P&S Heterogeneous Systems

Parallel Patterns: Reduction

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
14 April 2023
Performance Considerations
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
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

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();

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)

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
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

- **stride = 16**
- **stride = 8**
- **stride = 4**
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()`
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
Tree-Based Reduction on GPU (with Warp Shuffle)

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)

Warp shuffle
__shfl_sync(...);
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 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;
    }
}
```

Slide credit: Izzat El Hajj
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;
    }
}
```
Reduction with Thread Coarsening

```c
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {
    __shared__ int input_s[BLOCK_DIM];

    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x; // Global thread index
    const unsigned int gridSize = gridDim.x * blockDim.x; // Total number of threads

    int mySum = 0; // Local (per-thread) sum

    while (i < N) {
        mySum += input[i];
        i += gridSize;
    }

    // Load local sum to shared memory
    input_s[threadIdx.x] = mySum;
    __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();
    }

    ...
}
```
Search Space of Parallel Reduction

Over 85 different versions possible!

Recommended Readings (II)

  - Chapter 6 - Performance considerations
  - Chapter 10 - Reduction: And minimizing divergence
Shuffling Operations within a Warp

FIG. 13

HetSys Course: Lecture 6: Parallel Patterns: Reduction (Fall 2022)

Longer Lecture

https://youtu.be/tgVmqbE_eMc