

Performance engineering for real and complex tall & skinny matrix multiplication kernels on GPUs

Dominik Ernst¹ , Georg Hager¹, Jonas Thies² and Gerhard Wellein¹

The International Journal of High Performance Computing Applications 1–15

© The Author(s) 2020



Article reuse guidelines:

sagepub.com/journals-permissions

DOI: 10.1177/1094342020965661

journals.sagepub.com/home/hpc



Abstract

General matrix-matrix multiplications with double-precision real and complex entries (DGEMM and ZGEMM) in vendor-supplied BLAS libraries are best optimized for square matrices but often show bad performance for tall & skinny matrices, which are much taller than wide. NVIDIA's current CUBLAS implementation delivers only a fraction of the potential performance as indicated by the roofline model in this case. We describe the challenges and key characteristics of an implementation that can achieve close to optimal performance. We further evaluate different strategies of parallelization and thread distribution and devise a flexible, configurable mapping scheme. To ensure flexibility and allow for highly tailored implementations we use code generation combined with autotuning. For a large range of matrix sizes in the domain of interest we achieve at least 2/3 of the roofline performance and often substantially outperform state-of-the-art CUBLAS results on an NVIDIA Volta GPGPU.

Keywords

Performance engineering, complex, tall & skinny, matrix multiplication, CUDA, GPU

1. Introduction

1.1. Tall & skinny matrix multiplications

The general matrix-matrix multiplication (GEMM) is an essential linear algebra operation used in many numerical algorithms and hardware vendors usually supply an implementation that is well optimized for their hardware. In case of NVIDIA, this is part of CUBLAS (NVIDIA, 2019a). However, since these implementations are focused on mostly square matrices, they often perform poorly for matrices with unusual shapes.

This paper covers two types of matrix multiplications with *tall & skinny* matrices, i.e. matrices that are much taller than they are wide. We define *skinny* as having in the range of [1, 64] columns, and *tall* as having more than 10^6 rows. Both types of multiplications involve the two tall & skinny matrices **A** and **B**, with sizes $K \times M$ and $K \times N$, respectively, and K being the long dimension. The small dimensions M and N form a small matrix **C** with size $M \times N$.

The two variants are shown in Figures 1 and 2: The *Tall & Skinny Matrix Transposed times Tall & Skinny Matrix (TSMTTSM)* multiplication $\mathbf{A}^T \mathbf{B} = \mathbf{C}$ and the *Tall & Skinny Matrix times Matrix (TSM)* multiplication $\mathbf{AC} = \mathbf{B}$. We are interested in a highly efficient

implementation of these operations using double precision real and complex data types on the NVIDIA Volta GPGPU, used nowadays in many HPC systems.

1.2. Application

Row-major tall & skinny matrices are the result of combining several vectors to block vectors. *Block Vector Algorithms* are linear algebra algorithms that compute on multiple vectors simultaneously for improved performance. For instance, by combining multiple, consecutive *sparse matrix-vector* (SpMV) multiplications to a *sparse matrix-multiple-vector* (SpMMV) multiplication, the matrix entries are loaded only once and used for the multiple vectors, which reduces the overall memory traffic and consequently increases performance of this memory-bound

¹ Erlangen Regional Computing Center (RRZE), Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

² German Aerospace Center (DLR), Simulation and Software Technology, Köln, Germany

Corresponding author:

Dominik Ernst, Erlangen Regional Computing Center, Martensstraße 1, 91058 Erlangen, Germany.

Email: dominik.ernst@fau.de

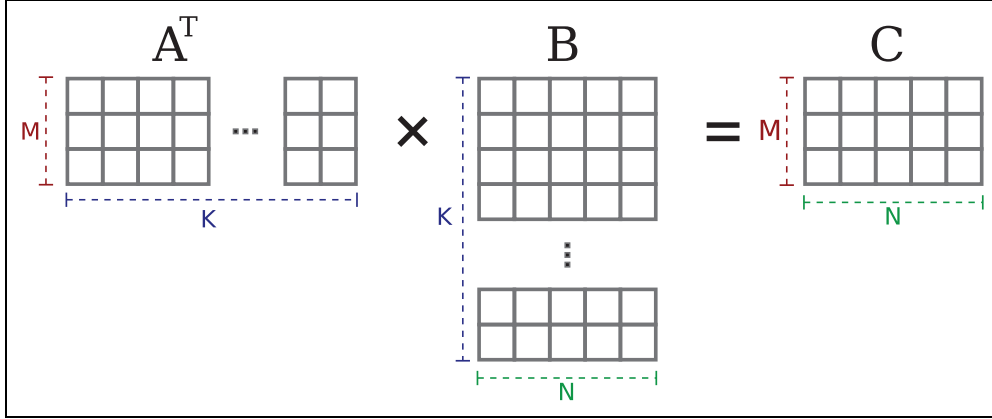


Figure 1. The TSMTTSM operation $\mathbf{A}^T \mathbf{B} = \mathbf{C}$ with \mathbf{A} and \mathbf{B} being tall & skinny matrices. Note that \mathbf{A} is transposed in the illustration.

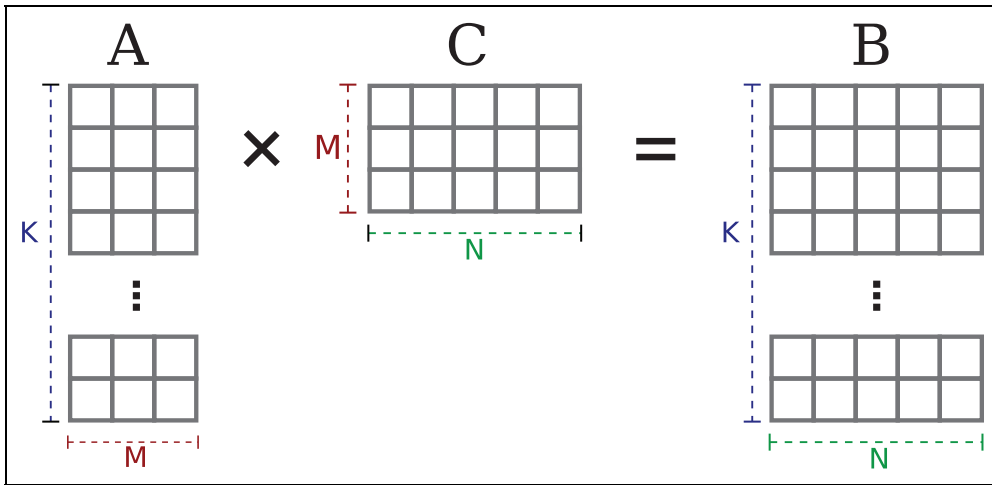


Figure 2. The TSMM operation $\mathbf{A} \mathbf{C} = \mathbf{B}$ with \mathbf{A} and \mathbf{B} being tall & skinny matrices.

operation. This has first been analytically shown by Gropp et al. (1999) and is used in many applications; see, e.g. Röhrig-Zöllner et al. (2015); Kreutzer et al. (2018).

The simultaneous computation on multiple vectors can also be used to gain numerical advantages. This has been shown for block vector versions of the Lanczos algorithm (see Cullum and Donath, 1974), of the biconjugate gradient algorithm (see O’Leary, 1980), and of the Jacobi-Davidson Method (see Röhrig-Zöllner et al., 2015), each of which use block vectors to compute multiple eigenvectors simultaneously. Many such algorithms require multiplications of block vectors. For example, both the TSMTTSM ($\mathbf{A}^T \mathbf{B}$) and TSMM ($\mathbf{A} \mathbf{C}$) occur in classical Gram-Schmidt orthogonalization of a number of vectors represented by \mathbf{B} against an orthogonal basis \mathbf{A} .

1.3. Roofline model

We use the roofline model by Williams et al. (2009) to obtain an upper limit for the performance of these kernels. In all cases, each of the three matrices has to be transferred between the memory and the chip at least once. Even though the directions of data transfers differ between the

kernels, the total data volume does not, as GPUs generally do not need a write-allocate transfer. Therefore the arithmetic intensity I_D is the same for both kernels if M and N are the same. $2MNK$ floating point operations are performed in a matrix-matrix multiplication, so for double precision the arithmetic intensity assuming $K \gg M, N$ and $M = N$ is

$$\begin{aligned}
 I_D &= \frac{2MNK}{(MK + NK + MN) \times 8 \text{ byte}} \frac{\text{flop}}{\text{byte}} \\
 &\stackrel{K \gg M, N}{\approx} \frac{2MN}{(M + N) \times 8 \text{ byte}} \frac{\text{flop}}{\text{byte}} \\
 &\stackrel{M=N}{=} \frac{M}{8} \frac{\text{flop}}{\text{byte}}.
 \end{aligned} \tag{1}$$

In this symmetric case, the arithmetic intensity grows linearly with M . We will show measurements only for this symmetric case, although the nonsymmetric case is not fundamentally different, with the intensity being proportional to the harmonic mean of both dimensions and consequently dominated by the smaller number. If the achievable memory bandwidth is b_s (see below in Section

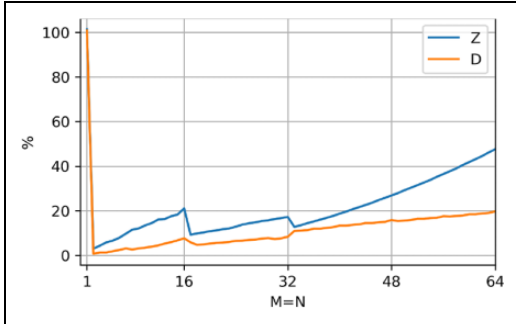


Figure 3. Percentage of roofline-predicted performance achieved by CUBLAS for the *TSMTTSM* kernel in the range $M = N \in [1, 64]$, complex (Z) and real (D) double precision, on a Tesla V100-PCIe-16GB.

1.6), the model predicts $P_{\max} = \min(I \times b_s, P_{\text{peak}})$ as an absolute upper performance limit. In the case of complex numbers, the data volume increases by $2\times$ and the number of floating-point operations by $4\times$, resulting in a doubled arithmetic intensity $I_Z = \frac{M}{4}$ flop/byte.

With proper loop optimizations in place, the GEMM is usually considered a classic example for a compute-bound problem with high arithmetic intensity. However, at $M, N = 1$, the arithmetic intensity of $1/8$ flop/byte is far to the left of the roofline knee of modern compute devices (typical values ranging from 5 flop/byte to 17 flop/byte) and strongly memory bound. This is not surprising given that a matrix multiplication with $M, N = 1$ is the same as a dot product. At the other end of the considered spectrum, at $M, N = 64$, the arithmetic intensity is 8 flop/byte, which is close to the roofline knee of a V100 GPU (see below in Section 1.6). Consequently, the performance character of the operation changes from extremely memory bound at $M, N = 1$ to simultaneously memory and compute bound at $M, N = 64$. An implementation with perfect performance thus needs to fully utilize the memory bandwidth at all sizes and additionally reach peak floating point performance for the large sizes. The very different performance characteristics make it hard to write an optimal implementation for both ends of the spectrum, i.e. different optimizations and specialization is required for both cases.

It is possible to judge the quality of an implementation’s performance as the percentage of the roofline limit. This metric is shown for CUBLAS in Figures 3 and 4, where the ratio of measured and roofline performance is plotted as a function of the matrix width. There is very little performance improvement headroom for CUBLAS’ *TSMM* implementation for real-valued matrices, but there is some opportunity for complex matrices. For the *TSMTTSM* kernel, there is a $2\times$ to $50\times$ gap to the upper limit, apart from $M, N = 1$, where NVIDIA obviously implemented a special case handling. Similarly to the BLAS nomenclature, we use the shorthand “D” for double precision real values and “Z” for double precision complex values.

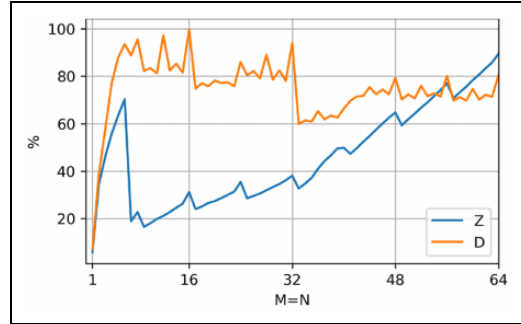


Figure 4. Percentage of roofline-predicted performance achieved by CUBLAS for the *TSMM* kernel in the range $M = N \in [1, 64]$, complex (Z) and real (D) double precision, on a Tesla V100-PCIe-16GB.

1.4. Contribution

This paper presents the necessary implementation techniques to achieve near-perfect (i.e., close to roofline) performance for two tall & skinny matrix-matrix multiplication variants on an NVIDIA V100 GPGPU with real- and complex-valued matrices.

To this end, two parallel reduction schemes are implemented and analyzed as to their suitability for small matrices.

A code generator is implemented that produces code for specific matrix sizes and tunes many configuration options specifically to that size. This allows to precompute many indexing and control flow expressions at compile time. As a result, our implementation outperforms state-of-the-art vendor implementations for most of the parameter range.

1.5. Related work

This work is an extended version of Ernst et al. (2020). In comparison to that paper we have added a different variant of matrix-matrix multiplication (*TSMM*), added a more in-depth performance analysis, extended the analysis to double precision complex data types, and examined a new *TSMTTSM* thread mapping scheme.

CUBLAS is NVIDIA’s BLAS implementation. The GEMM function interface in BLAS only accepts column-major matrices, but our inputs are row-major. The memory contents of a row-major tall & skinny matrix (TSM) \mathbf{A} can be reinterpreted as its transposed column major matrix version $\tilde{\mathbf{A}}$ without any computation. It is then possible to use the BLAS interface to compute $\tilde{\mathbf{A}}\tilde{\mathbf{B}}^T = \tilde{\mathbf{C}}^T \Leftrightarrow \mathbf{A}^T\mathbf{B}^{TT} = \mathbf{C}^{TT} \Leftrightarrow \mathbf{A}^T\mathbf{B} = \mathbf{C}$. This works equivalently for *TSMM* by executing $\tilde{\mathbf{C}}\tilde{\mathbf{A}} = \tilde{\mathbf{B}} \Leftrightarrow \mathbf{C}^T\mathbf{A}^T = \mathbf{B}^T \Leftrightarrow \mathbf{AC} = \mathbf{B}$. All shown data was measured with version 10.1 of CUDA and *CUBLAS*. We also checked for improvements in version 10.2 of *CUBLAS*, but no deviations larger than 1% were found.

CUTLASS (NVIDIA, 2019b) is a collection of primitives for multiplications especially of small matrices, which can be composed in different ways to form products of larger

matrices. One of these is the `splitK` kernel, which additionally parallelizes the inner summation of the matrix multiplication to increase parallelism for the *TSMTTSM* kernel. We adapted the “`06_splitK_gemm`” code sample from the library for benchmarking.

TSM2 (Chen et al., 2019) is another implementation that meets the challenges of tall & skinny matrix multiplication, albeit for a different application case (in our nomenclature it would be *MTSM*, the multiplication of a large, square matrix with a TSM). Some of the challenges like low arithmetic intensity, data reuse, and the need for a flexible thread mapping scheme are the same.

The Strassen algorithm (Strassen, 1969) for matrix-matrix multiplications was profitably employed by Huang et al. (2016, 2020) on CPUs and GPUs even for comparatively small matrices. However, we decided against employing the Strassen algorithm, because, as the authors of the latter paper point out, “it still trades memory operations (mops) for floating point operations (flops).” The extremely non-square nature of the matrices examined in our work makes the kernels memory bound; hence, the trade-off offered by the Strassen algorithm, i.e. fewer floating point operations in exchange for higher memory data volume, is unfavorable.

1.6. Hardware

In this work we use NVIDIA’s V100-PCIe-16GB GPGPU (Volta architecture) with CUDA 10.1. The hardware data was collected with our own CUDA micro benchmarks, which are available at Ernst (2019) together with more detailed data.

1.6.1. Occupancy. The V100-PCIe-16GB GPU consists of 80 Streaming Multiprocessors (SM), each of which has four quadrants with a scheduler and execution units. Similarly to CPUs with simultaneous multi-threading (SMT), each scheduler does not run just a single warp at a time but selects from up to 16 warps to schedule the next instruction. The large number of warps to pick from decreases the chance that there is no warp that can currently execute because of dependencies. The ratio of active warps on an SM to the maximum number of active warps supported by the SM is called *occupancy*. It is limited by the resources required by the warps, first and foremost the number of registers. The compiler can allocate a different number of registers for each program individually. While the compiler tries to minimize the number of allocated registers in order to maximize occupancy, this is often impossible beyond some point without spilling of registers to memory or generating non-optimal machine code.

For the maximum occupancy of 100% (64 warps per SM or 16 per quadrant), each thread must not allocate more than 32 registers. At the maximum amount of 256 registers per thread, only eight warps per SM or two warps per quadrant can be run simultaneously.

Table 1. Measured memory bandwidth on a Tesla V100-PCIe-16GB of a read-only kernel with different amount of load parallelism (ILP) and occupancies.

occupancy	ILP, Gbyte/s		
	1	4	16
1 block, 4 warps	3.0	10.1	16.3
6.25%	228	629	815
12.5%	419	824	877
25%	681	872	884
50%	834	884	887
100%	879	891	877

1.6.2. Memory bandwidth. Whereas the *TSM* operation has a read and a write stream and fits well to the “scale” kernel from the STREAM benchmarks (McCalpin, 1995), the *TSMTTSM* is read-only. We thus use a thread-local sum reduction to estimate the achievable memory bandwidth b_s (see Table 1). Read-only has a much higher maximum ceiling of about 880 Gbyte/s, compared to 820 Gbyte/s for a “scale” kernel. Maximum bandwidth is only attainable with sufficient parallelism, either through high occupancy or instruction level parallelism (ILP) in the form of multiple read streams, achieved here through unrolling.

1.6.3. Floating-point throughput. The V100 can execute one 32-wide double precision (DP) floating point multiply add (FMA) per cycle on each of its 80 streaming multiprocessors (SMs) and runs at a clock speed of 1.38 GHz for a DP peak of $80 \times 32 \times 2 \times 1.38 \text{ Gflop/s} = 7066 \text{ Gflop/s}$. One SM quadrant can process one instruction that is 32 warp lanes wide every four cycles at a latency of eight cycles. Full throughput can already be achieved with a single warp per quadrant if instructions are independent.

1.6.4. L1 cache. The L1 cache is instrumental in achieving the theoretically possible arithmetic intensity. Though load and DP FMA instructions have the same throughput of $1/\text{cy}/\text{SM}$, the actual L1 cache bandwidth of one 128-byte cache line per cycle means that the actual load instruction throughput is dependent on the number of touched cache lines. For example, a 32-wide, unit-stride DP load touches 2 cache lines and therefore takes two cycles. For that access pattern, the floating point to load instruction ratio would need to be at least 2:1 to attain peak performance.

1.6.5. Shared memory. The Shared Memory uses the same physical structure on the chip as the L1 cache. It has the same bandwidth, but lower access latency than the L1 cache.

2. General implementation strategies

2.1. Code generation

A single implementation cannot be suitable for all matrix sizes. In order to engineer the best code for each size, some

Listing 1. Naive matrix-matrix multiplication (MMM) pseudo code for $\mathbf{C} = \mathbf{A}^T \mathbf{B}$.

```

for m = 0...M:
  for n = 0...N:
    for k = 0...K:
      C[m][n] += A[k][m] * B [k][n]

```

form of meta programming is required. C++ templates allow some degree of meta programming but are limited in their expressiveness or require convoluted constructs. Usually the compiler unrolls and eliminates short loops with known iteration count in order to reduce loop overhead, to combine address calculations, to avoid indexed loads from arrays for the thread-local results, to deduplicate and batch loads, and much more. Direct generation of the intended code offers more control, however. For example, when using a thread count per row that is not a divisor of the matrix width, some threads would need to compute fewer results than others. This is achieved via guarding if statements around computations that would access out-of-bounds elements. These can be omitted wherever it is safe, i.e. all threads compute a valid value, in order to not compromise performance for even, dividing thread mappings. We therefore use a code generating script in Python, which allows to prototype new techniques much quicker and with more control. Many different parameters can be configured easily and benchmarked automatically, for example whether leap frogging and unrolling (see below in Section 3) are used, how the reduction is performed, and what thread mapping to set. The same reasoning for code generation is made by Herrero and Navarro (2006), Huang et al. (2017), and Benson and Ballard (2015).

2.2. Thread mapping options

The parallelization scheme, i.e. the way in which work is mapped to GPU threads, plays an important role for data flow in the memory hierarchy. The canonical formulation of an MMM is the three-level loop nest shown in Listing 1.

The iteration space of an MMM can be visualized as a cuboid spanned by the outer product of the two matrices being multiplied. For the *TSMTTSM* (Figure 5), the matrices \mathbf{A} and \mathbf{B} span the cube, and reduction along the long axis K results in the matrix \mathbf{C} . For the *TSMM* (Figure 6), the cube is flipped on its side, so the the matrices \mathbf{A} and \mathbf{C} span the cube and a reduction along the short side M results in \mathbf{B} .

This representation allows to visualize the locality of data transfers. Looking at a slice of the cube perpendicular to the long K axis spanned by one row of \mathbf{A} and \mathbf{B} , as depicted in Figures 7–9, shows all the data uses and computations. Each such slice contains $M \times N$ cells, which correspond to one FMA each, and requires the transfer of one row each of \mathbf{A} and \mathbf{B} , causing a data transfer of $M + N$ elements. The arithmetic intensity

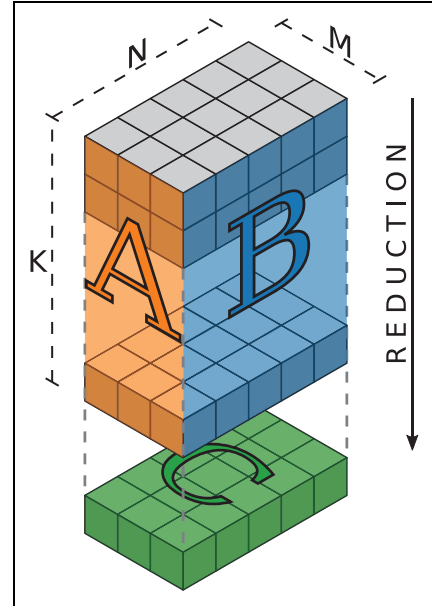


Figure 5. Illustration of the iteration space of the *TSMTTSM* operation $\mathbf{C} = \mathbf{A}^T \mathbf{B}$.

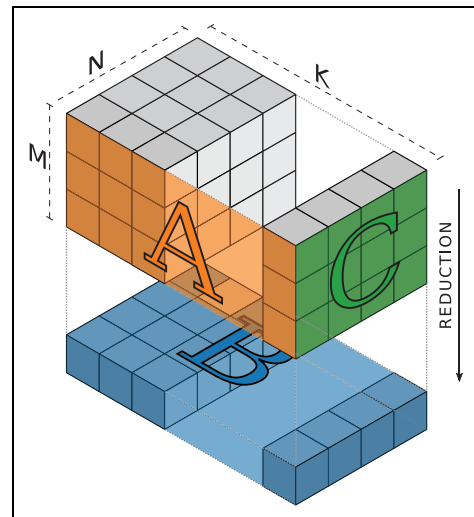


Figure 6. Illustration of the iteration space of the *TSMM* operation $\mathbf{B} = \mathbf{A}\mathbf{C}$.

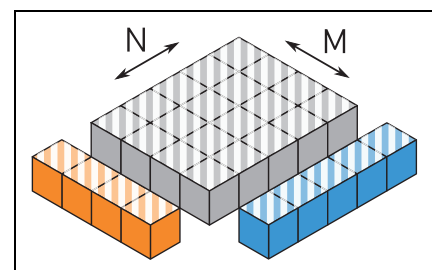


Figure 7. *TSMTTSM*: Parallelization over K only.

associated with the computations in one slice is the same as for the whole MMM kernel. We assume perfect caching, i.e. that \mathbf{A} and \mathbf{B} are transferred from memory

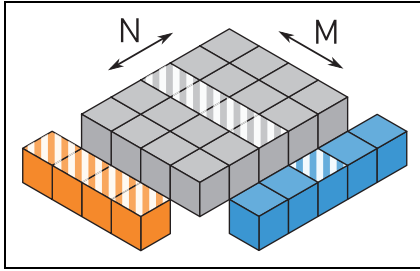


Figure 8. *TSMTTSM*: Parallelization over the K and N loop.

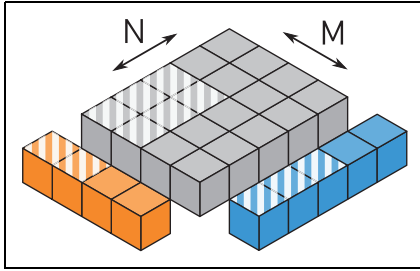


Figure 9. *TSMTTSM*: Parallelization over K and tiling of the two inner loops, here with tile size 2×3 .

just once and reused as many times as necessary throughout the calculation.

The fastest way to reuse values is to use a register and have the thread the register belongs to perform all required operations on this data. Data used by multiple threads can (preferably) be shared in the L1 cache for threads in the same thread block or in the L2 cache otherwise. This works only if some spatial and temporal access locality is in place. Therefore, the mapping of cells, i.e. work, to threads determines which thread requires what data for its computations and the locality of data access.

3. *TSMTTSM*

For the *TSMTTSM*, the two outer loops, which are completely independent and therefore well parallelizable, are usually the target of an implementation focused on square matrices. For skinny matrices, these loops are much too short to yield enough parallelism for a GPU. In consequence, the loop over the long K dimension has to be parallelized as well, which also involves parallelizing the sum inside the loop. There are many more terms in the parallel reduction than threads, so that each thread can first serially compute a thread local partial sum, which is afterwards reduced to a total sum (see Listing 2). Here, a so-called grid stripe loop, described by Harris (2013), is used to map rows to threads.

For data locality, the two small loops have to be moved into the K loop. Since they are short loops with constant loop trip count, they can be unrolled completely, which also allows to map the intermediates to local variables instead of indexing into a local array (see Listing 3). Depending on whether and how the two small loops are parallelized, each

Listing 2. *TSMTTSM* pseudo code, with the K loop parallelized as a grid stripe loop.

```
c_local[:, :] = 0

for (k = threadIdx; k < K; k += gridSize)
  for m = 0...M:
    for n = 0...N:
      c_local[m][n] += A[k][m] * B[k][n]

for m = 0...M:
  for n = 0...N:
    C[m][n]
    = global_reduction(c_local[m][n])
```

Listing 3. *TSMTTSM* pseudo code with parallelized K loop, after unrolling the two inner loops (here shown exemplarily for $M = N = 2$) and mapping array entries to variables. The global reduction is omitted for brevity.

```
for (k = threadIdx; k < K; k += gridSize)
  c0_0 += A[k][0] * B[k][0]
  c0_1 += A[k][0] * B[k][1]
  c1_0 += A[k][1] * B[k][0]
  c1_1 += A[k][1] * B[k][1]
```

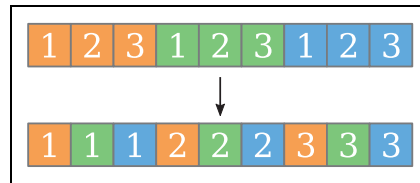


Figure 10. Example for transposing a 1D continuous thread mapping. The colors denote which thread the element belongs to, the numbers are their positions in the thread.

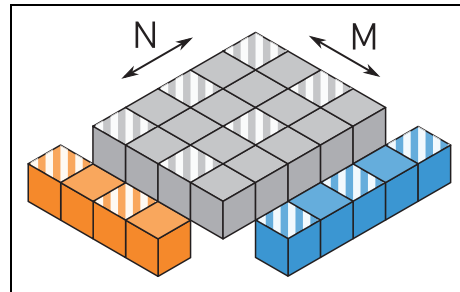


Figure 11. *TSMTTSM*: Parallelization over K and transposed tiling of the two inner loops, here with tile size 2×3 .

thread computes only some of these MN intermediates. Figures 7 to 11 visualize this by showing a slice of the multiplication cube and which values a single thread would compute. The number of loads that each thread has to do are the affected values in the row of **A** and **B**, also visible in the illustrations, while each highlighted cell in the slice stands for one line in the loop body of Listing 3, which

Listing 4. TSMTTSM pseudo code, with the K and N loop parallelized. The global reduction is omitted.

```

for (k = threadIdx / N; k < K;
     k += gridStride / N)
  n = threadIdx % N
  for m = 0...M:
    c_local[m][n] += A[k][m] * B[k][n]

```

Listing 5. TSMTTSM pseudo code, with tiled M and N loop using tile sizes T_M and T_N . The global reduction and row calculation in the K loop is omitted.

```

midx = (threadIdx / N) % M
nidx = threadIdx % N
for (...)
  for tm = 0...TM:
    for tn = 0...TN:
      m = midx * TM + tm
      n = nidx * TN + tn
      c[tm][tn] += A[k][m] * B[k][n]

```

corresponds to one FMA operations and one intermediate variable.

Since the L1 cache is not able to deliver one operand per FMA instruction, a high FMA-to-load ratio is desirable. This can be achieved by maximizing the area and the “squareness” of the area that is computed by a single thread. At the same time, more intermediate results per thread increase the register count, which can limit the occupancy and eventually lead to spilling.

The approach of only parallelizing the K loop (shown in Listing 2 and Figure 7) easily achieves this goal. While it maximizes the arithmetic intensity already in the L1 cache, the MN intermediate results occupy $2MN$ registers, so the maximum of 256 registers per thread is already exceeded at $M, N > 11$, causing spilling and poor performance.

Parallelizing one of the inner loops as well (Listing 4) leads to the pattern shown in Figure 8. The amount of registers required is only M here, so there is no spilling even at $M, N = 64$. However, the narrow shape results in an FMA/load ratio below 1 (i.e., a low arithmetic intensity in the L1 cache), as values from \mathbf{A} are used just once per load.

A better approach, which combines manageable register requirements with a more square form of the tile is to subdivide the two smaller loops into tiles (see Listing 5 and Figure 9). This mapping also allows for much more flexibility, as the tile sizes can be chosen small enough to avoid spilling or reach a certain occupancy goal but also large enough to create a high FMA/load ratio. Tile sizes that are not divisors of the small loop dimensions can be covered by generating guarding statements for tile entries that could possibly overlap to only be executed by threads with a tile index that does not extend beyond the border of the slice.

This is helpful for matrix dimensions that have few divisors, e.g. prime numbers.

Mapping a continuous range of values to a thread leads to strided loads, which can be detrimental to performance. The same entry in two consecutive threads’ partitions is always as far apart as the tile side length. A more advantageous, continuous load pattern can be achieved by transposing the threads’ tiles, as shown in Figure 10. The elements belonging to a thread do not form a contiguous range anymore but are interleaved. Therefore, when all participating threads load their n -th element, the addresses referenced by that load have a uniform stride. With a transposed mapping, the 2D TSMTTSM mapping from Figure 9 becomes Figure 11.

3.1. Leap frogging

On NVIDIA’s GPU architectures, load operations can overlap with each other. The execution will only stall at an instruction that requires an operand from an outstanding load. The compiler maximizes this overlap by moving all loads to the beginning of the loop body, followed by the floating-point (FP) instructions that consume the loaded values. Usually at least one or two of the loads come from memory and thus take longer to complete than other queued loads, so that execution stalls at the first FP instruction. A way to circumvent this stall is to load the inputs one loop iteration ahead into a separate set of *next* registers, while the computations still happen on the *current* values. At the end of the loop, the *next* values become the *current* values of the next loop iteration by assignment. These assignments are the first instructions that depend on the loads and thus the computations can happen while the loads are still in flight. This is a common technique for loops that iterate over data items, and a similar strategy is also used by Chen et al. (2019) for TSM2.

3.2. Global reduction

After each thread has serially computed its partial, thread-local result, a global reduction is required, which is considered overhead. Its runtime depends only on the thread count, though, whereas the time spent in the serial summation grows linearly with the row count. The time spent for the global reduction therefore becomes marginal for large row counts compared to the time for the serial summation. However, as shown by Thies et al. (2019), the performance at small row counts can still be relevant, as the available GPU memory may be shared by more data structures than just the two tall & skinny matrices, limiting the data set size.

Starting with the *Pascal* architecture, atomic add operations for double precision values are available for global memory, making global reductions more efficient than on older systems. Each thread can just use an `atomicAdd` of its partial value to update the final results. The throughput of global `atomicAdd` operations is limited by the amount of contention, which grows for smaller matrix sizes. We

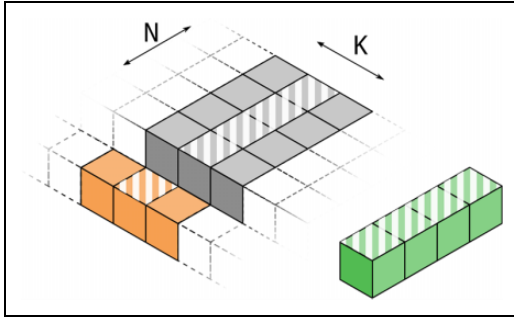


Figure 12. TSMM: Parallelization over K , a single thread computes a full result row of \mathbf{B} . Slice perpendicular to the M axis, the (long) K axis extends “indefinitely” on both sides.

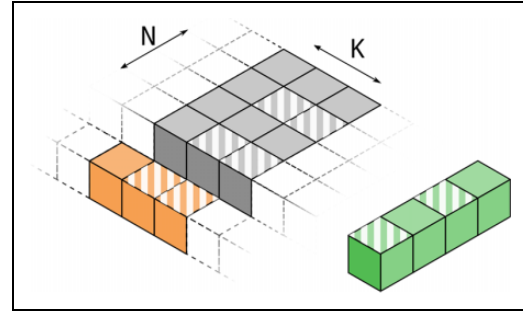


Figure 14. TSMM: Parallelization over K and N and $2\times$ unrolling, two threads compute two results each and on two rows of \mathbf{B} in a single iteration. Slice horizontal to the M axis, the (long) K axis extends “indefinitely” on both sides.

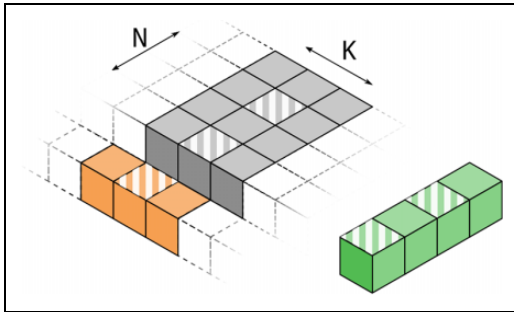


Figure 13. TSMM: Parallelization over K and N , two threads compute two results each. Slice horizontal to the M axis, the (long) K axis extends “indefinitely” on both sides.

improve on this global atomic reduction variant with a local atomic variant that reduces the amount of global `atomicAdd` operations by first computing thread-block-local partial results using shared memory atomics. This is followed by a global reduction of the local results. Additionally, we opportunistically reduce the amount of launched threads for small row counts.

4 TSMM

4.1. Thread mapping

In contrast to the *TSMTTSM* kernel, the summation is done along the short M axis, with no need for a global reduction. Though the short sum could be parallelized, this is not necessary in this case, as the other two loop dimensions supply sufficient parallelism. The visualizations in Figures 12–14 show slices perpendicular to the M axis, since this dimension will not be parallelized.

The first option is to only parallelize over the long K dimension as shown in Figure 12. Each entry in \mathbf{A} would be loaded once and then reused out of a register. The N sums that each thread computes require $2N$ registers, which is not a prohibitive number even at $N = 64$ but still does reduce occupancy. A more severe disadvantage are the strided stores. As each thread produces and stores a full row of \mathbf{B} , the addresses stored to by the different threads are far apart, leading to partially written cache lines. This in turn causes a write-allocate read stream of the result matrix

\mathbf{B} to ensure fully consistent cache lines, thereby reducing the arithmetic intensity of the kernel.

This can be avoided by parallelizing the N loop. Each thread computes a single result of the output row of \mathbf{B} . Because consecutive threads compute consecutive results, cache lines are always written fully and no write-allocate stream is necessary. The disadvantage is a low compute/load ratio. Each value from \mathbf{A} is loaded and used just once in each thread.

A more balanced approach is to have a smaller group of threads compute on each result row, with a few results computed by each thread. Each value loaded from \mathbf{A} is reused multiple times, once for each result computed by this thread. Using a transposed mapping as shown in Figure 13, each thread does not compute consecutive elements; results computed by threads are interleaved, so that consecutive elements are written and the amount of partial writes is reduced. This works best if the thread count is a multiple of four, which corresponds to the L1 cache line management granularity of 32 bytes. If N is not a multiple of four, the writes will necessarily be misaligned, with some cache lines being cut. Larger thread counts slightly reduce the impact of cut cache lines.

4.2. Data from \mathbf{C}

Our discussion of thread mappings and data transfers so far has ignored the entries of the matrix \mathbf{C} . These values are the same for every index of the K loop. The fastest would be to load all entries of \mathbf{C} into registers and reuse them from there, but this strategy would quickly exceed the number of available registers even at moderate M and N . Since they are accessed frequently and all threads in a thread block access similar values, the contents of \mathbf{C} should continuously stay in the L1 cache, making reloads of these values a question of L1 cache bandwidth and not memory latency. Depending on the number of threads per row and the alignment, each load from \mathbf{C} spans up to three 128-byte cache lines, which would then be used for a single FMA. This is higher than the sustainable ratio of one 128-byte cache line per FMA. A solution is to reuse each value loaded from \mathbf{C} by unrolling the K loop and pulling the unrolled iterations

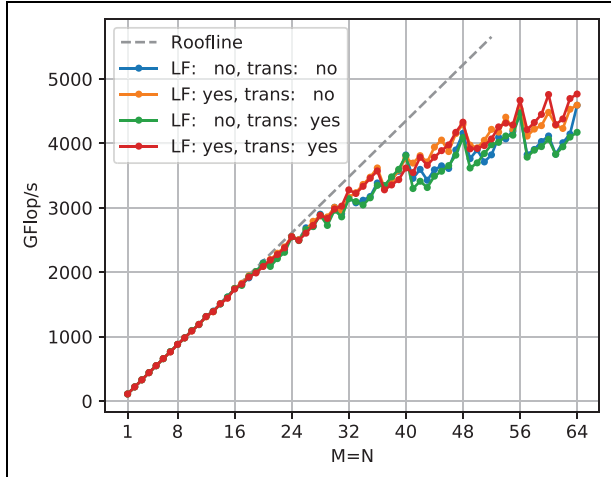


Figure 15. Performance comparison of real-valued double-precision TSMTTSM vs. quadratic tile size with $K=2^{29}/M$ on the V100 across the four different permutations of using leap frogging (LF) and transposed mapping (trans). The best performance for each matrix size and configuration is shown. The arithmetic peak performance of the device is 7.066 Tflop/s.

inside the M loop. Each iteration over K loads the same values of C , which can subsequently be used for multiple iterations per load.

The loads from C can also be sped up by using the shared memory to cache these loads. Threads in a thread block collaboratively load the contents of C into the shared memory at the beginning of the kernel. The loop over K is parallelized with a grid stride loop, where only as many threads as necessary for full occupancy are launched. Each kernel instantiation then computes on multiple rows of B . Therefore, loading C into shared memory can be amortized over many rows.

On the V100, the shared memory has the same bandwidth as the L1 cache, given that they occupy the same hardware structure. However, shared memory accesses guarantee cache hits, as they avoid conflict misses with other data. They also have a lower latency, since no tags have to be checked.

5. Results: TSMTTSM

5.1. Transposition and leap frogging

An exhaustive search was used to find the best tile size and configuration for each matrix size. The simpler mapping schemes are subsets of the tiled mapping. E.g. the mapping in Figure 8 corresponds to a tile size of $M \times 1$. Figure 15 shows the performance of the four configurations of using leap frogging and a transposed mapping. The performance agrees with the roofline prediction (dashed line) perfectly until $M, N = 20$. Until $M, N = 36$, the best performance stays within 95% of the limit. Beyond that, the growing arithmetic intensity does not translate into a proportional speedup anymore, although the performance is still about a factor of two away from peak. The best variants plateau at

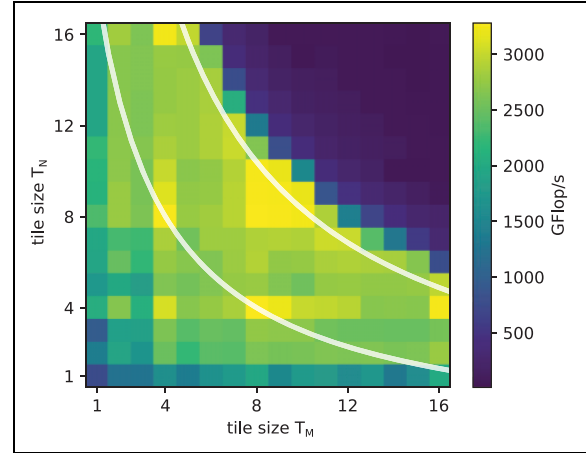


Figure 16. Performance of TSMTTSM for $M, N = 32$ and $K=2^{29}/M$ vs. tile sizes in M and N directions, using real-valued double-precision matrices, with leap frogging and transposed mapping. The two white lines are defined by $2 \times (T_M T_N + 2(T_M + T_N) + 8) = R$, with $R = 128, 256$ to mark approximate boundaries of register usage.

about 2/3 of peak. Both variants using leap frogging are clearly faster, but the transposed mapping is only a bit faster if leap frogging is used. This is in contrast to experiences with the *Kepler* GPU architecture, where strided loads are slower, and this kind of transformation is more beneficial. The best tile size changes when leap frogging is used as it requires more registers.

5.2. Tile sizes

Figure 16 shows the dependence of performance on the tile sizes T_M and T_N for the case $M, N = 32$ with leap frogging and transposed mapping. Performance drops off sharply if the tile sizes become too large and too many registers are used. The number of registers can be approximated by $2 \times (T_M T_N + 2(T_M + T_N) + 8)$, which accounts for the thread-local sums ($T_M T_N$), loaded values ($T_M + T_N$), and eight registers for other purposes. Leap frogging introduces a factor of two for the number of loaded values (for *current* and *next* values), and double precision values generally require two 32-bit registers for an overall factor of two. The graph shows the iso-lines of 128 and 256 registers, which represent the occupancy drop from 25% to 12.5% at 128 registers and the onset of spilling at 256 registers.

The best-performing tile sizes generally sit on or just below these lines, maximizing the area of the tile for a given occupancy. The dimensions are largely symmetric but not perfectly so, as threads are mapped to tiles in M direction first. There are clear patterns favoring powers of two as those are divisors of the matrix size 32 and avoid both the overhead of guarding statements and idle threads.

5.3. Analysis

According to the roofline model, at $M = N = 64$ the upper performance limit is

$$P = \frac{64 \text{ flop}}{8 \text{ byte}} \times 880 \text{ Gbyte/s} = 7060 \text{ Gflop/s}, \quad (2)$$

which is almost exactly the P_{Peak} of 7066 Gflop/s. However, our implementation cannot realize the roofline-predicted performance, and instead tops out at 4766 Gflop/s $\approx 2/3P_{Peak}$. The reason for the limitation is memory latency, which can be shown by a simple model: Whereas the memory latency for an idle memory interface measured with a pointer chasing benchmark (see Ernst, 2019) is only 435 cy, this latency increases as the load on the memory interface increases. For the values in Table 1, it is possible to calculate corresponding latency values according to Little's Law via

$$T_\ell = \frac{fN \times 8 \text{ byte}}{b}, \quad (3)$$

with f being the clock frequency, N the thread count, and b the memory bandwidth. For the unloaded case in the first row of Table 1 (ILP = 1), the latency according to (3) is $T_\ell \approx 470$ cy, which matches the measured pointer chasing latency quite well. The bandwidth of $b = 681$ Gbyte/s at 25% occupancy in the fourth row roughly corresponds to the highest observed memory bandwidth, based on the computational intensity, for $M, N = 64$, and result in $T_\ell \approx 664$ cy of memory latency.

The best tile size without leap frogging is 11×8 , which requires $11 \times 8 = 88$ FMA operations. These can be computed on a single quadrant in $88 \times 4 \text{ cy} = 352$ cy. At this large tile size, the register requirements of at least $2 \times 11 \times 8 = 176$ registers allow to run only eight warps, i.e. two warps per quadrant, simultaneously on a SM. One warp doing 352 cy of compute work finishes earlier than the other warp waiting for 664 cy for data from memory. It will then also wait for the next data to be loaded, which is a period of time where none of the two warps are issuing floating point operations, and therefore counts as wasted cycles.

Leap frogging does improve the situation, as even with a single warp the memory latency and compute times can overlap. However, additional registers are required to hold the data for the next iteration, which either necessitates smaller tile sizes or reduces occupancy, both of which are bad for overlapping. Overall, leap frogging is still beneficial, though.

Figure 17 shows an experiment that gives insight into the relationship of latency and occupancy. A modification of the generated kernels allows testing the impact of higher occupancies even for kernels with larger tile sizes, where the high register requirements usually limit the occupancy to the minimum of eight warps per SM. Instead of computing $T_M \times T_N$ intermediate results, all summands are summed up in just two accumulators. This does of course not compute the correct results any more, but all the instructions and loaded operands are the same, while reducing the register count so that 32 warps per SM can run concurrently. Another modification to the generated kernel

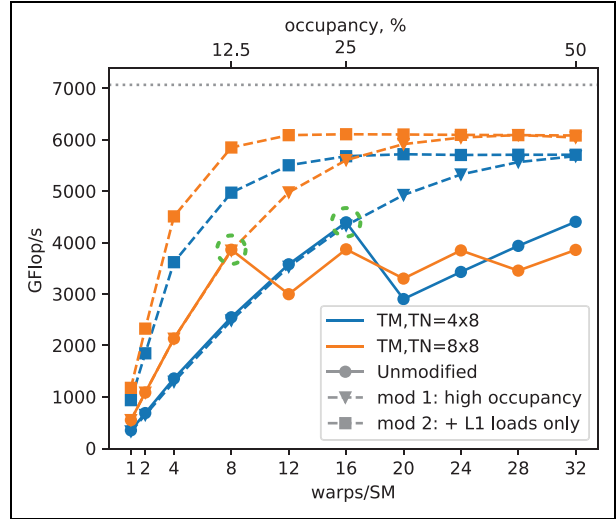


Figure 17. TSMTTSM performance vs. occupancy of real, correct kernels and two modified (incorrect) kernels at tile sizes of 4×8 and 8×8 , respectively. The first modification reduces register count, while the second kernel additionally reduces the data set so that it resides in L1 cache. Green circles mark the point with the highest performance of the unmodified kernels. (Real-valued double-precision matrices).

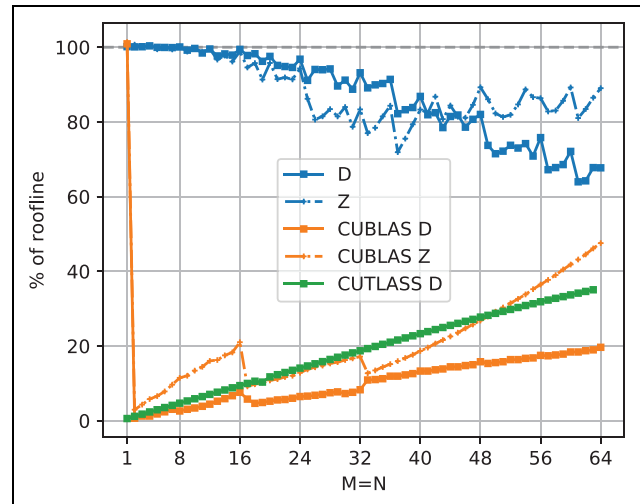


Figure 18. TSMTTSM percentage of roofline-predicted performance for real (D) and complex (Z) double-precision data in comparison with CUBLAS and CUTLASS.

introduces a division of the K loop row index by a large constant. In consequence, all loop iterations compute on data of very few rows, which makes almost all accesses L1 cache hits with the corresponding much smaller latency. Repeatedly using the same row is done in such a contrived way in order to prevent the compiler from pulling the loads in front of the loop.

With tile sizes of 8×4 and 8×8 , as used in this experiment, 16 and 8 warps per SM can run concurrently. At these occupancies (green circles in Figure 17), the respective real kernels (circle symbols) performance is highest, as the maximum possible number of thread blocks run concurrently.

With an increased number of launched thread blocks, the unmodified kernels’ performance does not increase anymore, as additional thread blocks do not run concurrently but are scheduled in a “second wave” of thread blocks. An imbalance in the number of thread blocks per wave leads to fluctuating performance.

The kernels modified for higher occupancy (triangle symbols) have the same performance as the unmodified kernels up to these points, but allow to see the hypothetical speedup if more thread blocks could run concurrently, which would be possible on a hypothetical V100 with four times as many registers.

The performance increase is linear in all cases up to four warps per SM, as this is the minimum to fill all four quadrants of an SM. For both tile sizes, the L1 load kernels (square symbols) profit somewhat from a second warp on each quadrant to overlap the remaining latency and overhead but quickly saturate at ceilings of 6080 Gflop/s and 5700 Gflop/s, respectively, which is not a latency effect any more. The reason for these lower roofs remains open, but we suspect that it is rooted in limited instruction throughput. We noticed that the gap to the device peak performance matches one missing DP FP operation per four non-DP FP operations, i.e. integer and load instructions. DP FP operations are supposed to execute on separate execution units, and so we can only speculate whether there is a restriction in co-issuing DP FP operations with integer and load instructions.

The two experiments with the normal, higher latency from memory (triangular symbols) need many more warps to overlap their longer latency to eventually saturate at the same level as the L1 load kernels. At least two to three times larger register files would be required to get there. At the same time, it also shows how devastating it would be if the register files were half as large, a situation that is not dissimilar to the older *Kepler* GPU architecture, where double the number of execution units were backed by a similar sized register file. The larger tile size saturates more quickly, because it amortizes the same latency over twice the number of floating-point operations. Note that in the end, both tile sizes have a similar real-world performance, as the higher possible occupancy of 16 warps per SM compared to 8 warps per SM balances the smaller amount of work per iteration.

This simple model also helps to explain the rather small benefits from using the transposed mapping. The transposed mapping changes the load pattern to contiguous blocks instead of long strides. This in turn reduces the number of touched cache lines, and increases the rate at which the L1 cache can serve the outstanding loads after the data has arrived from memory. However, this rate is only really a limiter at low FMA/load ratios, or at the beginning of the floating-point operation phase, where the FP units still wait for enough registers being filled for uninterrupted operation. The transposed mapping therefore only gives a small speedup in the phase that is mostly not the limiter, but at the same time also makes smaller tile sizes more feasible.

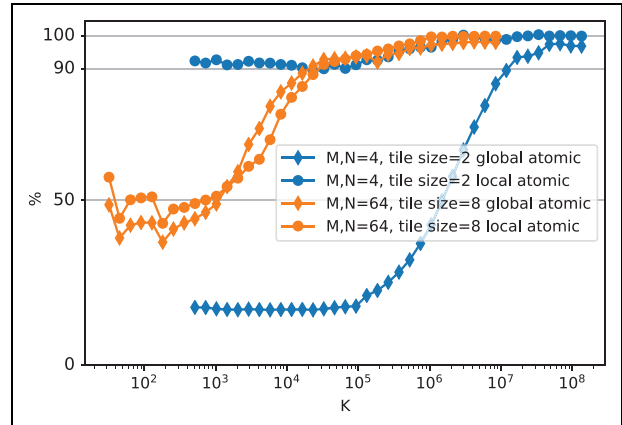


Figure 19. Global reduction impact for *TSMTTSM*: Performance when using each of the two global reduction variants as the percentage of the performance of a kernel without a global reduction, using two different matrix widths and tile sizes. (Real-valued double-precision matrices).

On the other hand, the strided access patterns of the nontransposed mapping touch most cache lines already on the first load, and therefore already cause most cache misses with the first load. Subsequent loads are cache hits. With the transposed mapping, with its contiguous blocks of addresses per load, cache misses are postponed until later loads, which starts the memory latency penalty later. That is why the configuration using the transposed mapping without leap frogging performs worst (see Figure 15). However, in combination with leap frogging it is faster than the two variants with the nontransposed mapping.

5.4. Comparison with libraries

Both CUBLAS’ and CUTLASS’ performance (see Figure 18) is far below the potential performance, except for $M, N = 1$, where CUBLAS seems to have a special detection for the dot product corner case. The utilization of potential performance increases as matrices become wider, which makes them more square and compute bound, bringing them closer to more standard scenarios.

In contrast, the presented implementation shows full efficiency for narrow, clearly memory bandwidth limited matrices, and utilization slightly drops off as matrices become more compute bound. For complex-valued matrices, the *TSMTTSM* becomes compute bound already at $M, N = 32$. Instruction throughput becomes the limiter much earlier than memory bandwidth and latency, which is why the utilization drops earlier. With increasing matrix size, it fully saturates the previously explained lower ceiling due to our speculated co-issue limitation between double-precision FP instructions and integer instructions.

5.5. Impact of reductions

Figure 19 shows the relative performance of our *TSMTTSM* implementation versus row count with respect to a baseline

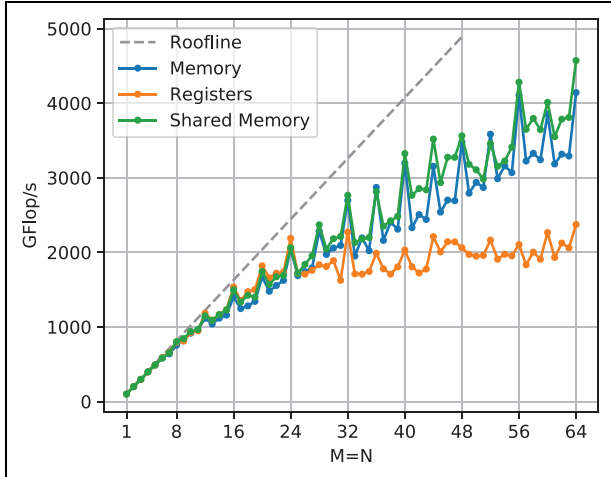


Figure 20. TSM performance comparison at $K=2^{29}/M$ among different sources for the matrix \mathbf{C} , showing the best-performing configuration of each method and matrix width. (Real-valued double-precision matrices).

without any reduction for a selection of inner matrix sizes and tile sizes, choosing either of the two reduction methods described in Section 3.2. As expected, the impact of the reduction generally decreases with increasing row count. The method with only global atomics is especially slow for the narrower matrices ($M, N = 4$). Many threads writing to a small amount of result values leads to contention and causes a noticeable impact even for a matrix filling the device memory ($K = 10^8$). The local atomic variant drastically reduces the number of writing threads, resulting in less than 10% overhead even for the smallest sizes and near-perfect performance for $K > 10^6$. For the wider matrices, the difference is smaller. The global atomic version is not as slow because writes spread out over more result values and the local atomic variant is not as fast because the larger tile size requires more work in the local reduction. Both variants incur less than 10% overhead just above $K = 10^4$, a point where only about 0.2% of the GPU memory is used.

6. Results: TSM

The described methods and parameters open up a large space of configurations. Each of the Figures 20, 21, and 22 shows a cross section of each configuration option by displaying the best-performing value for each choice for that configuration option.

6.1. Source of \mathbf{C}

The data in Figure 20 demonstrates that trying to keep the values of matrix \mathbf{C} in registers works well only for small M, N . The increasing register pressure at larger sizes reduces occupancy, which is especially bad if multiple results are computed per thread.

Reloading values from shared memory has a consistently small performance advantage especially for sizes

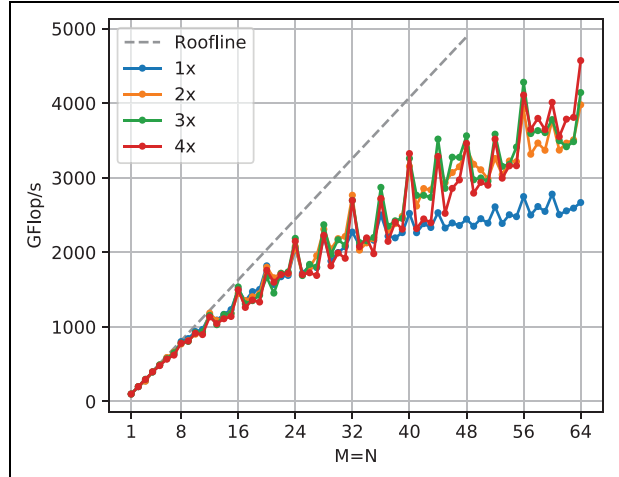


Figure 21. TSM performance comparison of different degrees of unrolling at $K=2^{29}/M$, showing the best-performing configuration for each unrolling depth (1, ..., 4) and matrix width. (Real-valued double-precision matrices).

that are not multiples of four, due to a smaller penalty because of misaligned loads. Each additional cache line that gets touched because of misalignment costs an additional cycle.

6.2. Unrolling

Although there is little improvement with further unrolling beyond $2\times$, as Figure 21 shows, unrolling at least once shows a clear speedup compared to no unrolling. Without unrolling, the shared memory bandwidth would limit the performance due to the high ratio of shared memory loads to FP DP instructions, and its latency could not be hidden as well with FP DP instructions from further iterations. Generally, a similar reasoning as for the *TSM* kernel applies, where computing more results per thread and higher unrolling counts increase the number of floating-point operations per iteration but also decrease the occupancy that would be needed to overlap the memory latency.

6.3. Thread count

Fewer threads per row mean more work per thread. For large matrix sizes, this can result in huge kernels with high register requirements, which is why Figure 22 does not show measurements for the whole matrix size range for one and two threads per row. These two thread counts are the slowest variants, as they show the effects of strided writes the most. With four threads writing consecutive values, there is at least a chance of writing a complete 32-byte cache line sector. The difference between 4, 8 or 16 threads is not large, although the larger thread counts perform slightly more consistently (i.e., with less fluctuation across M).

The performance analysis for *TSM* shows a clear preference for the small matrix dimension $M = N$ to be a

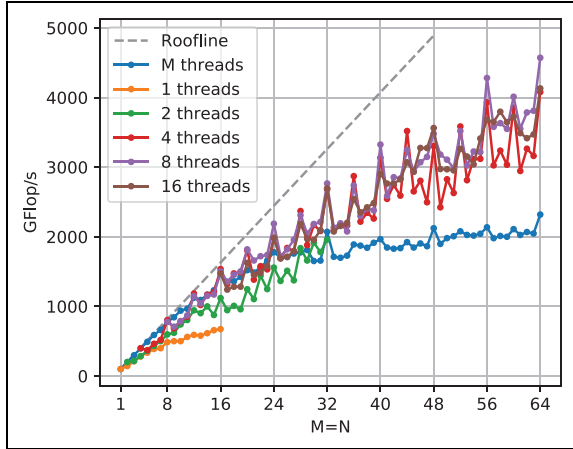


Figure 22. TSM performance comparison of different thread counts per row at $K = 2^{29}/M$, showing the best-performing configuration for each thread count and matrix width. (Real-valued double-precision matrices).

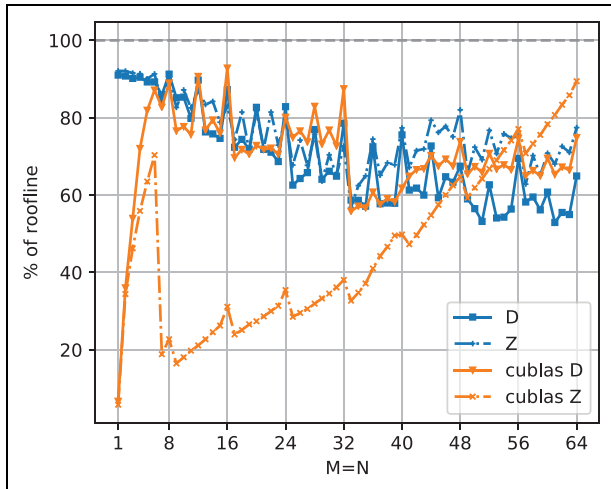


Figure 23. TSM percentage of roofline-predicted performance for real (D) and complex (Z) double-precision data in comparison with CUBLAS.

multiple of four. For this case, all writes of computed data to the matrix \mathbf{B} are aligned to $4 \times 8 \text{ byte} = 32 \text{ byte}$, which is the management granularity for L1 cache lines and the cache line length for the L2 cache. With this alignment, cache lines are fully written and there is no overhead for write allocation from memory. Misalignment is the major performance hurdle for matrix widths that are not multiples of four.

6.4. Comparison with libraries

Figure 23 shows that, except for very small M, N , CUBLAS performs very well for the real-valued TSM kernel. With increasing width, the development in utilization is very similar to the presented implementation. Our solution works similarly well for complex-valued matrices,

which is not the case for CUBLAS. Here, a strong performance drop for medium-wide matrices can be observed.

7. Conclusion and outlook

We have shown how to optimize the performance for two types of multiplication of double-precision, real and complex tall & skinny matrices on a V100 GPU. With matrices narrower than 32 columns, near-perfect performance in accordance with a roofline performance model could be achieved. Over the rest of the skinny range up to a width of 64, between 60% and 67% of the potential performance was attained. We used a code generator on top of a range of suitable thread mapping and tiling patterns, which enabled an exhaustive parameter space search. Two different ways to achieve fast, parallel device-wide reductions for long vectors have been devised in order to ensure a fast ramp-up of performance already for shorter matrices. An in-depth performance analysis was provided to explain observed deviations from the roofline limit. Our implementation outperforms the vendor-supplied CUBLAS and CUTLASS libraries by far or is on par with them for most of the observed parameter range.

In future work, in order to push the limits of the current implementation, shared memory could be integrated into the mapping scheme to speed up the many loads, especially scattered ones, that are served by the L1 cache.

The presented performance figures were obtained by parameter search. An advanced performance model, currently under development, could be fed with code characteristics such as load addresses and instruction counts generated with the actual code and then used to eliminate bad candidates much faster. It will also support a better understanding of performance limiters.

Prior work by us in this area is already part of the sparse matrix toolkit *GHOST* (Kreutzer et al., 2016) and we plan to integrate the presented work there as well.


Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the ESSEX-II project in the DFG Priority Programme SPPEXA.

ORCID iD

Dominik Ernst  <https://orcid.org/0000-0003-3547-0611>

References

- Benson AR and Ballard G (2015) A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices* 50(8): 42–53.

- Chen J, Xiong N, Liang X, et al. (2019) TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In: *Proceedings of the ACM international conference on super-computing, ICS '19*. New York, NY, USA: Association for Computing Machinery, pp. 106–116. ISBN 9781450360791. DOI: 10.1145/3330345.3330355.
- Cullum J and Donath WE (1974) A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices. In: *1974 IEEE conference on decision and control including the 13th symposium on adaptive processes*, Phoenix, AZ, USA, 20–22 November 1974, pp. 505–509.
- Ernst D (2019) CUDA microbenchmarks. Available at: <http://tiny.cc/cudabench> (accessed 2 February 2020).
- Ernst D, Hager G, Thies J, et al. (2020) Performance engineering for a tall & skinny matrix multiplication kernels on GPUs. In: R Wyrzykowski, E Deelman, J Dongarra, and K Karczewski (eds) *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing, pp. 505–515.
- Gropp WD, Kaushik DK, Keyes DE, et al. (1999) Towards realistic performance bounds for implicit CFD codes. In: *Proceedings of parallel CFD '99*, pp. 233–240. Amsterdam, the Netherlands: Elsevier.
- Harris M (2013) CUDA pro tip: write flexible kernels with grid-stride loops. Available at: devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (accessed 2 February 2020).
- Herrero JR and Navarro JJ (2006) Compiler-optimized kernels: an efficient alternative to hand-coded inner kernels. In: ML Gavrilova, O Gervasi, V Kumar, et al. (eds) *Computational science and its applications—ICCSA 2006*. Berlin, Heidelberg: Springer, pp. 762–771. ISBN 978-3-540-34080-5. DOI: 10.1007/11751649_84.
- Huang J, Rice L, Matthews DA, et al. (2017) Generating families of practical fast matrix multiplication algorithms. In: *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, Orlando, FL, USA, 29 May–2 June 2017, pp. 656–667.
- Huang J, Smith TM, Henry GM, et al. (2016) Strassen’s algorithm reloaded. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC '16*, Salt Lake City, UT, USA, 13–18 November 2016. Hoboken, NJ, USA: IEEE Press.
- Huang J, Yu CD and van de Geijn RA (2020) Strassen’s algorithm reloaded on GPUs. *ACM Transactions on Mathematical Software* 46(1): 1–22.
- Kreutzer M, Ernst D, Bishop AR, et al. (2018) Chebyshev filter diagonalization on modern manycore processors and GPGPUs. In: R Yokota, M Weiland, and D Keyes (eds) *High Performance Computing*. Cham: Springer International Publishing, pp. 329–349.
- Kreutzer M, Thies J, Röhrig-Zöllner M, et al. (2016) GHOST: building blocks for high performance sparse linear algebra on heterogeneous systems. *International Journal of Parallel Programming* 1–27. DOI: 10.1007/s10766-016-0464-z.
- McCalpin JD (1995) Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* 2: 19–25.
- NVIDIA (2019a) CUBLAS reference. Available at: <https://docs.nvidia.com/cuda/cublas> (accessed 5 May 2019).
- NVIDIA (2019b) CUTLASS. Available at: <https://github.com/NVIDIA/cutlass> (accessed 5 May 2019).
- O’Leary DP (1980) The block conjugate gradient algorithm and related methods. *Linear Algebra and Its Applications* 29: 293–322. Special Volume Dedicated to Alson S. Householder.
- Röhrig-Zöllner M, Thies J, Kreutzer M, et al. (2015) Increasing the performance of the Jacobi-Davidson method by blocking. *SIAM Journal on Scientific Computing* 37(6): C697–C722.
- Strassen V (1969) Gaussian elimination is not optimal. *Numerische Mathematik* 13(4): 354–356.
- Thies J, Röhrig-Zöllner M, Overmars N, et al. (2019) PHIST: a pipelined, hybrid-parallel iterative solver toolkit. Accepted for ACM Transactions on Mathematical Software. Available at: <https://elib.dlr.de/123323/>.
- Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 52(4): 65–76.

Author biographies

Dominik Ernst holds a Master Degree in Computational Engineering from the Friedrich-Alexander-Universität Erlangen-Nürnberg in Germany. He works as a research associate at the HPC division at Erlangen Regional Computing Center (RRZE). His research topics are optimization of GPU kernels and performance modeling of GPU architectures.

Georg Hager holds a PhD and a Habilitation degree in computational physics from the University of Greifswald. He is a senior researcher in the HPC division at Erlangen Regional Computing Center (RRZE), and an associate lecturer at the Institute of Physics of the University of Greifswald. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and structure formation in large-scale parallel codes. He was instrumental in developing and refining the Execution-Cache-Memory (ECM) performance model and energy consumption models for multicore processors. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide.

Jonas Thies received his PhD in mathematics from the University of Groningen, the Netherlands, in 2010. He joined the DLR facility for Simulation and Software Technology in 2013 to work in the DFG project ESSEX. Since 2017 he leads the group on parallel numerics. His research interests include programming techniques for modern

supercomputers, numerical linear algebra and computational fluid dynamics in engineering and geophysics.

Gerhard Wellein holds a PhD in solid state physics from the University of Bayreuth and is a regular professor at the Department for Computer Science at the University of Erlangen-Nuremberg, Germany. He heads the HPC division at Erlangen Regional Computing Center

(RRZE) and has more than 20 years of experience in teaching HPC techniques to students and scientists from computational science and engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.