# Morton Ordering of 2D Arrays for Parallelism and Efficient Access to Hierarchical Memory

*David W. Walker*

Cardiff School of Computer Science & Informatics

WalkerDW@cardiff.ac.uk

# Main Objectives of Talk

- To give a progress report on current research relating to efficient memory access in parallel and distributed computers.
- To describe the Morton order of memory layout for arrays.
- To present preliminary timing results for matrix multiplication, Cholesky factorization, and FFT, when Morton ordering is used.

# The Problem

- Modern computer systems have complex hierarchical memory systems with multiple layers (caches, main memory, remote memory, etc.)
- Applications need to be able to control placement and transfer of data in memory.
- Need a model of concurrent computation that takes hierarchical memory into account and balances load across heterogeneous processors.

# Matrix Multiply: ijk

```
void matMul_ijk(float* M, float* N, float* P, int Width)
{
    int i, j, k;
    for (i = 0; i < Width; ++i)
        for (j = 0; j < Width; ++j) {
            float sum = 0;
            for (k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

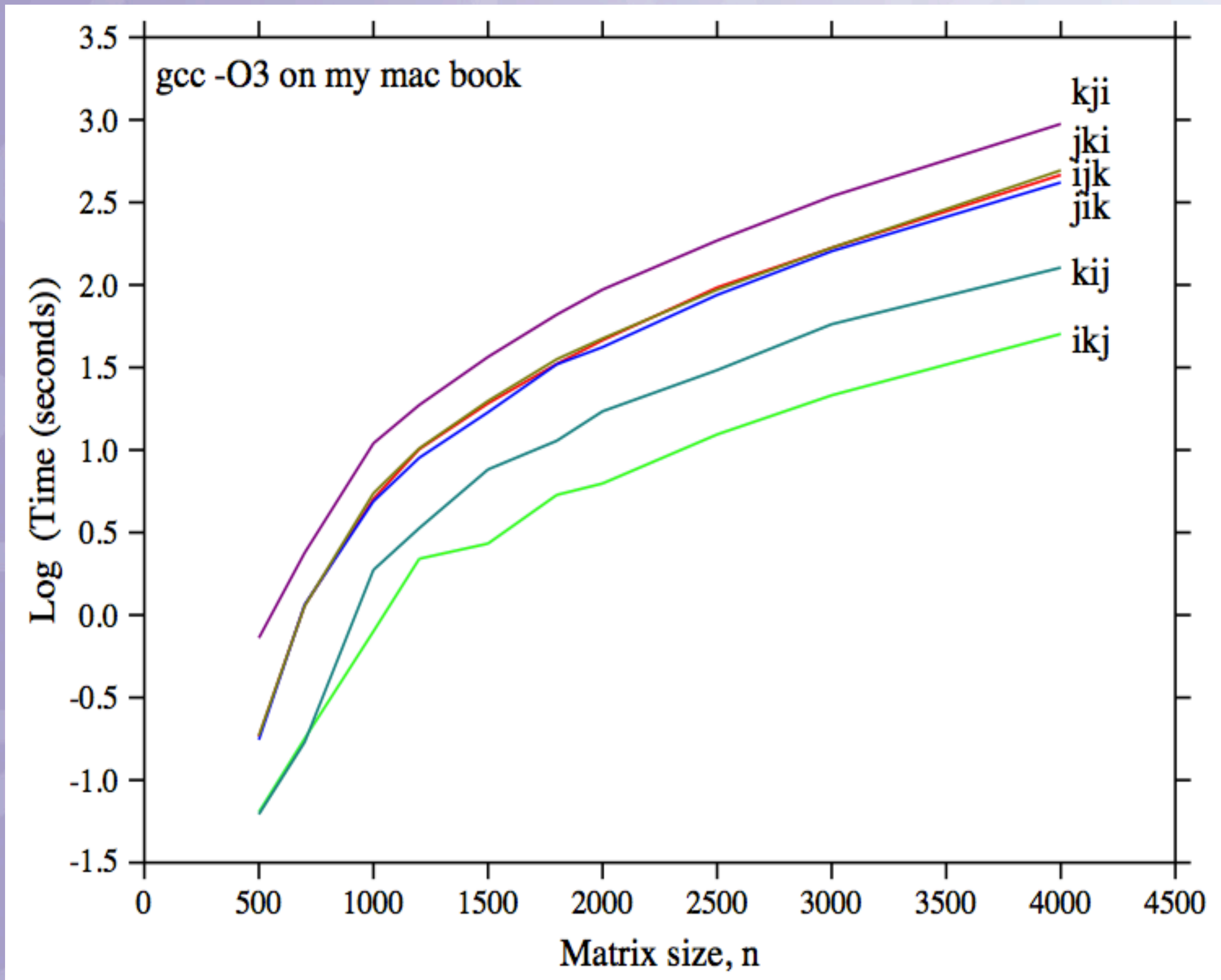
M is accessed by row, and N is accessed by column

# Matrix Multiply: ikj

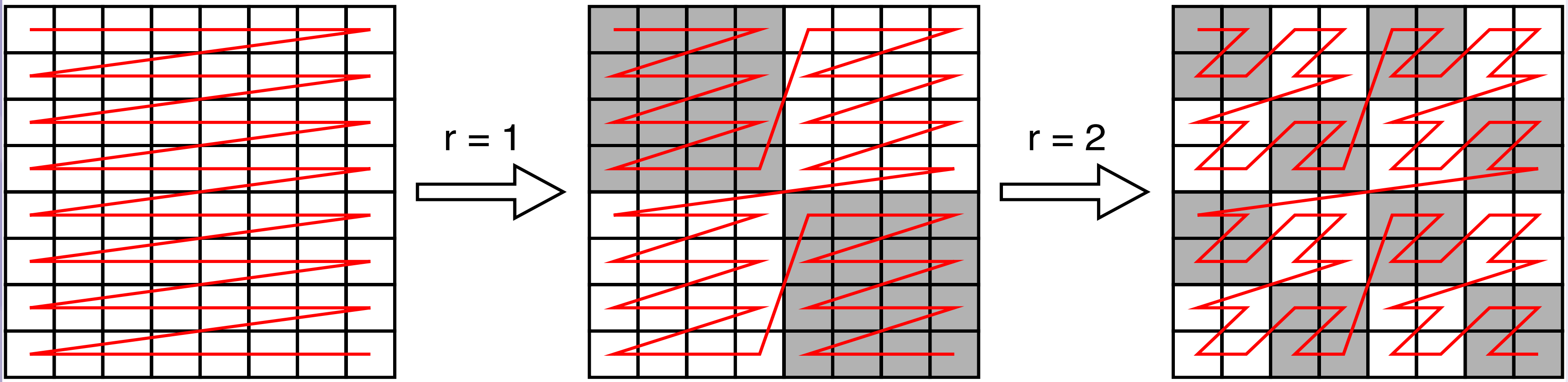
```
void matMul_ikj(float* M, float* N, float* P, float* work, int Width)
{
    int i, j, k;
    for (i = 0; i < Width; ++i){
        for (j = 0; j < Width; ++j) work[j] = 0;
        for (k = 0; k < Width; ++k) {
            float a = M[i * Width + k];
            for (j = 0; j < Width; ++j) {
                float b = N[k * Width + j];
                work[j] += a * b;
            }
        }
        for (j = 0; j < Width; ++j) P[i * Width + j] = work[j];
    }
}
```

M and N are both accessed by row.

# Matrix Multiplication Variants

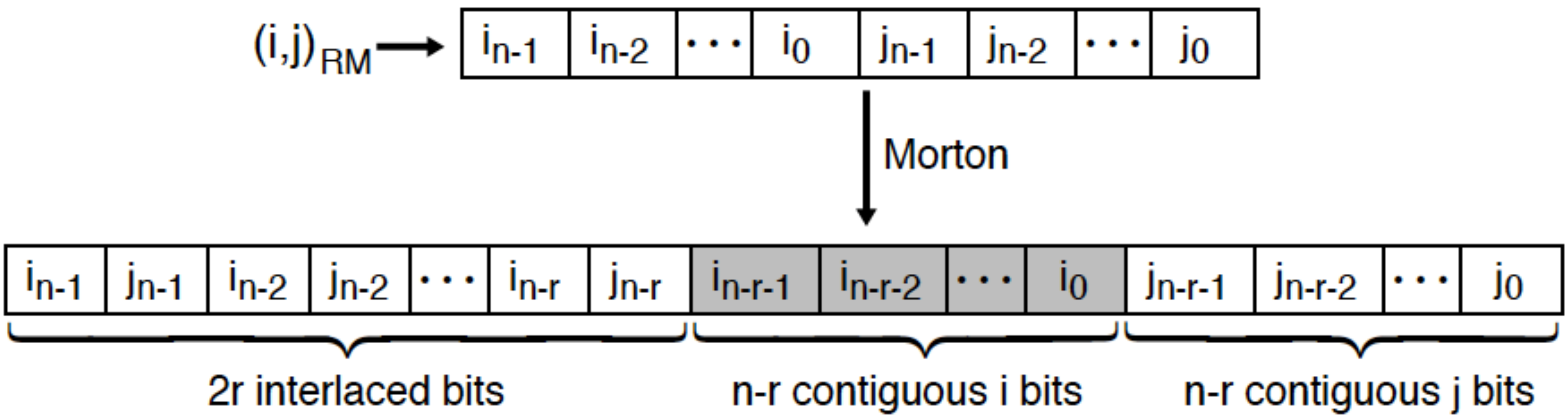


# Morton Order




- Can be done in-place using *unshuffle* operations
- Good locality of reference – should work well for hierarchical memories.
- Gives rise naturally to recursive parallel algorithms.

# Square $2^n$ x $2^n$ Arrays




Block size, $b = 2^{n-r}$

# Unshuffle Operation

- The unshuffle operation takes a shuffled sequence of items and unshuffles them:

$$a_1 b_1 a_2 b_2 \ldots a_n b_n \rightarrow a_1 a_2 \ldots a_n b_1 b_2 \ldots b_n$$

  where each $a_i$ is a contiguous vector of $\ell_a$ items, and each $b_i$ is a contiguous vector of $\ell_b$ items.

- Each $a_i b_i$ pair represents one row of the matrix.
- The unshuffle operation partitions the matrix over columns.

# Divide-And-Conquer Unshuffle

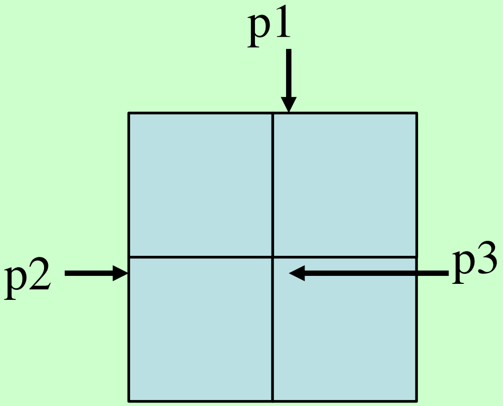
- Suppose n=8

1. Group as: $(a_1b_1a_2b_2)(a_3b_3a_4b_4)(a_5b_5a_6b_6)(a_7b_7a_8b_8)$
2. Swap first b vector with second a vector in each group: $(a_1a_2b_1b_2)(a_3a_4b_3b_4)(a_5a_6b_5b_6)(a_7a_8b_7b_8)$
3. Re-group as: $(a_1a_2b_1b_2a_3a_4b_3b_4)(a_5a_6b_5b_6a_7a_8b_7b_8)$
4. Swap first pair of b's with second pair of a's in each group: $(a_1a_2a_3a_4b_1b_2b_3b_4)(a_5a_6a_7a_8b_5b_6b_7b_8)$
5. Re-group as: $(a_1a_2a_3a_4b_1b_2b_3b_4a_5a_6a_7a_8b_5b_6b_7b_8)$
6. Swap first set of 4 b's with second set of 4 a's: $(a_1a_2a_3a_4a_5a_6a_7a_8b_1b_2b_3b_4b_5b_6b_7b_8)$

# Apply Morton Ordering to Matrix A

```
mortonOrder (A,n,b){
    if( b < n ){
        p1 = (n*n)/4
        p2 = 2*p1
        p3 = 3*p1
        unshuffle(A,n/2,n/2)
        unshuffle(A+p2,n/2,n/2)
        mortonOrder(A,n/2,b)
        mortonOrder(A+p1,n/2,b)
        mortonOrder(A+p2,n/2,b)
        mortonOrder(A+p3,n/2,b)
    }
}
```

n is matrix size, b is block size. Both are powers of 2.



# Examples of Related Work

- Previous work on *locality preserving hashing* and *space-filling curves.*
- *Sequoia:* programming language designed to facilitate the development of memory hierarchy aware parallel programs. Abstractly expose memory hierarchy in programming model. 10.1145/1188455.1188543
- Thiyagalingam, Beckmann and Kelly, 10.1002/cpe.1018
- Mellor-Crummey, Whalley and Kennedy, 10.1023/A:1011119519789

# Matrix Multiplication with RQB

- C=AB, for square matrices.
- Apply one level of Morton ordering to A and B
- For i and j = 0,1:

$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j}$$

- So C=AB can be done recursively
- MO means the matrices at each level of recursion are stored contiguously.

# Recursive Matrix Multiply

```
mm_Recursive (A,B,C,n,b){        // C = C + AB
    if(n==b){
        matmul(A,B,C,n)
    }
    else{
        mm_Recursive(A00,B00,C00,n/2,b)
        mm_Recursive(A01,B10,C00,n/2,b)
        mm_Recursive(A00,B01,C01n/2,b)
        mm_Recursive(A01,B11,C01,n/2,b)
        mm_Recursive(A10,B00,C10,n/2,b)
        mm_Recursive(A11,B10,C10,n/2,b)
        mm_Recursive(A10,B01,C11,n/2,b)
        mm_Recursive(A11,B11,C11,n/2,b)
    }
    return C
}
```

End of recursion. Choose b so matrices fit in cache.

| | |
|---|---|
| A00 | A01 |
| A10 | A11 |

Note: all the work happens in the leaves of the recursion tree.

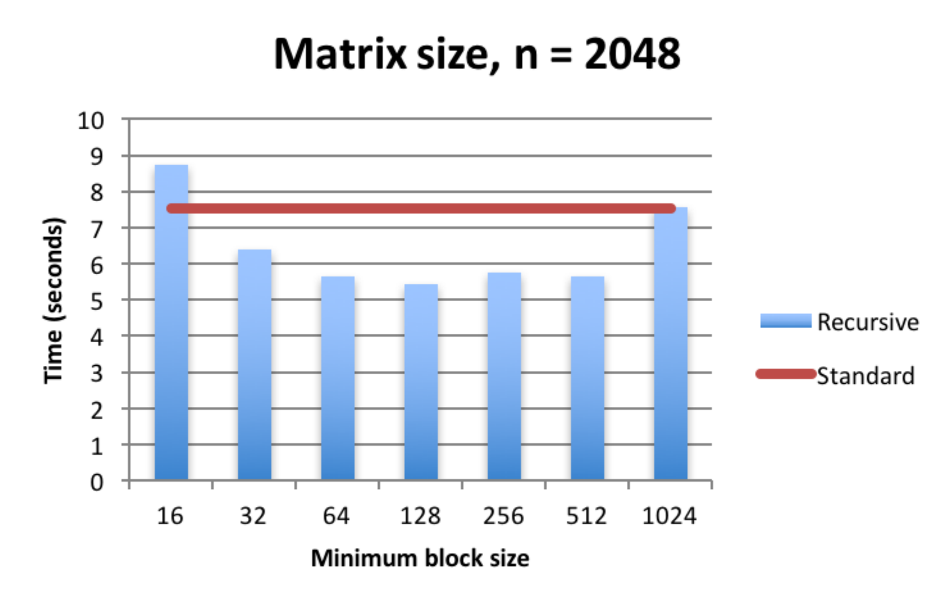
# Preliminary Timing Results

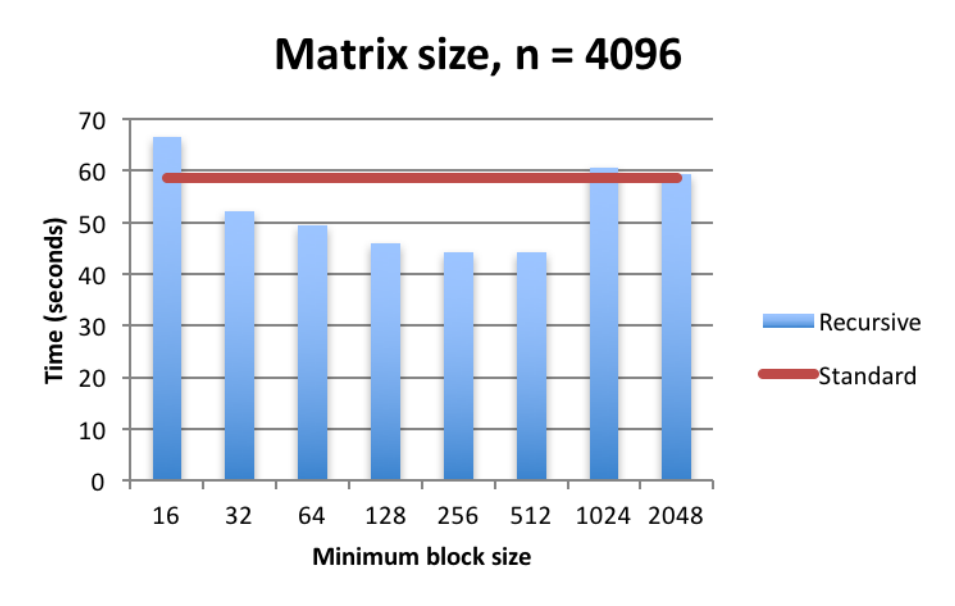
- Platform 1: MacBook, OS X 10.9.5
  - 2.66GHz Intel Core 2 Duo processor
  - 3Mb cache, 4Gb main memory
  - gcc v. 4.8.2, using –O3 flag
- Platform 2: MacBook, OS X 10.10.5
  - 2.5GHz Intel Core i7 (4 cores)
  - 256Kb L2 cache, 6Mb L3 cache
  - 16Mb main memory
  - gcc v. 4.8.5, using –O3 flag

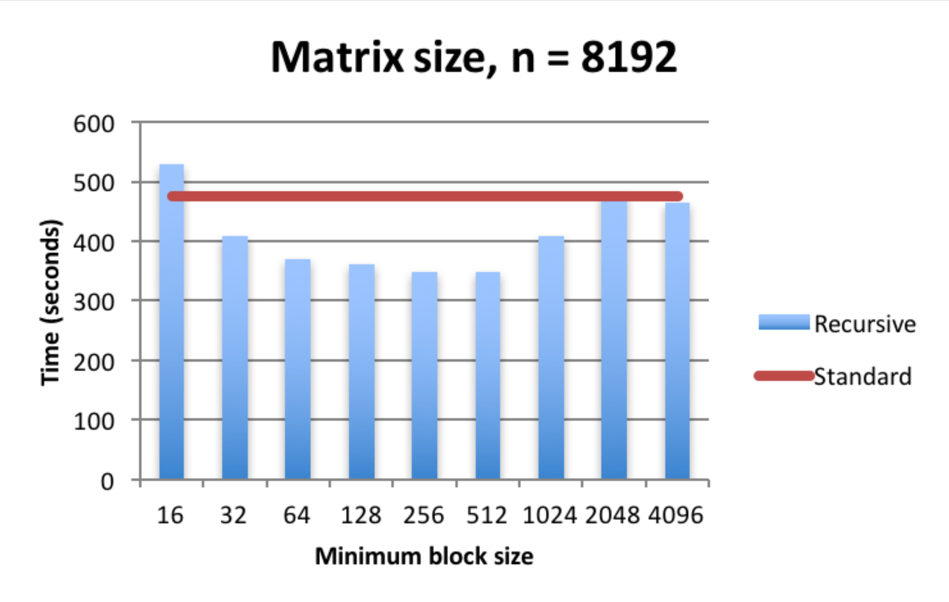
# Preliminary Timing Results

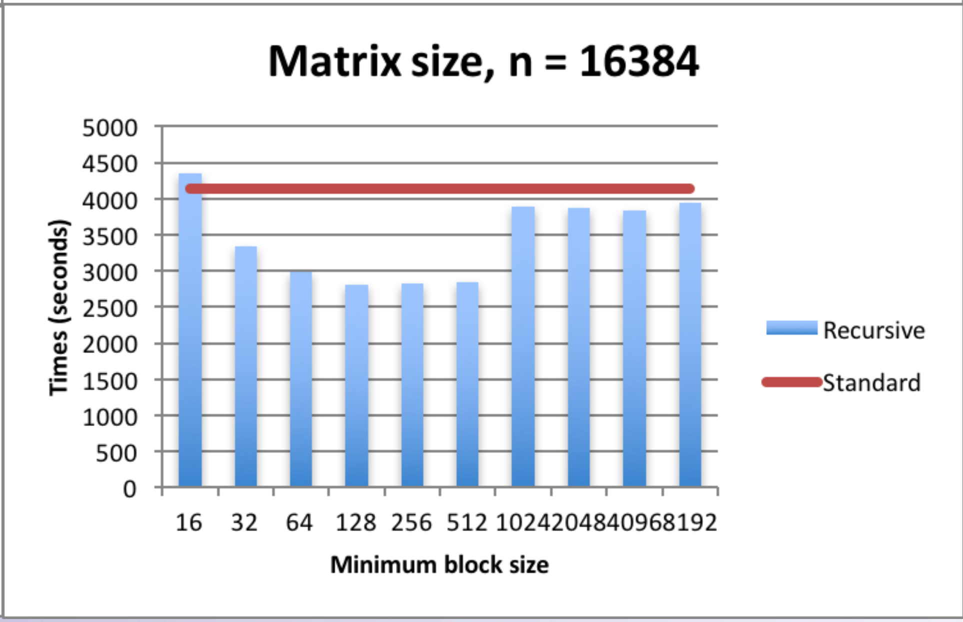
- Platform 3: Xeon E5-2620, Red Hat Enterprise Linux Server v6.2
  - 2Ghz Intel Xeon E5-2620 processor (6 cores)
  - 15Mb cache
  - gcc v. 4.8.5, using –O3 flag

# Platform 1

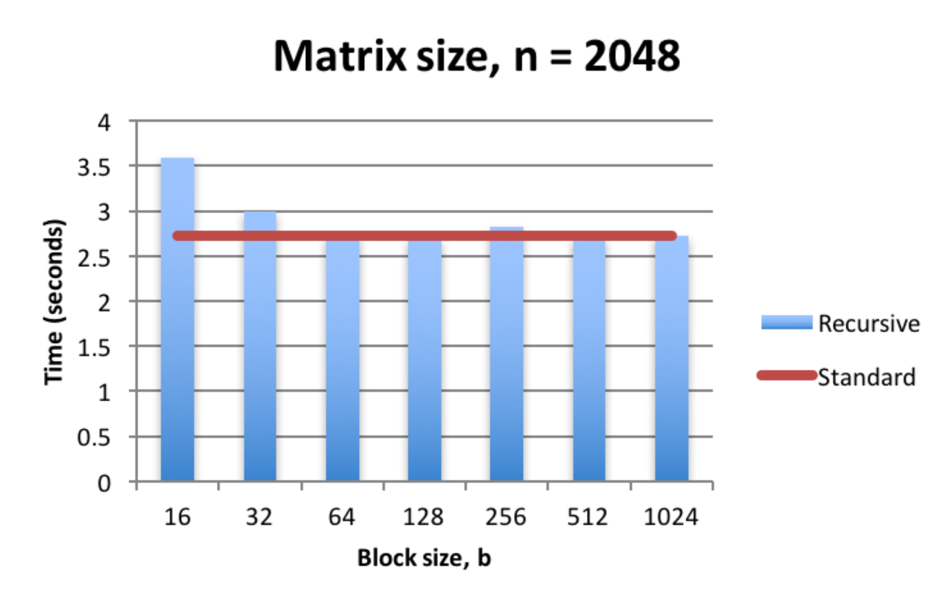

Matrix size, n = 2048
10
9
8
7
6
5
4
3
2
1
0
Time (seconds)
16
32
64
128
256
512
1024
Minimum block size
Recursive
Standard


Matrix size, n = 4096
70
60
50
40
30
20
10
0
Time (seconds)
16
32
64
128
256
512
1024
2048
Minimum block size
Recursive
Standard

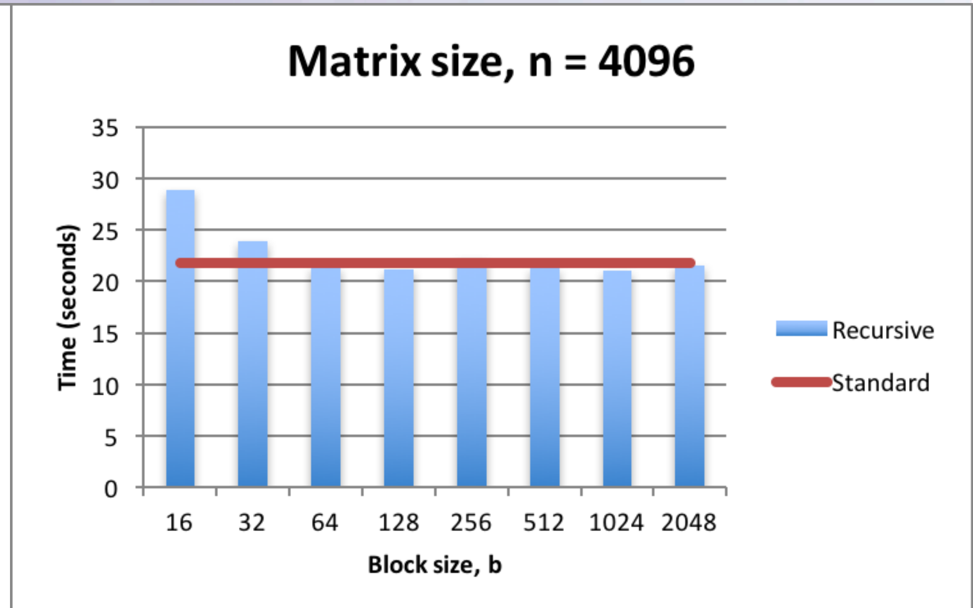

Matrix size, n = 8192
600
500
400
300
200
100
0
Time (seconds)
16
32
64
128
256
512
1024
2048
4096
Minimum block size
Recursive
Standard


Matrix size, n = 16384
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Times (seconds)
16
32
64
128
256
512
1024
2048
4096
8192
Minimum block size
Recursive
Standard

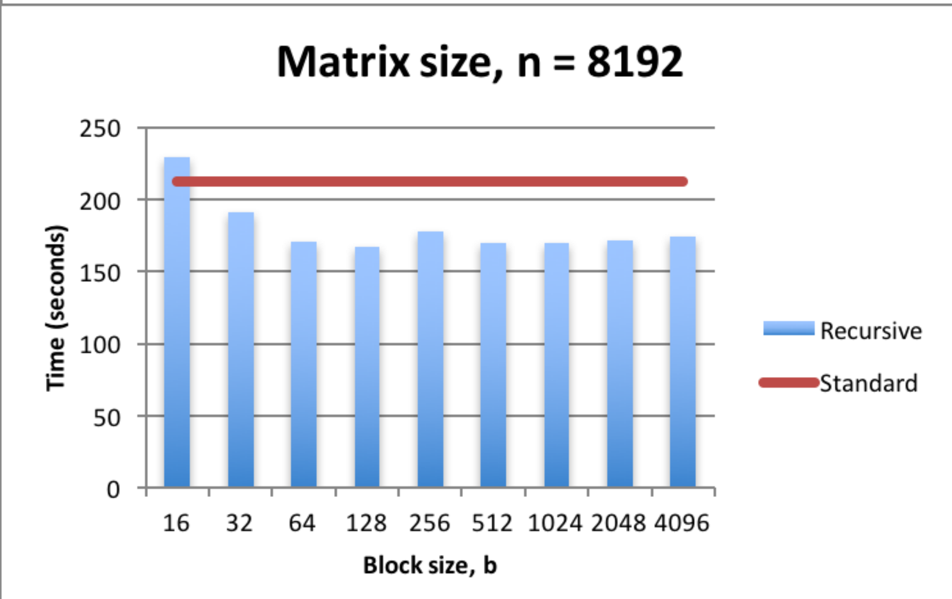
# Platform 2



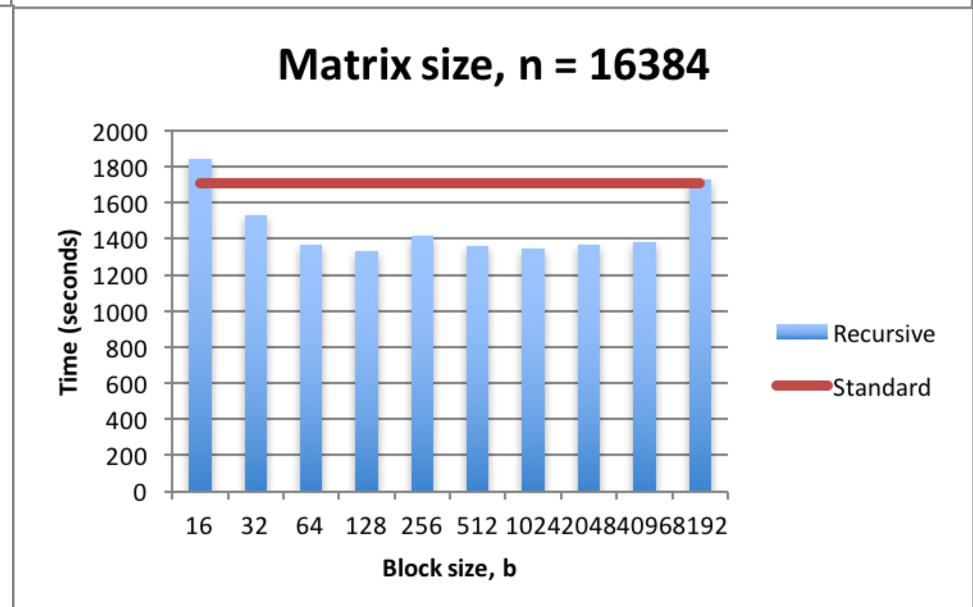
Matrix size, n = 2048
Time (seconds)
4
3.5
3
2.5
2
1.5
1
0.5
0
16
32
64
128
256
512
1024
Block size, b
Recursive
Standard

Matrix size, n = 4096
Time (seconds)
35
30
25
20
15
10
5
0
16
32
64
128
256
512
1024
2048
Block size, b
Recursive
Standard

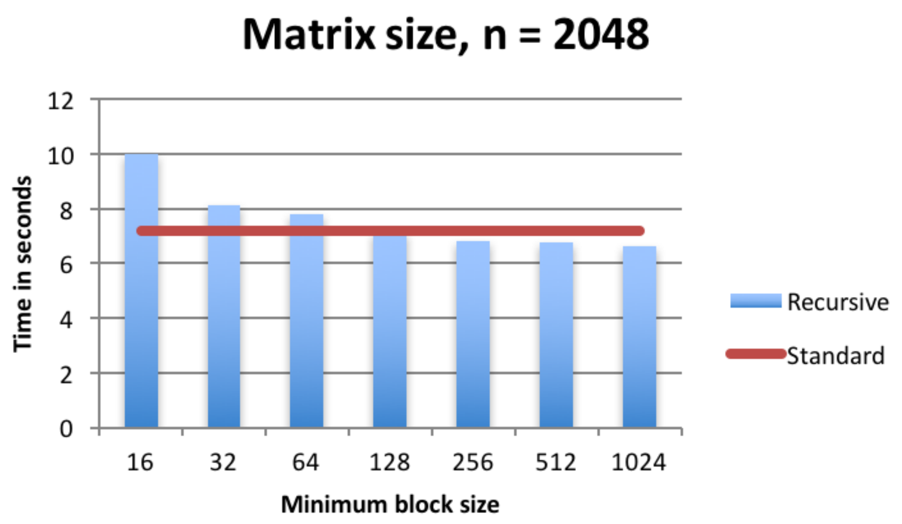



Matrix size, n = 8192
Time (seconds)
250
200
150
100
50
0
16
32
64
128
256
512
1024
2048
4096
Block size, b
Recursive
Standard

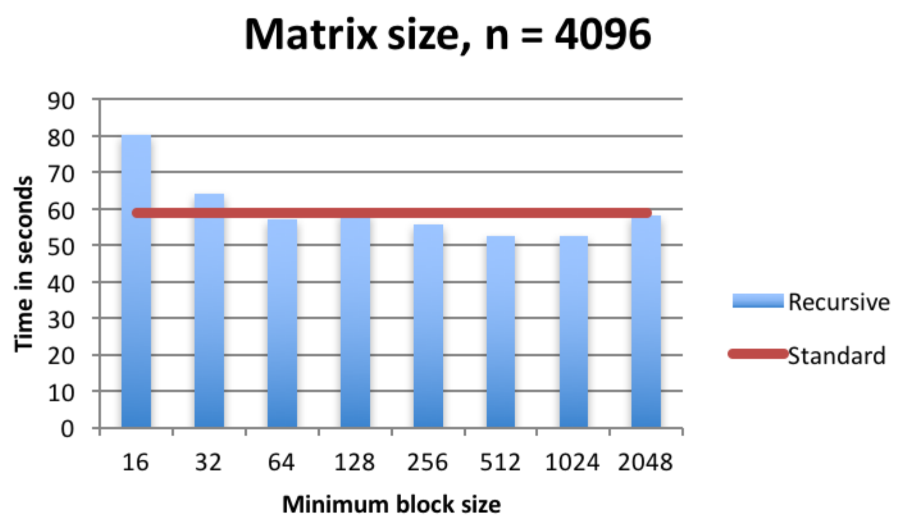



Matrix size, n = 16384
Time (seconds)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
16
32
64
128
256
512
1024
2048
4096
8192
Block size, b
Recursive
Standard

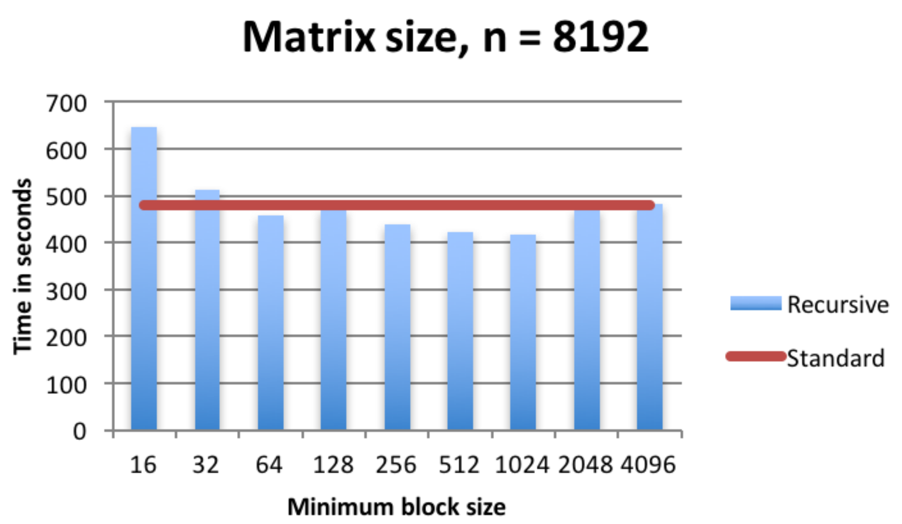
# Platform 3



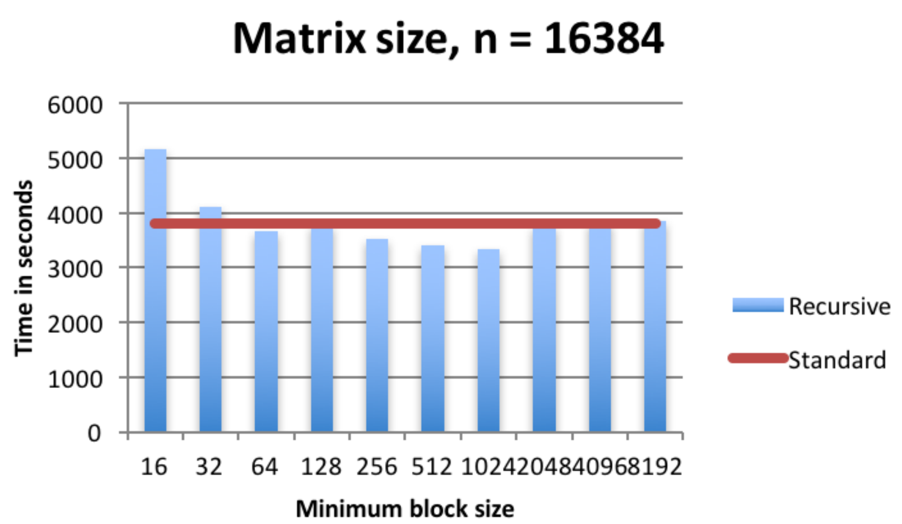
Matrix size, n = 2048
Time in seconds
0
2
4
6
8
10
12
16
32
64
128
256
512
1024
Minimum block size
Recursive
Standard

Matrix size, n = 4096
Time in seconds
0
10
20
30
40
50
60
70
80
90
16
32
64
128
256
512
1024
2048
Minimum block size
Recursive
Standard

Matrix size, n = 8192
Time in seconds
0
100
200
300
400
500
600
700
16
32
64
128
256
512
1024
2048
4096
Minimum block size
Recursive
Standard

Matrix size, n = 16384
Time in seconds
0
1000
2000
3000
4000
5000
6000
16
32
64
128
256
512
1024
2048
4096
8192
Minimum block size
Recursive
Standard

# 2D FFT with Morton Ordering

- The Fourier transform of an nxn array, X, can be expressed as:

  $$Y = F_n X F_n$$

  where element (p,q) of matrix $F_n$ is $w_n^{pq}$

  $$w_n = \exp(-2\pi i / n)$$

- The fast Fourier transform (FFT) recasts this dense matrix multiply in terms of sparse operations.

# 2D Fast Fourier Transform

$Y = F_n X F_n = F_n X F_n^T = A_t \ldots A_1 P_n^T X P_n A_1^T \ldots A_t^T$

- $t = \log_2(n)$
- $P_n^T$ is a permutation matrix such that $P_n^T X$ exchanges column k of X with column k', where k' is the t bits of k in reverse order.

Kronecker matrix product

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

$$\begin{aligned} A_q &= I_r \otimes B_L \\ B_L &= \begin{bmatrix} I_{L_*} & \Omega_{L_*} \\ I_{L_*} & -\Omega_{L_*} \end{bmatrix} \\ \Omega_{L_*} &= \mathrm{diag}(1, \omega_L, \ldots, \omega_L^{L_*-1}) \end{aligned}$$

- where $L = 2^q$, $r = n/L$, $L_* = L/2$
- $B_L$ is the “butterfly matrix”.
- $A_q$ is made up of r diagonal blocks of $B_L$

# Basis of recursive 2D FFT

$$F_n \Pi_{b,n} = B_{b,n}(I_{n/b} \otimes F_b)$$

$$\begin{aligned}
\Pi_{b,n} &= \Pi_n(I_2 \otimes \Pi_{n/2})(I_4 \otimes \Pi_{n/4}) \ldots (I_{n/(2b)} \otimes \Pi_{2b}) \\
B_{b,n} &= B_n(I_2 \otimes B_{n/2})(I_4 \otimes B_{n/4}) \ldots (I_{n/(2b)} \otimes B_{2b}) \\
&= A_t A_{t-1} \ldots A_{s+1}
\end{aligned}$$

- $b = 2^s$, $b<n$.
- $\Pi_n$ is a permutation matrix that performs a perfect shuffle index operation.
- $\Pi_{b,n}$ performs a partial bit reversal on indices.

$H_{b,n}$ permutes the columns and rows of X based on a partial bit-reversal of indices.

$$H_{b,n} = \Pi_{b,n}^T X \Pi_{b,n}$$

$$F_n X F_n = F_n X F_n^T = B_{b,n} \boxed{(I_{n/b} \otimes F_b) H_{b,n} (I_{n/b} \otimes F_b)} B_{b,n}^T$$

What is in the red box?
This is the result of partitioning the matrix into bxb blocks and performing a 2D FFT on each

Denote this by $K_{b,n}$

$K_{b,n}$ = 2D FFT of bxb blocks of partially bit-reversed matrix, X

$$F_n X F_n = F_n X F_n^T = A_t \dots A_{s+1} K_{b,n} A_{s+1}^T \dots A_t^T$$

$$Y_{b,n} = A_t \dots A_{s+1} K_{b,n}$$

$$F_n X F_n^T = \tilde{X} = Y_{b,n} A_{s+1}^T \dots A_t^T$$

$$\tilde{X}^T = A_t \dots A_{s+1} Y_{b,n}^T$$

1. Evaluate $Y_{b,n}$
2. Transpose
3. Evaluate $\tilde{X}^T$
4. Transpose

# Recursive 2D FFT

- Apply Morton ordering to X to give contiguous bxb blocks.
- Re-order rows and columns of re-ordered X according to partial bit-reverse permutation.
- Do 2D FFT of resulting bxb blocks.
- Recursively pre-multiply by $A_q$ to build up blocks of size (2b)x(2b), then (4b)x(4b), etc.
- Transpose.
- Recursively pre-multiply by $A_q$ to build up blocks of size (2b)x(2b), then (4b)x(4b), etc.
- Transpose.

# Recursive 2D FFT

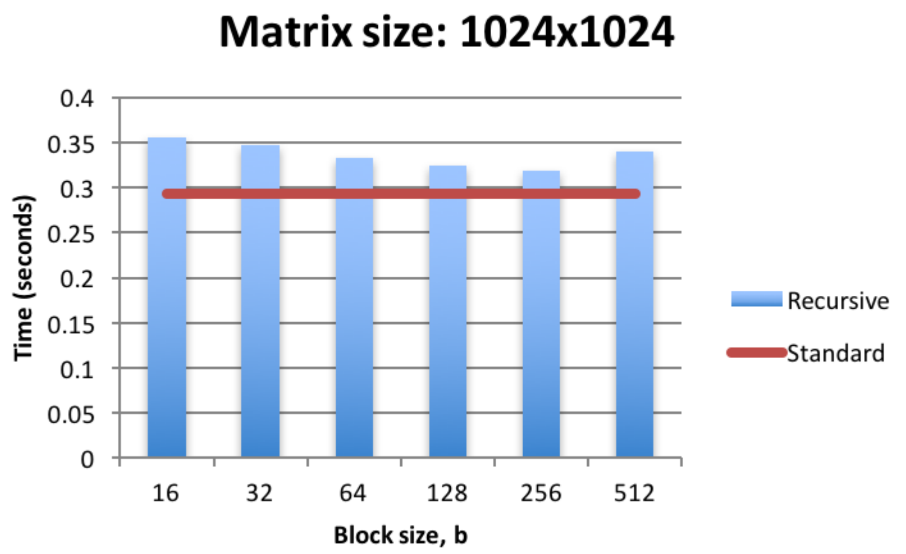
```
fft2D_Recursive (X,n,b,dofft){
    if(n==b){
        if(dofft) fft2D(X,n)
    }
    else{
        fft2D_Recursive(X00,n/2,b,dofft)
        fft2D_Recursive(X01,n/2,b,dofft)
        fft2D_Recursive(X10,n/2,b,dofft)
        fft2D_Recursive(X11,n/2,b,dofft)
        butterfly(X,n,b)
    }
    return
}
```

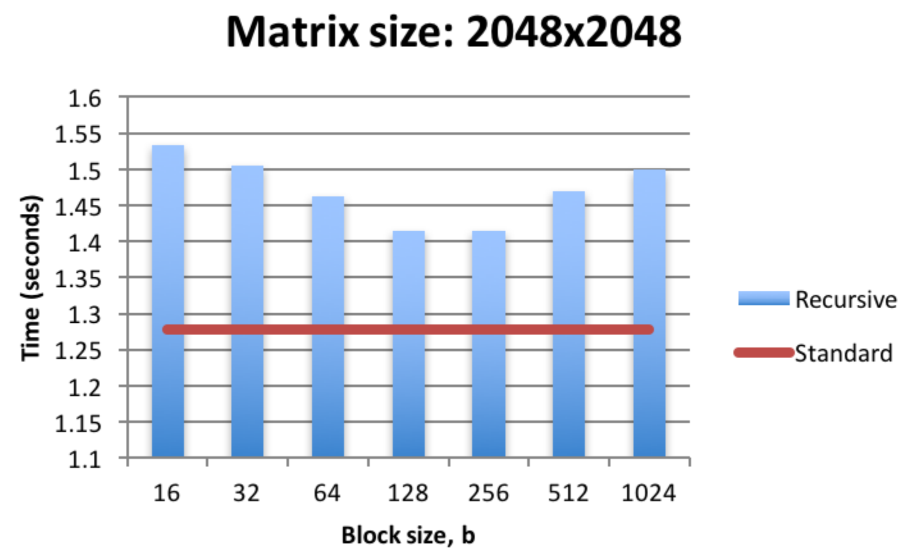
End of recursion. Choose b so matrices fit in cache.

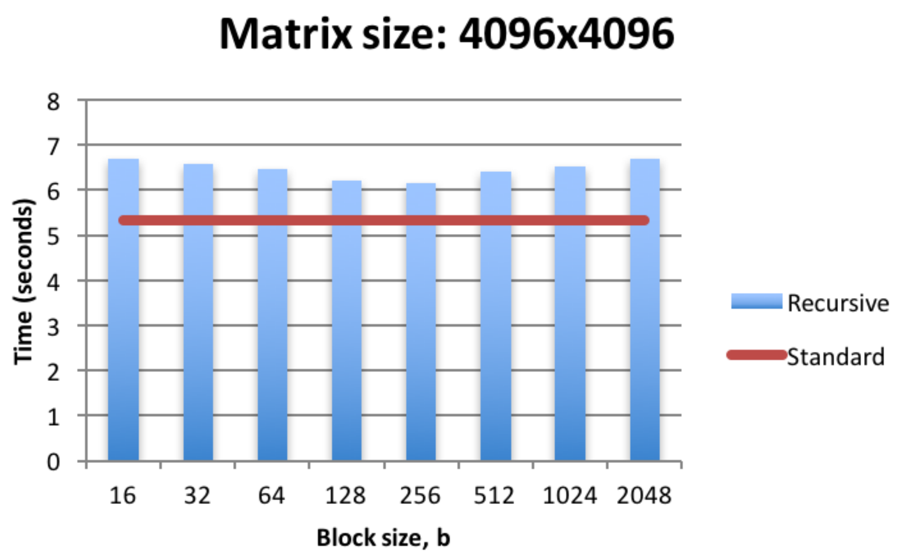
Note: includes work at each level of the recursion tree.

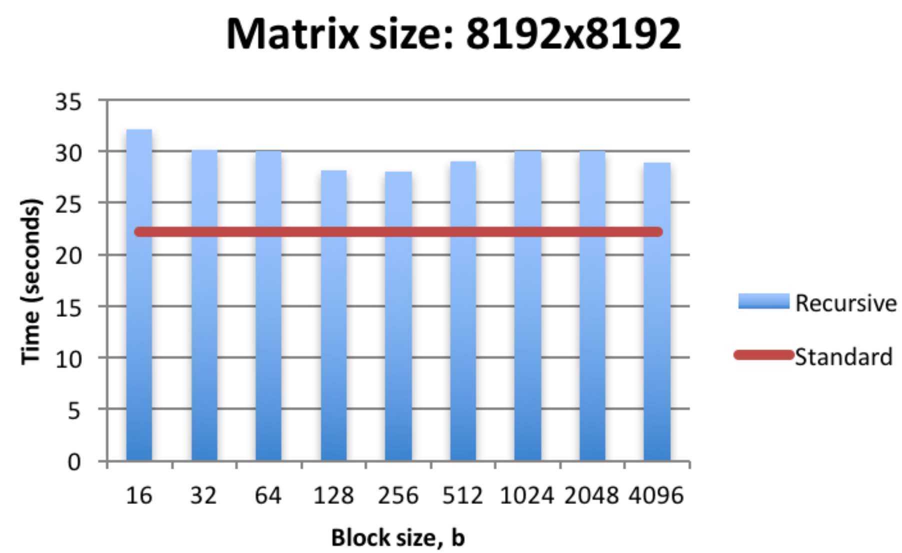
Butterfly routine applies butterfly matrix to nxn block, overwriting X.

# Platform 1 Timings: FFT

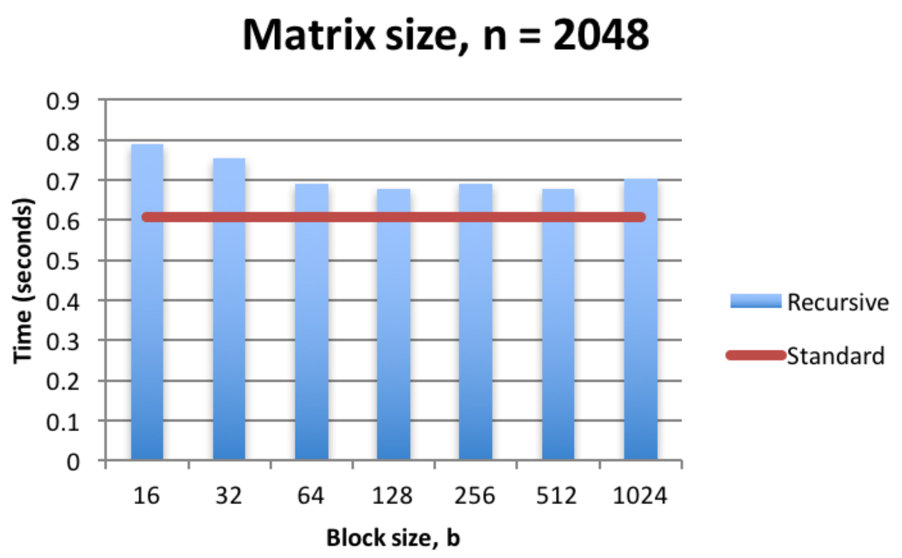


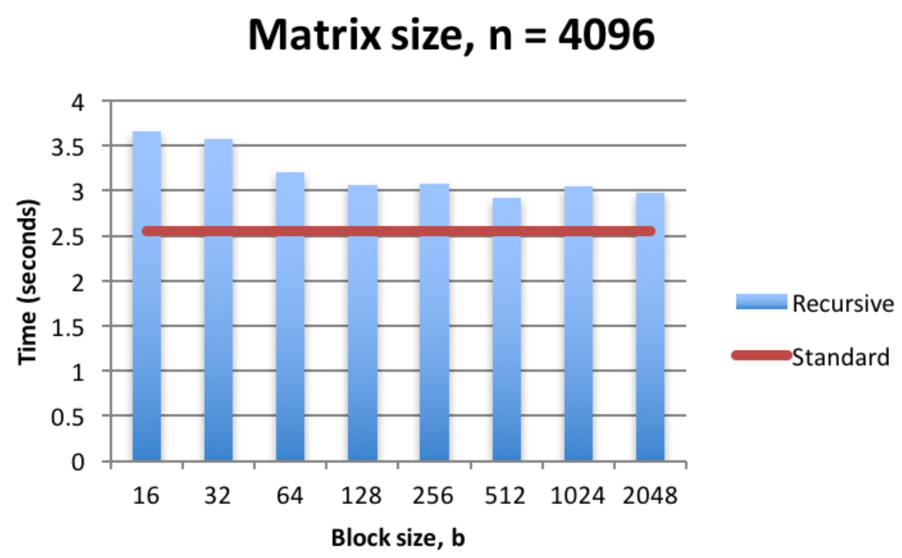



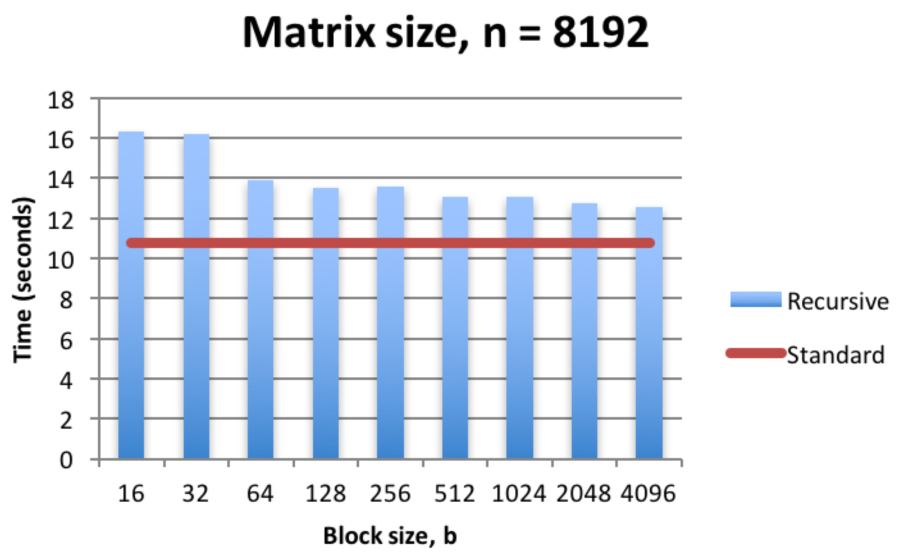



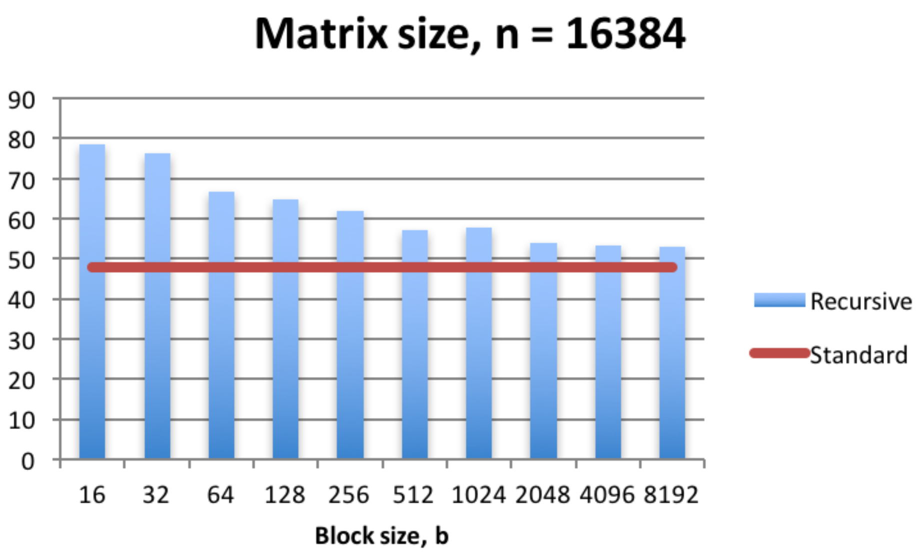



# Platform 2 Timings: FFT

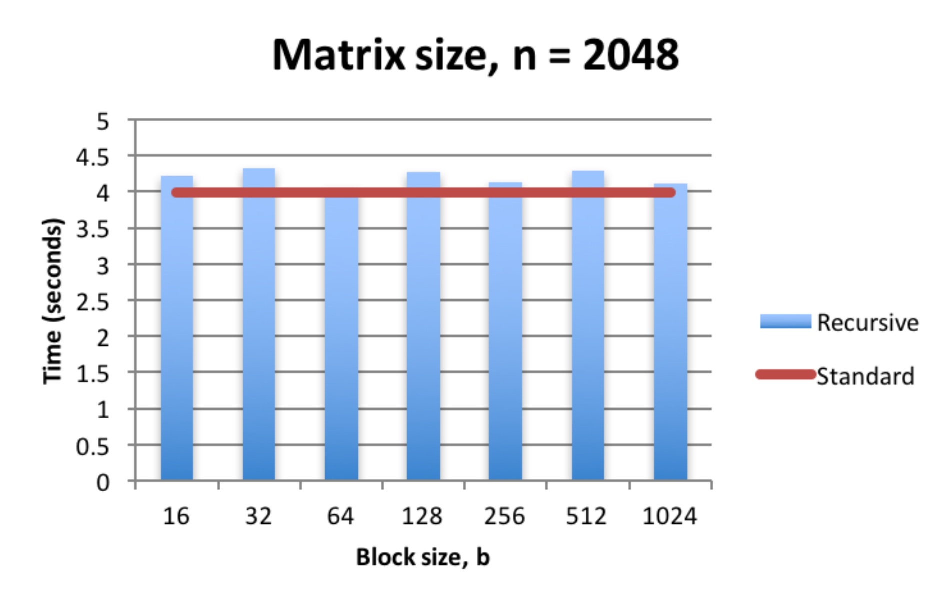


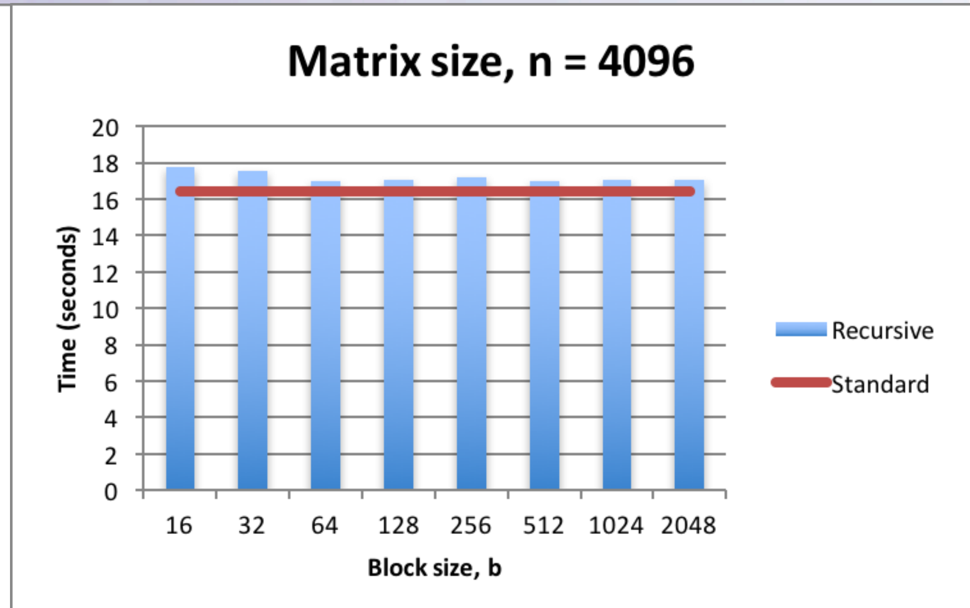



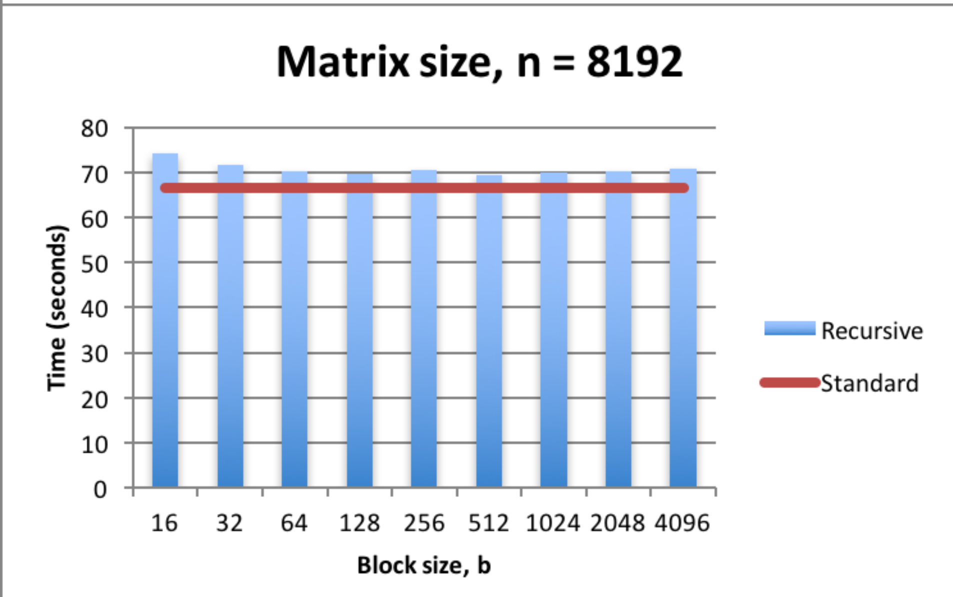



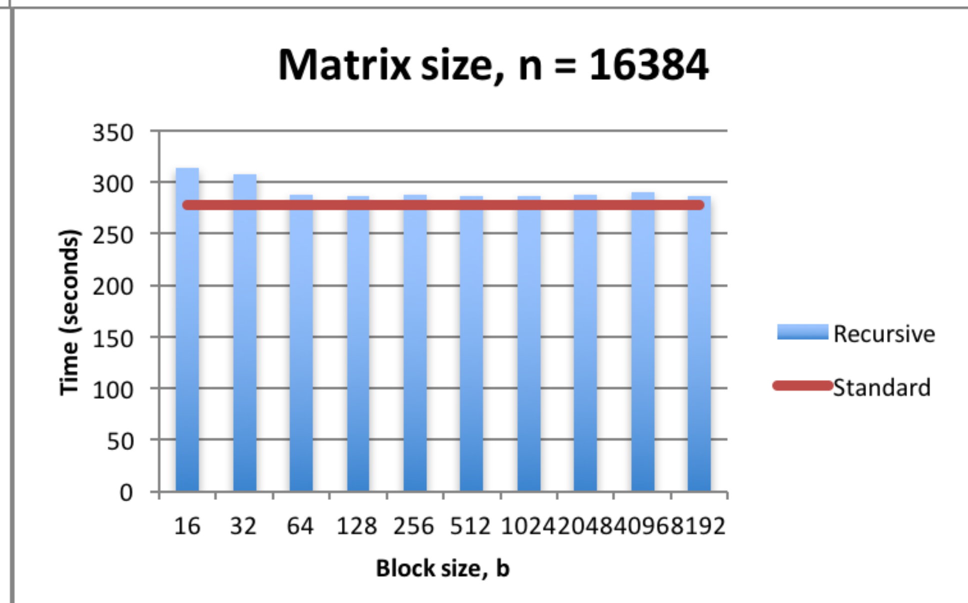



# Platform 3 Timings: FFT



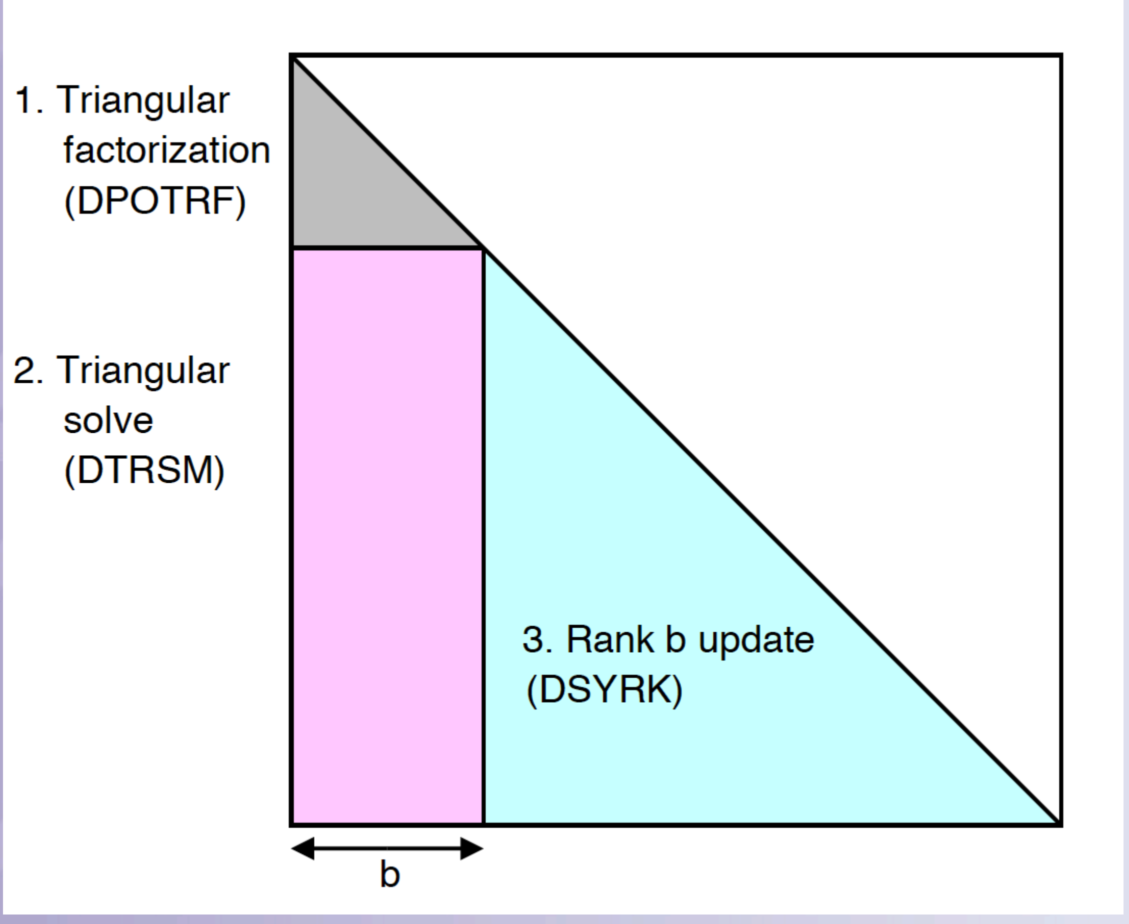



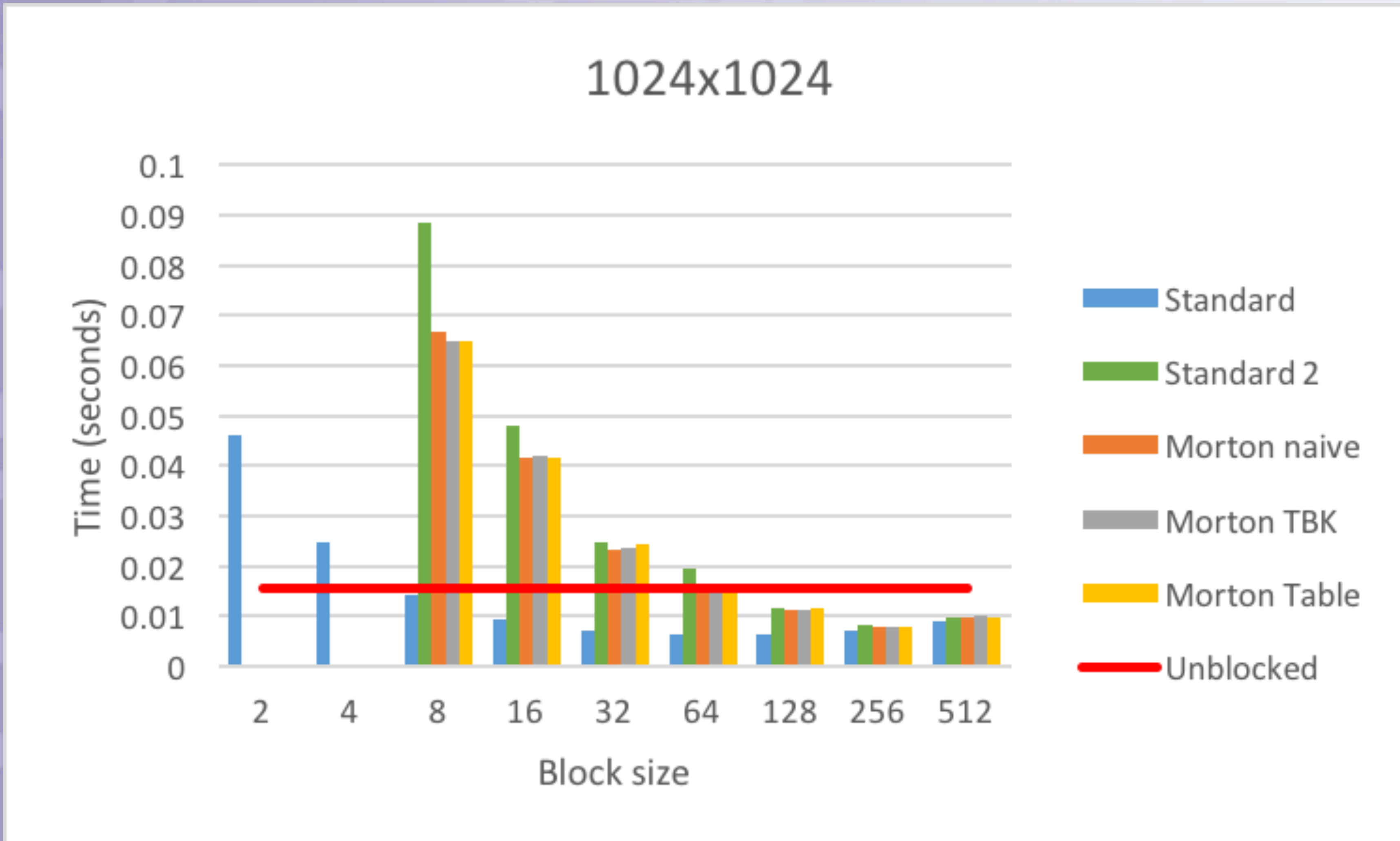
# FFT Results

- Morton ordering doesn’t improve FFT timings by as much as for matrix multiplication.
- Computation to data movement ratio is n for matrix multiply, and log(n) for FFT.
- FFT results are preliminary – still scope for improvements.
- Extra overhead in FFT arises from need to apply partial bit reverse, butterfly, and transpose operations to Morton ordered matrices.

# Cholesky Factorization

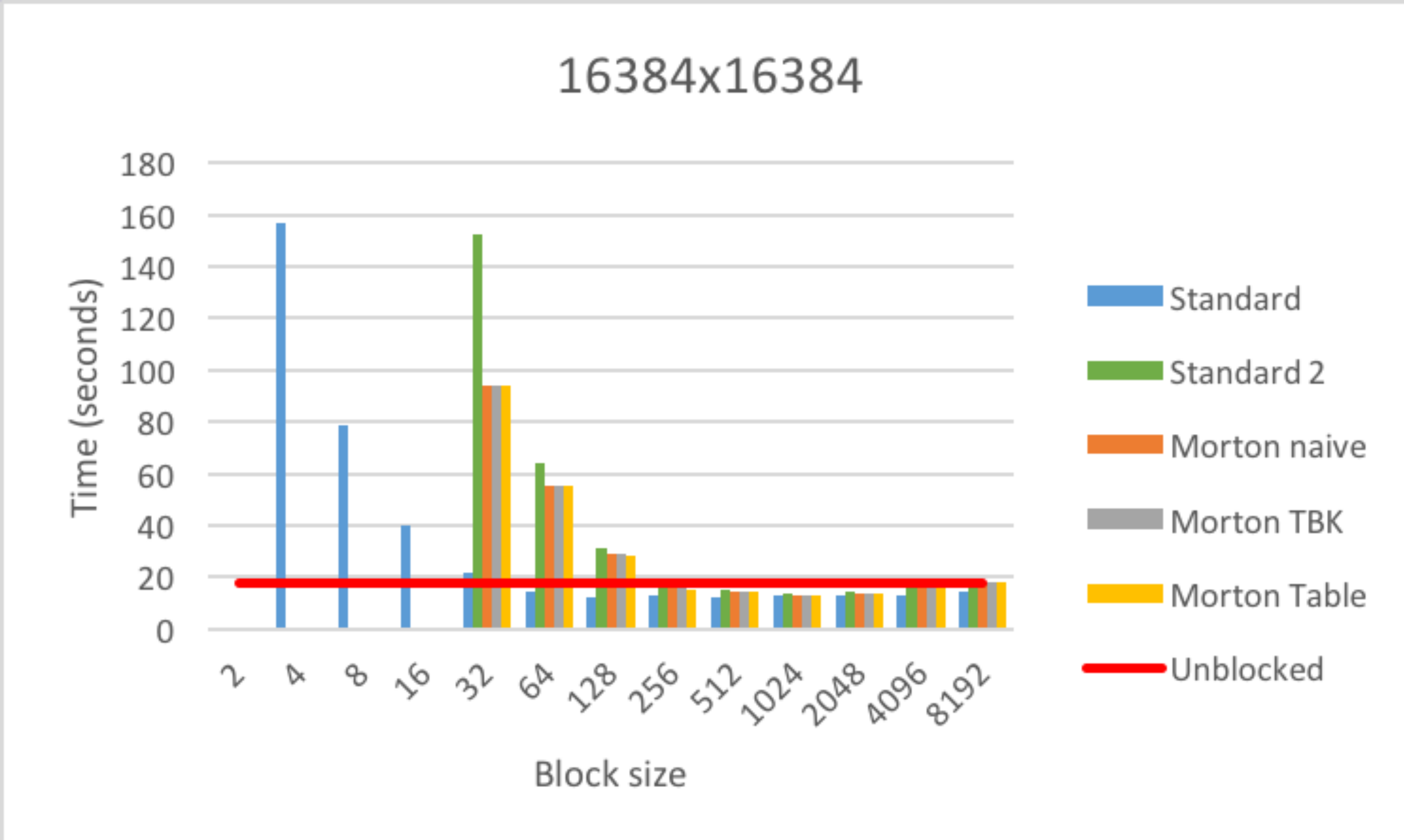


# Platform 2



1024x1024
Time (seconds)
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
2
4
8
16
32
64
128
256
512
Block size
Standard
Standard 2
Morton naive
Morton TBK
Morton Table
Unblocked

# Platform 2



16384x16384
Time (seconds)
180
160
140
120
100
80
60
40
20
0
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
Block size
Standard
Standard 2
Morton naive
Morton TBK
Morton Table
Unblocked

# Concluding Remarks

- Morton ordering and related recursive parallel algorithms may work well when hierarchical memory is handled programmatically. Extend MPI to control the memory hierarchy??
- Need to optimize FFT further.
- Next step will be to incorporate parallelism.
- Will look at frameworks and languages that use recursion to extract parallelism.

# Thank you for your attention.

# Any Questions?