Lebanese American University

Department of Computer Science & Mathematics

Parallel Programming

CSC 447 - Section 11

Assignment 3

Instructor: Hamdan Abdellatef

Elie Hanna

202101485

GitHub Link: https://github.com/eliehanna2/Parallel-Programming/tree/main/cuda-c-matrix-multiplication

1.1-Basic matrix multiplication:

The parallelization of matrix multiplication in the provided code is achieved by utilizing the CUDA programming model. CUDA allows us to leverage the computational power of the GPU by executing parallel tasks called threads. In the code, two kernel functions are defined: MatrixMulSquareKernel and MatrixMulRectangularKernel. These kernels are responsible for performing the matrix multiplication on the GPU.

For the square matrix case, the MatrixMulSquareKernel is invoked. The GPU grid and block dimensions are calculated based on the size of the matrices. Each thread within a block is assigned to compute a single element of the resulting matrix. The threads iterate over the corresponding rows and columns of the input matrices, accumulating the partial products to compute the final value. By distributing the workload among multiple threads, the computation can be performed in parallel.

Similarly, for rectangular matrices, the MatrixMulRectangularKernel is used. The grid and block dimensions are determined accordingly, considering the dimensions of the input matrices. Each thread within a block computes a single element of the output matrix by iterating over the rows and columns of the input matrices.

The two kernels, MatrixMulSquareKernel and MatrixMulRectangularKernel, differ in how they handle indexing and iteration over the input matrices. The MatrixMulSquareKernel is designed for square matrices, where the dimensions are the same. It utilizes a single loop that iterates over the range of the matrix width. The pseudocode for this kernel can be summarized as follows:

```
for Row in range(Width):
    for Col in range(Width):
        Pvalue = 0
        for k in range(Width):
            Pvalue += M[Row * Width + k] * N[k * Width + Col]
        P[Row * Width + Col] = Pvalue
```

On the other hand, the MatrixMulRectangularKernel is suitable for rectangular matrices with different dimensions. It adjusts the indexing and iteration based on the heights and widths of the input matrices. The pseudocode for this kernel can be outlined as follows:

```
for Row in range(HeightM):
    for Col in range(WidthN):
        Pvalue = 0
        for k in range(WidthM):
            Pvalue += M[Row * WidthM + k] * N[k * WidthN + Col]
        P[Row * WidthN + Col] = Pvalue
```

By launching these kernel functions with the specified grid and block dimensions, multiple threads execute concurrently on the GPU, performing the matrix multiplication in parallel. This parallel execution significantly improves the overall performance compared to sequential execution on the CPU.

1.2-Matrix multiplication with tiling:

To parallelize the computation, we utilized the concept of tiling or thread blocking. Tiling involves dividing the input matrices into smaller blocks, and each block is processed independently by a group of threads within a CUDA block. This allows for better memory access patterns and reduces the number of global memory accesses.

In both MatrixMulSquareKernel and MatrixMulRectangularKernel, we introduced the TILE_SIZE constant, which defines the size of the tile or block. The matrices are divided into smaller tiles, and each thread block is responsible for processing one tile.

MatrixMulSquareKernel:

```
for Row in range(HeightM):
    for Col in range(WidthN):
        Pvalue = 0
        for i in range(Width / TILE_SIZE):
            sharedM[ty][tx] = M[Row * Width + i * TILE_SIZE + tx]
            sharedN[ty][tx] = N[(i * TILE_SIZE + ty) * Width + Col]
            synchronize_threads()

            for k in range(TILE_SIZE):
```

```
            Pvalue += sharedM[ty][k] * sharedN[k][tx]
        synchronize_threads()
    P[Row * Width + Col] = Pvalue
```

In this pseudocode, we iterate over each element of the output matrix P, represented by Row and Col.We utilize shared memory by declaring sharedM and sharedN arrays of size TILE_SIZE x TILE_SIZE. Each thread in a block has its own local copy of these arrays stored in shared memory. The purpose of shared memory is to improve memory access patterns and reduce global memory access latency. It allows threads within a block to cooperatively load data into shared memory and reuse it efficiently. Within the inner loop, we perform tiling by loading a tile of size TILE_SIZE x TILE_SIZE from matrices M and N into shared memory. Each thread loads one element from M and N into its corresponding location in shared memory. After loading the tile, we synchronize the threads using synchronize_threads() to ensure that all threads have finished loading before proceeding to the computation phase.Then, each thread performs the matrix multiplication by iterating over the elements of the loaded tile in shared memory, multiplying the corresponding elements of sharedM and sharedN, and accumulating the result in Pvalue. Again, we synchronize the threads to make sure all threads have finished the computation before proceeding to the next iteration. Finally, we store the computed Pvalue in the output matrix P at the appropriate location (Row * Width + Col).

2-Matrix MulRectangular Kernel:

```
for Row in range(HeightM):
   for Col in range(WidthN):
      Pvalue = 0
      for i in range(WidthM / TILE_SIZE):
         sharedM[ty][tx] = M[Row * WidthM + i * TILE_SIZE + tx]
         sharedN[ty][tx] = N[(i * TILE_SIZE + ty) * WidthN + Col]
         synchronize_threads()

         for k in range(TILE_SIZE):
            Pvalue += sharedM[ty][k] * sharedN[k][tx]
         synchronize_threads()
      P[Row * WidthN + Col] = Pvalue
```

This pseudocode is similar to the previous one, but it handles rectangular matrices with different dimensions (WidthM, HeightM, WidthN, HeightN).The outer loops iterate over each element of the output matrix P, represented by Row and Col. The inner loops perform tiling by loading tiles of size TILE_SIZE x TILE_SIZE from matrices M and N into shared memory. The key difference is in how we calculate the indices for loading elements from matrices M and N into shared memory. We use the provided WidthM, HeightM, WidthN, and HeightN values to calculate the indices accordingly. The rest of the process, including the computation and storing the result in the output matrix, is the same as in the previous kernel. In both cases, the use of shared memory improves memory access patterns and reduces global memory access latency, leading to better performance by maximizing data reuse and minimizing global.

To optimize memory access patterns and reduce global memory latency, we introduced the concept of tiling and utilized shared memory. We declared sharedM and sharedN arrays of size TILE_SIZE x TILE_SIZE in shared memory. Each thread in a block had its own local copy of these arrays stored in shared memory. We loaded a tile of size TILE_SIZE x TILE_SIZE from matrices M and N into shared memory, with each thread loading one element into its corresponding location in shared memory.

By employing tiling and shared memory, we achieved better data locality and reduced the number of global memory accesses. The threads within a block cooperatively loaded data into shared memory and reused it efficiently during the computation phase, which significantly improved performance by minimizing memory access latency.

After loading the tile into shared memory, we synchronized the threads using __syncthreads() to ensure all threads had finished loading before proceeding to the computation phase. Within the inner loop, each thread performed matrix multiplication by iterating over the elements of the loaded tile in shared memory, multiplying the corresponding elements of sharedM and sharedN, and accumulating the result in a local variable, Pvalue.

Finally, we stored the computed Pvalue in the output matrix P at the appropriate location. The computation was parallelized by assigning different elements of the output matrix to different threads, and the use of shared memory and tiling improved memory access patterns and reduced memory latency, resulting in accelerated computation.

3-

```
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/Assignment3/Assignment3$ gcc -o matrixMultiplicationSequentialCode matrixMultiplicationSequentialCode.c
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/Assignment3/Assignment3$ ./matrixMultiplicationSequentialCode
Multiplication SEQUENTIAL TIME took  10.374840 seconds
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/Assignment3/Assignment3$
```

```
ignment3/Assignment3$ ./basicParallelMatrixMultiplication
Parallel Execution Time (No Tiling): 0.364720 seconds
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive -
ignment3/Assignment3$
```

```
ignment3/Assignment3$ ./ParallelMatrixMultiplicationWithTiling
Execution Time (Rectangular Matrix): 0.308210 seconds
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Le
ignment3/Assignment3$
```

FOR: **ROW: 1024 Width: 768 t**

Sequential time = 10.3748s

Parallel time (no tiling) = 0.3647s

Speedup Factor = 28.4474

Efficiency: block Dim(thread number) =	64	256	1024
Time execution =	0. 3647	0.3624	0.3485
Efficiency =	44.4	11.1	2.77

Parallel time (tiling) = 0.3082s

speedup factor = 33.6625 tiles = 16

Efficiency: block Dim (thread number) =	64	256	1024
Time execution =	0.3118	0.3082	0.2982
Efficiency =	52.5	13.1	3.28

4- Comparison of the two implementations:

The two implementations, one with tiling (MatrixMulSquareKernel) and one without tiling (MatrixMulRectangularKernel), differ in terms of their memory access patterns and performance characteristics. Here is a detailed comparison of the two implementations:

Memory Access Patterns:

Without Tiling: In the MatrixMulRectangularKernel, each thread accesses consecutive elements from the input matrices M and N in global memory. This leads to non-coalesced memory accesses as threads within a warp access memory location that are not contiguous.

With Tiling: In the MatrixMulSquareKernel, threads within a block cooperatively load a tile of elements from matrices M and N into shared memory. This results in coalesced memory accesses as threads access consecutive locations in shared memory, which can improve memory bandwidth utilization.

Memory Access Efficiency:

Without Tiling: The implementation without tiling suffers from high global memory latency due to non-coalesced memory accesses. Multiple memory transactions are required to retrieve consecutive elements, resulting in lower memory access efficiency.

With Tiling: The tiling implementation reduces global memory latency by utilizing shared memory. Threads within a block cooperatively load a tile of elements into shared memory, reducing the number of global memory accesses and improving memory access efficiency.

Computation Efficiency:

Without Tiling: The computation in the implementation without tiling is straightforward, with each thread independently performing matrix multiplication for its assigned element. However, the non-coalesced memory accesses can lead to inefficiencies and slower computation.

With Tiling: The tiling implementation improves computation efficiency by minimizing memory access latency. Threads within a block load data into shared memory, which can be reused multiple times during the computation phase. This reduces the overall memory access latency and can accelerate the computation.

Performance:

Without Tiling: The implementation without tiling may suffer from lower performance due to non-coalesced memory accesses and higher global memory latency.

With Tiling: The tiling implementation has the potential to achieve higher performance by reducing memory latency and improving memory access patterns. The use of shared memory and cooperative loading of data can lead to faster computation and improved overall performance.

5- Performance Comparison and Conclusions:

The results indicate that parallelizing the matrix multiplication operation significantly improves the execution time compared to the sequential implementation. Without tiling, the parallel execution achieves a speedup factor of approximately 28.4474, meaning it is about 28 times

faster than the sequential execution. However, the efficiency decreases as the block dimension (number of threads per block) increases. With a block dimension of 64, the efficiency is 44.4%, which decreases to 11.1% and 2.77% for block dimensions of 256 and 1024, respectively.

On the other hand, when tiling is applied, the parallel execution time decreases further to 0.3082 seconds, resulting in a higher speedup factor of approximately 33.6625. This improvement can be attributed to the utilization of shared memory and the reduction of global memory accesses. The efficiency also improves compared to the non-tiling approach, with values of 52.5%, 13.1%, and 3.28% for block dimensions of 64, 256, and 1024, respectively.

The tiling implementation's performance gains can be attributed to its ability to exploit shared memory, which enables threads within a block to access data more efficiently. By loading data into shared memory and reusing it during the computation phase, the tiling implementation minimizes global memory accesses and reduces memory latency. This results in improved memory bandwidth utilization and faster matrix multiplication.

In conclusion, the tiling technique with shared memory demonstrates the importance of optimizing memory access patterns and reducing memory latency for efficient GPU computations. By considering memory access patterns and leveraging shared memory, we can significantly improve performance in parallel computing scenarios. Also, the parallel execution achieves a speedup factor of approximately 28.4474, meaning it is about 28 times faster than the sequential execution. The tiling implementation serves as a valuable optimization strategy for matrix multiplication and similar computations, allowing for faster and more efficient execution on GPUs.