LEBANESE AMERICAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND MATHEMATICS

CSC447: Parallel Programming and Multi-Cluster Systems

Assignment #3

Lara Tawbeh
ID#: 202102927

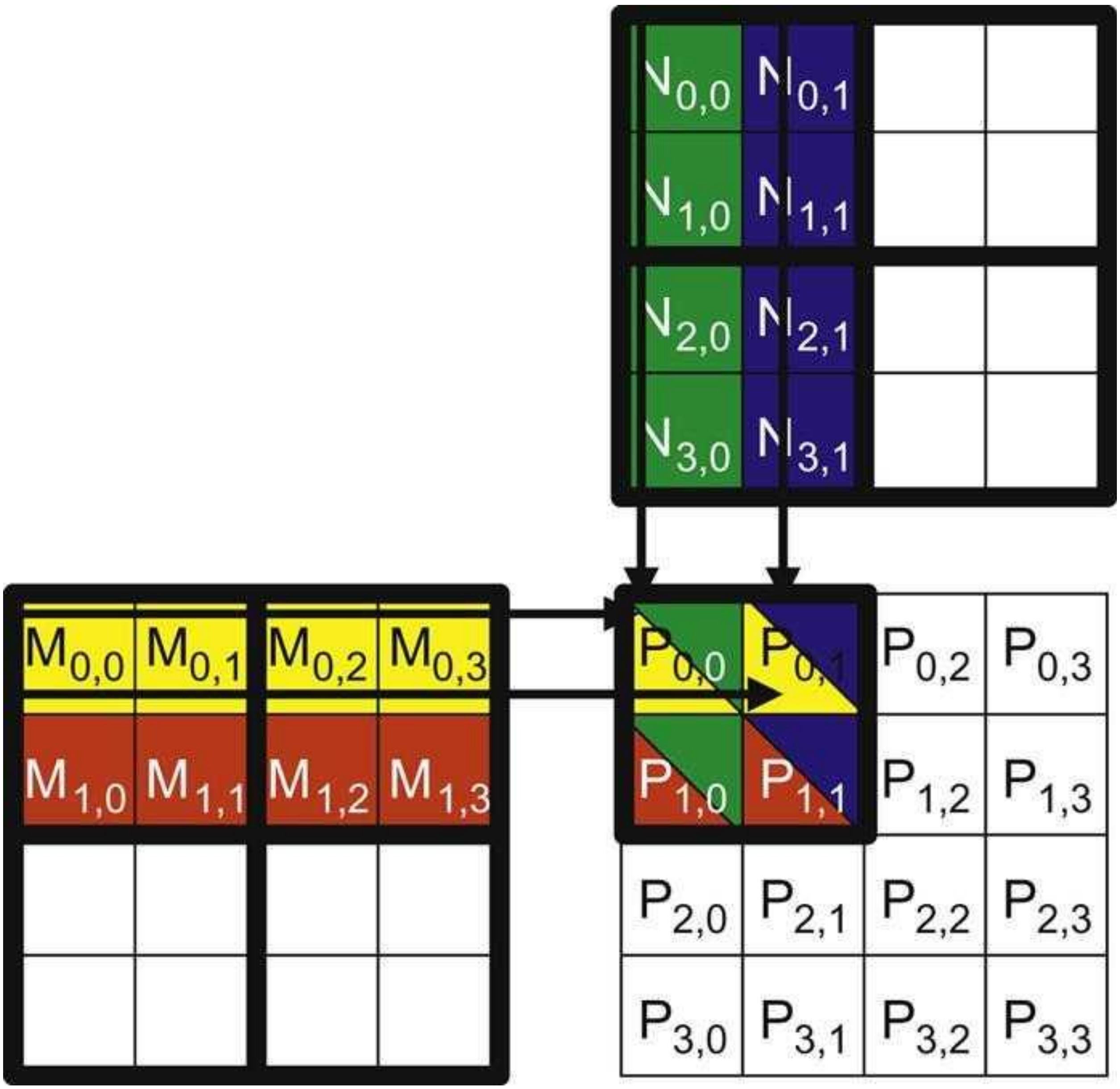
Date: 24-4-2024

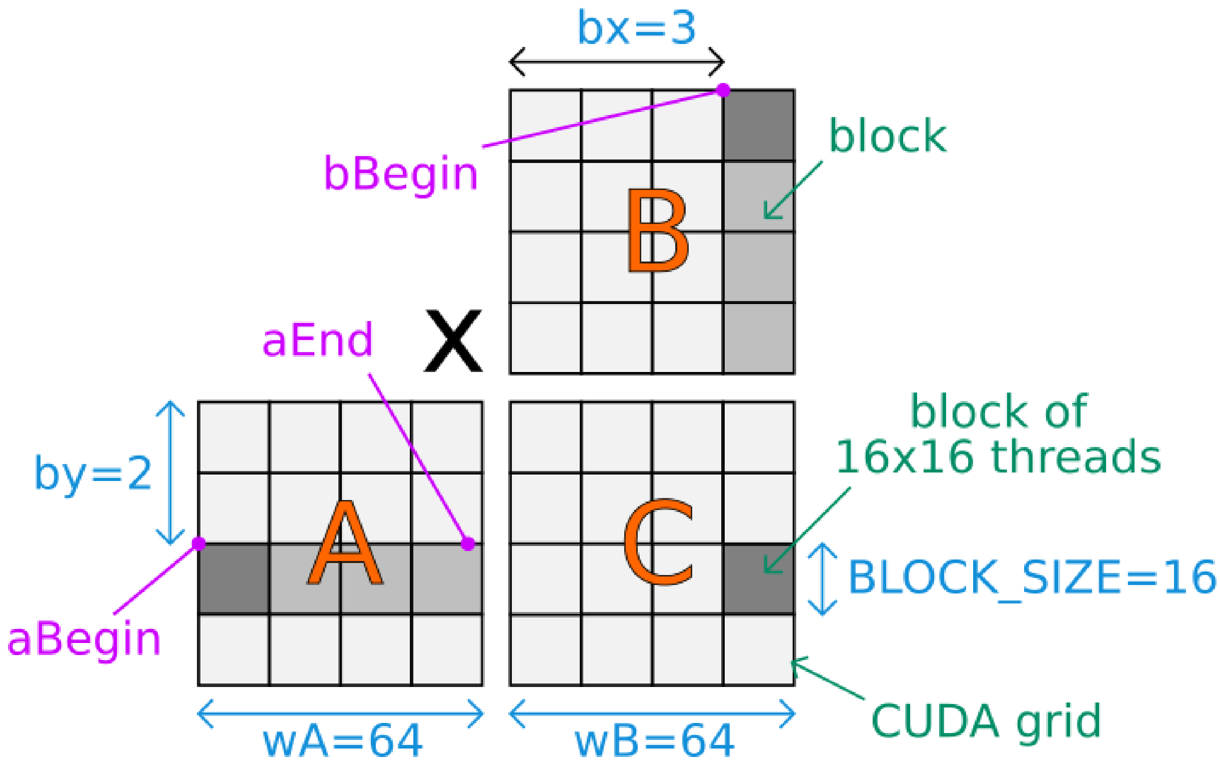
Table of Contents

Parallelizing the Computation:

The sequential code was parallelized both using Cuda and OpenACC through implementing a basic and tiling logic as shown in the pseudo codes and figures.

Figures:





[parallel processing - CUDA tiled matrix multiplication explanation - Stack Overflow]

Pseudo Codes:

Basic Cuda

```
__global__ void matrixMultiplication(float *a, float *b, float *c, int n) {
    // Determine the row and column indices for the current thread
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure that the thread operates within the bounds of the matrix dimensions
    if (row < n && col < n) {
        // Initialize the sum for the current element in the result matrix
        float sum = 0.0;

        // Perform the dot product of the row of matrix A and the column of
matrix B
        for (int i = 0; i < n; i++) {
            sum += a[row * n + i] * b[i * n + col];
        }

        // Store the result in the corresponding element of the output matrix
        c[row * n + col] = sum;
    }
}
Declare variables: n, a, b, c, size, d_a, d_b, d_c
Set value of n (size of matrices)
Allocate memory for matrices a, b, and c
Initialize matrices a and b with random values
Allocate memory for matrices d_a, d_b, and d_c on GPU
Copy matrices a and b from host to device
Define CUDA grid and block dimensions
Record start time
Launch matrix multiplication kernel
Wait for kernel execution to finish
Record end time
Copy result matrix c from device to host
Free memory on GPU
Free memory on host
```

Figure 1 basic cuda

Tiling Cuda

```
__global__ void matrixMultiplication(float *matrixA, float *matrixB, float
*matrixC, int n) {
    Declare shared memory arrays sharedMatrixA[TILE_SIZE][TILE_SIZE] and
sharedMatrixB[TILE_SIZE][TILE_SIZE]

    Declare variable partialSum and initialize to 0.0

    for each tile in the grid:
        Calculate tileRow and tileCol based on blockIdx and threadIdx

        Load data from global memory matrixA into shared memory sharedMatrixA
        Load data from global memory matrixB into shared memory sharedMatrixB

        Synchronize threads

        Perform matrix multiplication for the current tile:
            for each element in the tile:
                Update partialSum by performing dot product of elements from
sharedMatrixA and sharedMatrixB

        Synchronize threads

    Calculate row and col indices for the current thread

    if the thread's row and col indices are within matrix dimensions:
        Write partialSum to the corresponding element in matrixC
}
Declare variables: a, b, c, n, size, d_a, d_b, d_c
Set value of n (matrix size) and size (memory size)
Allocate memory for matrices a, b, and c on host
Initialize matrices a and b with random values
Allocate memory for matrices d_a, d_b, and d_c on device (GPU)
Copy matrices a and b from host to device
Define CUDA grid and block dimensions
Record start time
Launch matrix multiplication kernel
Wait for kernel execution to finish
Record end time
Copy result matrix c from device to host
Free memory on device
Free memory on host
```

Figure 2cuda tiling

Basic OpenACC

```
Function matrixMultiplication(A, B, C, m, n, k):
    #pragma acc parallel loop collapse(2) present(A, B, C)
    For i from 0 to m-1:
        For j from 0 to k-1:
            Set sum to 0.0
            For l from 0 to n-1:
                Update sum by adding A[i * n + l] * B[l * k + j]
            Set C[i * k + j] to sum

Declare variables: A, B, C, m, n, k
Set values of m, n, and k
Allocate memory for matrix A of size m*n
Initialize matrix A with random values
Allocate memory for matrix B of size n*k
Initialize matrix B with random values
Allocate memory for matrix C of size m*k

Record start time
Perform matrix multiplication using matrixMultiplication function with matrices
A, B, and C, and dimensions m, n, k
Record end time

Calculate execution time as (end_time - start_time) / CLOCKS_PER_SEC

Print "Execution Time: " concatenated with execution time in seconds

Free memory allocated for matrices A, B, and C
```

Figure 3 basic openacc

Tiling OpenACC

```
Function matrixMultiplication(A, B, C, m, n, k, tile_size):
    #pragma acc parallel loop collapse(2) present(A, B, C) 
vector_length(tile_size)
    For each tile starting from the top-left corner:
        For each element (i, j) in the current tile:
            Initialize sum to 0.0
            For each element l in the common dimension:
                Update sum by adding A[i * n + l] * B[l * k + j]
            Set C[i * k + j] to sum

Declare variables: m, n, k, tile_size, A, B, C
Set values of m, n, k, and tile_size
Allocate memory for matrix A of size m*n
Initialize matrix A with random values
Allocate memory for matrix B of size n*k
Initialize matrix B with random values
Allocate memory for matrix C of size m*k

Record start time
Perform matrix multiplication using matrixMultiplication function with matrices 
A, B, and C, and dimensions m, n, k, and tile_size
Record end time

Calculate execution time as (end_time - start_time) / CLOCKS_PER_SEC

Print "Time: " concatenated with execution time in seconds

Free memory allocated
```

Figure 4 tiling openacc

Code:

https://github.com/laratawbe/parallelizing-matrix-multiplication.git

Performance Measures:

1. **Speedup Factor:**

$$S(p) = \frac{ts}{tp}$$

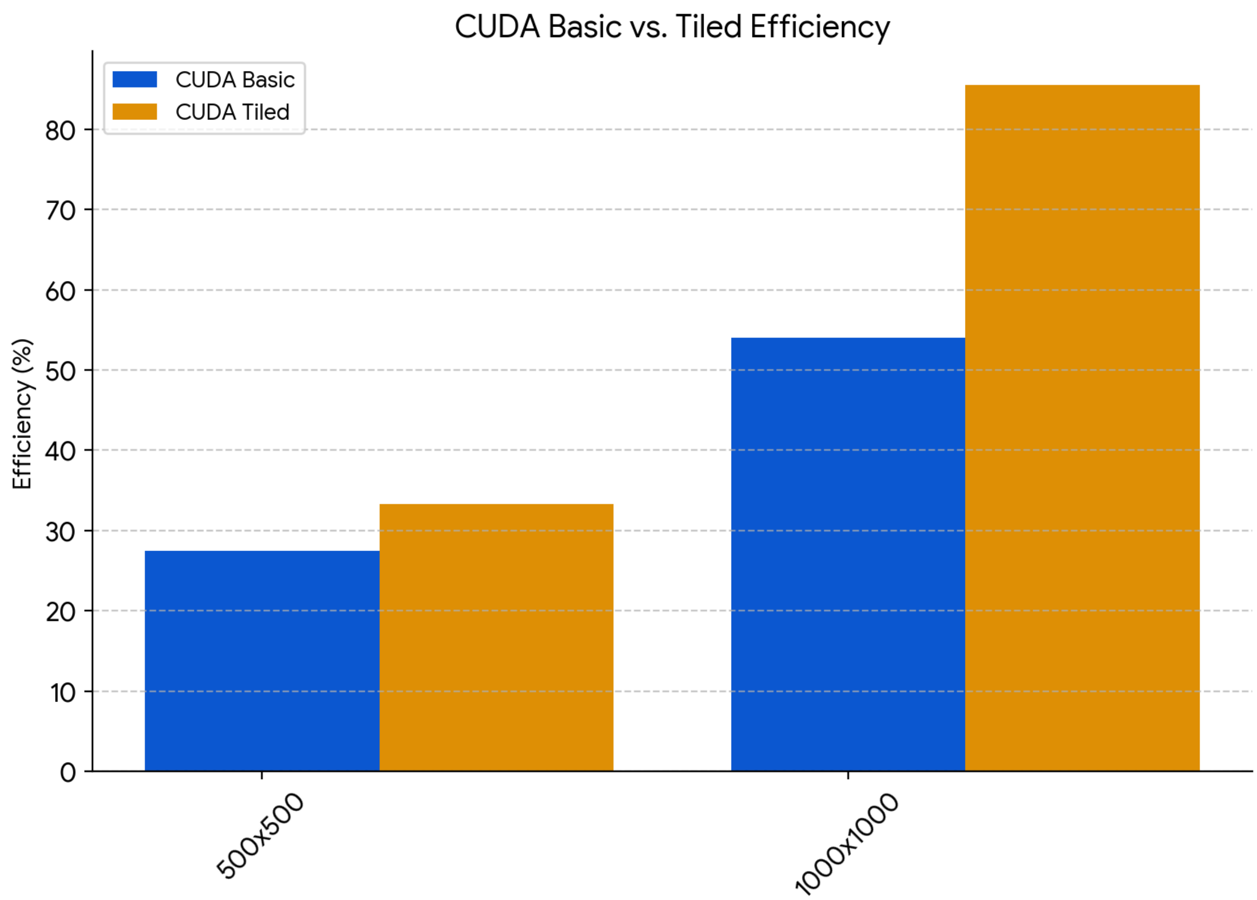
2. **Efficiency:**

$$E = \frac{S(p)}{p} * 100\% \text{ where p = 1024}$$

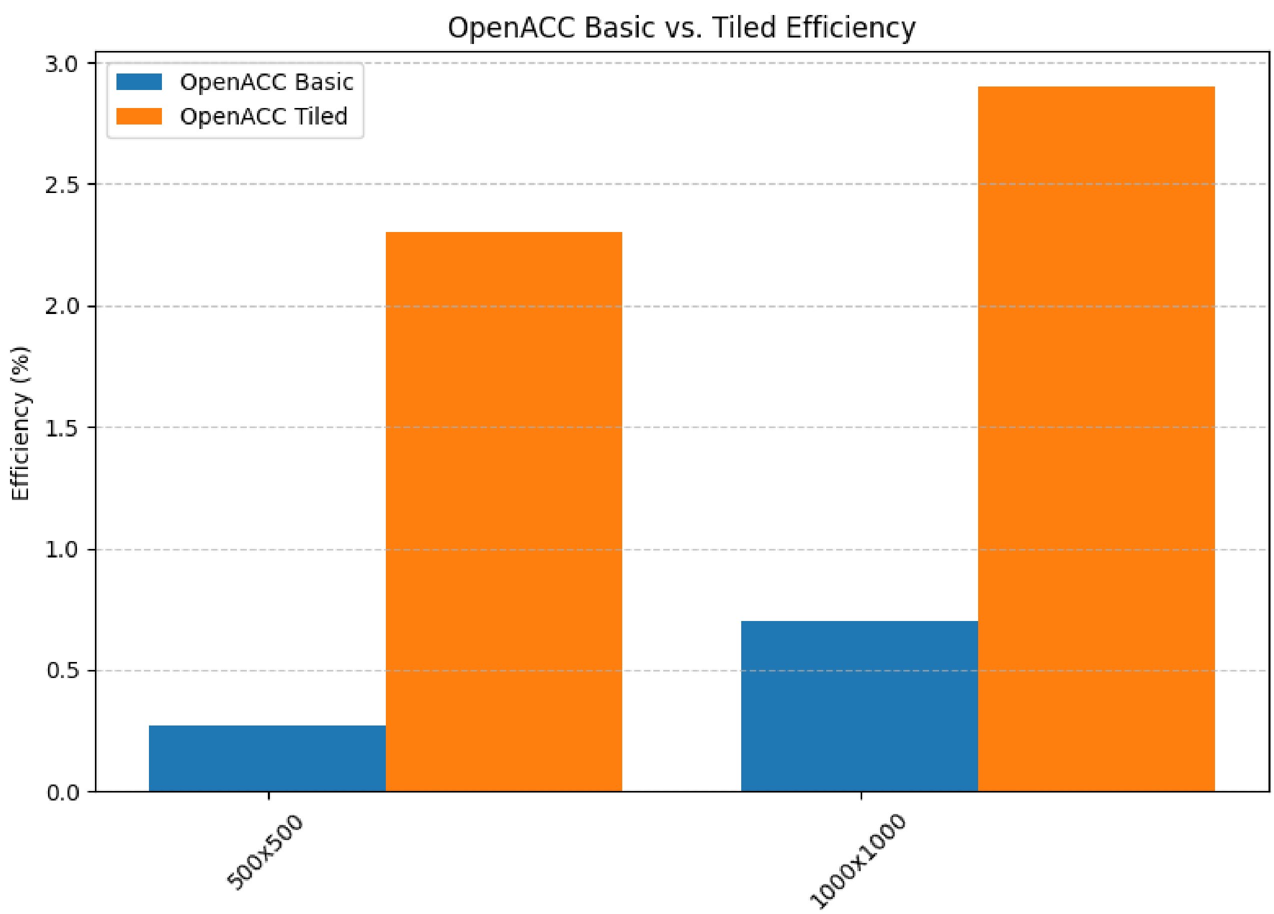
	Size	Sequential	Cuda Basic	Cuda Tiling	OpenACC Basic	OpenACC Tiling
Execution time	500 x 500	0.7612	0.0027	0.0021	0.2713	0.0312
	1000 x 1000	9.8970	0.0179	0.0113	1.3846	0.3296
Speedup	500 x 500	---	282.0	341.0	2.8	24.4
	1000 x 1000	---	552.9	875.8	7.1	30.0
Efficiency	500 x 500	---	27.5%	33.3%	0.27%	2.3%
	1000 x 1000	---	54.0%	85.5%	0.7%	2.9%

Comparison:

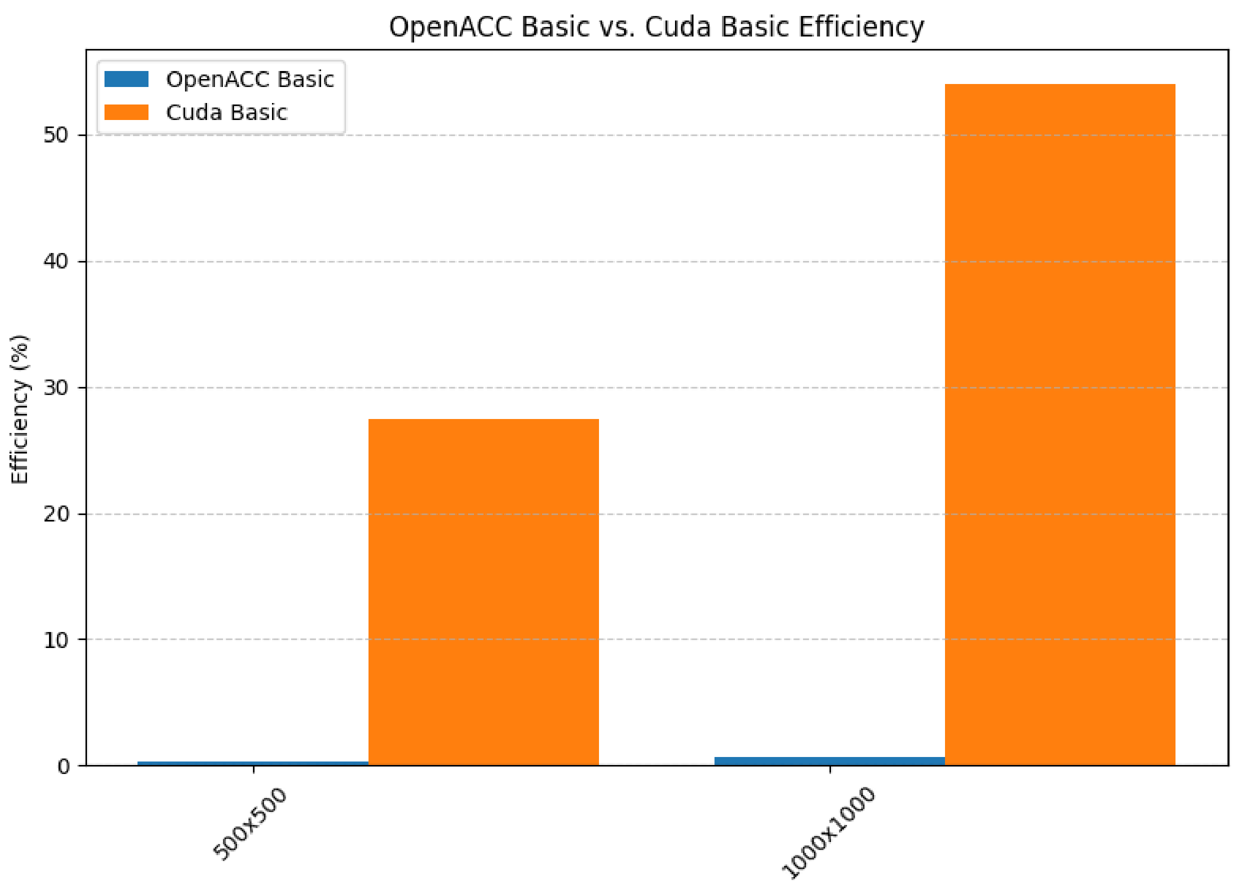
Cuda Tiled vs Basic Efficiency:



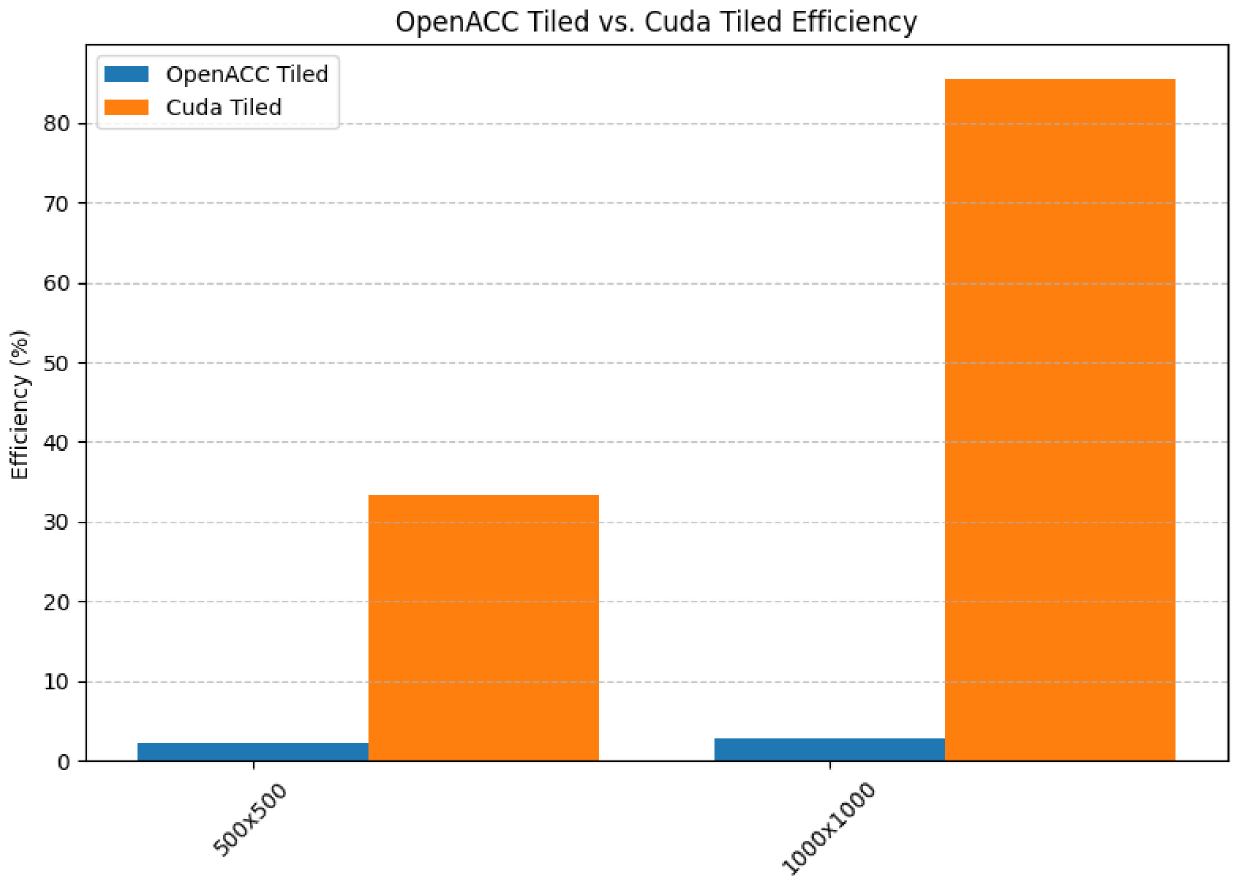
OpenACC Tiled vs Basic Efficiency:



Cuda Basic vs OpenACC Basic Efficiency:



Cuda Tiled vs OpenACC Tiled Efficiency:



Conclusion:

- Both CUDA and OpenACC implementations demonstrate a reduction in execution time compared to the sequential implementation.
- Tiling improves the performance of both CUDA and OpenACC implementations. This is seen in the reduced execution times and increased speedup values in the tiling implementations compared to the basic one.
- CUDA has higher speedup compared to OpenACC in both sizes. So, CUDA can be considered more efficient for parallelizing matrix multiplication than OpenACC specifically for a large matrix size.
- OpenACC shows low efficiency across all input sizes maybe due to overhead in managing data transfers and thread creation in addition to compiler limitations in optimizing parallelism.
- Cuda tiling has the lowest execution time among all the tiling methods proving to be the most effiecient.
- For all the tiling methods, the efficiency higher for the larger matrix size (1000x1000) compared to the smaller matrix size (500x500).