Matrix-Vector Multiplication Using Shared and Coalesced Memory Access

Erik Opavsky, Emircan Uysaler

June 12, 2012

Introduction

Matrix-vector multiplication is a fundamental operation in linear algebra. It is useful in countless fields of study, and is applicable to many problems in applied mathematics. The common notation for it is $Ax = y$, where A is a matrix of m rows and n columns, x is a vector of length n, and y is a vector of length m. Multiplying A with x makes each element of y the dot product of the corresponding row of A with x.

The goal of this project is to create a fast and efficient matrix-vector multiplication kernel for GPU computing in CUDA C. This is useful for computation with large matrices, on the scale of many millions of elements. Our project is primarily for use by Professor Jeffrey Blanchard for his work on other projects which involve many large matrix-vector multiplications, and thus need to be as fast as possible for large-scale testing to occur. We are making our software, complete with our testing suite, available for free use by anyone, in accordance with the GNU General Public License.

The baseline we are using to compare our algorithms against is a naive matrix-vector multiplication kernel, written by Professor Blanchard. It works by parallelizing the computations required to find each element of y. In this naive algorithm, there is a thread allocated for every row in A, giving us m total threads. Each thread loops through the columns of A, multiplying each element of its associated row of A with each element of x, and finding the sum of these. Once it has gone through all n columns of A, it has found the element of y for that row.

The naive algorithm is not very fast for multiple reasons. Each thread handles a large section of the matrix (n elements). This means we have fewer threads which each do a lot of work, and in the parallel programming paradigm, it is preferable to have more threads in parallel doing less work. Another reason this algorithm is slow is because it does not utilize CUDA's shared memory feature. Shared memory on the GPU is much faster than the global memory on the GPU. By copying x into shared memory, since it is accessed many times, we can greatly improve our matrix-vector multiplication's performance.

Our algorithm makes improvements to the naive algorithm by only going through smaller chunks of the rows of A with each thread. It also stores values which are used multiple times in shared memory. Finally, our algorithm takes advantage of memory coalescing to get maximum performance. We have three functions: $matVecMul$ which performs $Ax = y$, $matVecMulT$ which performs a transpose multiplication $A^T x = y$ by changing the way which indices of A are called, and $matVecMulTransposed$ which makes a transposed copy of A and then performs $matVecMul$ on it. These algorithms are described in detail below as Block & Multiply, Transposed Block & Multiply, and Transpose THEN Block & Multiply, respectively.

Note that column-major order of flattening the arrays is applied for adapting MATLAB environment.

Algorithms

Block & Multiply

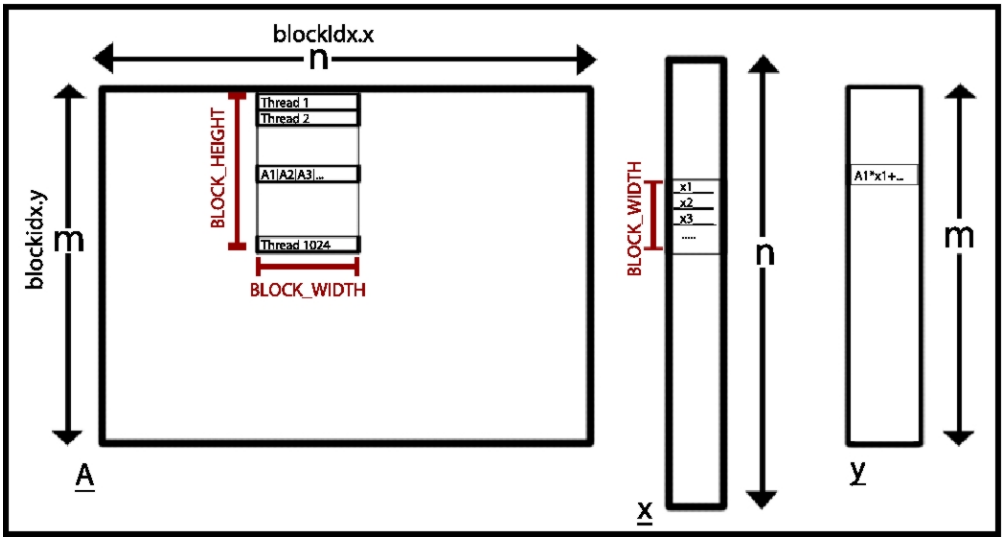

Figure 1: Blocking and threading of A, and multiplication indexing in Block&Multiply Algorithm.

On a high level analysis, we divide the matrix, A, into blocks which are $BLOCK\_HEIGHT$ elements tall, and $BLOCK\_WIDTH$ elements wide. Then, we take each row from the block, r_i, and find its dot product with the corresponding $BLOCK\_WIDTH$ section of the input vector, x. We add this to appropriate place in the output vector, y. After traversing all blocks of A in this fashion, we have $Ax = y$.

The ideal value for $BLOCK\_HEIGHT$ is hardware dependent, as this is the number of threads per block running in parallel. Since we are dealing with very large matrices, we want this value to be as large as possible. On modern GPU's, the maximum number of threads per block is 1024, and therefore 1024 is assigned as $BLOCK\_HEIGHT$.

We theorize that the ideal value for $BLOCK\_WIDTH$ is also hardware dependent. We are aiming to find the optimum value experimentally in the performance evaluation section of our analysis. A single block in our algorithm has $BLOCK\_HEIGHT$ threads, each of which compute the dot product of two vectors of length $BLOCK\_WIDTH$, and then add them to the corresponding index of y. In the process of finding the dot product, each thread goes through multiplication of $BLOCK\_WIDTH$ elements and summing up. In other words, each thread loops through $BLOCK\_WIDTH$ elements which results in additional computations as $BLOCK\_WIDTH$ increase.

Furthermore, we need to initialize all values in y to zero before running our algorithm, since we are doing the cumulative sum of all of the threads in a row in the corresponding index of y. However, for a large matrix of dimensions $m * n$, setting all of the values in the vector y of length m to zero is fairly fast, and does not add much overhead to our program.

On a lower, more CUDA-specific level analysis, we set the number of threads our kernel uses to be the value of $BLOCK\_HEIGHT$ from above, or 1024. We simply use this one dimensional indexing of threads, as that's all we need and can use. We use two dimensional blocking for our program, with the x dimension of our grid being $\frac{n}{BLOCK\_WIDTH}$ rounded up, and the y dimension of our grid being $\frac{m}{BLOCK\_HEIGHT}$ rounded up. This ensures that we will have enough blocks of threads to go through the entire matrix. We now have one thread for every $BLOCK\_WIDTH$ element row segment of A.

Upon execution of our $MatMulKernel$, we first allocate and initialize our shared memory. We also use three shared *int* variables to prevent having to make computations multiple times, and for faster access to these values:*blockElt*, *blockxInd*, and *blockyInd*. Note that these shared memory variables can be accessed from all threads of a block. We want only one thread to have to make these calculations one time, and then all of the threads in the block can quickly access that information from shared memory.

blockElt is the number of element that we loop through in each row. This value is set to $BLOCK\_WIDTH$, except when we reach the rightmost block. If we have a matrix with width that is not a multiple of

$BLOCK\_WIDTH$, we have less elements to loop through in the edging block. In that case, we mod out the matrix width, m, by $BLOCK\_WIDTH$.

blockxInd is the x index of the starting element of the block that is placed on the top-left corner. The index of the current column is given by $blockIdx.x * BLOCK\_WIDTH$. *blockyInd* is the y index of the starting element of the block that is placed on the top-left corner. The index of the current row is given by $blockIdx.y * BLOCK\_HEIGHT$.

After setting these variables, we then have the first *blockElts* threads copy the elements of x which are appropriate to the current block into shared memory. Now all the threads in every block can access these elements quickly. We then make a variable which is local to each thread which holds the running sum of the dot product of this segment of A and the shared memory array segment of x. We then do our dot product with a loop from 0 to *blockElts*, making sure we don't go outside of the boundaries of A. Finally, we do an atomic add of the value we just found to the appropriate index y. We do this atomically because blocks which have the same $blockIdx.y$ may try to add to y at the same time, so *atomicAdd* puts these additions in a queue, so none of them are lost. Once all threads on all blocks finish executing $MatMulKernel$, the answer to $A * x$ will be stored in y.

Algorithm 1 kernel *Block & Multiply*

Input: *out* vector, *in* vector, *a* multiplication matrix, matrixHeight, matrixWidth
if threadIdx.x == 0 **then**
| **if** $(blockIdx.x + 1) * BLOCK\_WIDTH \leq matrixWidth$
| | **then** shared $blockElt = matrixWidth\%$
| | **else** shared $blockElt =$
| **end**
| shared $blockxInd = blockIdx.x * BLOCK\_WIDTH$
| shared $blockyInd = blockIdx.y * BLOCK\_HEIGHT$
end
if $threadIdx.x < blockElt$ **then**
| shared $b[threadIdx.x]$ = in$[blockxInd + threadIdx.x]$
end
cSum=0
threadyInd = $blockyInd + threadIdx.x$
if $threadyInd < matrixHeight$ **then**
| **for** $i < blockElt$; $i++$
| | cSum += $b[i] * a[(blockxInd + i) * (matrixHeight) + (threadyInd)]$
| **end**
| $atomicAdd(out + threadyInd$, cSum)
end

Transposed Block & Multiply

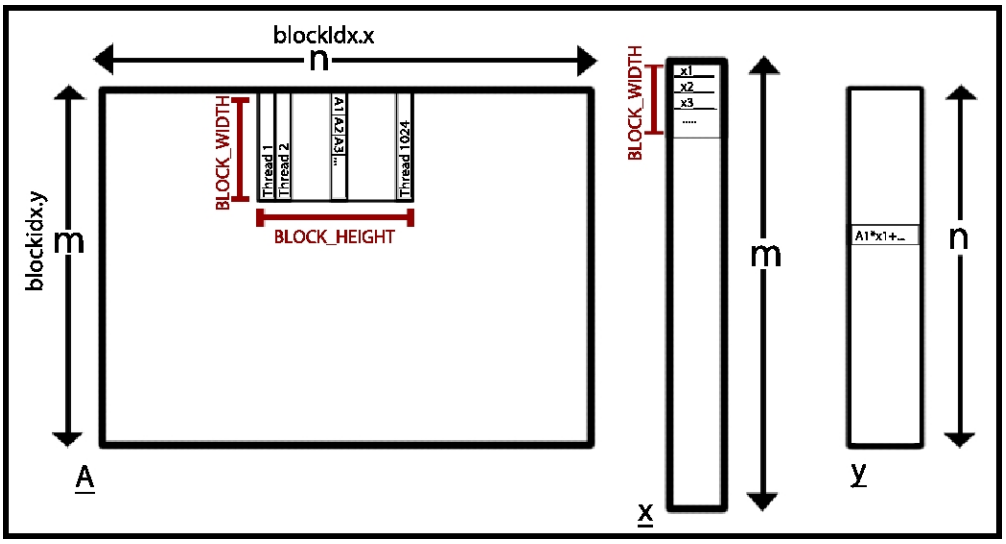

Figure 2: Blocking and threading of A, and multiplication indexing in Transposed Block&Multiply Algorithm.

In this algorithm, we are multiplying A^T with vector x to reach the result y. As a result, the way of blocking differs from the previous *Block&Multiply* algorithm. However, the multiplication approach and shared memory usage remains the same.

We divide the matrix, A, into blocks which are $BLOCK\_WIDTH$ elements tall, and $BLOCK\_HEIGHT$ elements wide, since we multiply x with the columns of A and we want to keep the convention that looping occurs through $BLOCK\_WIDTH$. We use two dimensional blocking , with the x dimension of our grid being $\frac{n}{BLOCK\_HEIGHT}$ rounded up, and the y dimension of our grid being $\frac{m}{BLOCK\_WIDTH}$ rounded up.

As in the *Block&Multiply* algorithm, we find the dot product of a $BLOCK\_WIDTH$ section of a column with the corresponding $BLOCK\_WIDTH$ section of x. We do an atomic add of the value we just found to the appropriate index y. We still do this atomically because blocks which have the same $blockIdx.yx$ may try to add to y at the same time.

Algorithm 2 kernel *Transposed Block & Multiply*

Input: *out* vector, *in* vector, *a* multiplication matrix, matrixHeight, matrixWidth
if threadIdx.x == 0 **then**
| **if** $(blockIdx.y + 1) * BLOCK\_WIDTH \leq matrixHeight$
| | **then** shared $blockElt = matrixHeight\%$
| | **else** shared $blockElt =$
| **end**
| shared $blockxInd = blockIdx.x * BLOCK\_HEIGHT$
| shared $blockyInd = blockIdx.y * BLOCK\_WIDTH$
end
if $threadIdx.x < blockElt$ **then**
| shared $b[threadIdx.x]$ = in$[blockxInd + threadIdx.x]$
end
cSum=0
threadxInd = $blockxInd + threadIdx.x$
if $threadxInd < matrixWidth$ **then**
| **for** $i < blockElt$; $i++$
| | cSum += $b[i] * a[(threadxInd) * (matrixHeight) + (blockyInd + i)]$
| **end**
| $atomicAdd(out + threadxInd$, cSum)
end

Transpose THEN Block & Multiply

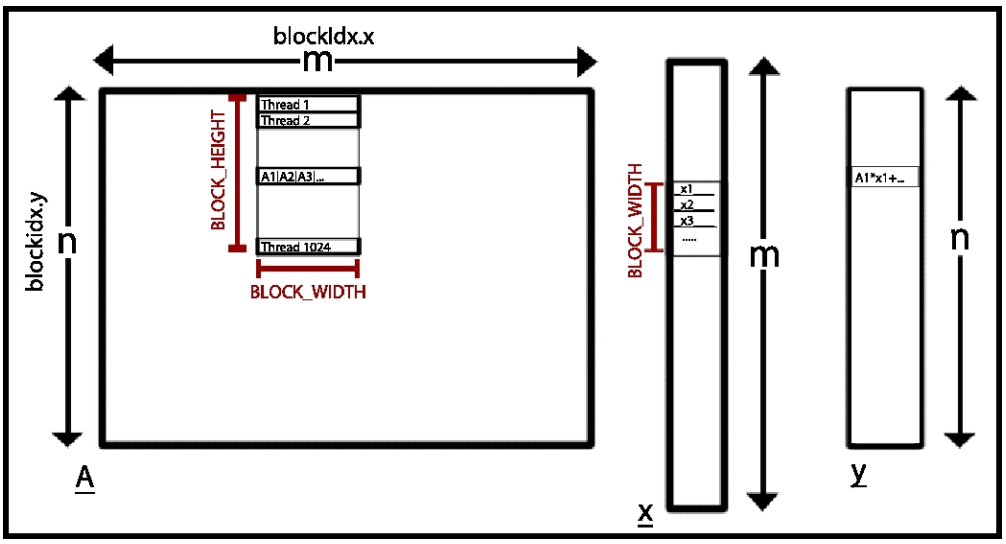

Figure 3: Blocking and threading of A^T, and multiplication indexing in Transposed THEN Block&Multiply Algorithm.

In this algorithm, we initially compute A^T and use *Block&Multiply* algorithm with matrix A^T and vector x to reach the result y. As a result, the way of blocking and use of memory differs from *Transposed Block &Multiply*.

We divide the matrix, A, into blocks which are $BLOCK\_HEIGHT$ elements tall, and $BLOCK\_WIDTH$ elements wide, just like *Block&Multiply* algorithm. We use two dimensional blocking , with the x dimension of our grid being $\frac{m}{BLOCK\_WIDTH}$ rounded up, and the y dimension of our grid being $\frac{n}{BLOCK\_HEIGHT}$ rounded up.

We use the *transpose* function provided in CUDA SDK and store the result in an array A^T same size as A. Finally, we input A^T instead of A in our *Block&Multiply* algorithm.

Analysis of Algorithms

Block & Multiply

In this algorithm, each thread is responsible for multiplying a $BLOCK\_WIDTH$ section of a row in the matrix with the corresponding section of vector and summing the elements up. In every block, consecutive threads are responsible for consecutive rows. Since the matrix we are given is column-ordered, the array indexing goes through the column first. As a result, consecutive threads in a block access consecutive memory addresses. Consequently, this sequence utilizes memory access by coalescing.

Matrix A			
0	*4*	*8*	*12*
1	5	9	13
2	6	10	14
3	7	11	15

Table 1: 2-dimensional column-major array indexing of A in Block&Multiply algorithm.

Memory	*0*	1	2	3	*4*	5	6	7	*8*	9	10	11	*12*	13	14	15

Table 2: 1-dimensional array indexing of A in Block&Multiply algorithm.

Thread#1	*0*	*4*	*8*	*12*
Thread#2	1	5	9	13
Thread#3	2	6	10	14
Thread#4	3	7	11	15

Table 3: Division of array elements into threads in Block&Multiply algorithm.

We use $m*n*sizeof(float)$ of GPU memory for A, $n*sizeof(float)$ for x and $m*sizeof(float)$ for y. In addition, we use $3*sizeof(int)+BLOCK\_WIDTH*sizeof(float)$ on shared memory of each block on GPU. The algorithm goes through the array once, resulting in an O(N) algorithm.

Transposed Block & Multiply

In this algorithm, each thread is responsible for multiplying a $BLOCK\_WIDTH$ section of a column in the matrix with the corresponding section of vector and summing the elements up. In every block, consecutive threads are responsible for consecutive columns. Since the matrix we are given is column-ordered, the array indexing goes through the column. Therefore, a thread needs to access consecutive memory accesses by itself, and memory coalescing is not applied. As a result, memory access is substantially decelerated.

Matrix A			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Table 4: 2-dimensional column-major array indexing of A in Transposed Block&Multiply algorithm.

Memory	*0*	*1*	*2*	*3*	*4*	5	6	7	*8*	9	10	11	*12*	13	14	15

Table 5: 1-dimensional array indexing of A in Transposed Block&Multiply algorithm.

Thread#1	*0*	*1*	*2*	*3*
Thread#2	4	5	6	7
Thread#3	8	9	10	11
Thread#4	12	13	14	15

Table 6: Division of array elements into threads in Transposed Block&Multiply algorithm.

We use $m*n*sizeof(float)$ of GPU memory for A, $n*sizeof(float)$ for x and $m*sizeof(float)$ for y. In addition, we use $3*sizeof(int)+BLOCK\_WIDTH*sizeof(float)$ on shared memory of each block on GPU. The algorithm goes through the array once, resulting in an O(N) algorithm.

Transpose THEN Block & Multiply

In this algorithm, coalesced memory access is applied by initially transposing the matrix and applying *Block&Multiply* algorithm. By transposing the matrix, the transposed consecutive elements are still stored in consecutive rows. Since *Block&Multiply* algorithm loops through a row, we use consecutive memory addresses for consecutive threads in a block.

Matrix A			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 7: 2-dimensional column-major array indexing of A in Transposed THEN Block&Multiply algorithm.

Matrix A^T			
0	*4*	*8*	*12*
1	5	9	13
2	6	10	14
3	7	11	15

Table 8: 2-dimensional column-major array indexing of A^T in Transposed THEN Block&Multiply algorithm.

Memory	*0*	1	2	3	*4*	5	6	7	*8*	9	10	11	*12*	13	14	15

Table 9: 1-dimensional array indexing of A^T in Transposed THEN Block&Multiply algorithm.

Thread#1	*0*	*4*	*8*	*12*
Thread#2	1	5	9	13
Thread#3	2	6	10	14
Thread#4	3	7	11	15

Table 10: Division of array elements into threads in Transposed THEN Block&Multiply algorithm.

We use $m * n * sizeof(float)$ of GPU memory for A, $m * n * sizeof(float)$ of GPU memory for A^T, $n * sizeof(float)$ for x and $m * sizeof(float)$ for y. Note that the memory allocation for the matrix doubles as a result of transpose. In addition, we use $3 * sizeof(int) + BLOCK\_WIDTH * sizeof(float)$ on shared memory of each block on GPU. The algorithm goes through the array twice, resulting in an O(N) algorithm.

Performance Evaluation

In our testing environment we let $n = 2^p$ for $p = 7 : 14$. For each n, we let $m = 2^q$ for $q = 1 : p$. These testing variables are determined considering the most common matrices advised by Professor Jeffrey Blanchard. We run 5 tests with such random m by n matrices. Random matrices are generated by using different seeds on curandGenerateUniform() function of CUDA Random library. Then, we run the kernel and take the average times using cudaEventRecord() and cudaEventElapsedTime() functions.

Optimal BLOCK_WIDTH

BLOCK_WIDTH	8	16	32	64	128	256
Block&Multiply (ms)	0.2132	0.1765	0.1558	0.1511	0.1547	0.1699
Transposed Block&Multiply (ms)	1.8483	1.9505	2.6159	2.5748	2.3792	2.2979
Transposed THEN Block&Multiply (ms)	0.4765	0.4642	0.4410	0.4321	0.4334	0.4501

Table 11: Average times for algorithms using different BLOCK_WIDTH values.

Increasing the size of$BLOCK_WIDTH$ results in a trade-off between computational complexity and memory access. As the size of $BLOCK\_WIDTH$ increases, the size of shared memory used in a block increases. All of the $BLOCK\_HEIGHT$ threads share this section of vector in the multiplication, and less memory access happens. However, increasing $BLOCK\_WIDTH$ directly increases the size of the loop of multiplications and additions done in a kernel. Therefore, we are expecting to find an optimal $BLOCK\_WIDTH$ which results in the fastest time. The results clearly suggest that 64 is the optimum value.

Comparison of Algorithms

Algorithm	Average Time (ms)
Naive Multiplication	0.8221
Block&Multiply	0.1511
Naive Transposed Multiplication	3.1646
Transposed Block&Multiply	2.5748
Transposed THEN Block&Multiply	0.4321

Table 12: Average times for algorithms.

We have faster result using our *Block&Multiply* algorithm than the naive multiplication algorithm in all of 294 test cases. Our *Transposed Block&Multiply* algorithm is faster than Naive Transposed Multiplication in 160 of 296 cases. Our *Block&Multiply* algorithm is about 5.5 times faster than Naive Multiplication. Our *Transposed Block&Multiply* algorithm is about 1.2 times faster than Naive Transposed Multiplication, whereas our *Transposed Block&Multiply* algorithm is about 7.3 times faster.

Conclusion

In this paper, we have introduced three vector-multiplication algorithms using shared memory and exhibited their performance against naive vector-multiplication algorithms. It is clear that the introduction of shared memory speeds up the algorithms significantly, as shown by our *Block&Multiply* algorithm. Furthermore, we came to the conclusion that coalesced memory is at least as important as using shared memory in the speed up process. Using shared memory we can only gain about 80% speed up in the *Transposed Block&Multiply* algorithm. However, if we initially transpose such that we are able to use coalesced memory, we have a 7 times faster algorithm, yet using about two times the original amount of memory. Our algorithm provides the correct results for any matrix dimension. When doing matrix-vector multiplication using floats, we provide fast parallel-processing algorithms that use coalesced and shared memory.