_d) {
14        unsigned int N = 1024;
15        foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
16    }
```

Example Problems

Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9; they spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time is spent waiting for the barrier?

Example Problems

Consider a GPU with the following hardware limits: 2048 threads per SM, 32 blocks per SM, and 64K (65,536) registers per SM. For each of the following kernel characteristics, specify whether the kernel can achieve full occupancy. If not, specify the limiting factor.

a. The kernel uses 128 threads per block and 30 registers per thread.
b. The kernel uses 32 threads per block and 29 registers per thread.
c. The kernel uses 256 threads per block and 34 registers per thread.

If a CUDA device's SM can take up to 1536 threads and up to 4 thread blocks, which of the following block configurations would result in the most number of threads in the SM?

a. 128 threads per block
b. 256 threads per block
c. 512 threads per block
d. 1024 threads per block

Assume a device that allows up to 64 blocks per SM and 2048 threads per SM. Indicate which of the following assignments per SM are possible. In the cases in which it is possible, indicate the occupancy level.

a. 8 blocks with 128 threads each
b. 16 blocks with 64 threads each
c. 32 blocks with 32 threads each
d. 64 blocks with 32 threads each
e. 32 blocks with 64 threads each