Lab 7: CUDA Reduction

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 11:59 PM on 10/31

Overview

This lab introduces GPU reduction kernels, an essential building block for many parallel algorithms. You will start with a naive global-memory reduction, optimize it using shared memory, and finally implement a warp-level reduction using CUDA primitives.

Learning Objectives

- Understand how reduction operations can be parallelized on GPUs.
- Compare global-memory, shared-memory, and warp-level reduction strategies.
- Use CUDA timing events and compute average runtime over multiple runs.
- Interpret synchronization and atomic operations in reduction kernels.

Euler Instruction

```
~$ ssh your_CAE_account@euler.engr.wisc.edu
~$ sbatch your_slurm_script.slurm
```

Do not run on the login node. Work locally, push to GitHub, and run on Euler using Slurm.

Submission Instruction

Specify your GitHub link here: https://github.com/boomchrisvanden/ECE455/HW07
https://github.com/YourGitHubName/ECE455/HW07

Problem 1: Naive Global-Memory Reduction

Task: Implement a simple GPU reduction kernel that uses only global memory and atomicAdd. Each thread adds one input element into a single global output variable.

Kernel

Filename: reduce_naive.cu

```
__global__ void reduce_naive(const int* in, int* out, size_t n_elems) {
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n_elems)
        atomicAdd(out, in[idx]);
}
```

Full code available: reduce_naive.cu

Slurm Script

Filename: reduce_naive.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=reduce_naive.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc reduce_naive.cu -o reduce_naive
./reduce_naive
```

Problem 2: Shared-Memory Reduction

Task: Optimize the reduction kernel by first reducing within each block using shared memory, then accumulating block sums globally.

Kernel

Filename: `reduce_shared.cu`

```
__global__ void reduce_shared(const int* in, int* out, size_t n_elems) {
    // Each block stores partial results in on-chip shared memory
    __shared__ int sdata[BLOCK_DIM];

    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Load one element per thread from global to shared memory
    // Threads beyond the valid range write 0 to avoid out-of-bound reads
    int x = (idx < n_elems) ? in[idx] : 0;
    sdata[tid] = x;
    __syncthreads();  // Wait for all threads to finish loading

    // --- In-block tree reduction ---
    // Start with stride = half block size, then repeatedly halve.
    // Each active thread adds the element stride positions ahead.
    // Example: stride=128 ==> thread 0 adds thread 128's value, etc.
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride)
            sdata[tid] += sdata[tid + stride];
        // Synchronize to make sure all additions complete
        // before using the updated shared memory in the next step.
        __syncthreads();
    }

    // After the loop, thread 0 holds the sum of all elements in this
       block.
    // It atomically adds that partial result to the global output.
    if (tid == 0)
        atomicAdd(out, sdata[0]);
}
```

Full code available: `reduce_shared.cu`

Slurm Script

Filename: `reduce_shared.slurm`

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=reduce_shared.output
```

```
cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc reduce_shared.cu -o reduce_shared
./reduce_shared
```

Problem 3: Warp-Level Reduction

Task: Implement a warp-level reduction using the CUDA intrinsic `__shfl_down_sync()` to exchange data between threads in the same warp. This version avoids shared-memory synchronization and reduces intra-warp latency.

Kernel

Filename: `reduce_warp.cu`

```
__device__ int warp_reduce_sum(int val) {
    // Each iteration halves the number of active lanes.
    // Example: for offset=16, lanes [0..15] add lanes [16..31]'s values.
    // The data moves between threads using warp shuffle instructions.
    for (int offset = 16; offset > 0; offset >>= 1)
        val += __shfl_down_sync(0xFFFFFFFF, val, offset);
    // After this loop, lane 0 of the warp holds the total sum of that
       warp.
    return val;
}

__global__ void reduce_warp(const int* in, int* out, size_t num_elems) {
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int val = (idx < num_elems) ? in[idx] : 0;

    // --- Perform warp-level reduction ---
    // Each warp (32 threads) computes a local partial sum.
    val = warp_reduce_sum(val);

    // --- Write partial results ---
    // Only the first thread in each warp (lane 0) performs the global
       atomic add.
    // This prevents multiple threads in the same warp from writing
       duplicates.
    if ((threadIdx.x & 31) == 0)
        atomicAdd(out, val);
}
```

Full code available: `reduce_warp.cu`

Explanation

Within a warp, threads cooperate implicitly without `__syncthreads()`. During the reduction:

- At each step, half the threads add values from the other half.
- For offset = 16, lanes 0–15 add lanes 16–31.
- For offset = 8, lanes 0–7 add lanes 8–15, and so on.
- Eventually, lane 0 contains the total sum for that warp.

Slurm Script

Filename: `reduce_warp.slurm`

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=reduce_warp.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc reduce_warp.cu -o reduce_warp
./reduce_warp
```

Problem 4: Reflection

Task: Briefly summarize the challenges you encountered in this lab.

I had a hard time minimizing global memory traffic within the kernel so I used ChatGPT to explain the topic to me so that I had a better understanding.