Strassen's Matrix Multiplication Acceleration using OPENMP, MPI and CUDA

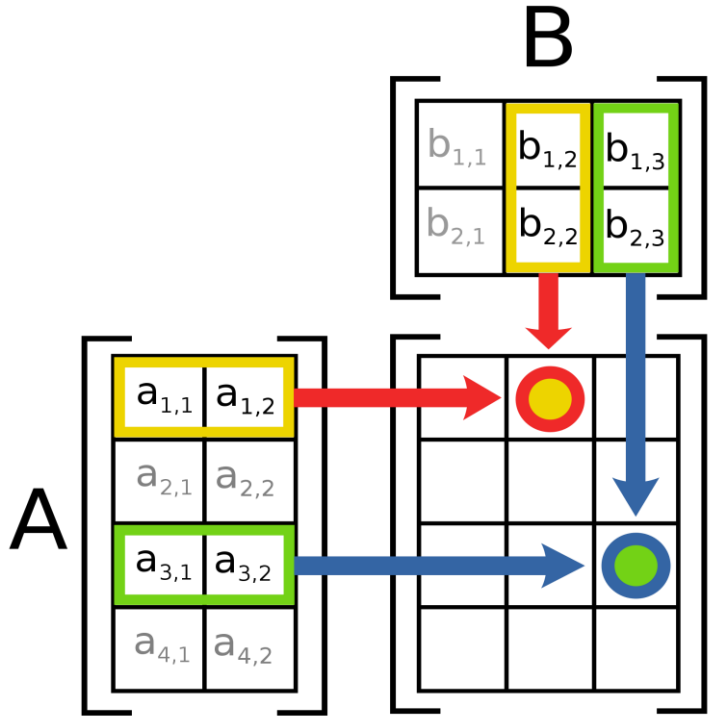
Williams Okeke

Holdings Ogon

Joshua Udobang

Standard Matrix Multiplication

- A matrix is a rectangular array of numbers. It can be represented as a two-dimensional grid, with rows and columns.
- Given matrix A of dimension m × n be given and B of dimension n × p, we obtain the matrix C = AB of dimension m × p.



- The entries of C are of the form

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Time Complexity = $O(mnp)$ = $\boldsymbol{O(n^3)}$ (when m = n = p)

WE CAN DO BETTER!

Strassen's Matrix Multiplication Algorithm

- Reduces the number of recursive calls(from 8 to 7)
- Reduces the number of multiplications carried out at each recursive level(computes more additions)
- Better cache locality and parallelism structure
- Faster time complexity

Input: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

1: **if** $n = 1$ **then**
2: $C = A \cdot B$
3: **else**
4: $M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
5: $M_2 = (A_{21} + A_{22}) \cdot B_{11}$
6: $M_3 = A_{11} \cdot (B_{12} - B_{22})$
7: $M_4 = A_{22} \cdot (B_{21} - B_{11})$
8: $M_5 = (A_{11} + A_{12}) \cdot B_{22}$
9: $M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
10: $M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
11: $C_{11} = M_1 + M_4 - M_5 + M_7$
12: $C_{12} = M_3 + M_5$
13: $C_{21} = M_2 + M_4$
14: $C_{22} = M_1 - M_2 + M_3 + M_6$

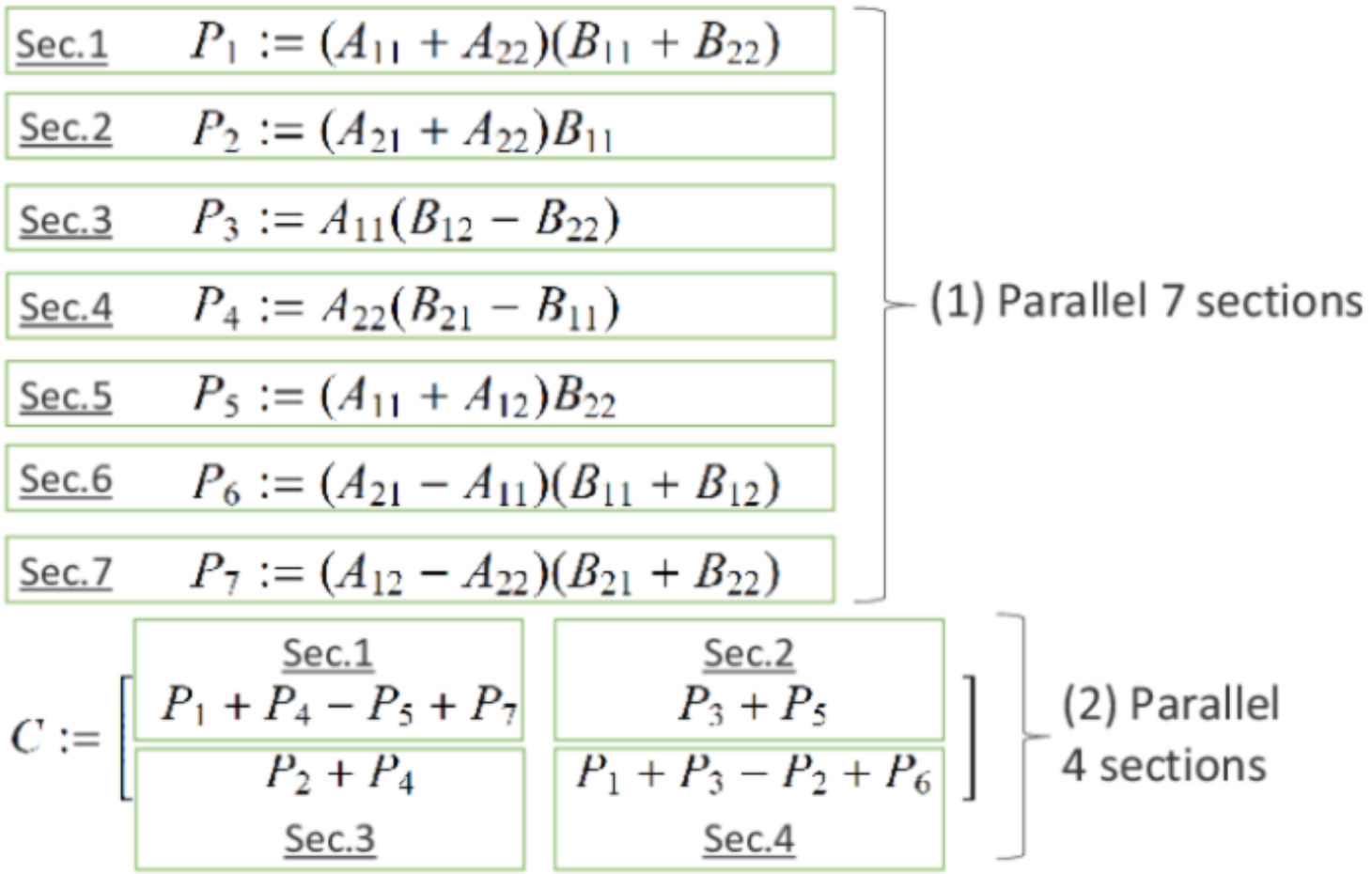
Output: $A \cdot B = C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

Analysis

$$T(n) = \begin{cases} b & n \le 2 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & n > 2 \end{cases}$$

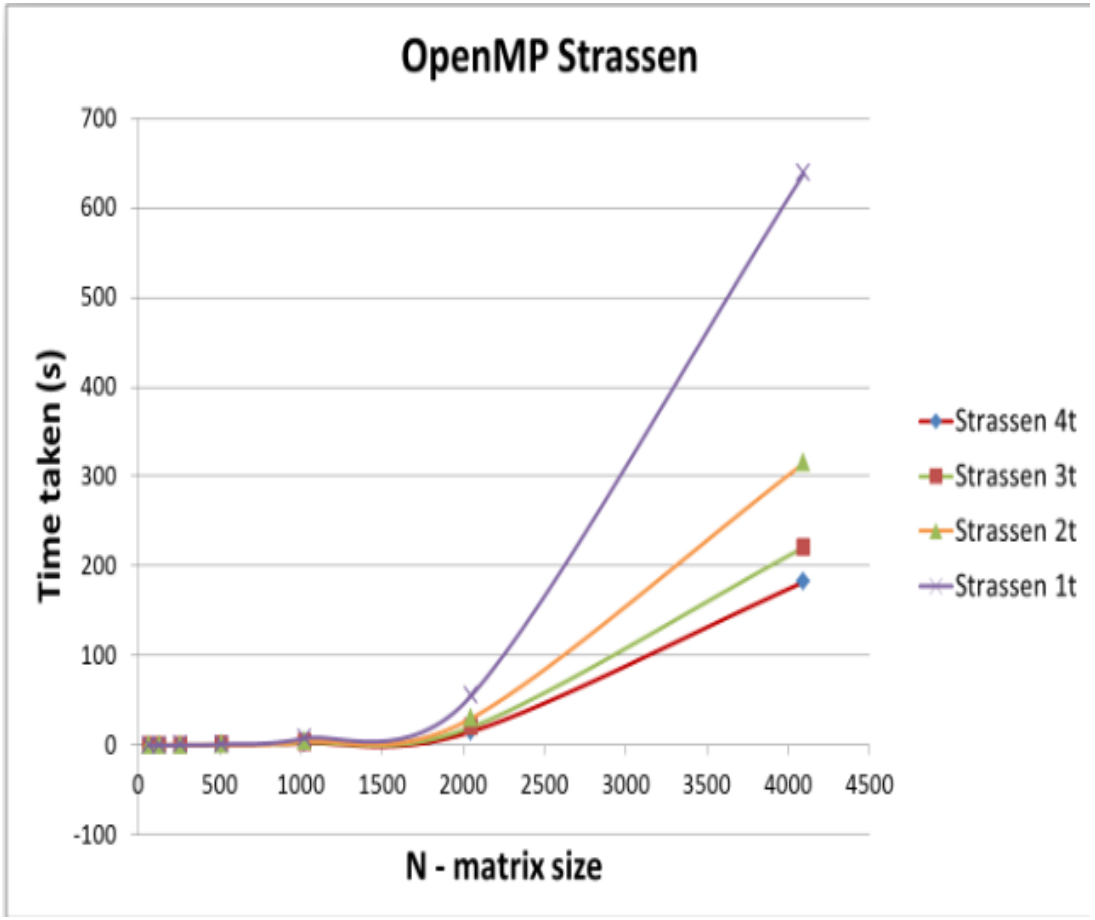
$$O(n^{\log_2 7}) \approx O(n^{2.81})$$

Intuition


Sec.1 $P_1 := (A_{11} + A_{22})(B_{11} + B_{22})$
Sec.2 $P_2 := (A_{21} + A_{22})B_{11}$
Sec.3 $P_3 := A_{11}(B_{12} - B_{22})$
Sec.4 $P_4 := A_{22}(B_{21} - B_{11})$
Sec.5 $P_5 := (A_{11} + A_{12})B_{22}$
Sec.6 $P_6 := (A_{21} - A_{11})(B_{11} + B_{12})$
Sec.7 $P_7 := (A_{12} - A_{22})(B_{21} + B_{22})$
(1) Parallel 7 sections
$C :=$
Sec.1 $P_1 + P_4 - P_5 + P_7$
Sec.2 $P_3 + P_5$
$P_2 + P_4$
Sec.3
$P_1 + P_3 - P_2 + P_6$
Sec.4
(2) Parallel 4 sections

OpenMP Implementation

- All iterative processes are parallelized
- Used 1, 2, 3 and 4 threads
- Minimum granularity set at 32x32 Matrix size.
- Highly memory intensive due to the numerous submatrices created in the recursive process.



Code Snippets

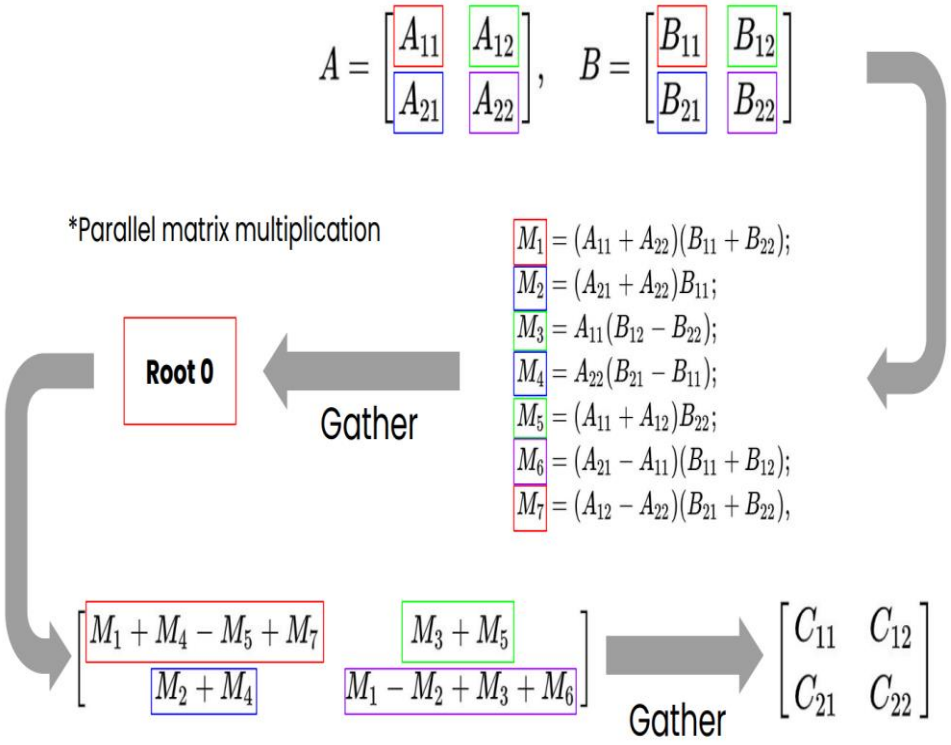
```
#pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            prod[i][j] = 0;
            for (int k = 0; k < n; k++)
            {
                prod[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }

    return prod;
}
```

```
    #pragma omp taskwait

    freeMatrix(m, s1);
    freeMatrix(m, s2);
    freeMatrix(m, s3);
    freeMatrix(m, s4);
    freeMatrix(m, s5);
    freeMatrix(m, s6);
    freeMatrix(m, s7);

    int** prod = combineMatrices(m, c11, c12, c21, c22);

    freeMatrix(m, c11);
    freeMatrix(m, c12);
    freeMatrix(m, c21);
    freeMatrix(m, c22);

    return prod;
}

bool check(int n, int** prod1, int** prod2)
{
    for (int i = 0; i < n; i++)
    {
```

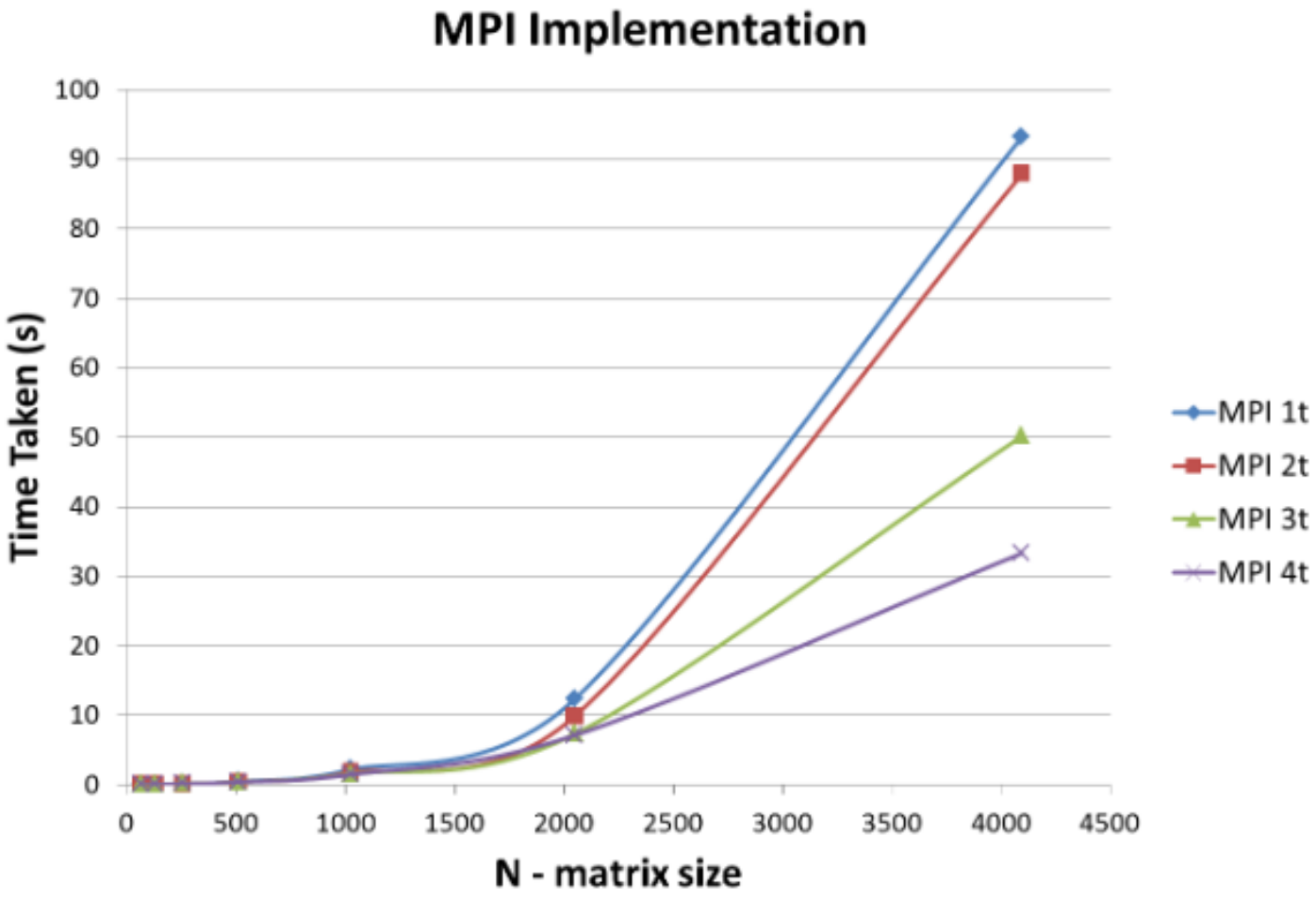
```
#pragma omp task shared(s1)
{
    int** bds = addMatrices(m, b, d, false);
    int** gha = addMatrices(m, g, h, true);
    s1 = strassen(m, bds, gha);
    freeMatrix(m, bds);
    freeMatrix(m, gha);
}

int** s2;
#pragma omp task shared(s2)
{
    int** ada = addMatrices(m, a, d, true);
    int** eha = addMatrices(m, e, h, true);
    s2 = strassen(m, ada, eha);
    freeMatrix(m, ada);
    freeMatrix(m, eha);
}

int** s3;
#pragma omp task shared(s3)
{
    int** acs = addMatrices(m, a, c, false);
    int** efa = addMatrices(m, e, f, true);
    s3 = strassen(m, acs, efa);
    freeMatrix(m, acs);
    freeMatrix(m, efa);
}

int** s4;
#pragma omp task shared(s4)
{
```

MPI Implementation

- initializing MPI, obtaining the total number of processes and the rank of the current process using communicators.
- Distribute submatrices of the input matrices to individual processes to ensure a balanced workload.
- Loacal Computation
- Exchange the necessary submatrices among processes using MPI send, receive and gather operations.



Snippets and Results




```
if (rank == 6)
{
    int** ges = addMatrices(m, g, e, false);
    s6 = strassen(m, d, ges);
    freeMatrix(m, ges);
    MPI_Send(&(s6[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
freeMatrix(m, g);

if (rank == 7)
{
    int** cda = addMatrices(m, c, d, true);
    s7 = strassen(m, cda, e);
    freeMatrix(m, cda);
    MPI_Send(&(s7[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
freeMatrix(m, c);
freeMatrix(m, d);
freeMatrix(m, e);

MPI_Barrier(MPI_COMM_WORLD);
```

Code

```
if (rank == 0)
{
    MPI_Recv(&(s1[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s2[0][0]), m * m, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s3[0][0]), m * m, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s4[0][0]), m * m, MPI_INT, 4, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s5[0][0]), m * m, MPI_INT, 5, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s6[0][0]), m * m, MPI_INT, 6, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&(s7[0][0]), m * m, MPI_INT, 7, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if (rank == 1)
{
    int** bds = addMatrices(m, b, d, false);
    int** gha = addMatrices(m, g, h, true);
    s1 = strassen(m, bds, gha);
    freeMatrix(m, bds);
    freeMatrix(m, gha);
    MPI_Send(&(s1[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

```
if (MPI_Init(&argc, &argv) != MPI_SUCCESS)
{
    printf("MPI-INIT Failed\n");
    return 0;
}

MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_process);

int n;
if (p_rank == 0)
{
    cout << endl;
    cout << "Enter the dimensions of the matrix: ";
    cin >> n;
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

int** mat1 = allocateMatrix(n);
int** mat2 = allocateMatrix(n);

if (p_rank == 0)
{
    fillMatrix(n, mat1);
    fillMatrix(n, mat2);
}

MPI_Bcast(&(mat1[0][0]), n * n, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&(mat2[0][0]), n * n, MPI_INT, 0, MPI_COMM_WORLD);
```

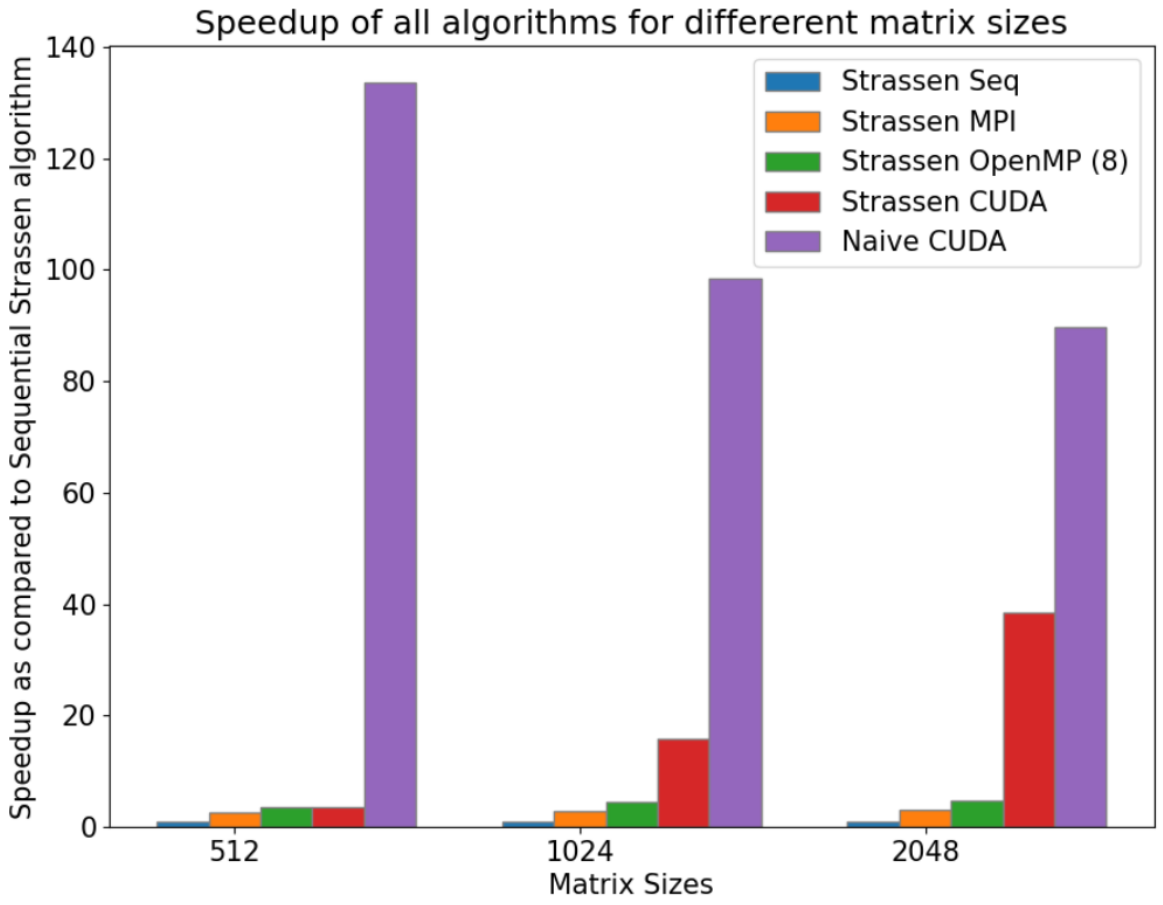
CUDA Implementation

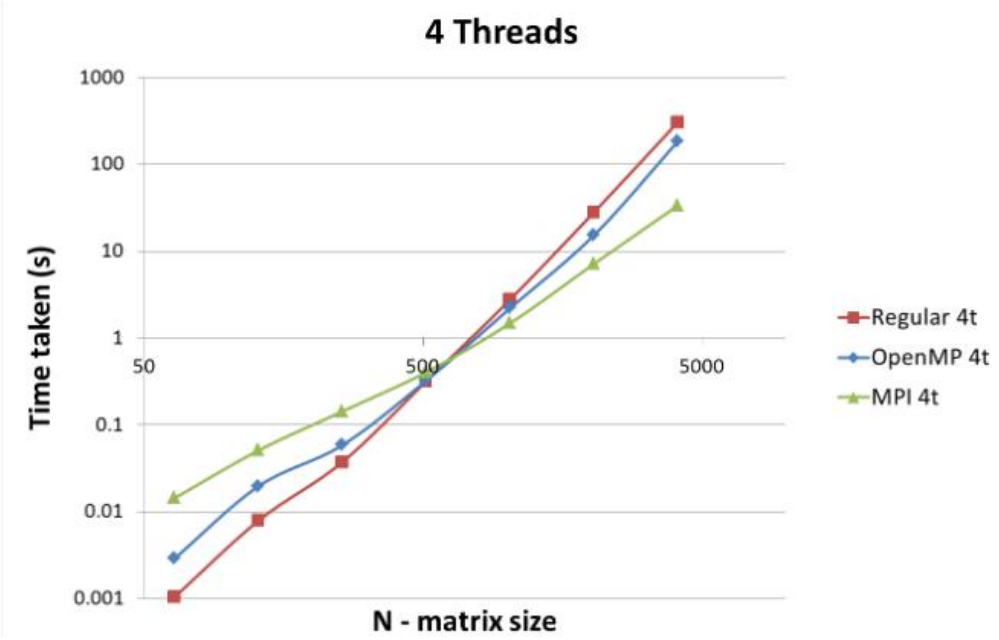
- Decompose the matrices into blocks.
- Use shared memory to store the submatrices.
- Use multiple threads to parallelize the computation.
- Use warp-level synchronization to reduce memory conflicts.

```
__global__
void classicalMatmul(ring* A, ring* B, ring* C, const int dim)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (row < dim && col < dim)
        {
                ring sum = 0;
                for (int k = 0; k < dim; ++k)
                {
                        sum += A[row*dim + k] * B[k*dim + col];
                }
                C[row*dim + col] = sum;
        }
}
```

```
cudaMalloc(&d_mat1, bytes);
cudaMalloc(&d_mat2, bytes);
cudaMalloc(&d_product, bytes);

cudaMemcpy(d_mat1, h_mat1, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_mat2, h_mat2, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_product, h_product, bytes, cudaMemcpyHostToDevice);

int threads = min(1024, n);
int blocks = (n * n) / threads;
dim3 gridSize(blocks, 1, 1);
dim3 blockSize(threads, 1, 1);

clock_t start, end;
start = clock();

matrixMultiplication<<<gridSize, blockSize>>>(d_mat1, d_mat2, d_product, n);
cudaDeviceSynchronize();
```

Comparisons of Parallelization Methods





Observations

- Significant speedup for large matrices
- Strassen algorithm slower for smaller matrices
- A combination of MPI and openMP can speed up even more.
- Granularity changes for different matrix sizes to avoid memory overflow.
- A practical downside of Strassen – potential for numerical instability

THANKS FOR YOUR ATTENTION