COMP 633 Parallel Computing - Fall 2013 PA2

Duo Zhao <duozhao@cs.unc.edu>

1. Summary:

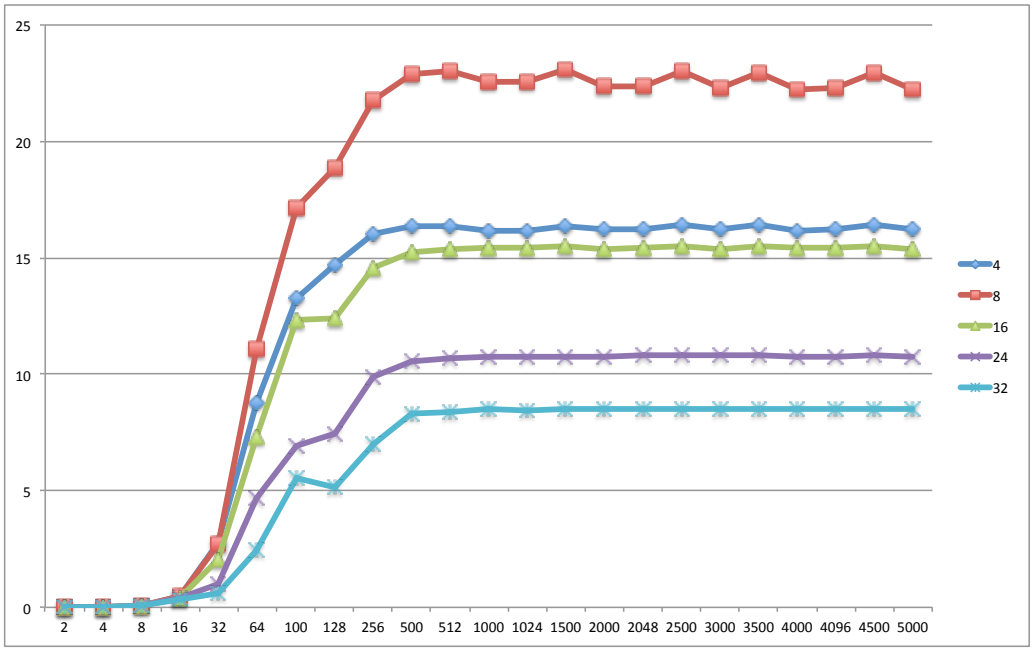
This project works on the problem of matrix multiplication in the form of $M \cdot M^T$. In general, two types block division have been explored in CUDA matrix computation, the square tiled version (v002~007) and the column stripe version (v008, v009) as well as using the cuda BLAS library API to compute the same problem. The experiments show the best performance achieves when the block width is 8. The cublasSgemm_v2 functions is superb to compute this problem. The GFlops is measured in both CPU-GPU transfer time inclusive and exclusive. The dimension is limited to 5000. Up to 20000 dimension is tested for some cases.

2. Implementation Details of Kernel Functions

2.1 cuda_MxMT_v001(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v001(float *d_mr, float *d_m, int d){
    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int j = bdx * blockDim.x + tdx;
    int i = bdy * blockDim.y + tdy;
    if (i < d && j < d){
        float sum = 0;
        for (int k = 0; k < d; k++){
            sum += d_m[i*d+k] * d_m[j*d+k];
        }
        d_mr[i*d+j] = sum;
    }
}
```

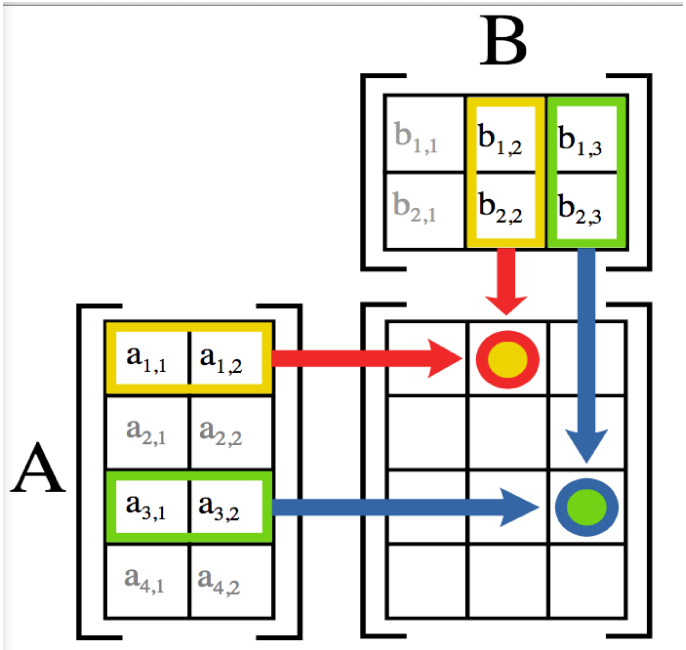
This is the naive version of computing the matrix multiplication simply by mapping each entry in the target matrix to a CUDA thread. Each thread iterate through two rows in the source matrix and performing a vector inner product. Invalid indices are filtered by the if statement. Note in this problem, we are assuming the grid is large enough to hold the entire matrix, otherwise the if-statement is supposed to change to a while statement and reuse each thread to performance warp-up tasks. **For the following implementations, the version axis is in the units of GFlops and the horizontal axis is the dimension of the matrix. Different lines are from different size of blocks.**



2.2 cuda_MxMT_v002(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v002 (float *d_mr, float *d_m, int d){
    float rval = 0;
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];

    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;

    for (int bk = 0; bk < gridDim.y; bk++){
        b_mr[tdx][tdy] = d_m[(bdx*BLOCK_WIDTH + tdx)*d + bk * BLOCK_WIDTH + tdy]; // (bdx*w+tdx, bk*w+tdy)
        b_mc[tdx][tdy] = d_m[(bdy*BLOCK_WIDTH + tdx)*d + bk * BLOCK_WIDTH + tdy];   // (bdy*w+tdx, bk*w+tdy)
        __syncthreads();
        for (int tk = 0; tk < BLOCK_WIDTH; tk++){
            rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
        }
        __syncthreads();
    }

    d_mr[rpos] =  rval;// + 0.1*row + 0.01*col;
}
```

The second version is attempting to load the data from the global memory block-wise before performing computations. Since the problem is essentially performing vector inner product of any two rows (i-th and j-th) and write the result to Row i and Column j of the target matrix. The target matrix is evenly partitioned into tiled blocks. Each matrix block in the target matrix corresponds to a CUDA thread block. Each CUDA thread block iterates over two corresponding rows of the source matrix. It first performs a in-block summation and then moves into the next source matrix block, when the column block index advances by one.



The above figure is taken from wikipedia: (http://en.wikipedia.org/wiki/File:Matrix_multiplication_diagram_2.svg)

2.3 cuda_MxMT_v003(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v003 (float *d_mr, float *d_m, int d){
    float rval = 0;
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];

    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;

    int srcRowR = bdx*BLOCK_WIDTH + tdx;
    int srcRowC = bdy*BLOCK_WIDTH + tdx;
    for (int bk = 0; bk < gridDim.y; bk++){
```

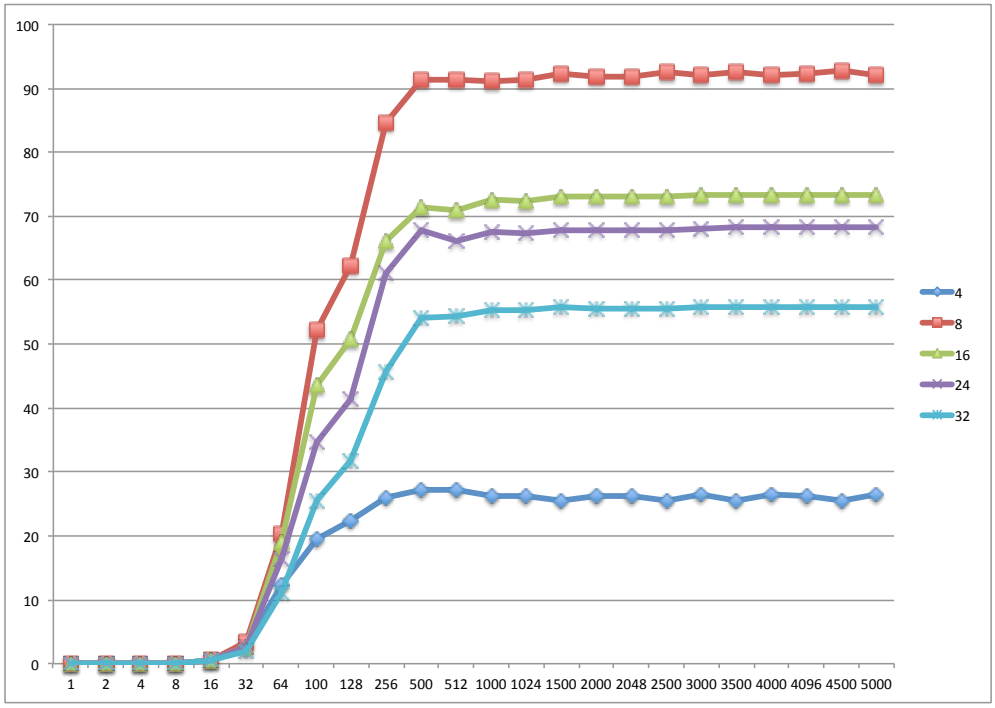
```
        int srcCol = bk * BLOCK_WIDTH + tdy;
        if (srcRowR < d && srcCol < d)
            b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)
        if (srcRowC < d && srcCol < d)
            b_mc[tdx][tdy] = d_m[srcRowC*d + srcCol];   // (bdy*w+tdx, bk*w+tdy)

        __syncthreads();
        if (row < d && col < d){
            for (int tk = 0; tk < BLOCK_WIDTH; tk++){
                if (bk*BLOCK_WIDTH + tk < d)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
            }
        }
        __syncthreads();
    }
    if (row < d && col < d){
        d_mr[rpos] =  rval;// + 0.1*row + 0.01*col;
    }
}
```

The second version only works for the block size divides the matrix dimensions exactly, while the third version generalize the second version to generate matrices and examines the boundary cases carefully. Inappropriate conditional branch may lead to thread divergence and has a negative impact on the overall performance. The version has the problem of unusually thread divergence while taking care of the end of row and almost reduces the performance by half. This version is discard for analysis.

2.4 cuda_MxMT_v004(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v004 (float *d_mr, float *d_m, int d){
    float rval = 0;
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];

    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;

    int srcRowR = bdx*BLOCK_WIDTH + tdx;
    int srcRowC = bdy*BLOCK_WIDTH + tdx;
    for (int bk = 0; bk < gridDim.y; ++bk){
        int srcCol = bk * BLOCK_WIDTH + tdy;
        if (srcRowR < d && srcCol < d)
            b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)
        if (srcRowC < d && srcCol < d)
            b_mc[tdx][tdy] = d_m[srcRowC*d + srcCol];   // (bdy*w+tdx, bk*w+tdy)

        __syncthreads();
        if (row < d && col < d){
            if (bk != gridDim.y - 1) // condition outside the for loop
                for (int tk = 0; tk < BLOCK_WIDTH; tk++)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
            else
                for (int tk = 0; tk < d % BLOCK_WIDTH; tk++)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
        }
        __syncthreads();
    }
    if (row < d && col < d){
        d_mr[rpos] =  rval;// + 0.1*row + 0.01*col;
    }
}
```

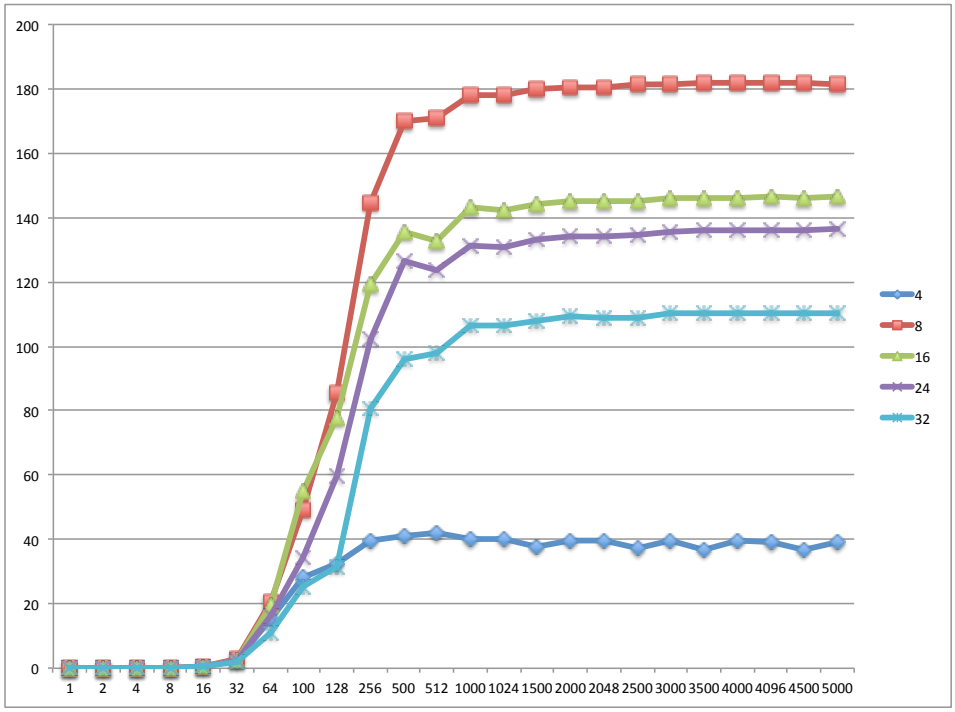
Between the barriers, __syncthreads(), by making branches outside the for loop has little performance impact on the performance, this is an mature version for computation in tiled block format.



2.5 cuda_MxMT_v005(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v005 (float *d_mr, float *d_m, int d){
    float rval = 0;
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];

    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    if (bdx > bdy)
        return;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;
    int rposPair = col * d + row;
    int srcRowR = bdx*BLOCK_WIDTH + tdx;
    int srcRowC = bdy*BLOCK_WIDTH + tdx;
    for (int bk = 0; bk < gridDim.y; ++bk){
        int srcCol = bk * BLOCK_WIDTH + tdy;
        if (srcRowR < d && srcCol < d)
            b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)
        if (srcRowC < d && srcCol < d)
            b_mc[tdx][tdy] = d_m[srcRowC*d + srcCol];   // (bdy*w+tdx, bk*w+tdy)

        __syncthreads();
        if (row < d && col < d){
            if (bk != gridDim.y - 1)
                for (int tk = 0; tk < BLOCK_WIDTH; tk++)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
            else
                for (int tk = 0; tk < d % BLOCK_WIDTH; tk++)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
        }
        __syncthreads();
    }
    if (row < d && col < d){
        d_mr[rposPair] = d_mr[rpos] =  rval;// + 0.1*row + 0.01*col;
    }
}
```

By observing that the target matrix is Symmetric, we discard half of the block computation. Instead, we generate the other half of the target matrix by directly coping from the other half. This technique almost doubles the performance.

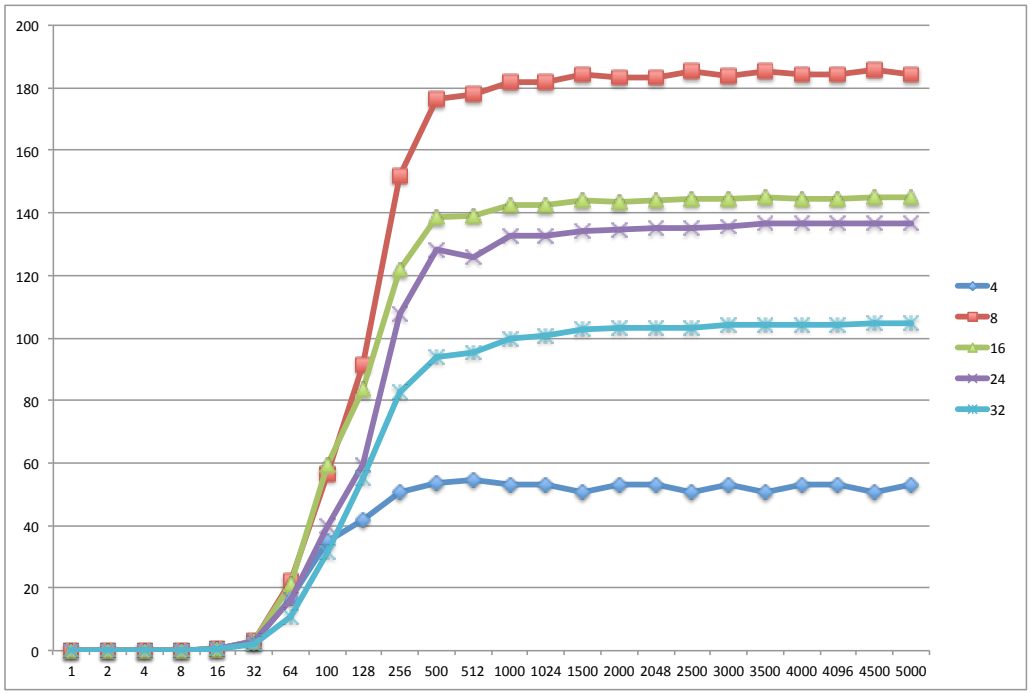


2.6 cuda_MxMT_v006(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v006 (float *d_mr, float *d_m, int d){
    float rval = 0;
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];

    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;
    int rposPair = col * d + row;
    int srcRowR = bdx*BLOCK_WIDTH + tdx;
    int srcRowC = bdy*BLOCK_WIDTH + tdx;
    if (bdx > bdy){
        return;
    }
    else if (bdx < bdy)
        for (int bk = 0; bk < gridDim.y; ++bk){
            int srcCol = bk * BLOCK_WIDTH + tdy;
            if (srcRowR < d && srcCol < d)
                b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)
            if (srcRowC < d && srcCol < d )
                b_mc[tdx][tdy] = d_m[srcRowC*d + srcCol];   // (bdy*w+tdx, bk*w+tdy)

            __syncthreads();
            if (row < d && col < d){
                if (bk != gridDim.y - 1)
                    for (int tk = 0; tk < BLOCK_WIDTH; tk++)
                        rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
                else
                    for (int tk = 0; tk < d % BLOCK_WIDTH; tk++)
                        rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
            }
            __syncthreads();
        }
    else
        for (int bk = 0; bk < gridDim.y; ++bk){
            int srcCol = bk * BLOCK_WIDTH + tdy;
            if (srcRowR < d && srcCol < d)
                b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)

            __syncthreads();
            if (row < d && col < d){
                if (bk != gridDim.y - 1)
                    for (int tk = 0; tk < BLOCK_WIDTH; tk++){
                        rval +=   b_mr[tdx][tk] * b_mr[tdy][tk];
                    }
                else
                    for (int tk = 0; tk < d % BLOCK_WIDTH; tk++){
```

```
                    rval +=  b_mr[tdx][tk] * b_mr[tdy][tk];
                }
            }
            __syncthreads();
        }

    if (row < d && col < d){
        d_mr[rposPair] = d_mr[rpos] =  rval;
    }
}
```

Continuing working on the symmetry of the target matrix, it's easy to observe that we do not have load two rows of blocks, since they are the same. We treat the diagonal blocks as a separate case and it shows slightly performance improvement.

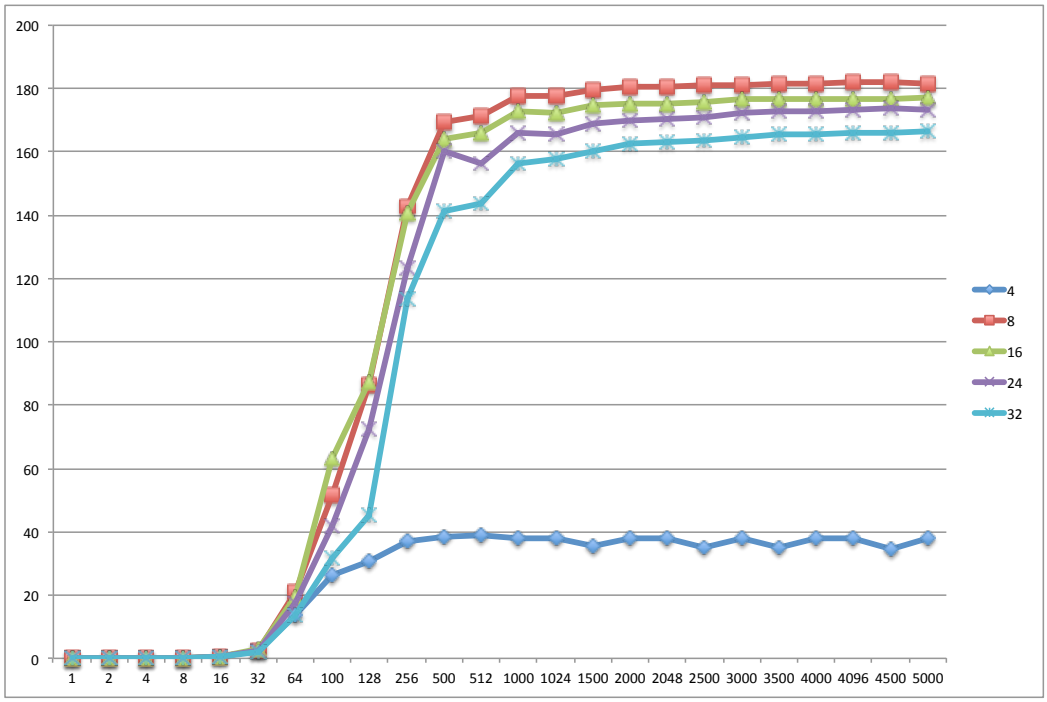


2.7 cuda_MxMT_v007(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v007 (float *d_mr, float *d_m, int d){
    __shared__ float b_mr[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float b_mc[BLOCK_WIDTH][BLOCK_WIDTH];


    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    if (bdx > bdy)
        return;
    int row = bdx * blockDim.x + tdx;
    int col = bdy * blockDim.y + tdy;
    int rpos = row * d + col;
    int rposPair = col * d + row;
    int srcRowR = bdx*BLOCK_WIDTH + tdx;
    int srcRowC = bdy*BLOCK_WIDTH + tdx;

    float rval = 0;
    for (int bk = 0; bk < gridDim.y; ++bk){
        int srcCol = bk * BLOCK_WIDTH + tdy;
        if (srcRowR < d && srcCol < d)
            b_mr[tdx][tdy] = d_m[srcRowR*d + srcCol]; // (bdx*w+tdx, bk*w+tdy)
        if (srcRowC < d && srcCol < d)
            b_mc[tdx][tdy] = d_m[srcRowC*d + srcCol];   // (bdy*w+tdx, bk*w+tdy)

        __syncthreads();
        if (row < d && col < d){
            if (bk != gridDim.y - 1){
                rval +=  b_mr[tdx][0] * b_mc[tdy][0];
                rval +=  b_mr[tdx][1] * b_mc[tdy][1];
                rval +=  b_mr[tdx][2] * b_mc[tdy][2];
                rval +=  b_mr[tdx][3] * b_mc[tdy][3];
                rval +=  b_mr[tdx][4] * b_mc[tdy][4];
                rval +=  b_mr[tdx][5] * b_mc[tdy][5];
                rval +=  b_mr[tdx][6] * b_mc[tdy][6];
                rval +=  b_mr[tdx][7] * b_mc[tdy][7];
            }
```

```
            else
                for (int tk = 0; tk < d % BLOCK_WIDTH; tk++)
                    rval +=  b_mr[tdx][tk] * b_mc[tdy][tk];
        }
        __syncthreads();
    }
    if (row < d && col < d){
        d_mr[rposPair] = d_mr[rpos] =  rval;// + 0.1*row + 0.01*col;
    }
}
```

By some reference, it's said loop unrolling will increase the performance. However, by my experiment, it's not the case in my above unrolling version.

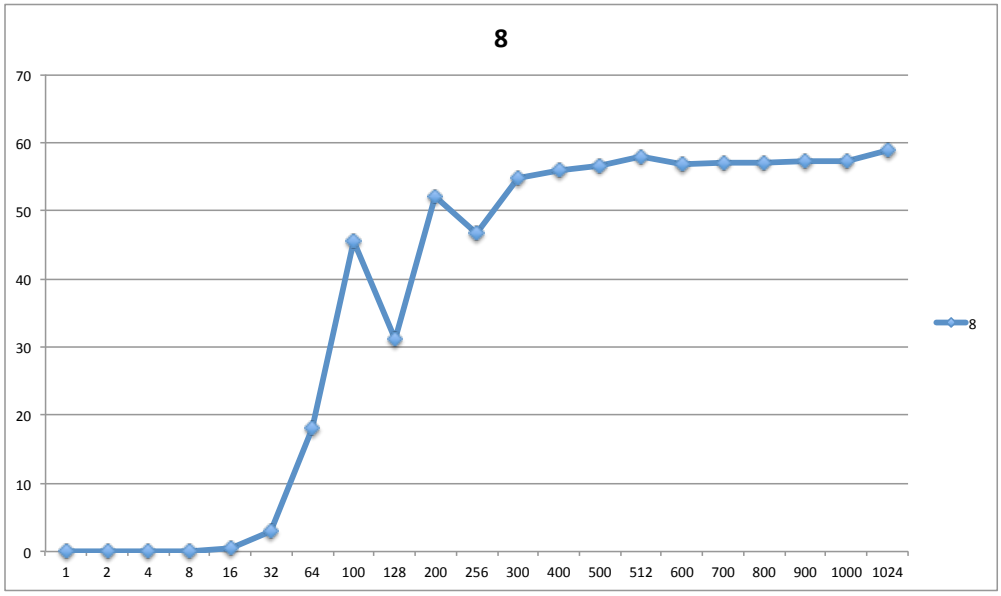


2.8 cuda_MxMT_v008(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v008(float *d_mr, float *d_m, int d){
    __shared__ float bm[1024][BLOCK_WIDTH];
    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int gtdx = bdx * blockDim.x + tdx;
    int gtdy = bdy * blockDim.y + tdy;

    for (int rowStart = 0; rowStart < d; rowStart += blockDim.x){
        int rowRead = rowStart + tdx;
        if (rowRead < d && gtdy < d){
            bm[rowRead][tdy] = d_m[rowRead * d + gtdy];
        }
    }
    if (tdx == 0 && tdy == 0){

    }
    __syncthreads();

    //linear thread block space
    int tid = threadIdx.x * blockDim.y + threadIdx.y;
    int roundEnd = ceilf((float)(d*d)/(blockDim.x * blockDim.y));
    int widthEnd = d % BLOCK_WIDTH;
    for (int r = 0; r < roundEnd; r++){
        float sum = 0;
        int i = (r * blockDim.x * blockDim.y + tid) / d;
        int j = (r * blockDim.x * blockDim.y + tid) % d;
        if (i < d && j < d){
            if (bdy == gridDim.y - 1){
                for (int k = 0; k < widthEnd; k++){
                    sum += bm[i][k] * bm[j][k];
                }
            }
            else{
                for (int k = 0; k < BLOCK_WIDTH; k++){
                    sum += bm[i][k] * bm[j][k];
                }
```

```
            }
            atomicAdd(&d_mr[i*d + j], sum);
        }
    }
}
```

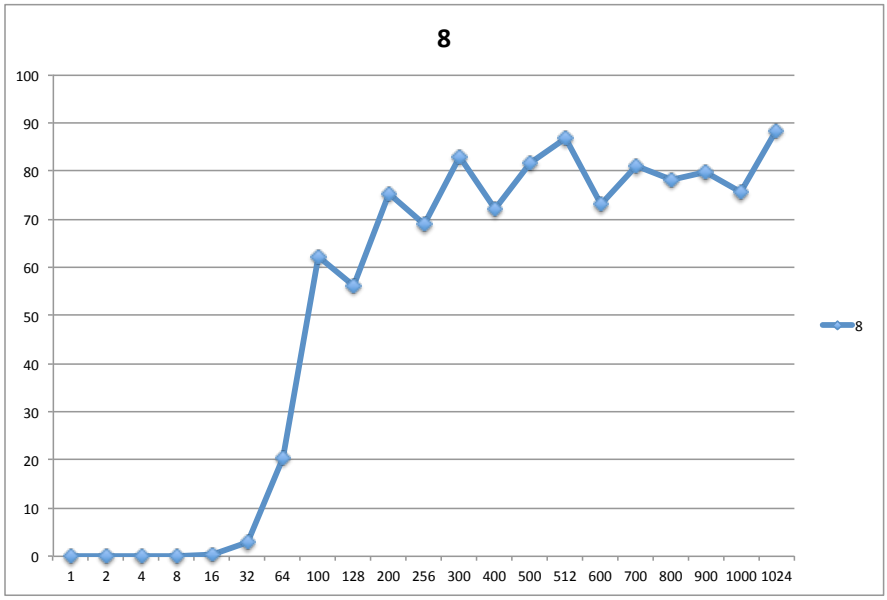
Instead of using the square block, adopting the structure of column blocks in the mid-term. However, my experiment shows it cannot beat the tiled version.



2.9 cuda_MxMT_v009(src/cudaMxMT.cu)

```
__global__ void cuda_MxMT_v009(float *d_mr, float *d_m, int d){
    __shared__ float bm[BLOCK_WIDTH][1024];
    int bdx = blockIdx.x, bdy = blockIdx.y;
    int tdx = threadIdx.x, tdy = threadIdx.y;
    int gtdx = bdx * blockDim.x + tdx;
    int gtdy = bdy * blockDim.y + tdy;

    for (int rowStart = 0; rowStart < d; rowStart += blockDim.x){
        int rowRead = rowStart + tdx;
        if (rowRead < d && gtdy < d){
            bm[tdy][rowRead] = d_m[rowRead * d + gtdy];
        }
    }
    if (tdx == 0 && tdy == 0){

    }
    __syncthreads();

    //linear thread block space
    int tid = threadIdx.x * blockDim.y + threadIdx.y;
    int roundEnd = ceilf((float)(d*d)/(blockDim.x * blockDim.y));
    int widthEnd = d % BLOCK_WIDTH;
    for (int r = 0; r < roundEnd; r++){
        float sum = 0;
        int i = (r * blockDim.x * blockDim.y + tid) / d;
        int j = (r * blockDim.x * blockDim.y + tid) % d;
        if (i < d && j < d){
            if (bdy == gridDim.y - 1){
                for (int k = 0; k < widthEnd; k++){
                    sum += bm[k][i] * bm[k][j];
                }
            }
            else{
                for (int k = 0; k < BLOCK_WIDTH; k++){
                    sum += bm[k][i] * bm[k][j];
                }
            }
            atomicAdd(&d_mr[i*d + j], sum);
        }
    }
}
```

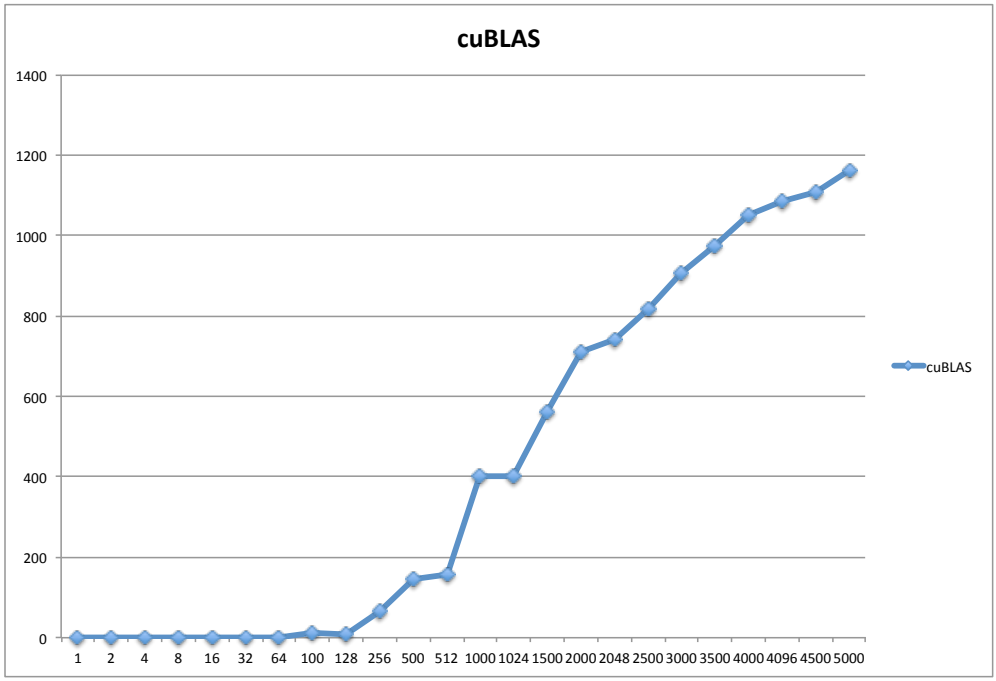
This is an improved version of the v008. While reading the blocks from the global memory, we store it in the shared memory in a transposed way. The reason for this arrangement is while the threads in a thread block reading the shared memory, the aces will be in coalesced pattern, instead of a striped one. This shows a great improvement over v008. However, it's still far from the titled version v005.



2.10 cuBLAS_MxMT_device(src/cudaBLAS_MxMT.cu)

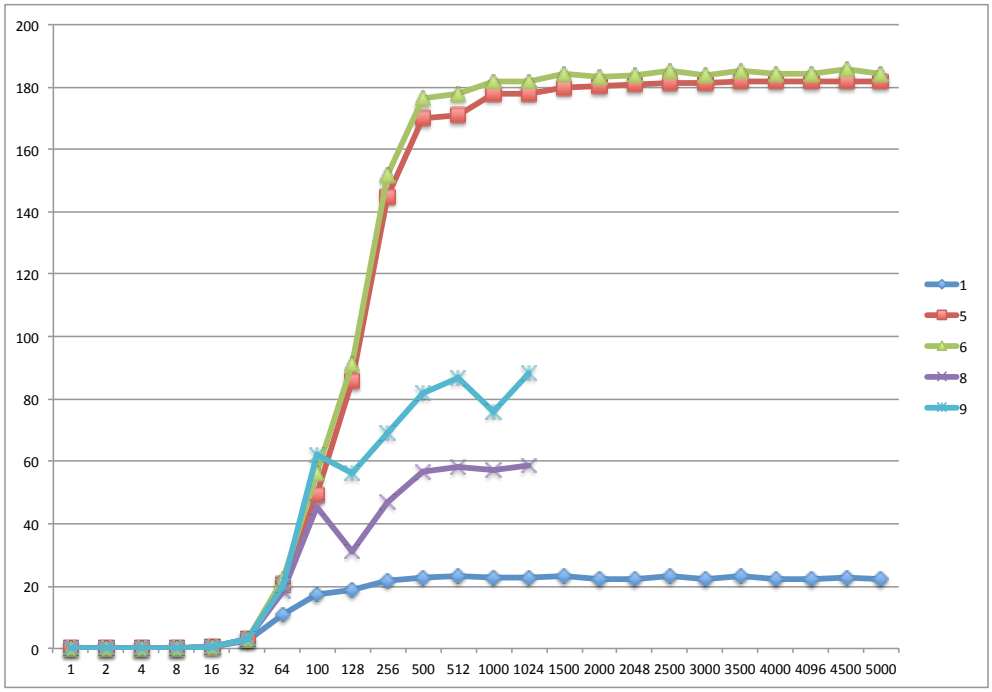
```
void cuBLAS_MxMT_device(float *d_r, float *d_m, int d){
    cublasHandle_t handle;
    cublasCreate(&handle);
    float alpha = 1.0f, beta = 1.0f;
    cublasSgemm_v2(handle,
            CUBLAS_OP_T, CUBLAS_OP_N,
            d, d ,d,
            &alpha,
            d_m, d,
            d_m, d,
            &beta,
            d_r, d);
}
```

This one is by using the cudaBLAS library. The library works pretty well for this problem and has excellent performance than any of my methods above.

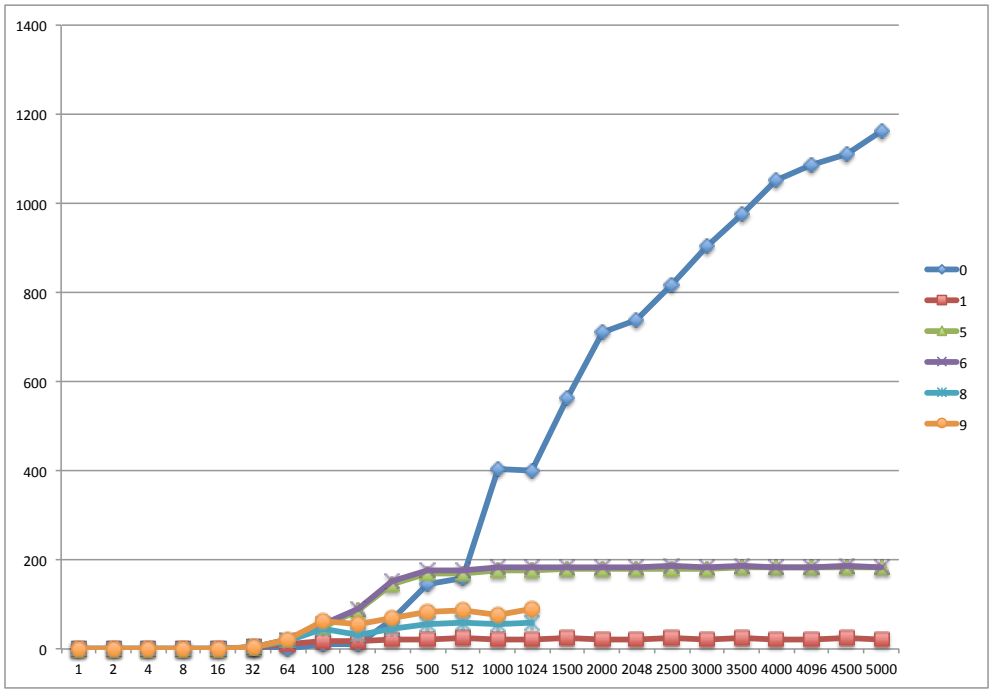


3. Performance Comparison:

For v1, v4, v6, v8, v9 are non-trivial versions, since all these cases got their own best performance with the block size is 8 by 8. The following graph shows the comparing of these methods. The v8 and v9 only support up to 1024 dimensions. The other versions support arbitrary dimensions.



Non of the above implementations perform as well as the cuBLAS_v2 library.



4. Sanity Check

The sanity check for this project is done by comparison with the sequential code. i.e.

```
void seq_MxMT(float *mr, float *m, int d){
    for (int i = 0; i < d; i++){
        for (int j = 0; j < d; j++){
            float sum = 0;
            for (int k = 0; k < d; k++){
                sum += m[i*d+k] * m[j*d+k];
            }
            mr[i*d+j] = sum;
        }
    }
}
```

The error is computed as L2 difference. Note there might be tolerable floating point errors.

```
float MatrixL2Diff(float *mt, float *ms, int d){
    float sum = 0;
    for (int i = 0; i < d * d; i++){
        float diff = mt[i] - ms[i];
        sum += diff * diff;
    }
    return sum;
}
```

Dimension	0	1	5	6	8	9	Grand Total
10	0.000	0.000	0.000	0.000	0.000	0.000	0.000
20	0.000	0.000	0.000	0.000	0.000	0.000	0.000
30	0.000	0.000	0.000	0.000	0.015	0.015	0.031
50	0.000	0.000	0.000	0.000	0.220	0.220	0.440
100	0.000	0.000	0.000	0.000	3.495	3.492	6.987
500	0.000	0.000	0.000	0.000	44.268	46.389	90.657

There are no observable errors for the tiled matrix version, v001, v005, v006. For v008 and v009 there are cumulative errors due floating point operations.

For small dimension, the program can output the computation result for checking.

```
$ 6 7 1 1 1 1
Source Matrix
m={
    {         0,          1,          2,          3,          4,          5,          6},
    {         7,          8,          9,         10,         11,         12,         13},
    {        14,         15,         16,         17,         18,         19,         20},
    {        21,         22,         23,         24,         25,         26,         27},
    {        28,         29,         30,         31,         32,         33,         34},
    {        35,         36,         37,         38,         39,         40,         41},
    {        42,         43,         44,         45,         46,         47,         48}
};
Target Matrix from GPU
m={
    {        91,        238,        385,        532,        679,        826,        973},
    {       238,        728,       1218,       1708,       2198,       2688,       3178},
    {       385,       1218,       2051,       2884,       3717,       4550,       5383},
    {       532,       1708,       2884,       4060,       5236,       6412,       7588},
    {       679,       2198,       3717,       5236,       6755,       8274,       9793},
    {       826,       2688,       4550,       6412,       8274,      10136,      11998},
    {       973,       3178,       5383,       7588,       9793,      11998,      14203}
};
Target Matrix from CPU
m={
    {        91,        238,        385,        532,        679,        826,        973},
    {       238,        728,       1218,       1708,       2198,       2688,       3178},
    {       385,       1218,       2051,       2884,       3717,       4550,       5383},
    {       532,       1708,       2884,       4060,       5236,       6412,       7588},
    {       679,       2198,       3717,       5236,       6755,       8274,       9793},
    {       826,       2688,       4550,       6412,       8274,      10136,      11998},
    {       973,       3178,       5383,       7588,       9793,      11998,      14203}
};
```