Cuda Supported Matrix Multiplication

Jingtao Song
jis117@eng.ucsd.edu

Zhikang Fan
zhf047@eng.ucsd.edu

1 Models

When doing matrix multiplication using multiple threads, we want to design an algorithm to hide latency, i.e., let threads do other operations while waiting for latency. Latency is usually 20 cycles for arithmetic operations, while around 400 cycles for memory operations [4]. [1] [3] [2] suggests to run more threads per multiprocessor / thread block in the same time to get a higher occupancy. However, [4] points out that higher occupancy does not necessarily leads to a lower latency. We conduct experiments on these ideas and find out the following ideas useful to hide latency.

- More threads
- More work load per thread, thus fewer threads needed
- Using shared memory
- More registers

In this paper, we experiment on these insights.

1.1 Single Thread Single Output (STSO)

To let more threads running in parallel, we split the matrix into multiple blocks, where each entry of a block is computed by a single thread. In other words, for each block, all the entries are calculated in parallel by their own thread. Figure 1 shows an example of a block with 4 x 4 entries.

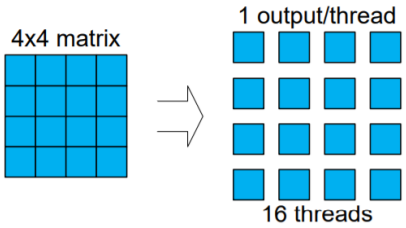

Figure 1: STSO Example: 4x4 block

Generally, given a $N \times N$ matrix, we split it into many *BLOCK_SIZE* x *BLOCK_SIZE* blocks. When *BLOCK_SIZE* is indivisible by N, we add one more row and one more column of blocks. For each thread, check whether or not its entry is within the range of original matrix. If not, do nothing. Note that for all the models we've tried, we use this approach (adding one more row/column of blocks).

To calculate C = A x B, for each time, we load a block (*BLOCK_SIZE* x *BLOCK_SIZE* of entries) of matrix A into shared memory, load a block of matrix B into shared memory, and compute the corresponding block of entries for matrix C. Let $C_{i,j}$, $A_{i,k}$, $B_{k,j}$ denote blocks for matrices C, A, and B respectively. Specifically, STSO is as follows:

STSO

$$For\ k \in \{0, ...BLOCK\_SIZE - 1\}$$
$$C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j}$$

Figure 2 shows the result for STSO:

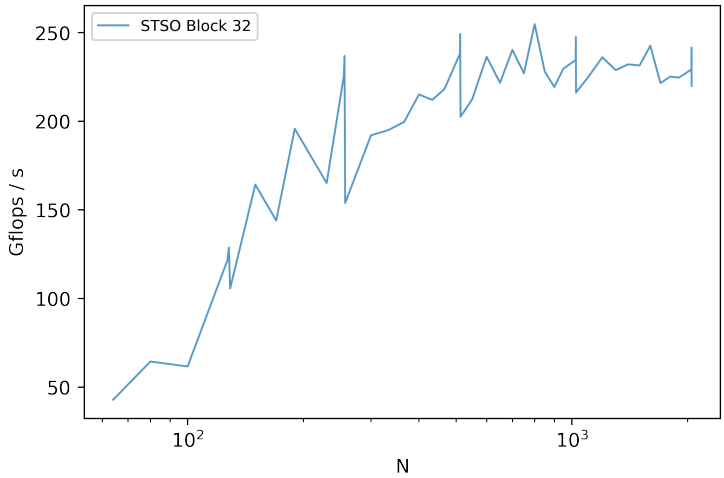

Figure 2: STSO Result: 32x32 block

We can see that the result is far from satisfactory (around 230 GFlops on large Ns). We are utilizing the GPU in a naive way, ss each thread only computes for one single entry of a block, we need huge number of threads to calculate matrix multiplication. Although more threads could hide latency by improving occupancy, [4] points out that with too many threads, we actually decrease instruction-level-parallelism. Not to mention that with too many threads running on, we also increase the number of thread switches, which is also time-consuming.

Besides, by doing a 32x32 block, we're only using $2 \times (32 \times 32) \times 8 = 16KB$ shared memory, which is way beyond Kepler's $48KB$ maximum capacity. We'll see how to improve these problems in the following sections.

1.2 Single Thread Multiple Outputs (STMO)

Although running more threads could get a higher occupancy, switches between threads always create non-negligible latency. If we assign too less work load for each of the thread, there are potentially much more thread switches during computing. One insight is to ask one single thread do more operations, i.e., one thread could compute more outputs. Figure 3 illustrates an example with 2 outputs per thread in a 4 x 4 block.

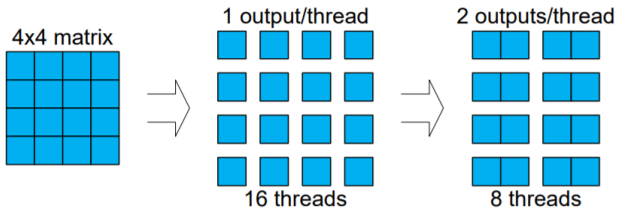

Figure 3: STMO Example: 2 outputs/thread in a 4x4 block

The shared memory part is exactly the same, we still load *BLOCK_SIZE* x *BLOCK_SIZE* sub-matrices into shared memory for matrix A and B respectively. The only difference is, instead of considering one single entry per thread, each thread generates *OP_PER_THREAD* outputs. The algorithm is as follows:

STMO

$$
\begin{aligned}
&Init \\
&\quad cOut = \{0\} \times OP\_PER\_THREAD \\
&\quad STRIDE = BLOCK\_SIZE \ / \ OP\_PER\_THREAD \\
&\quad ty = threadIdx.ty \\
&\quad tx = threadIdx.tx \\
&For\ k \in \{0, ...BLOCK\_SIZE - 1\} \\
&\quad For\ i \in \{0, ...OP\_PER\_THREAD - 1\} \\
&\quad\quad cOut[i] := cOut[i] \ + \ AS_{ty+STRIDE\times i,k} \times BS_{k,tx}
\end{aligned}
$$

where AS and BS are shared memory loaded for each block from A and B.

Figure 4 shows the result for STMO on different *OP_PER_THREAD*:

There is a lot of improvement over STSO. Now the performance reaches around 350 GFlops on large Ns. By doing STMO, the total number of threads drops from N^2 to $\frac{N^2}{OP\_PER\_THREAD}$ and each thread does roughly *OP_PER_THREAD* times more operation than STSO.

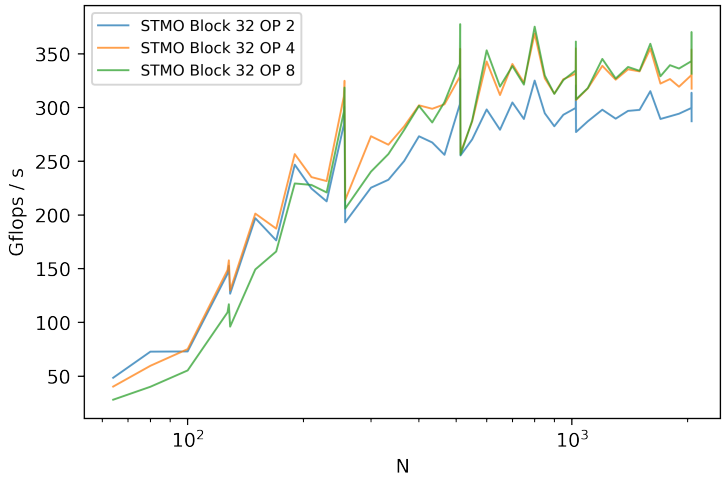

Figure 4: STMO Result: 32x32 block

1.3 Single Thread Multiple Vertical Blocks (STMVB)

Recall what we have tried in **STMO**, we assign each thread with more workload, while different blocks remain somewhat independent. In other words, threads from different blocks have no interactions. Figure 5 shows the running process for STMO. To get result for the red colored block in matrix C, at each time, we multiply the red colored blocks in matrix A and B. then add this multiplication to C. We iterate the red colored block of A inside the orange colored line of A from left to right, while in the meantime, iterate the corresponding red colored block of B in the orange column (of B). Arrays in Figure 5 shows the moving directions for red blocks.

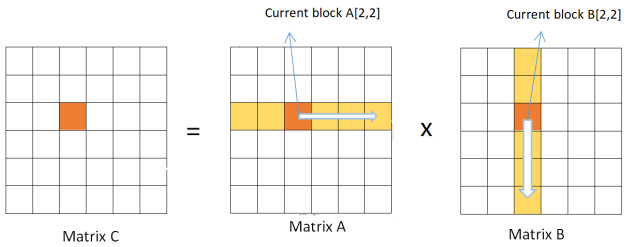

Figure 5: STMO running process

Another interesting idea is that, instead of putting more workload inside a block towards each thread, we could assign workloads from other blocks to the current thread. The difference is that, in STMO, each thread calculate operations in her own block, while in STMVB, each thread compute results based on multiple blocks. Figure 6 gives an example of this process for computing 2 blocks per thread.

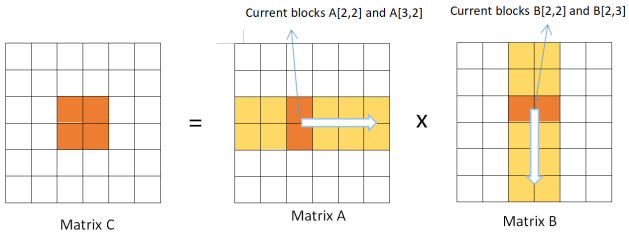

Figure 6: STMVB running process (2 blocks/thread)

In STMO, each time we calculate results for one single block, while in STMVB, say each thread cares for *PARALLEL* blocks in the same time, we generate *PARALLEL* x *PARALLEL* blocks for output. Figure 7 shows the result for STMVB on different *BLOCK_SIZE* and *PARALLEL* parameters.

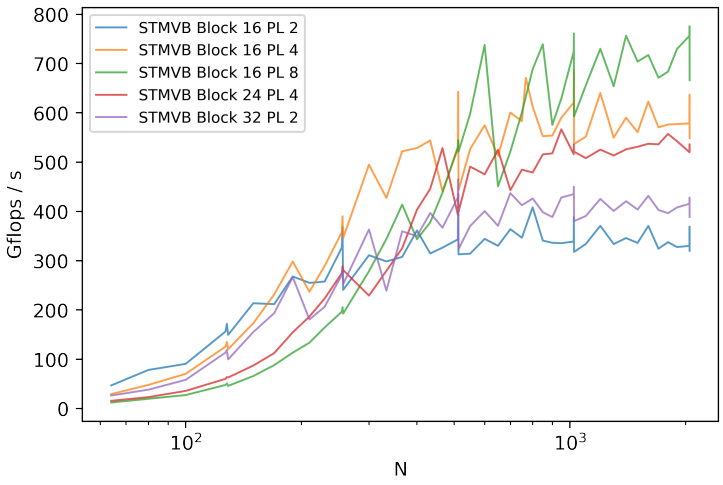

Figure 7: STMVB Result

We can see that the performance is now satisfactory. With *BLOCK_SIZE* = 16 and *PARALLEL* = 8, we get around 700 GFlops on large Ns. Now the GPU uses $2 \times (16 \times 16 \times 8) \times 8 = 32KB$ of shared memory, and the number of threads is $\frac{N^2}{64}$. However, this doesn't work well on small Ns such as 256 or 128, it may because we've reduced the number of threads too much such that it doesn't

achieve enough parallelism by reducing the occupancy. So in the final turn-in version, we picked *BLOCK_SIZE* = 16 and *PARALLEL* = 4 as the parameters in order to achieve a smooth performance.

1.4 Other Attempts

We made many attempts to improve the performance, some worked while some didn't. Here we list several parts that is worth mentioning:

1.4.1 Using predicate

To avoid thread divergence, we used predicate to replace *if* judgments as much as possible.

1.4.2 Common Subexpression Elimination (CSE)

Because each thread calculates *PARALLEL* $\times$ *PARALLEL* values of C in a double loop, there are lots of loop-invariant expressions like start index of matrices, condition value of out-of-bound judgment, etc. Moving these expression outside the loop using CSE helps reduce the number of computation.

1.4.3 Single Thread Multiple Horizontal Blocks (STMHB)

Same idea with STMVB, yet in another fashion. In STMVB, we load *PARALLEL* x 1 blocks from A, load 1x *PARALLEL* blocks from B and compute *PARALLEL* x *PARALLEL* blocks of outputs for C. On the contrary, in STMHB, we load 1 x *PARALLEL* blocks from A and *PARALLEL* x 1 blocks from B, and finally we get 1 block of output for C.

The performance of STMHB is even worse than STMO, which is quite understandable. (1) STMHB only generates one block of outputs, which is no better than STMO; (2) each thread only cares about one entry for each block, leading to a scenario in which workload per thread is actually decreased; (3) loading values from different blocks could be more inefficient compared to loading from one single block.

1.4.4 Transpose Matrix

For all the models we've experimented, the first thing we need to do is to load some sub-matrices from A and B into shared memory. Then we iterate values for each row of AS (sub-matrix of A in shared memory) and each column of BS (sub-matrix of B in shared memory). Based on my personal experience on CPU, matrices are stored in row-major. To use more benefit of memory locality, use values from a row rather than a column could help us get better performance. In this case, as AS is already called in row-manner, we only transpose BS. In English, when we load sub-matrices from B into shared memory, we put $B_{i,j}$ into $BS_{j,i}$ (rather than $BS_{i,j}$).

However, as the title of this section hints, this trick doesn't provide us with any noticeable improvement. This actually results from the different architectures between CPUs and GPUs. In CPUs, we use cache lines to load from memory, while for GPUs, shared memories are already some kind of cache in CPU, thus there is no need to transpose matrices for the use of locality.

2 STMVB with Different Block Sizes

We conduct experiments for STMVB with *BLOCK_SIZE* of 8, 16 and 24. We set *PARALLEL* fixed at 4, which means that each thread computes 4×4 blocks of outputs.

In order to meet the requirements (that our program should keeps running for at least 5 seconds), we did some experiments on the relationship between running time and N. We found a model of $r = 20000e^{-0.0035N}$ such that for every value of N between 64 and 2048, the running time is roughly 5 seconds.

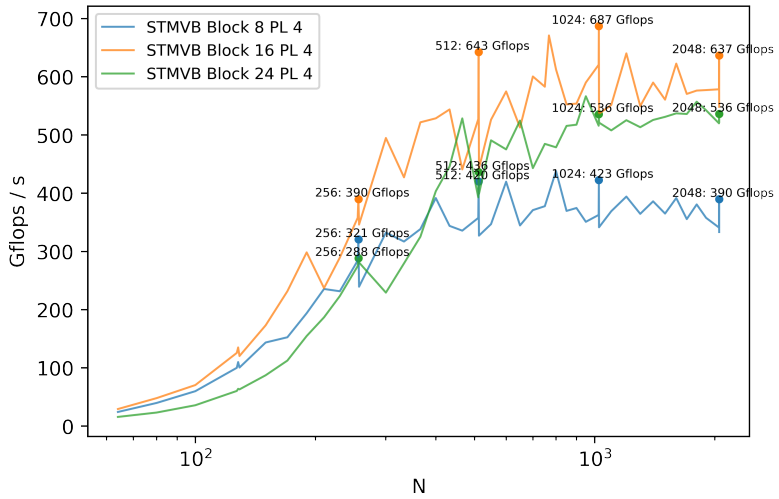

Figure 8: STMVB Result with Different Block Size

Intuitively, larger block sizes can lead to better performance because it utilizes more shared memory. However, too large a block size may degrade the performance. We can see that although *BLOCK_SIZE* = 24 fully uses the $48KB$ shared memory, it doesn't perform the best on average. As previously discussed, large block size weakens the performance on small Ns because it reduces parallelism. In fact, as [1] points out, while one multiprocessor is tackling with a block of threads, CUDA scheduler actually assigns some other blocks to it in a waiting queue. Whenever the multiprocessor sits idle, multiprocessor could calculate blocks in that waiting queue. In a word, multiprocessors could always stay busy by switching blocks when threads in one block have to wait. In our case, when *BLOCK_SIZE* is too large, we have fewer active blocks, leading to a situation where there are not enough blocks in the 'waiting queue'. As multiprocessors become more likely to sit idle, the performance would drop a little bit. Therefore in our final turn-in version we choose *BLOCK_SIZE* = 16.

3 STMVB vs Naive

In this section, we compare the naive solution with our best result achieved on matrices with different sizes. Table 1 shows this difference.

n	Naive (GFlops)	STMVB (GFlops)	(*BLOCK_SIZE*, *PARALLEL*)
256	88	390	(16, 4)
512	93	643	(16, 4)
1024	88	761	(16, 8)
2048	86	776	(16, 8)

Table 1: Naive v.s. STMVB on different matrix sizes

The improvement is amazingly huge, which is comprehensible. (1) Instead of making use of on-chip shared memory, naive solution always loads from global memory, which is much more slower. (2) By adding more workload to each thread, we increase instruction-level-parallelism (ILP). In the meanwhile, as we have reduced the number of necessary threads for computation, we save lots of time in thread switches.

4 STMVB vs BLAS

We use STMVB with *BLOCK_SIZE* = 16 (which performs the best on $N = 1024$) and *PARALLEL* = 4 in this section.

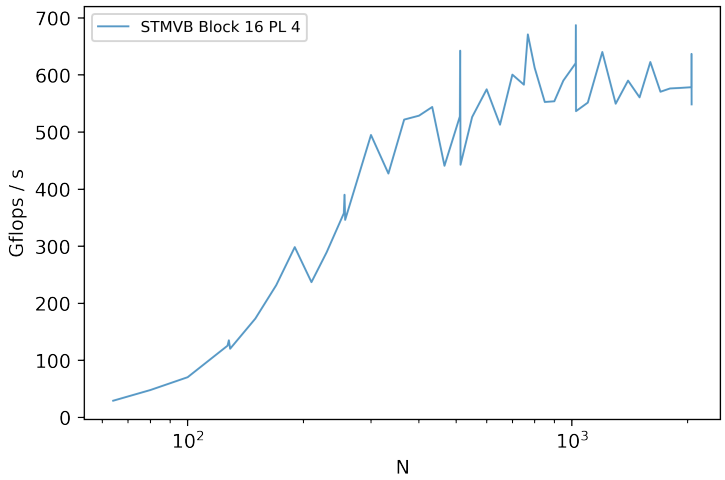

Figure 9: Best Result on N = 1024

n	BLAS (GFlops)	STMVB (GFlops)	n	BLAS (GFlops)	STMVB (GFlops)
64	N/A	29	800	N/A	613
128	N/A	135	900	N/A	554
190	N/A	298	1023	73.7	621
256	5.84	390	1024	73.6	687
300	N/A	495	1025	73.5	537
400	N/A	528	1200	N/A	640
512	17.4	643	1600	N/A	623
600	N/A	575	2047	171	579
700	N/A	601	2048	182	637
768	45.3	671	2049	175	549

Table 2: BLAS v.s. STMVB on different matrix sizes

Our STMVB model outperforms way better than BLAS, especially when we calculate multiplications for large matrices, in which cases we not only have enough threads running in parallel to get a higher occupancy to hide latency, but make better use of instruction level parallelism as well.

There are indeed some irregularities. E.g., with a 1024 x 1024 matrix, we achieved 687 GFlops, while for a matrix with shape of 1025 x 1025, performance drops to 537 GFlops. It's quite easy to understand, as 1024 is divisible by 16 (*BLOCK_SIZE*), while for 1025, we need to pad blocks to compute values for the last row and column. And in these padded blocks, most of the threads are doing nothing useful, since they are out of boundary.

5 Speedup Ratio

In Figure 10 we show the speedup ratio over BLAS.

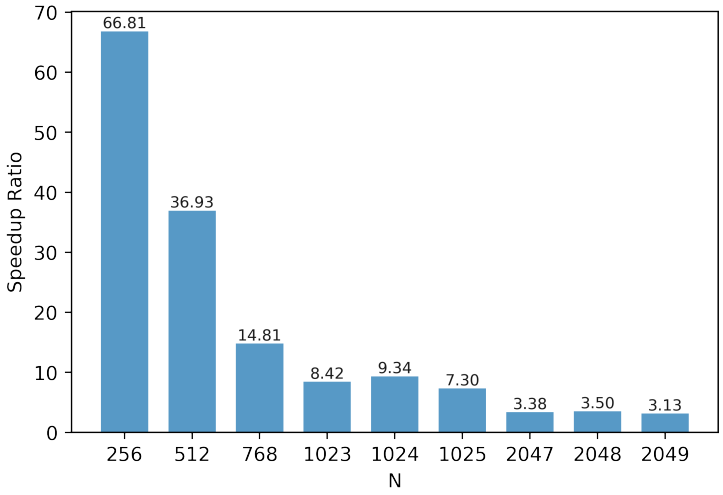

Figure 10: Speedup Ratio

STMVB works dramatically better than BLAS for small matrices (over 66 times better). With matrix size increasing, speedup ratio falls exponentially ($N < 2000$) and finally gets stable (around 3 times better). Recall roofline model, where there is a roof "ceiling" for the best performance. While STMVB has already reached that "ceiling", BLAS is still on its way. Thus speedup ratio first decreases, and finally remains stable.

6 Roofline Model

We take addition and multiplication as one operation. Given a $n \times n$ matrix, operation intensity is number of operations divided by number of memory operations. For our STMVB, this is:

$$q = \frac{n^3}{\frac{n^2}{p^2}(2\frac{np}{b} + p^2)} = \frac{n}{2\frac{n}{pb} + 1}$$

where b is *BLOCK_SIZE* and p is *PARALLEL*. For $n = 1024$, $q \approx 31$.

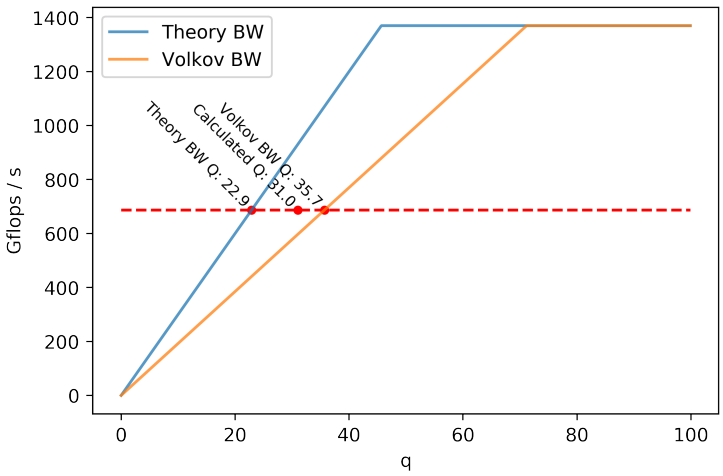

Figure 11: Roofline model

For theory BW, we estimate $q = 22.9$; for Volkov BW, we estimate $q = 35.7$. Our calculated result is closer to the Volkov BW. With a lower BW, our estimate of q becomes larger. As to maintain the

performance with a lower BW, we need to do more calculations per memory operation. An interesting fact about q is that $\lim_{n\to\infty} q = \frac{pb}{2} = 32$, so there is a limit on the q for our model, large n doesn't help to improve the performance.

7 Future Work

We would combine STMO with STMVB and check its performance. That is, each single thread not only calculates values for nearby blocks, but computes multiple outputs in each block as well.

We would also conduct research on adaptive block sizes. That is, apply smaller block size for smaller matrices.

8 Conclusion

In this paper, we experiment different ideas to hide latency for matrix multiplication. Based on our experiments, we state that (1) shared memory is more efficient than global memory; (2) we could hide latency by adding workload for each of the threads, as we increase ILP and reduce the number of thread switches; (3) more active threads could help us improve occupancy, which would lead to a lower latency. It's a trade-off between higher occupancy and more workload per thread.

References

[1] Robert Hochberg. Matrix multiplication with cuda-a basic introduction to the cuda programming model. 2012.

[2] Kirk Hwu. Matrix-matrix product.

[3] Vlad Kindratenko. Ncsa gpu programming tutorial day 3.

[4] Vasily Volkov. Better performance at lower occupancy. 2010.