Lab 6: Shared Memory Optimization

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 11:59 PM on 10/24

Overview

This lab focuses on using **shared memory** to optimize CUDA kernels. You will first write a warm-up kernel that loads data into shared memory for basic computation, then apply shared-memory tiling to matrix multiplication, and finally compare global vs. shared memory performance.

Learning Objectives

- Understand the role of CUDA shared memory.
- Implement a tiled shared-memory matrix multiplication.
- Compare runtime and scalability with a global-memory kernel.
- Summarize performance improvements and debugging challenges.

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/HW06
https://github.com/YourGitHubName/ECE455/HW06

Problem 1: Shared Memory Warm-Up

Task: Write a CUDA kernel that demonstrates basic use of shared memory:

1. Load a block of elements from global memory into shared memory.
2. Square each element inside shared memory.
3. Write the results back to global memory.

Kernel

Filename: shared_warmup.cu

```
template < typename T >
__global__ void square_shared_kernel(const T * in , T * out , size_t N ) {
    __shared__ T tile [ BLOCK_DIM ];

    size_t idx = blockIdx . x * blockDim . x + threadIdx . x ;
    if ( idx >= N ) return ;

    // 1. Load from global to shared memory tile [ threadIdx . x ] = in [ idx ]; __syncthreads () ;


    // 2. Compute in shared memory
    tile [ threadIdx . x ] = tile [ threadIdx . x ] * tile [ threadIdx . x ]; __syncthreads () ;

    // 3. Write back to global memory
    out [ idx ] = tile [ threadIdx . x ];
}
```

Full source and main function: GitHub Gist

Slurm Script

Filename: shared_warmup.slurm

```
#!/ usr/bin/env zsh
# SBATCH-- partition = instruction
#SBATCH -- time = 00 :01: 00
#SBATCH -- ntasks =1
#SBATCH -- cpus - per - task =1
#SBATCH -- gpus - per - task =1
# SBATCH-- output = shared_warmup . output

cd $SLURM_SUBMIT_DIR
module load nvidia / cuda
nvcc shared_warmup . cu -o shared_wa
rmup ./shared_warmup
```

Problem 2: Tiled Matrix Multiplication with Shared Memory

Task: Implement a matrix multiplication kernel using shared-memory tiling. Each thread block should load a TILE_SIZE × TILE_SIZE tile of matrices A and B into shared memory, synchronize threads, and compute the corresponding tile of C.

Kernel

Filename: mm_tiled.cu

```
template <typename T>
__global__ void mm_tiled(const T* A, const T* B, T* C, int N) {
    // Allocate shared-memory tiles for A and B
    __shared__ T tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ T tile_B[TILE_SIZE][TILE_SIZE];

    // Compute the row and column index this thread is responsible for int row =
    blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    T val = 0;

    // Loop over all tiles required to compute one C-tile
    for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; ++t) {

        // Load one tile of A and one tile of B from global to shared memory

        if (row < N && (t * TILE_SIZE + threadIdx.x) < N)
            tile_A[threadIdx.y][threadIdx.x] =
                A[row * N + t * TILE_SIZE + threadIdx.x];
        else
            tile_A[threadIdx.y][threadIdx.x] = 0;

        if (col < N && (t * TILE_SIZE + threadIdx.y) < N)
            tile_B[threadIdx.y][threadIdx.x] =
                B[(t * TILE_SIZE + threadIdx.y) * N + col];
        else
            tile_B[threadIdx.y][threadIdx.x] = 0;

        __syncthreads(); // Wait until all threads load their tile

        // Multiply the two tiles
        for (int k = 0; k < TILE_SIZE; ++k)
            val += tile_A[threadIdx.y][k] * tile_B[k][threadIdx.x];

        __syncthreads(); // Wait before loading the next tile
    }

    // Write result to global memory
    if (row < N && col < N)
        C[row * N + col] = val;
}
```

Full source (with validation and timing): GitHub Gist

Slurm Script

Filename: mm_tiled.slurm

```
#!/ usr/bin/env  zsh
# SBATCH-- p ar ti ti on = i n s t r u c t i o n
#SBATCH  -- time = 00 :0 1: 00
#SBATCH  -- ntasks =1
#SBATCH  -- cpus - per - task =1
#SBATCH  -- gpus - per - task =1
#SBATCH  -- output = mm_tiled . output

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

Problem 3: Global vs. Shared Memory Performance

Task: Compare the runtime of a naive global-memory matrix multiplication and a tiled shared-memory version. Measure both runtimes using CUDA events and report the observed speedup.

Naive Global-Memory Kernel

Filename: mm_compare_tiled_vs_naive.cu

```
template <typename T>
__global__ void mm_naive(const T* A, const T* B, T* C, int N) { int tid =
    blockIdx.x * blockDim.x + threadIdx.x;
    int total_elems = N * N;
    if (tid >= total_elems) return;

    int row = tid / N;
    int col = tid % N;

    T val = 0;
    for (int k = 0; k < N; ++k)
        val += A[row * N + k] * B[k * N + col];

    C[tid] = val;
}
```

Tiled Shared-Memory Kernel

```
template <typename T>
__global__ void mm_tiled(const T* A, const T* B, T* C, int N) {
    // Declare shared-memory tiles
    __shared__ T tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ T tile_B[TILE_SIZE][TILE_SIZE];

    // Compute this thread's global row/col index
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    T val = 0; // Accumulator for the result

    // Loop through all tiles of A and B needed for this output block
    for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; ++t) {
        // Load a tile of A and a tile of B into shared memory
        if (row < N && (t * TILE_SIZE + threadIdx.x) < N)
            tile_A[threadIdx.y][threadIdx.x] =
                A[row * N + t * TILE_SIZE + threadIdx.x];
        else
            tile_A[threadIdx.y][threadIdx.x] = 0;

        if (col < N && (t * TILE_SIZE + threadIdx.y) < N)
            tile_B[threadIdx.y][threadIdx.x] =
                B[(t * TILE_SIZE + threadIdx.y) * N + col];
        else
            tile_B[threadIdx.y][threadIdx.x] = 0;
```

```
        __syncthreads(); // Synchronize all threads before computing

        // Compute partial products for this tile for (int k = 0; k <
        TILE_SIZE; ++k)
            val += tile_A[threadIdx.y][k] * tile_B[k][threadIdx.x];

        __syncthreads(); // Wait before loading next tile
    }

    // Write result to global memory
    if (row < N && col < N)
        C[row * N + col] = val;
}
```

Full source (with validation and timing): GitHub Gist

Slurm Script

Filename: mm_compare_tiled_vs_naive.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=mm_compare_tiled_vs_naive.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc mm_compare_tiled_vs_naive.cu -o mm_compare_tiled
_vs_naive ./mm_compare_tiled_vs_naive
```

Problem 4: Reflection

Task: Summarize the challenges you faced in this lab.

I was having trouble getting the indexing, tiling loop, and edge handling right conceptually, so I used ChatGPT to better understand the topic.