Lecture 2: CUDA Programming
Let’s look at a sequential program in C first:

```c
void add_vector(float *A, float *B, float *C, int N){
    for (index = 0; index < N; index++)
}

void main() { ...
    add_vector(A, B, C, N);
    ...}
```

Add vector A and vector B to vector C.
Now let’s look at the parallel implementation in CUDA:

Add vector A and vector B to vector C.

```c
__global__ void add_vector (float *A, float *B, float *C, int N) {
    if (tid < N)
}
```

```c
void main ( ) { ...
    add_vector <<<dimGrid, dimBlock>>> (A, B, C, N);
    ...}
```

/ilab/users/zz124/cs515_2017/samples/0_Simple/vectorAdd
CPU Program

Add vector A and vector B to vector C.

```c
void add_vector (float *A, float *B, float *C, int N){
    ...
    for ( index = 0; index < N; index++ )
}

void main ( ) { ...
    add_vector (A, B, C, N);
    ...
}
```

GPU Program

Add vector A and vector B to vector C.

```c
__global__ void add_vector (float *A, float *B, float *C, int N){
    ...
    if ( tid < N )
}

void main ( ) { ...
    add_vector <<<dimGrid, dimBlock>>>, (A, B, C, N);
    ...
}
Review: Programming in CUDA

• Now the parallel implementation in CUDA:

```c
_global_ void add_vector (float *A, float *B, float *C, int N)
{
    tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < N)
}

void main () {
    … //allocate memory on GPU and initialization
    add_vector <<<dimGrid, dimBlock>>>(A, B, C, N);
    … // free memory on GPU
    …}
```

Sample code: /ilab/users/zz124/cs515_2017/samples/0_Simple/vectorAdd
Review: Programming in CUDA

- CPU Computation and GPU Computation:

  ![Diagram of CPU Host and GPU Device with Kernel Invocation and CPU Work](image-url)
Review: Thread View

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency *shared memory* (scratch-pad memory)
  - Two threads from two different blocks cannot cooperate by shared memory
- A thread block is organized as a set of warps
  - A warp is a **SIMD** thread group
  - A warp has 32 threads

Kernels can also run simultaneously if specified as asynchronous streams.
Review: Single Instruction Multiple Data (SIMD)

- \( C[\text{index}] = A[\text{index}] + B[\text{index}] \);
  
  \begin{align*}
  \text{thread 0:} & \quad C[0] = A[0] + B[0]; \\
  \text{....} & \\
  \end{align*}

- **Flynn’s taxonomy for modern processors**
  
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (MIMD)
  - Multiple Instruction Single Data (MISD)
  - Multiple Instruction Multiple Data (MIMD)

**GPU — SIMT**

(Single Instruction Multiple Thread)

*Simultaneous execution of many SIMD thread groups*
Review: GPU Architecture

- **Nvidia GeForce 8800 GTX**
  - 128 streaming processors (SP) each at 575 MHz
  - Each streaming multiprocessor (SM) has 16 KB shared memory.
Review: Thread Blocks on a SM

- Threads are assigned to SMs in BLOCK granularity
  - The number of blocks per SM limited by resource constraints
  - One SM in C2075 can host up to 1536 threads:
    - 256 (threads/block) * 6 blocks or
    - 128 (threads/block) * 12 blocks, and etc.
- Threads run concurrently
- SM assigns/maintains thread id #s
- SM manages/schedules thread execution
Review: Thread Life Cycle in Hardware

- Thread blocks are distributed to all the SMs
  - Typically more than 1 block per SM
- One Streaming Multi-processor schedules and executes warps that are ready
- As a thread block completes, resources are freed
  - More thread blocks will be distributed to SMs
Review: Thread Scheduling and Execution

- Warps are scheduling units within a SM

  - If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there?
  - 24

- At any point in time, only a limited number of those warps will be selected for execution on one SM
Review: SM Warp Scheduling

- SM hardware implements “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 (round-robin/age)
  - All threads in a warp execute the same instruction when selected

- Typically 4 clock cycles needed to dispatch the same instruction for all threads in a warp
Review: SM Warp Scheduling

• SM hardware implements “zero-overhead” Warp scheduling

**Question:**

Assume one global memory access is needed for every 4 instructions, at one time only one instruction can be issued, and it takes 4 cycles to dispatch an instruction for one warp.

*What is the minimum number of warps are needed to tolerate the 320-cycle global memory latency?*
Review: SM Warp Scheduling

Question:

Assume one global memory access is needed for every 4 instructions, at one time only one instruction can be issued, and it takes 4 cycles to dispatch an instruction for one warp.

What is the minimum number of warps are needed to tolerate the 320-cycle global memory latency?
CUDA Memory Model

• On-chip memory
  - Registers
  - Shared Memory (SMEM)
  - Instruction Cache, Constant Cache, Texture/L1 Cache
  - Small latencies, i.e., ~ 20 cycles or less
  - L2 (last level cache), ~ 400-1000 cycles latency

• Off-chip memory
  - Local Memory: register spills, activation record, thread-private variable
  - Constant Memory: automatically maps to a constance cache by hardware
  - Texture Memory: automatically maps to texture cache by hardware
  - Global memory: main device memory, read/write, visible to all threads in the same GPU program.

Optional Reading: Wong et al. “Demystifying GPU microarchitecture through microbenchmarking”. ISPASS’10
CUDA Software Cache

• **Software Cache (SMEM)**
  - A thread block gets equal-size shared memory partition
  - Requires explicit shared memory allocation
  - Requires explicit data movement management

• **Software Cache v.s. Hardware Cache**
  - Replacement: implicit by hardware V.S. explicit by software
  - Cache eviction policy: least recent use (LRU) policy V.S. programmer specified
  - Addressing model: complex hash function V.S. programmer specified
  - Partition model: no fixed-size partition V.S. fixed-size partition for different thread blocks
  - Interaction with concurrency: no impact on concurrency V.S. affects concurrency level

*Optional Reading: Silberstein et al. "Efficient computation of sum-products on GPUs through software-managed cache.,” ICS’08*
CUDA Synchronization Primitive

• Within a thread block
  - Barrier: `__syncthreads()`
  - All computation by threads in the thread block before the barrier complete before any computation by threads after the barrier begins
  - Barriers are a conservative way to express dependencies
  - Barriers divide computation into phases
  - In conditional code, the condition must be uniform across the block

• Across thread blocks
  - Implicit synchronization at the end of a GPU kernel
  - A barrier in the middle of a GPU kernel can be implemented using atomic compare-and-swap (CAS) and memory fence operations

Optional Reading: Xiao et al. “Inter-block GPU communication via fast barrier synchronization”, IPDPS'10
Case Study 1: Matrix Transpose
Case Study 1: Matrix Transpose
Case Study 1: Matrix Transpose

• Simple Implementation

```c
__global__ void transpose_naive(float *outputData,
                                 float *inputData,
                                 int width,
                                 int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        outputData[index_out] = inputData[index_in];
    }
}
```

One thread is assigned to one a cell in the input matrix
Performance Tuning — Global Memory Coalescing

• Data is fetched as a contiguous memory chunk (a cache block/line)
  - A cache block/line is typically 128 byte
  - The purpose is to utilize the wide DRAM burst for maximum memory level parallelism (MLP)

• A thread warp is ready when all its threads’ operands are ready
  - Coalescing data operands for a thread warp into as few contiguous memory chunks as possible
  - Assume we have a memory access pattern for a thread \( \ldots = A[P[tid]] \):

```
P[] = { 0, 1, 2, 3, 4, 5, 6, 7 }
```

- Coalesced Accesses

```
A[]: [ 0, 1, 2, 3, 4, 5, 6, 7 ]
```
a cache block.

```
P[] = { 0, 5, 1, 7, 4, 3, 6, 2 }
```

- Non-Coalesced Accesses

```
A[]: [ 0, 1, 2, 3, 4, 5, 6, 7 ]
```
a cache block.
Any non-coalesced memory access?

```c
__global__ void transpose_naive(float *outputData,
                                float *inputData,
                                int width,
                                int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        outputData[ index_out ] = inputData[ index_in ];
    }
}
```

One thread is assigned to one a cell in the input matrix.
Case Study 1: Matrix Transpose

Any non-coalesced memory access?

```c
__global__ void transpose_naive(float *outputData,
                               float *inputData,
                               int width,
                               int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        outputData[index_out] = inputData[index_in];
    }
}
```

One thread is assigned to one a cell in the input matrix.
Case Study 1: Matrix Transpose

Any non-coalesced memory access?

```c
__global__ void transpose_naive(float *outputData,
                                float *inputData,
                                int width,
                                int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        outputData[index_out] = inputData[index_in];
    }
}
```

1 Million Elements

Naive Implementation

**Performance:** Throughput = 8.0226 GB/s, Time = 0.97381 ms
Case Study 1: Matrix Transpose

• An Improved Implementation

```c
__global__ void transpose(float *outputData, float *inputData, int width, int height) {
  __shared__ float block[BLOCK_DIM][BLOCK_DIM + 1];

  unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
  unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
  if (xIndex < width) && (yIndex < height) {
    unsigned int index_in = yIndex * width + xIndex;
    block[threadIdx.y][threadIdx.x] = inputData[index_in];
  }
  __syncthreads();

  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
  if (xIndex < height) && (yIndex < width) {
    unsigned int index_out = yIndex * height + xIndex;
    outputData[index_out] = block[threadIdx.x][threadIdx.y];
  }
}
```

Use shared memory to improve global memory coalescing.
Case Study 1: Matrix Transpose

• An Improved Implementation

```
__global__ void transpose(float *outputData, float *inputData, int width, int height) {
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if( (xIndex < width) && (yIndex < height) ) {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = inputData[index_in];
    }

    __syncthreads();

    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if( (xIndex < height) && (yIndex < width) ) {
        unsigned int index_out = yIndex * height + xIndex;
        outputData[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Use shared memory to improve global memory coalescing

both read/write accesses are coalesced
Case Study 1: Matrix Transpose

• An Improved Implementation

```c
__global__ void transpose(float *outputData, float *inputData, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;

    if( (xIndex < width) && (yIndex < height) )
    {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = inputData[index_in];
    }

    __syncthreads();

    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

    if( (xIndex < height) && (yIndex < width) )
    {
        unsigned int index_out = yIndex * height + xIndex;
        outputData[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

1 Million Elements

**Naive Implementation**

**Performance:** Throughput = 8.0226 GB/s, Time = 0.97381 ms

**Improved implementation**

**Performance:** Throughput = 11.5080 GB/s, Time = 0.67887 ms
Case Study 1: Matrix Transpose

• An Improved Implementation

```c
__global__ void transpose(float *outputData, float *inputData, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    unsigned int xIndex = blockIdx.x* BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;

    if( ( xIndex < width) && (yIndex < height) )
    {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = inputData[index_in];
    }

    __syncthreads();

    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

    if( (xIndex < height) && (yIndex < width))
    {
        unsigned int index_out = yIndex * height + xIndex;
        outputData[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Use shared memory to improve global memory coalescing

both read/write accesses are coalesced
GPU Performance Hazards
Performance Hazard I — Control Divergence

• Thread Divergence Caused by Control Flows

```c
A[ ]:
  0 0 6 0 4 2 6 0

if (A[tid]) {
  do some work;
} else {
  do nothing;
}
```

May degrade performance by up to warp_size times.
(warp size = 32 in modern GPUs)

Optional Reading: Zhang et al. "On-the-Fly Elimination of Dynamic Irregularities for GPU Computing", ASPLOS'11
Performance Hazard II — Global Memory Coalescing

• **Data is fetched as a contiguous memory chunk (a cache block/line)**
  
  - A cache block/line is typically 128 byte
  - The purpose is to utilize the wide DRAM burst for maximum memory level parallelism (MLP)

• **A thread warp is ready when all its threads’ operands are ready**

  - Coalescing data operands for a thread warp into as few contiguous memory chunks as possible
  - Assume we have a memory access pattern for a thread $... = A[P[tid]]$:

```
P[] = { 0, 1, 2, 3, 4, 5, 6, 7}
```

Coalesced Accesses

```
tid: 0 1 2 3 4 5 6 7
```

```
A[]: [ ][ ][ ][ ][ ][ ][ ][ ]
```

A cache block.

Non-Coalesced Accesses

```
P[] = { 0, 5, 1, 7, 4, 3, 6, 2}
```

```
tid: 0 1 2 3 4 5 6 7
```

```
A[]: [ ][ ][ ][ ][ ][ ][ ][ ]
```

A cache block.
Performance Hazard III: Shared Memory Bank Conflicts

- **Shared Memory**
  - Typically 16 or 32 banks
  - Successive 32-bit words are assigned to successive banks
  - A fixed stride access may cause bank conflicts
  - Maximizing bank level parallelism is important

- **Reduce Bank Conflicts**
  - Padding, might waste some shared memory space.
  - Data layout transformation, i.e., array of struct to struct of array.
  - Thread-data remapping, i.e., the tasks that are mapped to different banks are executed at the same time.
Performance Hazard III: Shared Memory Bank Conflicts

- No Bank Conflicts

[Diagram showing bank connections for threads without conflicts]

- No Bank Conflicts

[Diagram showing bank connections for threads without conflicts]
Performance Hazard III: Shared Memory Bank Conflicts

- 2-way Bank Conflicts
- 8-way Bank Conflicts

Diagram showing 2-way and 8-way bank conflicts with threads accessing shared memory banks.
Case Study 2 — Parallel Reduction

• What is a reduction operation?
  - A binary operation and a roll-up operation
  - Let $A = [a_0, a_1, a_2, a_3, \ldots, a(n-1)]$, let $\oplus$ be an associative binary operator with identity element $I$

  $\text{reduce}(\oplus, A) = [a_0 \oplus a_1 \oplus a_2 \oplus a_3 \ldots a_N]$

  ```
  int sum = 0;
  for (int i = 0; i < N; i++)
  sum = sum + A[i];
  ```

Sequential  Parallel

![Diagram of parallel reduction](image)
Case Study 2 — Parallel Reduction

• First Implementation

```c
int tid = threadIdx.x;
for (s = 1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Any GPU performance hazard here?

Thread Divergence
Case Study 2 — Parallel Reduction

First Implementation

\[ \text{tid} = \text{threadIdx.x}; \]

for \( s = 1; s < \text{blockDim.x}; s *= 2 \) {
  if \( \text{tid} \% (2*s) == 0 \) {
    \text{sdata[tid]} += \text{sdata[tid+s]};
  }
  \text{__syncthreads();}
}

Any GPU performance hazard here?

Thread Divergence

Reducing 16 Million Elements

Implementation 1:
\textbf{Performance:} Throughput = 1.5325 GB/s, Time = 0.04379 s

Implementation 2:
\textbf{Performance:} Throughput = 1.9286 GB/s, Time = 0.03480 s

Implementation 3:
\textbf{Performance:} Throughput = 2.6062 GB/s, Time = 0.02575 s
Case Study 2 — Parallel Reduction

• Reduce Thread Divergence

Values (shared memory)

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Stride 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Stride 2</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3</td>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Stride 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 4 8</td>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Stride 8</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

Second Version

First Version

Thread-Task Mapping Changed!
Case Study 2 — Parallel Reduction

- Reduce Thread Divergence

First Version

```c
	tid = threadIdx.x;

	for (s=1; s < blockDim.x; s *= 2) {
		int index = 2 * s * tid;
		if (index < blockDim.x) {
			sdata[index] += sdata[index + s];
		}
		__syncthreads();
	}
```

Second Version

```c
	tid = threadIdx.x;

	for (s=1; s < blockDim.x; s *= 2) {
		int index = 2 * s * tid;
		if (index < blockDim.x) {
			sdata[index] += sdata[index + s];
		}
		__syncthreads();
	}
```

Thread-Task Mapping Changed!
Case Study 2 — Parallel Reduction

- Implementation 2

```c
threadIdx.x;
for (s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * threadIdx.x;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
}
__syncthreads();
```

Any performance hazard?

Potential Shared Memory Bank Conflicts

Reducing 16 Million Elements

Implementation 1:
**Performance:** Throughput = 1.5325 GB/s, Time = 0.04379 s

Implementation 2:
**Performance:** Throughput = 1.9286 GB/s, Time = 0.03480 s
Case Study 2 — Parallel Reduction

- Reduce Shared Memory Bank Conflicts

Let Logically Contiguous Thread Access Contiguous Shared Memory Locations
Case Study 2 — Parallel Reduction

• Third Implementation (utilize implicit intra-warp synchronization)

```c
    tid = threadIdx.x;
    for (s = blockDim.x/2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
    }
    __syncthreads();
```

Reducing 16 Million Elements

Implementation 1:
**Performance:** Throughput = 1.5325 GB/s, Time = 0.04379 s

Implementation 2:
**Performance:** Throughput = 1.9286 GB/s, Time = 0.03480 s

Implementation 3:
**Performance:** Throughput = 2.6062 GB/s, Time = 0.02575 s

Code: /ilab/users/zz124/cs515_2017/samples/6_Advanced/reduction
Reading Assignments

• “Optimization Parallel Reduction in CUDA” by Mark Harris, NVIDIA
• “Parallel Prefix Sum (Scan) with CUDA”, GPU Gem3, Chapter 39, by Mark Harris, Shubhabrata Sengupta, and John Owens.
• “Performance Optimization” by Paulius Micikevicius, NVIDIA
• CUDA C/C++ Basics
• Loop unrolling for GPU Computing
• CUDA Programming Guide, Chapter 1-5.
• CUDA C Best Practices Guide.
Optional Reading Assignments

• Silberstein et al. “Efficient computation of sum-products on GPUs through software-managed cache”. ICS’08.
• Xiao et al. “Inter-block GPU communication via fast barrier synchronization”. IPDPS’10.
• Zhang et al. ”On-the-Fly Elimination of Dynamic Irregularities for GPU Computing”. ASPLOS’11.