P&S Heterogeneous Systems

Parallel Patterns: Reduction

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Fall 2022
7 November 2022
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

[Diagram showing the memory hierarchy with Grid (Device) at the top, followed by Block (0, 0) and Block (1, 0) with Shared memory, Registers, and Threads (0, 0) and (1, 0). Below are the Global / Texture & Surface memory and Constant memory, connected to the 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
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?
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 $\rightarrow$ N iterations
  
  \[
  \begin{array}{cccccccccc}
  \end{array}
  \]

  \[
  \text{sum} = 0; \quad \text{// Initialize with identity value}
  \]

  \[
  \text{for}(i = 0; i < N; ++i) \{
  \]

  \[
  \text{sum += A[i];} \quad \text{// Accumulate elements of input array A[]} 
  \]

  \}

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


Iteration 1

Iteration 2

Iteration 3

Iteration 4

Partial results in temporary storage

log(N) iterations
Tree-Based Reduction on GPU

- **Block 0**
  - **Warp 0**
  - **Warp 1**

- **Block 1**
  - **Warp 0**
  - **Warp 1**
  - **A[N-1]**

- Partial results in shared memory (or registers)

**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**
Vector Reduction: Naïve Mapping (I)

0...7

0+1
2+3
4+5
6+7
8+9
10+11

0
1
2
3
4
5
6
7
8
9
10
11

Thread 0
Thread 2
Thread 4
Thread 6
Thread 8
Thread 10

1

2

3

iterations

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?
Divergence-Free Mapping (I)

- All active threads belong to the same warp

Slide credit: Hwu & Kirk
Divergence-Free Mapping (II)

- Program with high SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

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

Warp utilization is maximized
Divergence-Free Mapping (III)

- Program with **high SIMD utilization**

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 0; stride >>= 1){
    __syncthreads();

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

We can use **warp shuffle** to avoid shared memory accesses and **__syncthreads()**

Stride examples:

- **Stride = 16**
- **Stride = 8**
- **Stride = 4**
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(mask, var, srcLane)`
    - Direct copy from indexed lane
  - `__shfl_up_sync(mask, var, delta)`
    - Copy from a lane with lower ID relative to caller
  - `__shfl_down_sync(mask, var, delta)`
    - Copy from a lane with higher ID relative to caller
  - `__shfl_xor_sync(mask, var, laneMask)`
    - Copy from a lane based on bitwise XOR of own lane ID
Threads running on processing engines have access to a local register file (LRF)

And shared memory banks (SRF)

**FIG. 13**

Nickolls et al., "Single Interconnect Providing Read and Write Access to a Memory Shared by Concurrent Threads," US7680988B1
Read from Shared Memory Bank

FIG. 13

Nickolls et al., "Single Interconnect Providing Read and Write Access to a Memory Shared by Concurrent Threads," US7680988B1
Write to Shared Memory Bank

FIG. 13

Nickolls et al., "Single Interconnect Providing Read and Write Access to a Memory Shared by Concurrent Threads," US7680988B1
Shuffling Operations within a Warp

FIG. 13

Nickolls et al., "Single Interconnect Providing Read and Write Access to a Memory Shared by Concurrent Threads," US7680988B1
Program with high SIMD utilization

```c
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

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

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

- Partial results in shared memory (or registers)
- 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

Warp shuffle
__shfl_sync(...);

Block 0

Block 1

Warp 0

Warp 1

Warp 0

Warp 1
Reduction with Warp Shuffle

```c
__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;
    }
}
```
Warp Reduce Functions

- Ampere (cc 8.x) adds native support for warp-wide reduction operations

---

B.21. Warp Reduce Functions

The __reduce_sync(unsigned mask, T value) intrinsics perform a reduction operation on the data provided in value after synchronizing threads named in mask. T can be unsigned or signed for [add, min, max] and unsigned only for [and, or, xor] operations.

Supported by devices of compute capability 8.x or higher.

### B.21.1. Synopsis

```c
// add/min/max
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);
int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);

// and/or/xor
unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

### B.21.2. Description

- **__reduce_add_sync**, **__reduce_min_sync**, **__reduce_max_sync**
  Returns the result of applying an arithmetic add, min, or max reduction operation on the values provided in value by each thread named in mask.

- **__reduce_and_sync**, **__reduce_or_sync**, **__reduce_xor_sync**
  Returns the result of applying a logical AND, OR, or XOR reduction operation on the values provided in value by each thread named in mask.

The mask indicates the threads participating in the call. A bit, representing the thread’s lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

Tree-Based Reduction on GPU (with Warp Reduce)

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

Partial results in shared memory (or registers)
Reduction with Warp Shuffle

```c
__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;
    }
}
```
Reduction with Warp Reduce

```c
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {
    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ int 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 with warp reduce instruction
    int sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
        // warp reduce intrinsic for cc 8.0 or higher
        sum = __reduce_add_sync(0xffffffff, sum);
    }

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

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

- Bitwise functions
  - And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)*

* Datatypes for different atomic operations in [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts

No atomic conflict = concurrent updates

Atomic conflict = serialized updates
Recall: Uses of Atomic Operations

- **Computation**
  - Atomics on an array that will be the output of the kernel
  - Example
    - Histogram, reduction

- **Synchronization**
  - Atomics on memory locations that are used for synchronization or coordination
  - Example
    - Counters, locks, flags...

- Use them to prevent **data races** when more than one thread need to update the same memory location
**Image Histogram**

- Histograms are widely used in **image processing**
  - Some **computation before voting** in the histogram may be needed

```
For (each pixel i in image I){
    Pixel = I[i]  // Read pixel
    Pixel’ = Computation(Pixel)  // Optional computation
    Histogram[Pixel’]++  // Vote in histogram bin
}
```

- Parallel threads frequently incur **atomic conflicts** in image histogram computation

---

Diagram showing the input data distribution among threads and the voting process in the histogram.
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

- **Fermi GTX 580**
  - Version 0: 10.81 GB/s
  - Version 1: 14.92 GB/s
  - Version 2: 21.40 GB/s
  - Version 3: 49.10 GB/s
  - Version 4: 62.40 GB/s
  - Version 5: 81.62 GB/s
  - Version 6: 159.45 GB/s

- **Kepler K20**
  - Version 0: 15.98 GB/s
  - Version 1: 20.69 GB/s
  - Version 2: 29.25 GB/s
  - Version 3: 57.98 GB/s
  - Version 4: 85.36 GB/s
  - Version 5: 95.31 GB/s
  - Version 6: 125.02 GB/s

- **Maxwell GTX 980**
  - Version 0: 20.35 GB/s
  - Version 1: 39.02 GB/s
  - Version 2: 48.07 GB/s
  - Version 3: 86.52 GB/s
  - Version 4: 106.66 GB/s
  - Version 5: 106.49 GB/s
  - Version 6: 165.30 GB/s
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 shared memory atomic operation per thread
10 Versions of Reduction

Fermi GTX 580

Kepler K20

Maxwell GTX 980
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Affiliation</th>
<th>Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon Garcia De Gonzalo</td>
<td>CS and Coordinated Science Lab, UIUC</td>
<td><a href="mailto:grcdgnz2@illinois.edu">grcdgnz2@illinois.edu</a></td>
</tr>
<tr>
<td>Sitao Huang</td>
<td>ECE and Coordinated Science Lab, UIUC</td>
<td><a href="mailto:shuang91@illinois.edu">shuang91@illinois.edu</a></td>
</tr>
<tr>
<td>Juan Gómez-Luna</td>
<td>Computer Science, ETH Zurich</td>
<td><a href="mailto:juang@ethz.ch">juang@ethz.ch</a></td>
</tr>
<tr>
<td>Simon Hammond</td>
<td>Scalable Computer Architecture, Sandia National Laboratories</td>
<td><a href="mailto:sdhammo@sandia.gov">sdhammo@sandia.gov</a></td>
</tr>
<tr>
<td>Onur Mutlu</td>
<td>Computer Science, ETH Zurich</td>
<td><a href="mailto:omutlu@ethz.ch">omutlu@ethz.ch</a></td>
</tr>
<tr>
<td>Wen-mei Hwu</td>
<td>ECE and Coordinated Science Lab, UIUC</td>
<td><a href="mailto:w-hwu@illinois.edu">w-hwu@illinois.edu</a></td>
</tr>
</tbody>
</table>
Parallel Reduction with Tensor Cores

- Reduction can be expressed as a dot product operation and get accelerated by GPU tensor core units
  - \[ \text{sum} = A_0 \times B_0 + A_1 \times B_1 + \ldots + A_{N-1} \times B_{N-1} \]
  - With all \( B_i = 1 \), the result will be the sum of array \( A \)

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

Dakkak et al., "Accelerating Reduction and Scan Using Tensor Core Units," ICS 2019
Recommended Readings (I)

  - Chapter 5: Performance considerations
  - Chapter 9 - Parallel patterns — parallel histogram computation: An introduction to atomic operations and privatization
Recommended Readings (II)

  - Chapter 6 - Performance considerations
  - Chapter 10 - Reduction: And minimizing divergence
P&S Heterogeneous Systems
Parallel Patterns: Reduction

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2022
7 November 2022