Performance Considerations
Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

Parallel Kernel (device)
```
KernelA<<< nBlk, nThr >>>(args);
```

Serial Code (host)

Parallel Kernel (device)
```
KernelB<<< nBlk, nThr >>>(args);
```
Memory Hierarchy in CUDA Programs

Grid (Device)

Block (0, 0)

- Shared memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Block (1, 0)

- Shared memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Global / Texture & Surface memory

Constant memory

Host
Latency Hiding and Occupancy

- **FGMT** can hide **long latency operations** (e.g., memory accesses)
- **Occupancy**: ratio of **active warps** to the maximum number of warps per GPU core
Memory Coalescing (I)

- When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as memory coalescing

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence
Memory Coalescing (II)

- When accessing global memory, we want to make sure that concurrent threads access nearby memory locations.
- Peak bandwidth utilization occurs when all threads in a warp access one cache line (or several consecutive cache lines).

---

Slide credit: Hwu & Kirk
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values

Slide credit: Hwu & Kirk
SIMD Utilization
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have **conditional control flow instructions**
- Threads can execute different control flow paths

Slide credit: Tor Aamodt
Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps

- **Branch divergence** occurs when threads inside warps branch to different execution paths

This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?

Slide credit: Tor Aamodt
Intra-warp divergence

Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
}
else{
    Do_that(threadIdx.x);
}
Increasing SIMD Utilization

- **Divergence-free execution**

```c
Compute(threadIdx.x);
if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
}
else{
    Do_that((threadIdx.x%32)*2+1);
}
```
Reduction Operation
Reduction Operation

- A **reduction** operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

- **Properties of reduction**
  - Associativity
  - Commutativity
  - Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Dean and Ghemawat, “MapReduce: Simplified Data Processing of Large Clusters,” OSDI 2004
Sequential Reduction

- A sequential implementation of reduction only needs a `for` loop to go through the whole input array
  - N elements → N iterations

```
sum = 0; // Initialize with identity value
for(i = 0; i < N; ++i) {
    sum += A[i]; // Accumulate elements of input array A[]
}
```

- Many independent operations
  - A parallel implementation can calculate multiple partial sums, and then reduce them
Tree-Based Reduction

log(N) iterations

Partial results in temporary storage
Tree-Based Reduction on GPU

Block 0

Intra-block synchronization
__syncthreads();

Inter-block synchronization
- Kernel termination and
  - Final reduction on CPU, or
  - Launch new reduction kernel on GPU
- Atomic operations in global memory

Block 1

Partial results in shared memory (or registers)
Vector Reduction: Naïve Mapping (I)

Slide credit: Hwu & Kirk
Program with low SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = 1; stride < blockDim.x; stride *= 2){
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```

How to avoid the warp underutilization?

- **stride = 1**
- **stride = 2**
- **stride = 4**
Divergence-Free Mapping (I)

- All active threads belong to the same warp
Divergence-Free Mapping (II)

Program with **high SIMD utilization**

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 1; stride >>= 1){
    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

Warp utilization is maximized
Warp Shuffle Functions

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
  - Faster than using shared memory and __syncthreads() to share across threads in the same block

- Variants:
  - __shfl_sync()  
    - Direct copy from indexed lane
  - __shfl_up_sync()  
    - Copy from a lane with lower ID relative to caller
  - __shfl_down_sync()  
    - Copy from a lane with higher ID relative to caller
  - __shfl_xor_sync()  
    - Copy from a lane based on bitwise XOR of own lane ID

Slide credit: Izzat El Hajj
Read and Write Access to GPU Shared Memory


FIG. 13
Shuffling Operations within a Warp
Divergence-Free Mapping (III)

- Program with high SIMD utilization

```c
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 1; stride >>= 1){
    __syncthreads();

    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

We can use warp shuffle to avoid shared memory accesses and `__syncthreads()`.
Reduction with Warp Shuffle

__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
    }
    for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
        sum += __shfl_down_sync(0xffffffff, sum, stride);
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}

Slide credit: Izzat El Hajj
CUDA provides atomic instructions on shared memory and global memory.
- They perform read-modify-write operations atomically.

Arithmetic functions
- Add, sub, max, min, exch, inc, dec, CAS

```c
int atomicAdd(int*, int);
```

Bitwise functions
- And, or, xor

Integer, uint, ull, and float
Atomic Operations (II)

Atomic operations serialize the execution if there are atomic conflicts.

No atomic conflict = concurrent updates

Atomic conflict = serialized updates
Histogram calculation on GPU requires atomic operations

For (each pixel $i$ in image $I$)

1. $Pixel = I[i]$  // Read pixel
2. $Pixel' = \text{Computation}(Pixel)$  // Optional computation
3. $\text{Histogram} [Pixel']++$  // Vote in histogram bin
Optimized Parallel Reduction

- 7 versions in CUDA samples: Tree-based reduction in shared memory
  - Version 0: No whole warps active
  - Version 1: Contiguous threads, but many bank conflicts
  - Version 2: No bank conflicts
  - Version 3: First level of reduction when reading from global memory
  - Version 4: Warp shuffle or unrolling of final warp
  - Version 5: Warp shuffle or complete unrolling
  - Version 6: Multiple elements per thread sequentially

https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction
## 7 Versions of Reduction

<table>
<thead>
<tr>
<th>Version</th>
<th>Fermi GTX 580</th>
<th>Throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.81</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>14.92</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>21.40</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>49.10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>62.40</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>81.62</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>159.45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Kepler K20</th>
<th>Throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15.98</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>20.69</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>29.25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>57.98</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>85.36</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>95.31</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>125.02</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Maxwell GTX 980</th>
<th>Throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.35</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>39.02</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>48.07</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>86.52</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>106.66</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>106.49</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>165.30</td>
<td></td>
</tr>
</tbody>
</table>
Reduction with Atomic Operations

- 3 new versions of reduction based on 3 previous versions
  - Version 0: No whole warps active
  - Version 3: First level of reduction when reading from global memory
  - Version 6: Multiple elements per thread sequentially

- New versions 7, 8, and 9
  - Replace the for loop (tree-based reduction) with one atomic operation per thread
## 10 Versions of Reduction

<table>
<thead>
<tr>
<th>Version</th>
<th>Throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fermi GTX 580</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10.81</td>
</tr>
<tr>
<td>7</td>
<td>2.28</td>
</tr>
<tr>
<td>1</td>
<td>14.92</td>
</tr>
<tr>
<td>2</td>
<td>21.40</td>
</tr>
<tr>
<td>3</td>
<td>49.10</td>
</tr>
<tr>
<td>8</td>
<td>5.86</td>
</tr>
<tr>
<td>4</td>
<td>62.40</td>
</tr>
<tr>
<td>5</td>
<td>81.62</td>
</tr>
<tr>
<td>6</td>
<td>159.45</td>
</tr>
<tr>
<td>9</td>
<td>152.44</td>
</tr>
</tbody>
</table>

| **Kepler K20** | |
| 0 | 15.98 |
| 7 | 3.76 |
| 1 | 20.69 |
| 2 | 29.25 |
| 3 | 57.98 |
| 8 | 4.37 |
| 4 | 85.36 |
| 5 | 95.31 |
| 6 | 125.02 |
| 9 | 118.20 |

| **Maxwell GTX 980** | |
| 0 | 20.35 |
| 7 | 56.57 |
| 1 | 39.02 |
| 2 | 48.07 |
| 3 | 86.52 |
| 8 | 93.56 |
| 4 | 106.66 |
| 5 | 106.49 |
| 6 | 165.30 |
| 9 | 165.46 |
10 Versions of Reduction

We save lines of code
Search Space of Parallel Reduction

Over 85 different versions possible!
Automatic Generation of Parallel Reduction


Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs

Simon Garcia De Gonzalo
CS and Coordinated Science Lab
UIUC
grdgnz2@illinois.edu

Sitao Huang
ECE and Coordinated Science Lab
UIUC
shuang91@illinois.edu

Juan Gómez-Luna
Computer Science
ETH Zurich
juang@ethz.ch

Simon Hammond
Scalable Computer Architecture
Sandia National Laboratories
sdhammo@sandia.gov

Onur Mutlu
Computer Science
ETH Zurich
omutlu@ethz.ch

Wen-mei Hwu
ECE and Coordinated Science Lab
UIUC
w-hwu@illinois.edu
Parallel Reduction with Tensor Cores

Accelerating Reduction and Scan Using Tensor Core Units

Abdul Dakkak, Cheng Li
University of Illinois Urbana-Champaign
Urbana, Illinois
{dakkak,cli99}@illinois.edu

Isaac Gelado
NVIDIA Corporation
Santa Clara, California
igelado@nvidia.com

Jinjun Xiong
IBM T. J. Watson Research Center
Yorktown Heights, New York
jinjun@us.ibm.com

Wen-mei Hwu
University of Illinois Urbana-Champaign
Urbana, Illinois
w-hwu@illinois.edu

ICS 2019
Recommended Readings

  - Chapter 5: Performance considerations
  - Chapter 9 - Parallel patterns — parallel histogram computation: An introduction to atomic operations and privatization
P&S Heterogeneous Systems

Parallel Patterns: Reduction

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2021
11 November 2021