Project Report

Name: Venkata Sai Phanindra Mandadapu

Optimized Matrix Multiplication on GPU Using Tiling and Shared Memory Techniques.

Abstract:
This project explores the optimization of matrix multiplication on GPUs through tiling and shared memory techniques, enhancing computational efficiency and scalability. It contrasts a naive GPU-based approach with an optimized method across three different matrix sizes to illustrate the benefits of optimized memory handling and parallel processing.

1. Introduction

Matrix multiplication is central to many scientific, engineering, and machine learning applications. The computationally intensive nature of matrix multiplication necessitates optimization to enhance performance, particularly when processing large data sets. GPUs offer substantial parallel processing power, making them ideal for accelerating operations such as matrix multiplication. However, naive implementations often underutilize GPU memory hierarchies, leading to performance bottlenecks. This project addresses these challenges by implementing tiling and shared memory optimizations to improve execution efficiency.

Approach:

The project utilizes CUDA programming to implement two versions of matrix multiplication on an NVIDIA GPU: a naive version and an optimized version. The naive version computes matrix multiplication directly using global memory, while the optimized version uses tiling techniques and shared memory to minimize global memory accesses and maximize cache utilization.

Naive Implementation:

The initial phase of the project involved developing a naive matrix multiplication kernel using CUDA, executed on an NVIDIA GPU. This naive version serves as a baseline for comparison against the optimized version. Each thread in this implementation is responsible for computing one element of the result matrix by straightforwardly multiplying the corresponding row and column from two input matrices.

Code snippet:

```
__global__ void matrixMultNaive(float *A, float *B, float *C, int N) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (col < N && row < N) {
      for (int i = 0; i < N; i++) {
         sum += A[row * N + i] * B[i * N + col];
      }
      C[row * N + col] = sum;
```

```
    }
}
```

Optimized Implementation:

The optimized version divides the input matrices into tiles that are loaded into shared memory. This approach reduces the frequency and volume of global memory accesses. Each block of threads computes a tile of the result matrix by accumulating results from multiple tiled multiplications, significantly reducing memory access latency and increasing throughput.

Code snippet:

```
__global__ void matrixMultOpt(float *A, float *B, float *C, int N) {
    __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];
    int tx = threadIdx.x, ty = threadIdx.y;
    int row = blockIdx.y * TILE_WIDTH + ty;
    int col = blockIdx.x * TILE_WIDTH + tx;
    float sum = 0.0;
    for (int m = 0; m < N/TILE_WIDTH; ++m) {
        tileA[ty][tx] = A[row * N + m * TILE_WIDTH + tx];
        tileB[ty][tx] = B[(m * TILE_WIDTH + ty) * N + col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k) {
            sum += tileA[ty][k] * tileB[k][tx];
        }
        __syncthreads();
    }
    C[row * N + col] = sum;
}
```

Results

The performance of the naive and optimized implementations was evaluated by measuring the execution times for matrix sizes ranging from 512 to 4096. The execution times were recorded as follows:

Matrix Size	**Naive Implementation**	**Optimized Implementation**
512x512	1.2717ms	0.85137ms
1024x1024	9.2484ms	5.476ms
2048x2048	75.0469ms	42.0904ms
3072x3072	228.3017ms	139.11325ms
4096x4096	393.9755ms	274.743ms

The data clearly shows that the optimized implementation consistently outperforms the naive approach, cutting execution times by approximately 41% across all tested matrix sizes.
The significant reduction in execution time for the optimized implementation confirms the efficacy of using tiling and shared memory in exploiting the GPU architecture more effectively. The reduced execution times demonstrate enhanced scalability and efficiency, crucial for applications requiring large-scale data computations.

Conclusion

This project successfully demonstrates the advantages of optimizing matrix multiplication using tiling and shared memory on GPUs. Future work could explore further optimizations, such as varying tile sizes and incorporating more complex memory management techniques, to enhance performance even further.