Matrix Multiplication: A Study

Kristi Stefa

Introduction

- Task is to handle matrix multiplication in an efficient way, namely for large sizes
- Want to measure how parallelism can be used to improve performance while keeping accuracy
- Also need for improvement on "normal" multiplication algorithm
  - $O(n^3)$ for conventional square matrices of n x n size

```
for (int i = 0; i < size; i++){
    for (int j = 0; j < size; j++){
        for (int k = 0; k < size; k++){
            C[i*size + j] += A[i*size + k] * B[k*size + j];
        }
    }
}
```

Strassen Algorithm

- Introduction of a different algorithm, one where work is done on submatrices
  - Requirement of matrix being square and size $n = 2^k$
  - Uses A and B matrices to construct 7 factors to solve C matrix
  - Complexity from 8 multiplications to 7 + additions: $O(n^{\log_2 7})$ or $n^{2.80}$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

Room for parallelism

- TBB:parallel_for is the perfect candidate for both the conventional algorithm and the Strassen algorithm
  - Utilization in ranges of 1D and 3D to sweep a vector or a matrix

```
parallel_for(blocked_range3d<int>(0, size, 0, size, 0, size),
  [&](const blocked_range3d<int> &r ) {
    for(int i=r.pages().begin(), i_end=r.pages().end(); i<i_end; i++){
      for(int j=r.rows().begin(), j_end=r.rows().end(); j<j_end; j++){
          for(int k=r.cols().begin(), k_end=r.cols().end(); k<k_end; k++){
              C[i*size + j] += A[i*size + k] * B[k*size + j];
          }
      }
    }
});
```

- TBB: parallel_reduce would be possible but would increase the complexity of usage
  - Would be used in tandem with a 2D range for the task of multiplication

```
parallel_for(int(0), int(subSize*subSize), [&] (int i){  //M1 = (A11 + A22)(B11 + B22)
  temp1[i] = A.quad11[i] + A.quad22[i];
  temp2[i] = B.quad11[i] + B.quad22[i];
});
tbbMatMult(temp1, temp2, M1, subSize);
```

Test Setup and Use Cases

- The input images were converted to binary by Matlab and the test filter ("B" matrix) was generated in Matlab and on the board
  - Cases were 128x128, 256x256, 512x512, 1024x1024
  - Matlab filter was designed for element-wise multiplication and board code generated for proper matrix-wise filter
  - Results were verified by Matlab and an import of the .bof file

```
for (int y = 0; y < matSize; y++){
    for (int x = 0; x < matSize; x++){
        if (x%2 && y==x){
            filterData[y*matSize + x] = 1;
        }
        else                    //simple filter
        {
            filterData[y*matSize + x] = 0;
        }
    }
}
```

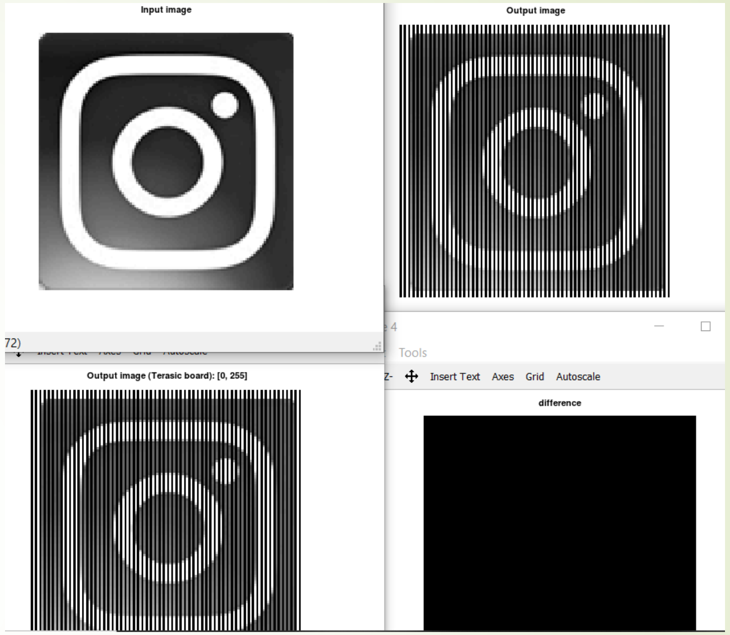
0	0	0	0
0	1	0	0
0	0	0	0
0	0	0	1

```
NIM = zeros(sY, sX);
for y = 1:sY
  for x = 1:sX
    if (!(mod(x,2)))  %simulates intended filter
      NIM(y,x) = 1;
```

0	1	0	1
0	1	0	1
0	1	0	1
0	1	0	1

Test setup (contd)

- Pictured to the right are the results for the small (128x128) case and a printout for the 256x256 case
- Filter degrades the image horizontally and successful implementation produces a black difference

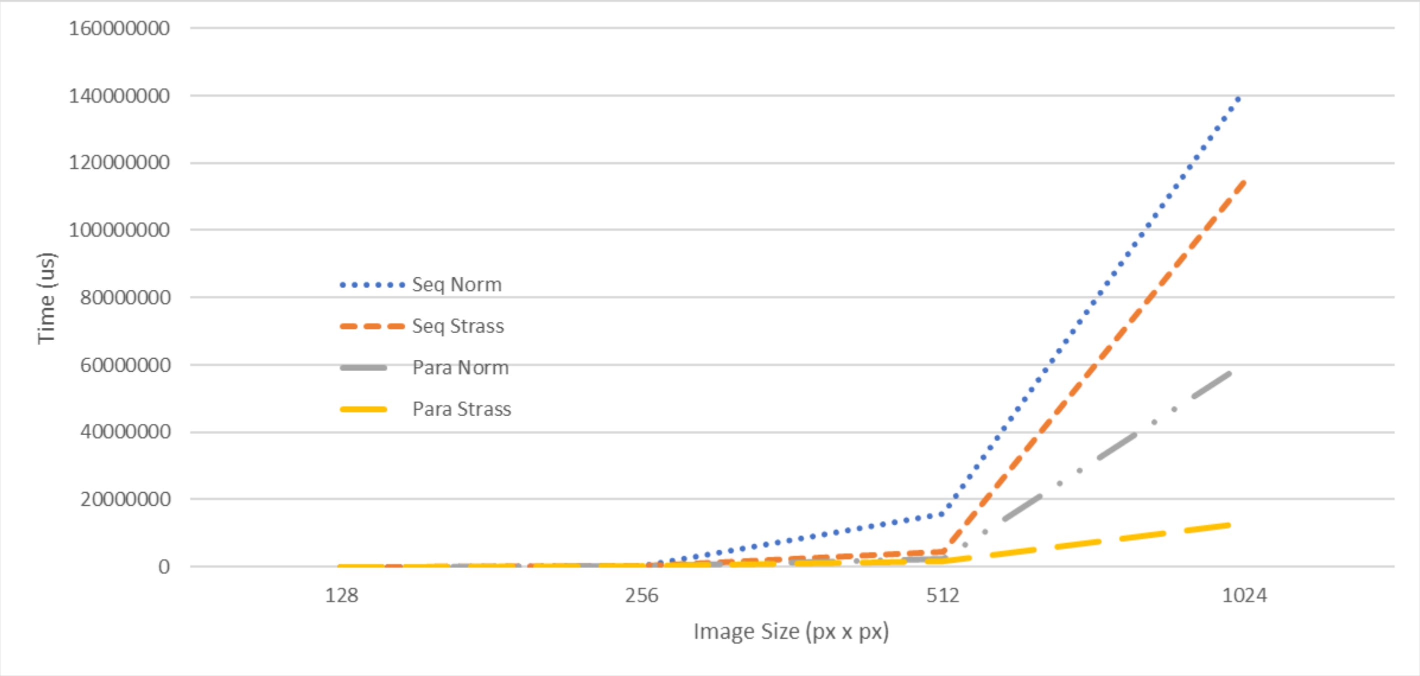



```
kristi@kristi-6930:~/Desktop/ECE5900/FINAL$ ./compareMM
(read_binfile) Size of each element: 1 bytes
(read_binfile) Input binary file: # of elements read = 65536
---Elapsed time (sequential norm): 968182 us
---Elapsed time (sequential Strassen): 838087 us
---Elapsed time (parallel norm): 577718 us
---Elapsed time (parallel strass): 484988 us
Output binary file: # of elements written = 65536
```

Bonus case & error

- If the filter doesn’t match between the board and Matlab, the following case can be observed
  - Half the element value are negative (Matlab – board), thus the sign dropping and the image appearing the same

Timing results (us)

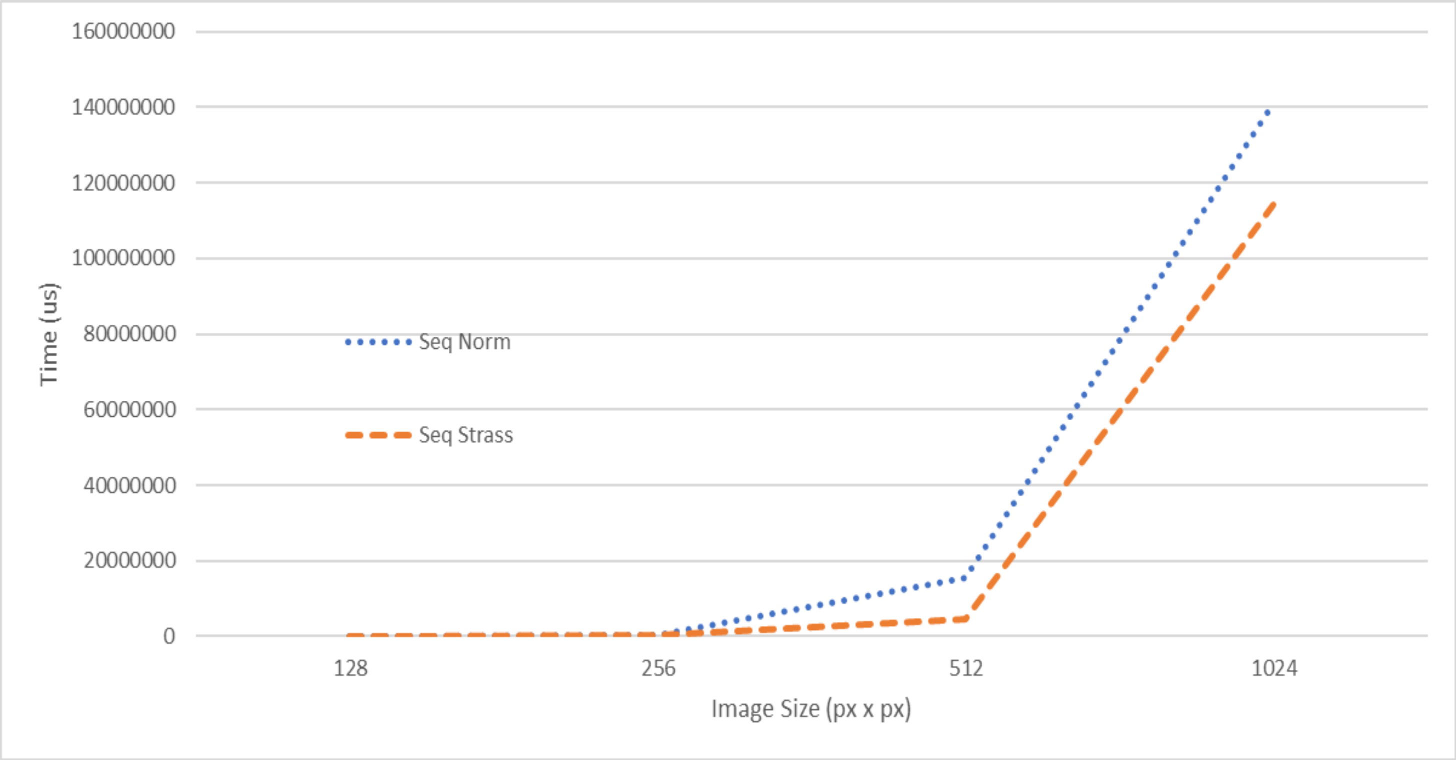


Size	Seq Norm	Seq Strass	Para Norm	Para Strass
128	37009.1	12521.7	26648.1	31785.4
256	279550	219712	212504	147237
512	15593915	4379200	2215604	1477464
1024	141.2 M	114.2 M	60116914	12895696

Seq Norm vs Para Strass	Speedup
128	1.16x
256	1.89x
512	10.55x
1024	10.94

Sequential Comparison

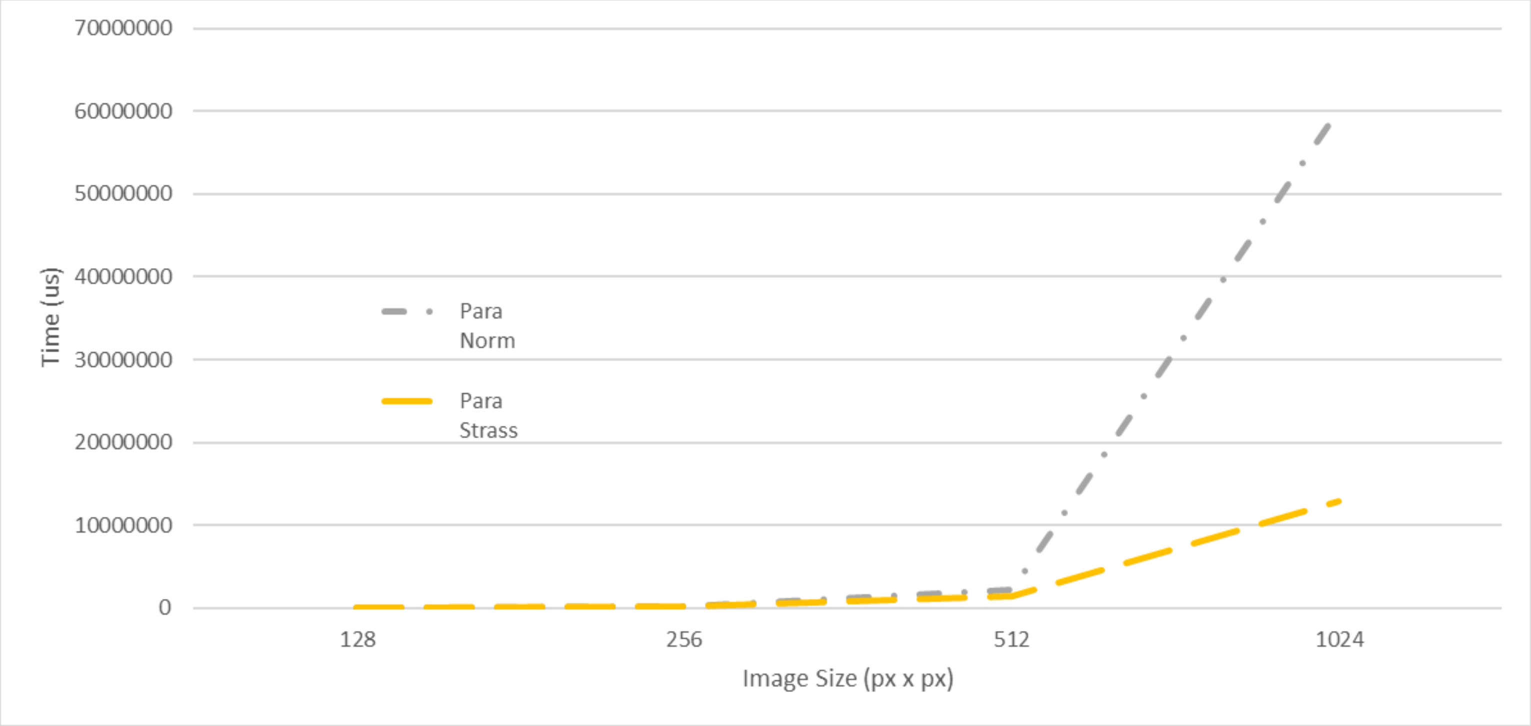
Size	Seq Norm	Seq Strass
128	37009.1	12521.7
256	279550	219712
512	15593915	4379200
1024	141.2 M	114.2 M



Sequential Strass faster than anything else by 2x for 128 case

Parallel Comparison

Size	Para Norm	Para Strass
128	26648.1	31785.4
256	212504	147237
512	2215604	1477464
1024	60116914	12895696



Closing remarks

- Strassen provides some very significant speedup at all ranges at the cost of being complex
  - True Strassen requires recursion down to submatrices of size 1, becomes absurd at higher sizes
  - Current studies find that less and less Strassen steps are needed before switching to an optimized “simple” multiplication method for the best performance
- Parallelism, given knowledge of TBB, is exceedingly straightforward to implement and provides substantial speedup on all parts of the algorithms