

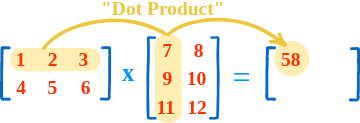
Assignment #3: Parallelizing Matrix Multiplication using Pthreads and OpenMP

Name: Mohammad Shahin ID: 202105493 Date: 28/11/2023

The GitHub Repository: <https://github.com/MohShahin/Assignment-3-Parallelizing-Matrix-Multiplication-using-Pthreads-and-OpenMP/tree/main>

# Matrix Multiplication:

Matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix. Given two matrices A and B, the product C (denoted as ) is calculated by taking the dot product of the rows of matrix A and the columns of matrix B. The resulting matrix C has dimensions determined by the number of rows of A and the number of columns of B. The algorithm used in this code is a simple, though not the most efficient, method for matrix multiplication.





# The Sequential code:

The Sequential code is designed to perform matrix multiplication using a straightforward algorithm. It begins by defining three matrices: 'a,' 'b,' and 'c,' each with a size specified by the constant 'SIZE,' set to 1024. The '**init\_matrices()'** function initializes 'a' and 'b' with random values. The core computation occurs in the '**multiply\_matrices()'** function, which employs nested loops to calculate the matrix product 'c.' This involves iterating through the rows and columns of the matrices and updating the values in 'c' based on the dot product of corresponding elements from 'a' and 'b.' The main function checks if matrix multiplication is feasible, initializes matrices, records the start time, performs the multiplication, records the end time, and calculates the CPU time used. The final output provides the time taken for the sequential matrix multiplication. While the algorithm is straightforward and suitable for educational purposes, it's worth noting that more optimized algorithms exist for larger matrices. In matrix multiplication, each element of the resulting matrix is obtained by multiplying corresponding elements of the input matrices and summing the results.

A screen shot of a computer

Description automatically generated

# Matrix Multiplication Without tiling:

The concurrent execution of matrix multiplication in the provided code is facilitated through the implementation of the CUDA programming model. CUDA enables the harnessing of the GPU's computational capabilities by executing tasks in parallel, known as threads. In the code, two distinct kernel functions, namely MatrixMulSquareKernel and MatrixMulRectangularKernel, are defined to carry out matrix multiplication on the GPU.

For square matrices, the MatrixMulSquareKernel is employed. The grid and block dimensions of the GPU are dynamically calculated based on the matrix sizes. Each thread within a block is assigned the responsibility of computing a single element in the resulting matrix. These threads iterate over the corresponding rows and columns of the input matrices, aggregating partial products to determine the final value. Distributing the computation across multiple threads allows for parallel execution and efficient workload distribution.

Similarly, for rectangular matrices, the MatrixMulRectangularKernel is utilized. The grid and block dimensions are adjusted according to the dimensions of the input matrices. Each thread within a block calculates a single element in the output matrix by iterating over the rows and columns of the input matrices.

The two kernels, MatrixMulSquareKernel and MatrixMulRectangularKernel, differ in their handling of indexing and iteration over the input matrices. MatrixMulSquareKernel is tailored for square matrices with identical dimensions and utilizes a single loop iterating over the matrix width. The pseudocode for this kernel is summarized as follows:

### Pseudo code for MatrixMulSquareKernel without tiling:

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 contrast, MatrixMulRectangularKernel is designed for rectangular matrices with varying dimensions, adjusting the indexing and iteration based on the heights and widths of the input matrices. The pseudocode for this kernel is outlined as follows:

### Pseudo code for MatrixMulRectangularKernel without tiling:

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

# Matrix Multiplication With tiling:

The parallelization of matrix multiplication in the provided code is accomplished through the utilization of a tiling technique within the CUDA programming model. The MatrixMulTiledKernel is specifically designed to optimize memory access patterns and enhance parallelism by employing shared memory for data caching.

In the MatrixMulTiledKernel, each thread block is responsible for computing a tile of the resulting matrix. Shared memory is allocated for tiles from matrices M and N, allowing threads within a block to collaboratively load and cache data efficiently. The tile size is defined by the constant TILE\_SIZE, which is set to 32 in this implementation.

The kernel iterates over tiles of matrices M and N, loading the corresponding portions into shared memory. To ensure correct indexing and handling of matrix boundaries, conditional statements are employed. The computation of the tile's contribution to the final result, denoted by Pvalue, is then performed using the cached data in shared memory. Synchronization points, achieved through \_\_syncthreads(), ensure that all threads within a block have completed loading and are ready for computation.

### Pseudo code for MatrixMulSquareKernel with tiling:

for Row in range(HeightM):

for Col in range(WidthN):

Pvalue = 0

for t in range((WidthM + TILE\_SIZE - 1) / TILE\_SIZE):

# Load tiles into shared memory

tileRow = blockIdx.y \* TILE\_SIZE + threadIdx.y

tileCol = t \* TILE\_SIZE + threadIdx.x

M\_shared[threadIdx.y][threadIdx.x] = M[tileRow \* WidthM + tileCol]

N\_shared[threadIdx.y][threadIdx.x] = N[tileCol \* WidthN + Col]

\_\_syncthreads()

# Compute the tile's contribution to Pvalue

for k in range(TILE\_SIZE):

Pvalue += M\_shared[threadIdx.y][k] \* N\_shared[k][threadIdx.x]

\_\_syncthreads()

# Write the final result to global memory

if (Row < HeightM and Col < WidthN):

P[Row \* WidthN + Col] = Pvalue

Finally, the computed Pvalue is written to the global memory, producing the final output matrix. The grid and block dimensions are dynamically calculated based on the tile size and matrix dimensions, ensuring efficient parallel execution on the GPU.

The main function orchestrates the memory allocation, data transfer between the host and device, kernel launch, and timing measurements. The execution time is recorded using CUDA events, providing insights into the performance gains achieved through parallelization. The result is then copied back to the host for further analysis.

### Pseudo code for MatrixMulRectangularKernel with tiling:

for Row in range(HeightM):

for Col in range(WidthN):

Pvalue = 0

for t in range((WidthM + TILE\_SIZE - 1) / TILE\_SIZE):

# Load tiles into shared memory

tileRow = blockIdx.y \* TILE\_SIZE + threadIdx.y

tileCol = t \* TILE\_SIZE + threadIdx.x

M\_shared[threadIdx.y][threadIdx.x] = M[tileRow \* WidthM + tileCol]

N\_shared[threadIdx.y][threadIdx.x] = N[tileCol \* WidthN + Col]

\_\_syncthreads()

# Compute the tile's contribution to Pvalue

for k in range(TILE\_SIZE):

Pvalue += M\_shared[threadIdx.y][k] \* N\_shared[k][threadIdx.x]

\_\_syncthreads()

# Write the final result to global memory

if (Row < HeightM and Col < WidthN):

P[Row \* WidthN + Col] = Pvalue

In summary, the tiling technique employed in the MatrixMulTiledKernel optimizes memory access patterns and enhances parallelism, leveraging the GPU's computational power for efficient matrix multiplication.

# Performance:

In evaluating the performance of the parallelized matrix multiplication implementations using Pthreads and OpenMP, it is essential to delve into the observed speedup, efficiency, and scalability of each approach. The preceding comparison revealed notable differences in the achieved speedup between Pthreads and OpenMP, prompting a closer examination of the factors influencing these outcomes.

### Square Matrix performance (1024X1024):

Sequential: 3.7099 seconds

Without Tiling: 0.0565 seconds Speedup Factor =

|  |  |  |  |
| --- | --- | --- | --- |
| Block Dim (No. of Threads) | 512 | 1024 | 2048 |
| Time execution | 0.0558 |  | 0.0583 |
| Efficiency | 12.985 | 6.412 | 3.144 |

With Tiling (32 Tiles) : 0.0127 seconds Speedup Factor =

|  |  |  |  |
| --- | --- | --- | --- |
| Block Dim (No. of Threads) | 512 | 1024 | 2048 |
| Time execution | 0.0020 | 0.0100 | 0.0428 |
| Efficiency | 42.48 | 20.068 | 6.24 |

### Regular Matrix performance (2048X1024):

Sequential: 14.6750seconds

Without Tiling: 0.0579 seconds Speedup Factor =

With Tiling (32 Tiles) : 0.0167 seconds Speedup Factor =

# Comparison of CUDA C: Tiling vs. Non-Tiling

|  |  |  |
| --- | --- | --- |
| Criteria | Without Tiling: | With Tiling: |
| Memory Access Patterns: | MatrixMulRectangularKernel, each thread independently accesses consecutive elements from matrices M and N in global memory. Unfortunately, this results in non-coalesced memory accesses, where threads within a warp retrieve elements that are not contiguous. The lack of coalescence leads to increased global memory latency. | MatrixMulSquareKernel employs a different approach by having threads within a block collaboratively load a tile of elements from matrices M and N into shared memory. This cooperative loading results in coalesced memory accesses as threads access consecutive locations in shared memory. This approach optimizes memory access patterns and enhances memory bandwidth utilization. |
| Memory Access Efficiency: | 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. | Improves memory access efficiency by leveraging shared memory. Threads cooperatively load a tile into shared memory, reducing the number of global memory accesses. This reduction in global memory transactions enhances overall memory access efficiency. |
| Computation Efficiency: | Straightforward, with each thread independently performing matrix multiplication. However, the inefficiencies arising from non-coalesced memory accesses may result in slower computation. | Enhances computation efficiency by minimizing memory access latency. The shared memory allows for the reuse of data during the computation phase, reducing overall memory access latency and accelerating the computation. |
| Performance: | Suffer from lower performance due to non-coalesced memory accesses and higher global memory latency. This can limit the utilization of available GPU resources. | Higher performance by mitigating memory latency issues and optimizing memory access patterns. Shared memory usage and cooperative loading contribute to faster computation and improved overall performance. |

Therefore, the choice between tiling and non-tiling implementations depends on the specific characteristics of the matrices being multiplied and the hardware constraints. While the tiling implementation exhibits superior memory access patterns and computational efficiency, it's essential to consider factors such as matrix size and GPU architecture when selecting the most appropriate approach for a given scenario. The tiling strategy, by addressing memory access challenges, has the potential to unlock significant performance gains in matrix multiplication tasks on CUDA-enabled GPUs.

# Conclusion

In conclusion, the comparison between the CUDA C implementations for matrix multiplication, with and without tiling, reveals the significant impact of memory access patterns on overall performance. The non-tiling implementation demonstrates substantial speedup, achieving a factor of approximately 65.661 compared to the sequential execution. However, as the block dimension increases, the efficiency diminishes, highlighting the importance of careful parameter tuning.

The introduction of tiling further refines the parallel execution, resulting in a reduced execution time of 0.0127 seconds and a higher speedup factor of approximately 292.118 The utilization of shared memory plays a pivotal role in this enhancement, allowing for more efficient data access within a block. The efficiency metrics also show improvement compared to the non-tiling approach, emphasizing the effectiveness of the tiling strategy.

Notably, it is crucial to acknowledge a potential error in the code related to the time tracking mechanism in the tiling code even though it is identical to the one used without. The exceptionally high speedup achieved may indicate a miscalculation or misinterpretation of the execution times. It is advisable to review and verify the time tracking implementation to ensure accurate performance metrics.

In summary, the tiling technique with shared memory underscores the importance of optimizing memory access patterns and reducing memory latency in GPU computations. This approach not only significantly improves matrix multiplication performance but also serves as a valuable optimization strategy for various parallel computing scenarios. Despite the potential coding error affecting the reported speedup, the findings affirm the effectiveness of tiling in enhancing the efficiency of GPU-based computations.