**Lebanese American University**

**Department of Computer Science & Mathematics**

**Parallel Programming**

**CSC 447-Section 12**



**Instructor: Hamdan Abdullatif**

**Hadi Zorkot**

**202106640**

**1.1-Matrix Multiplication without tilling:**

Matrix multiplication, as executed in the given code using the CUDA programming model and without employing tiling, leverages the parallel processing potential of GPUs. To achieve this parallelization, two crucial kernel functions, namely MatrixMulSquareKernel and MatrixMulRectangularKernel, play a pivotal role.

For square matrices, MatrixMulSquareKernel is specifically designed. This kernel adopts a simple strategy, utilizing a singular loop to traverse the width of the matrix. The pseudocode succinctly outlines the essential procedures for this process.

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

In this kernel, every thread within a block is tasked with calculating a solitary element in the resultant matrix. Through the allocation of tasks among threads and the utilization of parallelism, the computational speed is notably enhanced in contrast to a sequential CPU implementation.

For rectangular matrices, MatrixMulRectangularKernel is crafted to adapt its indexing and iteration according to the heights and widths of the input matrices. The pseudocode detailing the operations of this kernel is provided below:

for row in range(heightM):

for col in range(widthN):

Pvalue = 0

for i in range(width):

Pvalue += M[row \* widthM + i] \* N[i\* widthN + col]

P[row \* widthN + col] = Pvalue

In this context, the grid and block dimensions are determined to align with the sizes of the input matrices. The threads within a block cooperate to concurrently calculate the elements of the output matrix. By initiating these kernel functions with carefully chosen grid and block dimensions, the GPU can simultaneously execute multiple threads, efficiently parallelizing the matrix multiplication process. This parallel execution capitalizes on the GPU's computational prowess, resulting in significant performance enhancements compared to a sequential approach on a CPU.

**1.2-matrix multiplication with tilling:**

Undoubtedly, the integration of tiling, also known as thread blocking, stands as a pivotal optimization strategy for parallelizing matrix multiplication. This involves breaking down the input matrices into smaller blocks and independently processing each block using a set of threads within a CUDA block. This approach enhances memory access patterns and minimizes global memory accesses, resulting in a notable improvement in overall performance.

Let's explore how tiling is implemented in the MatrixMulSquareKernel and MatrixMulRectangularKernel:

MatrixMulSquareKernel with Tiling:

Within the MatrixMulSquareKernel, tiling is introduced through the constant TILE\_SIZE, defining the dimensions of each tile or block. The pseudocode for the kernel incorporating tiling is summarized as follows:

size = some\_value

for tileRow in range(0, width, size):

for tileCol in range(0, width, size):

for row in range(tileRow, min(tileRow + size, width)):

for col in range(tileCol, min(tileCol + size, width)):

Pvalue = 0

for i in range(width):

Pvalue += M[row \* width + i] \* N[i \* width + col]

P[row \*width + col] = Pvalue

In this scenario, the matrix is subdivided into smaller tiles, and each CUDA block handles one tile in sequence. This restructuring contributes to better memory access patterns and a reduction in global memory accesses, thereby augmenting parallelism.

MatrixMulRectangularKernel with Tiling:

Likewise, the MatrixMulRectangularKernel integrates tiling through the use of the constant TILE\_SIZE. The pseudocode for performing rectangular matrix multiplication with tiling is outlined as follows:

size = value

for tileRow in range(0, heightM, size):

for tileCol in range(0, widthN, size):

for row in range(tileRow, min(tileRow + size, heightM)):

for col in range(tileCol, min(tileCol + size, widthN)):

Pvalue = 0

for i in range(widthM):

Pvalue += M[row \* widthM + i] \* N[i \* widthN + col]

P[row \* widthN + col] = Pvalue

The technique of tiling is implemented on rectangular matrices, where they are partitioned into smaller tiles, and each CUDA block is assigned the task of handling one tile at a time.

The introduction of tiling optimizes the computation for parallel execution on the GPU. The smaller, autonomous tiles promote data locality, diminish global memory accesses, and foster improved parallelism, resulting in enhanced performance compared to versions without tiling.

MatrixMulSquareKernel**:**

for row in range(heightM):

for col in range(widthN):

Pvalue = 0

for j in range(width / size):

sharedM[ty][tx] = M[row \* width + j \* size + tx]

sharedN[ty][tx] = N[(j \* size + ty) \* width + col]

synchronize\_threads()

for k in range(size):

Pvalue += sharedM[ty][k] \* sharedN[k][tx]

synchronize\_threads()

P[row \* width + col] = Pvalue

The provided pseudocode offers a comprehensive illustration of how shared memory and tiling are employed to enhance matrix multiplication within the CUDA kernel. Let's dissect the essential elements outlined in the pseudocode:

size = svalue # Define the size of the tile

# Declare shared memory arrays for matrices M and N

sharedM = shared memory of size size x size

sharedN = shared memory of size size x size

# Iterate over each tile of the output matrix P

for tileRow in range(0, height, size):

for tileCol in range(0,width, size):

# Initialize shared memory with elements from matrices M and N

for i in range(size):

for j in range(size):

sharedM[i][j] = M[row + i][k]

sharedN[i][j] = N[k][col + j]

# Synchronize threads to ensure all threads have finished loading

synchronize\_threads()

# Perform matrix multiplication on the loaded tile in shared memory

for i in range(size):

for j in range(size):

Pvalue = 0

for k in range(size):

Pvalue += sharedM[i][k] \* sharedN[k][j]

# Synchronize threads to ensure all threads have finished computation

synchronize\_threads()

# Store the computed Pvalue in the output matrix P

P[tileRow + i][tileCol + j] = Pvalue

This pseudocode delineates the sequence of steps involving loading a size x size tile from matrices M and N into shared memory, executing matrix multiplication on the loaded tile, and storing the result in the output matrix P.

Key considerations include:

**Shared Memory Utilization:** The sharedM and sharedN arrays are declared within shared memory, providing a quicker and more efficiently accessed memory space for threads within the same block.

**Tiling Approach:** The outer loops iterate through each tile of the output matrix P. In these loops, the pseudocode loads tiles sized size x size from matrices M and N into shared memory.

**Synchronization:** The synchronize\_threads() function ensures that all threads have completed loading and computation before progressing to the subsequent phase. This synchronization is crucial for the accurate execution of the parallel algorithm. **Matrix Multiplication:** The inner loops execute matrix multiplication on the tile loaded in shared memory. Each thread is responsible for calculating one element of the output matrix P, contributing to the overall parallelism of the computation.

In summary, this pseudocode represents an optimized CUDA kernel for matrix multiplication, leveraging shared memory and tiling to enhance memory access patterns and diminish global memory access latency. This leads to improved parallel performance on the GPU.

**2-Matrix MulRectangular Kernel:**

for Row in range(heightM):

for Col in range(widthN):

Pvalue = 0

for i in range(widthM / size):

sharedM[ty][tx] = M[Row \* widthM + i \* size + tx]

sharedN[ty][tx] = N[(i \* size + ty) \* widthN + Col]

synchronize\_threads()

for k in range(size):

Pvalue += sharedM[ty][k] \* sharedN[k][tx]

synchronize\_threads()

P[row \* widthN + Col] = Pvalue

This pseudocode is akin to the previous one but is tailored for handling rectangular matrices with varying dimensions (widthM, heightM, widthN, heightN). The outer loops traverse each element of the output matrix P, denoted by row and col. The inner loops implement tiling by loading tiles sized size x size from matrices M and N into shared memory. The notable difference lies in how we compute indices for loading elements from matrices M and N into shared memory, utilizing the provided widthM, heightM, widthN, and heightN values. The subsequent processes, including computation and storing results in the output matrix, mirror those of the previous kernel. Shared memory usage in both cases enhances memory access patterns, reducing global memory access latency and thereby improving performance through data reuse maximization and global reduction.

To optimize memory access patterns and diminish global memory latency, we introduced the concept of tiling and harnessed shared memory. In shared memory, we declared sharedM and sharedN arrays of size size x size. Each thread in a block possessed its local copy of these arrays stored in shared memory. Loading a size x size tile from matrices M and N into shared memory involved each thread loading one element into its corresponding shared memory location. Employing tiling and shared memory provided enhanced data locality and diminished global memory accesses. Threads in a block collaboratively loaded data into shared memory, efficiently reusing it during computation, significantly improving performance by minimizing memory access latency.

After loading the tile into shared memory, thread synchronization using syncthreads() ensured all threads completed loading before advancing to the computation phase. In the inner loop, each thread executed matrix multiplication by iterating over elements of the loaded tile in shared memory, multiplying corresponding elements of sharedM and sharedN, and accumulating the result in a local variable, Pvalue.

Subsequently, computed Pvalue was stored in the output matrix P at the appropriate location. Parallelization was achieved by assigning different threads to compute different elements of the output matrix. The use of shared memory and tiling not only improved memory access patterns but also reduced memory latency, resulting in accelerated computation.

**3-**

Sequential time: 10.3748s Parallel time (no tiling) = 0.3647s, Speedup factor = 28.4474

Efficiency

Block Dim (thread number) = 64: Time execution = 0.3647, Efficiency = 64.7 Block Dim (thread number) = 256: Time execution = 0.3624, Efficiency = 31.5 Block Dim (thread number) = 1024: Time execution = 0.3485, Efficiency = 12.2

Parallel time (tiling) = 0.3082s, Speedup factor = 33.6625, Tiles = 16

Efficiency:

Block Dim (thread number) = 64: Time execution = 0.3118, Efficiency = 44.7 Block Dim (thread number) = 256: Time execution = 0.3082, Efficiency = 29.05 Block Dim (thread number) = 1024: Time execution = 0.2982, Efficiency = 5.61

**4- Comparison of the two implementations:**

Here are some key takeaways from comparison and conclusions:

**Memory Access Patterns:**

**Without Tiling:** Inefficient utilization of memory bandwidth due to non-coalesced memory accesses.

**With Tiling:** Collaborative loading of a tile into shared memory enhances memory access patterns, resulting in coalesced memory accesses and improved bandwidth utilization.

**Memory Access Efficiency:**

**Without Tiling:** Elevated global memory latency attributed to non-coalesced accesses.

**With Tiling:** Reduction in global memory latency through shared memory utilization, leading to enhanced efficiency.

**Computation Efficiency:**

**Without Tiling:** Straightforward computation marred by inefficiencies from non-coalesced memory accesses.

**With Tiling:** Enhanced computation efficiency facilitated by shared memory and cooperative loading, contributing to an overall reduction in memory access latency.

**Performance:**

**Without Tiling:** Significant speedup in comparison to sequential execution (28.4474x), albeit with diminishing efficiency for larger block dimensions.

**With Tiling:** Further performance enhancement, achieving a higher speedup factor (33.6625x) and maintaining superior efficiency across varying block dimensions.