ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cuCOSMA: Near Optimal Matrix-Matrix Multiplication in CUDA C++

Bachelor Thesis

Neville Walo

September 19, 2020

Supervisor: Prof. Dr. T. Hoefler
Advisors: Dr. N. Dryden, Dr. T. Ben-Nun, G. Kwasniewski

Scalable Parallel Computing Laboratory
Department of Computer Science, ETH Zürich

Abstract

Matrix-matrix multiplication is one of the most important routines in scientific computing. Implementing this operation efficiently is a complex challenge that that needs to take the memory model and architectural details into account. There are several high-performance matrix-matrix multiplication implementations for GPUs like cuBLAS, CUTLASS and rocBLAS. However, none of them is optimal for all applications. Here we present our implementation of COSMA [31] on a single GPU. Assuming that each matrix row is properly aligned and that the matrix dimensions are available at compile time, we wrote a matrix-matrix multiplication kernel that is faster than CUTLASS and outperforms cuBLAS and rocBLAS in specific situations. Furthermore, we integrated the findings of COSMA into a schedule generator, but it showed that the communication volume alone is not the right metric to choose tile sizes for GPUs and that other characteristics, such as occupancy and L2 cache hit rate, also have a significant impact on performance. Our results demonstrate that matrix-matrix multiplication on GPU can be further improved, from which many applications can benefit.

Acknowledgements

Uppermost, I would like to thank my advisors, Nikoli Dryden, Tal Ben-Nun and Grzegorz Kwasniewski for their support throughout the last months and for providing guidance and feedback during this project. Furthermore, I want to thank Prof. Dr. T. Hoefler for supervising my thesis.

Contents

Chapter 1

Introduction

Matrix-matrix multiplication is one of the most important operations in scientific computing. It has many use cases in linear algebra, machine learning, graph processing and computational sciences. The acceleration of this routine is therefore of great importance for many areas.

Distributing matrix-matrix multiplication efficiently, even within a single GPU or CPU, is a complex challenge that needs to take the memory model and architectural details into account. Libraries such as MKL and cuBLAS provide optimized implementations of matrix-matrix multiplication. However, they are often only optimized for specific, mostly square, sizes, which does not cover the full span of real-world workloads.

Optimizing matrix-matrix multiplication has already come a long way [27, 33, 28, 32, 29], but in 2019 a new competitor entered the field: COSMA [31], a parallel matrix-matrix multiplication algorithm that is near communication-optimal for all combinations of matrix dimensions, processor counts, and memory sizes. It outperformed all established libraries and algorithms and set a new record for the fastest distributed matrix-matrix multiplication on CPUs. The key idea behind COSMA [31] is to derive an optimal sequential schedule and then parallelize it, preserving I/O optimality. Compared to other 2D or 3D algorithms, which fix a processor decomposition in advance and then map it to the matrix dimensions, COSMA takes the matrix dimension, processor counts and memory sizes into account to create an I/O optimal schedule.

We generalize this approach to accelerate matrix-matrix multiplication on a single GPU, which can be thought of similarly to a distributed system with multiple processors and local and global memory. We assume that the matrix dimensions and the specification of the GPU are known in advance to generate an optimal schedule and that each row of a matrix was aligned and padded accordingly.

We provide the following contributions:

- A schedule generator which takes as input the matrix dimensions and the specification of a GPU to produce communication optimal tile sizes.
- A kernel written in CUDA C++ and HIP which performs an SGEMM (**S**ingle precision **GE**neral **M**atrix **M**ultiply) using tile sizes and matrix dimensions which are known at compile time.
- An extensive performance evaluation on a NVIDIA Tesla V100 32GB and AMD Radeon Instinct MI50 of cuCOSMA, CUTLASS [8], cuBLAS [1], hipCOSMA and rocBLAS [26].

Chapter 2

Background

This chapter presents the basics of matrix-matrix multiplication, the approach taken by COSMA, how to implement an efficient GEMM in CUDA and the approaches by current state-of-the-art implementations.

2.1 Matrix-Matrix Multiplication

This section describes the challenges that arise when trying to implement a parallel matrix-matrix multiplication and the solution proposed by COSMA.

Suppose a system with p processors, each with a fast memory of size S (measured in how many elements of a matrix it can store) and unlimited slow memory. The goal is to perform the following matrix-matrix multiplication $C = A * B$ with $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$. Further, assume that $S < \min\{m \cdot k,\ k \cdot n,\ m \cdot n\}$, such that none of the matrices fit into a single processor's fast memory. Now the problem arises how to distribute the multiplication among different processors.

As shown in Listing 2.1, a naive matrix-matrix multiplication has three for-loops, each of which represents a dimension that could be decomposed. Parallelizing the M and N dimension is easier in comparison to the K dimension. Listing 2.2 shows how the parallelization of the M and N dimensions looks like when programming with OpenMP. Note that every processor receives a unique pair of i and j and it is therefore easy to parallelize. If the K dimension is also split, the partial results must be reduced at the end, since multiple processors might have the same pair of i and j. When moving on to distributed computing, everything becomes more complicated, as now the data itself has to be divided and distributed among the processors.

The question of how to decompose a matrix-matrix multiplication with p processors has been raised for some time. For some special cases, such as square matrices or the number of processors is a power of two, there were

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    C[i][j] = 0;
    for (int k = 0; k < K; k++) {
      C[i][j] += A[i][k] * B[k][i];
    }
  }
}
```

Listing 2.1: Matrix-matrix Multiplication with 3 for loops.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    C[i][j] = 0;
    for (int k = 0; k < K; k++) {
      C[i][j] += A[i][k] * B[k][i];
    }
  }
}
```

Listing 2.2: Matrix-matrix Multiplication parallelizing the M and N dimensions.

already optimal solutions [27, 32]. Before COSMA there was no implementation that was optimal for all problem sizes and hardware configurations.

2.1.1 COSMA

This subsection briefly explains how COSMA (Communication Optimal S-partition-based Matrix multiplication Algorithm [31]) works. However, it will concentrate more on the surface and what is important for understanding cuCOSMA instead of going into the mathematical details. For more information see [31].

COSMA uses the following domain and grid size

$$\text{Grid: } \left[\frac{m}{a} \times \frac{n}{a} \times \frac{k}{b}\right], \quad \text{Domain size: } [a \times a \times b] \ , \tag{2.1}$$

where

$$a = \min\left\{\sqrt{S}, \sqrt[3]{\frac{mnk}{p}}\right\}, \quad b = \max\left\{\frac{mnk}{pS}, \sqrt[3]{\frac{mnk}{p}}\right\} . \tag{2.2}$$

Kwasniewski et al. used the red-blue pebble game [30] to model matrix-matrix multiplication dependencies and derived a tight sequential and parallel I/O lower bound proof. COSMA is I/O optimal for all combinations of parameters up to the factor of $\sqrt{S}/(\sqrt{S+1}-1)$. The schedule interpolates between a 2D and a 3D schedule, depending on whether S is large

enough. An important additional point about COSMA is that it allows a certain percentage of processors not to be used if this will result in a lower communication volume (see Fig. 5 in [31]).

2.2 Efficient GEMM in CUDA

This section presents how to implement an efficient GEMM in CUDA. We assume that the reader is familiar with CUDA C++ programming and the CUDA memory model [5].

Note that the following figures are not always drawn to scale.

2.2.1 Hierarchical GEMM Structure

To implement an efficient GEMM for GPUs, best use should be made of the different memory hierarchies. After loading data from slower to faster memory, the data has to be reused in as many maths operations as possible before loading again data from slower memory. In CUDA this means loading data from global memory to shared memory to the register file. A good approach in CUDA is to decompose the computation into *thread block tiles*, *warp tiles* and *thread tiles*, which closely mirrors the NVIDIA CUDA programming model. Fig. 2.1 shows the complete GEMM hierarchy of an example multiplication using single precision floating-point numbers, which will be now explained in more detail.

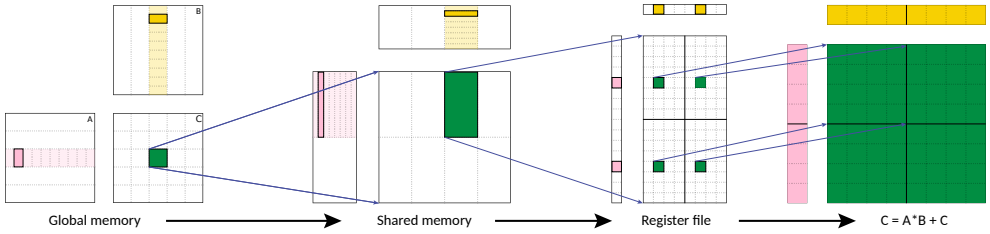


Figure 2.1: The complete GEMM hierarchy.

The parameters used for this example are:

- M: 640
- N: 640
- K: 640
- `THREADBLOCK_TILE_M`: 128
- `THREADBLOCK_TILE_N`: 128
- `THREADBLOCK_TILE_K`: 640
- `WARP_TILE_M`: 64
- `WARP_TILE_N`: 32
- `THREAD_TILE_N`: 8
- `THREAD_TILE_M`: 8
- `LOAD_K`: 8

M, N and K are the dimension of the matrices. `THREADBLOCK_TILE_M`, `THREADBLOCK_TILE_N` and `THREADBLOCK_TILE_K` define how much of the matrix multiplication a single thread block computes. `WARP_TILE_M` and `WARP_TILE_N` define how much of a thread block a single warp computes. `THREAD_TILE_M` and `THREAD_TILE_N` define how much of a warp block a single thread computes. `LOAD_K` defines the tile size of the tiles that are worked on in one iteration of the main loop. This example assumes a warp size of 32.

Thread Block Tile

Each thread block computes a portion of the output C by looping over the K dimension and iteratively loading tiles of the input matrices A and B to compute an accumulated matrix product. Fig. 2.2 shows the computation of a single thread block and the data it uses. At the thread block level in each iteration in the main loop, one tile of size `LOAD_K` of A and B is loaded from global memory into shared memory, while the accumulator of C resides in the register file, the fastest memory, since it is going to be updated once per maths operation.

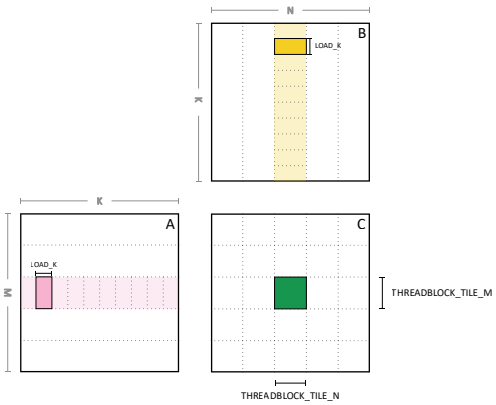


Figure 2.2: A 640x640 * 640x640 GEMM decomposed into the computation performed by 128x128 thread blocks. The data used by one thread block is highlighted. The part of C that the thread block computes is shown in green. To achieve this, it uses the pink part of A and the yellow part of B. In each iteration of the main loop, `LOAD_K` rows/columns are loaded from global into shared memory.

Warp Tile

The thread block tile is then further divided into warp tiles as shown in Fig. 2.3. Once the tiles of A and B are stored in shared memory, each warp again loops over the K dimension and loads a small fragment into the register file. Since multiple threads request the same data from shared memory, a warp's broadcast ability is used to reduce the number of times shared memory is accessed.

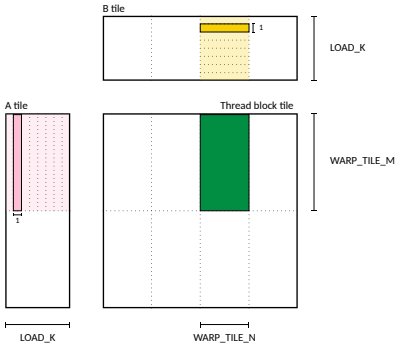


Figure 2.3: Decomposition of a 128x128 thread block tile into 64x32 warp tiles. The data used by one warp is highlighted. The part of C that the warp computes is shown in green. To achieve this, it uses the pink part of A and the yellow part of B. In each iteration of the loop, 1 row/column is loaded from shared memory into registers.

Thread Tile

The warp tile is divided into thread tiles, which are computed by individual threads. The straightforward way to divide a 64x32 warp tile into 8x8 thread tiles can be found in Fig. 2.4. However, the decomposition in Fig. 2.4 is not optimal as the stride to access the shared and global memory is 8. For 32-bit floating-point numbers, the optimal value for minimizing bank conflicts and maximizing efficiency is 4, since 4 floats = 128 bits, which is the largest amount that can be moved by a single memory instruction in CUDA. This better decomposition can be found in Fig. 2.5. Note that all threads in the same row or column load the same data from shared memory into the register file. This will result in an overall 8x8 thread tile, as seen in Fig. 2.6.

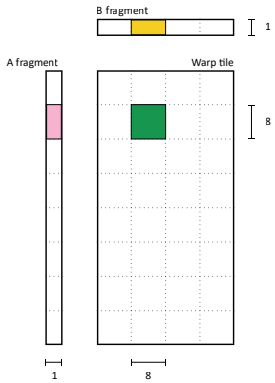


Figure 2.4: Non-optimal decomposition of a 64x32 warp tile into 8x8 thread tiles. The data used by one thread is highlighted.

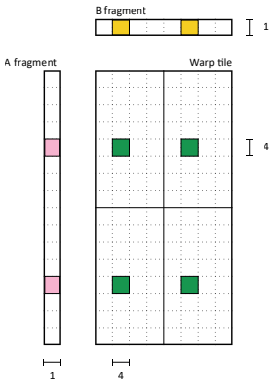


Figure 2.5: Optimal decomposition of a 64x32 warp tile into 8x8 thread tiles. The data used by one thread is highlighted.

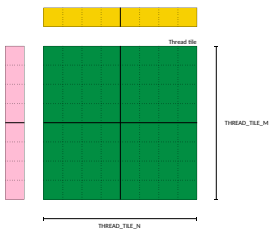


Figure 2.6: The 8x8 thread tile where the multiplication happens.

2.2.2 Optimizations

In this subsection some further techniques are introduced to make GEMM more efficient.

Split K

As seen in Fig. 2.2, this configuration launches 25 thread blocks to compute the result of $C = A * B$. If this kernel is run on a NVIDIA Telsa V100 which has 80 SMs (Streaming Multiprocessors), the kernel does not use 55 SMs and thus can only achieve 31.25% of peak performance.

Now there are two solutions for this problem. The first and easier one is to decrease `THREADBLOCK_TILE_M` or `THREADBLOCK_TILE_N`, which means more thread blocks are launched. As long as the memory bandwidth will not become a bottleneck (see Section 3.1.4), and the overhead of the additional started thread block is smaller than the performance gain, this will result in higher percentage of peak performance. There may be cases where this approach does not work well, for example if M and N are small in comparison to K or it is not possible to further decrease `THREADBLOCK_TILE_M` nor `THREADBLOCK_TILE_N`. In that case is possible to split the K dimension: Instead of launching only one thread block per tile in C, multiple thread blocks are started for the same block in C. Every thread block is responsible for one part of the K dimension. Fig. 2.7 shows an example for SPLIT_K = 2 and `THREADBLOCK_TILE_K` = 320. The thread block now only iterates through half of the K dimension while another thread block is responsible for the second half of the K dimension. Since there are now multiple thread blocks responsible for the same tile in C, the partial results of the individual thread blocks must be reduced to get the correct result.

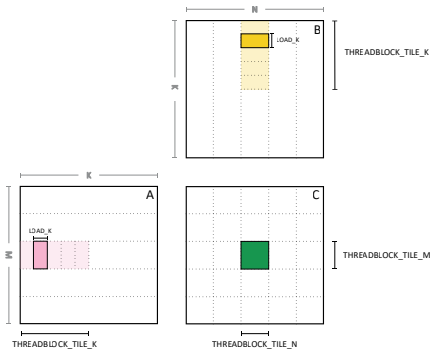


Figure 2.7: An example for one 128x128 thread block with SPLIT_K = 2 and `THREADBLOCK_TILE_K` = 320. The data used by one thread block is highlighted. The part of C that the thread block computes is shown in green. To achieve this, it uses the pink part of A and the yellow part of B. In each iteration of the main loop, `LOAD_K` rows/columns are loaded from global into shared memory. This thread block iterates only over half of the K dimension, the other half is computed by another thread block. The partial results must be reduced.

Slice K

An additional optimization employed by cuBLAS is Slice K. Slice K is essentially the same as Split K, but while Split K works on the thread block level, Slice K works on the warp level. Using Slice K, it is possible to start more threads and using more warps, if the problem would otherwise be too small, to utilize more resources of the GPU. If slice K is used, a warp is not responsible for the whole K dimension, but only one slice of it. Similar to Split K, a reduction must be made at the end.

Software Pipelining / Double Buffering

The hierarchical GEMM structure (Section 2.2.1) uses many registers to store the accumulator and fragments. This limits the achievable occupancy of the kernel and therefore the GPU's ability to hide memory latency and other stalls. As previously discussed, the data is loaded from global memory to shared memory to the register file. To hide these data movement latencies, software pipelining can be introduced to overlap memory accesses with computation.

The global memory and shared memory loads can be double-buffered. This comes at a cost of twice the amount of shared memory and twice the number of fragments used per thread. As shown in Fig. 2.8, all stages can be executed concurrently and the output of each stage can be fed to its dependent stage during the next iteration. Once the data is stored into shared memory, a call to *__syncthreads()* makes sure there are no race conditions. This approach has one more benefit; one call to *__syncthreads()* can be eliminated, since one shared memory buffer is read while the other one is written to.

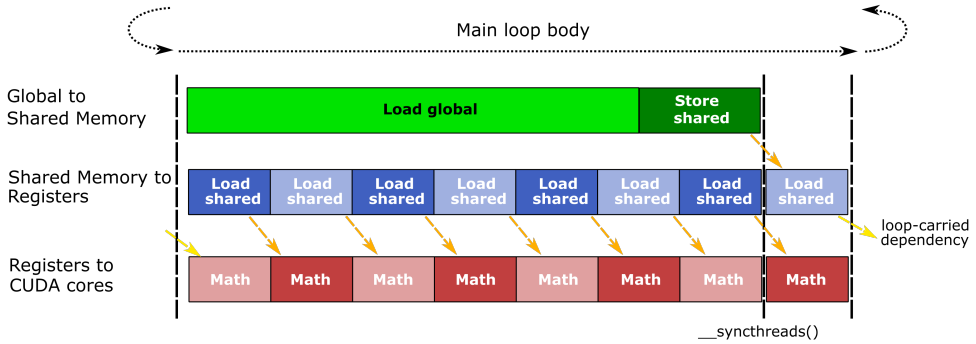


Figure 2.8: Three concurrent streams of instructions interleaved in the main loop. Source: [9]

Thread Block Swizzle

In Fig. 2.2, 25 thread blocks are required to cover the entire square matrix C. The straight-forward way is to use a grid with dimensions $(5, 5, 1)$ and a

simple mapping from thread block ids to the problem (using *blockIdx.x* and *blockIdx.y* directly). This will result either in a classic row-major (Table 2.1) or in a classic column-major layout, depending on how the launch order of the thread blocks is implemented by the driver or hardware.

Table 2.1: Thread block layout: Row-major

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

Table 2.2: Thread block map: `SWIZZLE` = 2

(0,0)	(0,2)	(0,4)	(0,6)	(0,8)
(0,1)	(0,3)	(0,5)	(0,7)	(0,9)
(1,0)	(1,2)	(1,4)	(1,6)	(1,8)
(1,1)	(1,3)	(1,5)	(1,7)	(1,9)
(2,0)	(2,2)	(2,4)	(2,6)	(2,8)

Instead of working on one row at a time, it might be beneficial to work on several rows at once, with the aim of increasing the L2 cache hit rate. This is the approach used by CUTLASS [10]. We have achieved this by introducing a `SWIZZLE` factor which describes on how many rows to work with simultaneously.

The normal grid has the dimensions $(Grid_X, Grid_Y, 1)$ and the usual *blockIdx.x* and *blockIdx.y* are used to map the thread block the corresponding tile in C. The swizzled grid has the following dimensions:

$$(Grid_X * \texttt{SWIZZLE},\ (Grid_Y + \texttt{SWIZZLE} - 1)\ /\ \texttt{SWIZZLE},\ 1)\ . \tag{2.3}$$

The ids are remapped follows:

$$blockIdx.x_{\texttt{SWIZZLE}} = blockIdx.x\ /\ \texttt{SWIZZLE} \tag{2.4}$$

and

$$blockIdx.y_{\texttt{SWIZZLE}} = (blockIdx.y * \texttt{SWIZZLE}) + (blockIdx.x \bmod \texttt{SWIZZLE})\ . \tag{2.5}$$

Note that all variables are integers.

Table 2.2 shows an example for `SWIZZLE` = 2. If `SWIZZLE` does not evenly divide $Grid_Y$, one additional check is required that $blockIdx.y_{\texttt{SWIZZLE}} \leq blockIdx.y$, otherwise there might be an out-of-bounds error, depending on how robustly the kernel is implemented. In cuCOSMA it is necessary to make this check because the kernel only performs a bounds check if the thread block tiles do not evenly divide the matrix dimensions.

Thread Mapping

This section discusses how to map threads to thread tiles within a warp tile for maximum efficiency. Suppose a 32x16 warp tile with 4x4 thread tiles, as seen in Fig. 2.9.

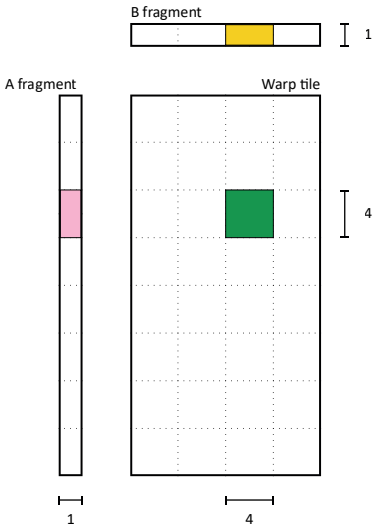


Figure 2.9: The decomposition of a 32x16 warp tile into 4x4 thread tiles.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

Table 2.3: Thread map: Row-major

0	1	16	17
2	3	18	19
4	5	20	21
6	7	22	23
8	9	24	25
10	11	26	27
12	13	28	29
14	15	30	31

Table 2.4: Thread map: Optimal. Source: [13]

First, the thread id of thread inside a warp is computed:

$$threadIdx_{warp} = threadIdx.x \bmod 32 \,. \tag{2.6}$$

Note that threads are only launched in the x dimension ($threadIdx.y$ and $threadIdx.z$ are always 1).

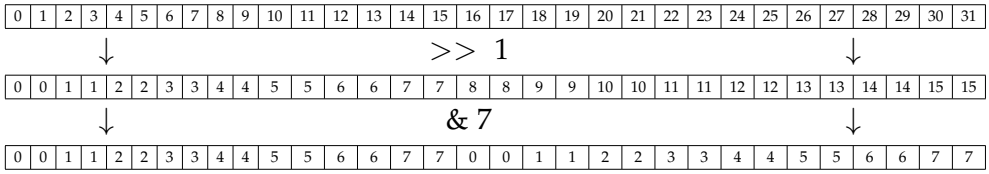
The simplest way to map threads to thread tiles is to use a row-major (Table 2.3) or column-major mapping. However, it turns out that on NVIDIA GPUs neither of these is optimal. The row-major approach doubles the amount of shared memory transactions for loading B, while the column-major approach doubles the amount of shared memory transactions for loading A. To get the optimal transaction count for loading both A and B, a different mapping has to be chosen. One such mapping can be found in Table 2.4.

The most convenient way to calculate the mapping in Table 2.4 is through bitwise operations.

LaneIdy The LaneIdy is the same for two consecutive threads and is then always increased by 1 for the next two. It is also limited by the height of the warp tile. Table 2.5 shows how the LaneIdy is constructed, first the ids are shifted by one to the right such that two consecutive threads have same id

and there is always an increase of 1 after two threads. To make sure that the LaneIdy does not go beyond the height of the warp tile, the *and* operation is used to select the lower 3 bits of the id ($7_{10} = 111_2$).

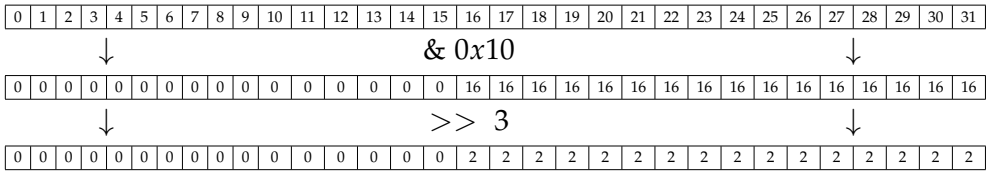
Table 2.5: How LaneIdy is computed.



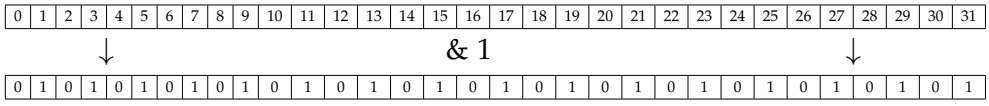
LaneIdx The LaneIdx alternates between 0 and 1 for the first 16 threads and between 2 and 3 for the remaining half. First the displacement is calculated for the difference between the two halves, see 2.6a. The first *and* operation sets all values to 16 if the thread id is $\geq$ 15, and afterwards the value is divided by 8 so that the upper half of the threads has the value 2, ($10_{16} = 10000_2$). To perform the alternation, an *and* operation is used to cut off the last bit, see 2.6b. To get the final result, the *or* operations is used to add up the intermediate results, see 2.6c.

Table 2.6: How LaneIdx is computed.

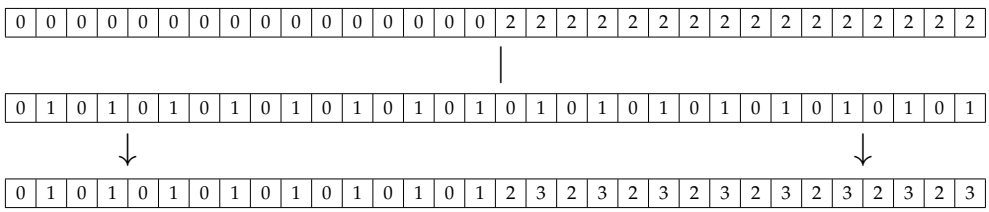
(a)



(b)



(c)



In conclusion, the coordinates of a thread can be computed as

$$LaneIdx = (((threadIdx_{warp} \ \& \ 0x10) \ >> \ 3) \mid (threadIdx_{warp} \ \& \ 1)) \quad (2.7)$$

and

$$LaneIdy = ((threadIdx_{warp} \ >> \ 1) \ \& \ 7) \ . \quad (2.8)$$

We generalized this approach to work with all shapes of warp and thread tiles.

We do not know if this is the best possible mapping. We could not find a better one and using the presented mapping will result in a shared memory transaction count that is equal of that of CUTLASS and cuBLAS. There are probably other mappings that can achieve the same.

We have not found out the exact reason that causes that certain mappings result in more loading transactions than others, but since the loading from shared memory into register relies heavily on broadcasting, our best guess would be that there is some kind of limitation in the broadcasting mechanism inside NVIDIA GPUs, as already pointed out by [13].

Epilogue

While the layout described in Fig. 2.5 is well suited to perform the multiplication efficiently, this layout is not optimal for writing the result back to global memory, as the values are not nicely distributed for coalesced memory accesses. This problem can be solved by using vector stores, if this is not possible, another solution is required.

The epilogue is an additional phase in which threads of the same warp exchange data through shared memory to access global memory using a more efficient access pattern. We will now demonstrate the epilogue stage of a 16x32 warp tile with 4x4 thread tiles, as shown in Fig. 2.10.

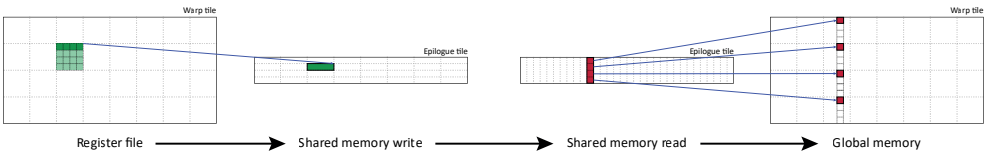


Figure 2.10: The whole epilogue process. Each warp shuffles its values by using shared memory.

The whole procedure is split into 4 iterations, one for each row of the thread tile. In the first iteration, each thread writes its top row into the epilogue tile, which is stored in shared memory, using the same mapping as in the multiplication. The epilogue tile has the same N dimension as the warp tile, the M dimension is collapsed by 4, see Fig. 2.11.

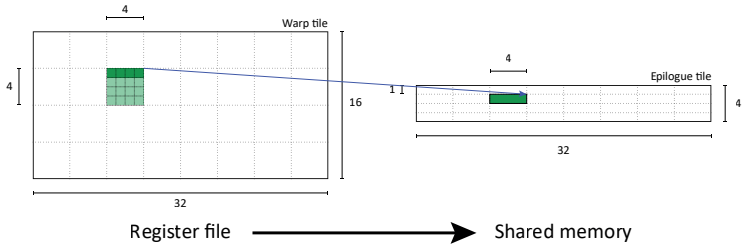


Figure 2.11: Each thread writes one row in the epilogue tile.

Afterwards, the threads are rearranged in a way that each thread reads a column of height 4 from the epilogue tile which allows now for coalesced storing to global memory, see Fig. 2.12.

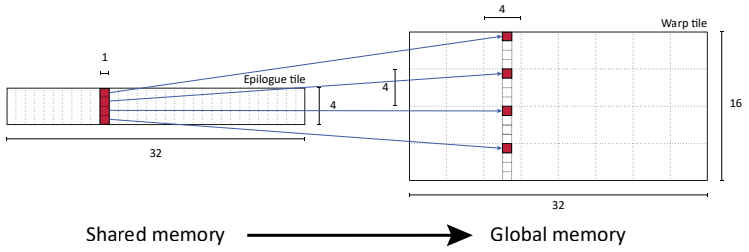


Figure 2.12: Each thread reads one column from the epilogue tile and stores the value into global memory.

The mapping shown can be generalized for all thread and warp tiles, but it is not equally efficient for all. With badly chosen tile sizes bank conflicts will occur and storing to global memory is still not quite optimal. For 4x4 thread tiles, a 16x32 warp tile is ideal for a row-major layout and a 32x16 warp tile is optimal for a column-major layout.

2.3 State-of-the-Art MMM Algorithms

In this section the approaches by current state-of-the-art implementations are described.

2.3.1 cuBLAS

cuBLAS [1] is a library developed by NVIDIA that integrates BLAS (Basic Linear Algebra Subprograms) into the CUDA environment.

The GEMM implementation from cuBLAS works as follows: There are several handwritten assembly kernels ([2] says 24 for CUDA cores and 15 for tensor cores.) for different tile sizes and at runtime a heuristic is applied with the goal to select the best kernel for the given problem. Since the cuBLAS kernels are written directly in assembly, it is usually not possible to

outperform cuBLAS with a compiler-generated assembly using the same tile sizes for the same problem. However, cuBLAS' disadvantage is that there are only a certain number of pre-written kernels, so there are problems for which the kernels do not fit well. If the kernel should be modified, e.g. to use an element-wise function as epilogue, this would have to be done directly in the assembly or in an additional kernel. Furthermore, since the heuristic to choose the kernel is run at runtime, the heuristic cannot take too long to decide what kernel to launch and might choose poorly. cuBLAS represents the state of the art in GEMM performance.

2.3.2 CUTLASS

CUTLASS [8] (CUDA Templates for Linear Algebra Subroutines) is an open source collection of CUDA C++ templates and abstractions for implementing high-performance GEMM computations developed by NVIDIA.

The main difference to cuBLAS is that the compiler generates the assembly and that the tile sizes have to be determined by the user. The main disadvantage is that CUTLASS decides at compile time which kernel to run without knowing the matrix dimensions and the alignment of the matrices, this leads to the fact that CUTLASS cannot use vector loads for loading from global memory. Furthermore there is the question of how to choose the best tile sizes for a given problem, which CUTLASS does not answer. This is a particular problem because the average user might have difficulty predicting which configuration will perform best. Another limitation of CUTLASS is that it only supports thread tiles which are multiples of 4, there are again problems for which CUTLASS might not fit well. Since CUTLASS is written in CUDA C++, adding additional functionality is straightforward.

2.3.3 rocBLAS

rocBLAS is AMD's library for BLAS on ROCm (Radeon Open Compute platform). It is implemented in the HIP programming language and optimized for AMD's GPUs. It has the same advantages and disadvantages as CUTLASS. However, compared to CUTLASS it is not possible to add custom tile sizes and epilogues. We observed that rocBLAS always uses the same tile sizes for all problems, which is suboptimal.

Chapter 3

cuCOSMA

In this chapter, we will describe our implementation of COSMA on a single GPU, which includes a schedule generator and a matrix-matrix multiplication kernel.

3.1 Schedule Generator

This section describes how the schedule generator works. Note that the presented schedule generator only focuses on NVIDIA GPUs, but can be easily adopted for AMD graphics cards.

The schedule generator is not based on a theorem that proves that it always returns the fastest/optimal tile sizes, unlike COSMA. In COSMA, the matrix-matrix multiplication were represented as a red-blue pebble game to create an I/O lower bound proof. The schedule generator is a collection of heuristics that were collected during this thesis, while keeping the main idea of COSMA in mind: minimizing the communication volume.

We created an easily customizable framework to try out many techniques to generate schedules.

3.1.1 Overview

A basic overview, on how the schedule generator works, can be found in Fig. 3.1.

As input the schedule generator receives the matrix dimensions, additional scaling parameters and the specification of the GPU on which the multiplication is performed. The schedule generator then loops over all possible configurations and verifies if the schedule meets all desired constraints. If this is the case, the schedule generator ranks the schedule according to certain heuristics and saves the best schedule found. The output of the schedule

generator describes the configuration of the kernel to be used for this problem. All these parameters must be added to the kernel at compile time to allow the compiler to perform all necessary optimizations.

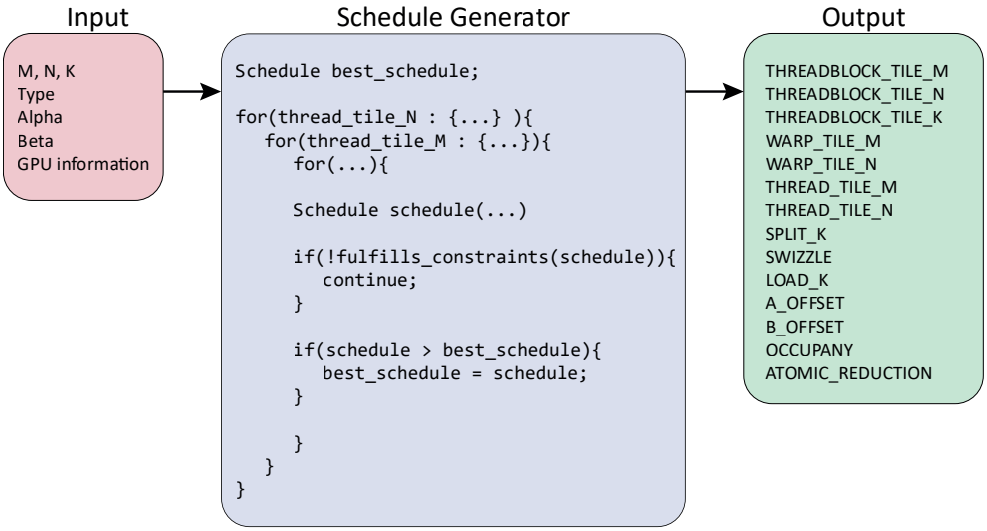


Figure 3.1: Basic overview of the schedule generator.

3.1.2 Input

As input the schedule generator receives the matrix dimensions M, N and K and the type (float, double, int, etc.) to be used. Furthermore, it receives alpha and beta as input because the goal is to calculate $C = \alpha \cdot (A * B) + \beta \cdot C$ with $A \in \mathbb{R}^{mxk}$, $B \in \mathbb{R}^{kxn}$, $C \in \mathbb{R}^{mxn}$ and $\alpha, \beta \in \mathbb{R}$. It is also possible to use an element-wise function instead of scaling, such as the sigmoid function. The remaining input parameters are summarized with *GPU information*, they describe the configuration of the GPU which performs the multiplication. All the information about the GPU can either be found in the CUDA C++ Programming Guide [11] or directly on the GPU by using *cudaGetDeviceProperties* [7]. A more detailed list of the values used can be found in the appendix: Section A.3.

One important input not included is the layouts of the individual matrices (row or column-major layout). This thesis focuses only on row-major matrices and therefore this aspect was ignored.

3.1.3 Schedule Generator

A simplification of how the schedule generator is implemented can be found the appendix in Listing A.1.

We believe that it is not feasible to find a closed formula, as in Equation (2.2) and Equation (2.1), to decide which tile sizes will perform best. The main problem is the following: On the CPU there are a certain number of processors and one block is assigned to each CPU. On the GPUs thread blocks are assigned to SMs and often it is necessary to assign several blocks to one SM. So, it is not possible to measure how good the schedule is by looking at how many thread blocks are launched or how many SMs are occupied. Furthermore, we have no direct control over how the GPU schedules work.

As shown in Listing A.1, there is one for loop for nearly every variable of the output. The implementation is a brute force method to find out which schedule is best, since the cuCOSMA kernel can support nearly all combinations of tile sizes. If the tile sizes are limited, the schedule generator can be simplified. `A_OFFSET` and `B_OFFSET` are additional paddings in shared memory to avoid bank conflicts and can be determined independently like `SWIZZLE`. In our case (row-major layout and single precision floating-point numbers): `A_OFFSET` $= 4$ and `B_OFFSET` $= 0$.

The first for loop iterates over possible `LOAD_K` parameters. For 32-bit floating-point numbers it is usually 8, however, for small K it can be chosen even smaller.

Then the schedule generator searches for the best tile sizes to divide the GEMM into thread, warp and thread block tiles. Smaller tiles are used as a base for a bigger one, so the bigger tiles are always multiples of the smaller ones. The tile sizes are limited by the available registers of the GPU, this makes the schedule generator more efficient as the runtime is independent of the matrix dimension.

The last loop iterates over the `SPLIT_K` parameter. The `SPLIT_K` parameter is limited in the following way: Assume a hypothetical multiplication with K tending to infinity, then the upper limit is chosen such that all threads of the GPU can participate in the multiplication. The multiplication must be distributable to all available SMs and warps. The factor 2 is present because at least an occupancy of 2 is enforced by the schedule generator.

Now all that remains is to discuss what kind of constraints are imposed on the schedule and how it is decided that one schedule is better than another.

3.1.4 Constraints

The imposed constraints are verified by the schedule generator as soon as possible to make it more efficient, and not only in the innermost loop.

At first, the number of constraints in the schedule generator were kept to a minimum and most of the tile size selection was performed through the ranking system. However, it is difficult to predict which tile sizes will per-

form best. Finally, the ranking system was simplified, but more constraints were added. Since the cuCOSMA kernel is flexible, it is likely that some of the best configurations are not covered by the schedule generator.

Thread Tiles

The first constraint is what thread tile sizes are allowed in the schedule. This property also depends heavily on the capabilities of the kernel. The cuCOSMA kernel can support all possible thread tile sizes (as long as they do not run into the register limit), while for example the kernel of CUTLASS only supports square thread tiles sizes that are multiples of 4.

Assuming that the kernel can support all combinations of thread tiles, it sometimes makes sense not to consider all possibilities. There is a compromise: more specialized tile sizes may fit a specific problem better, but make the main loop in the kernel more inefficient because vector loads cannot always be used.

We will now present some combinations of thread tiles and their advantages and disadvantages.

No Restrictions The easiest choice is to impose no restrictions on the thread tiles and allow the schedule generator to try all possible configuration. This allows the thread tiles to be precisely adapted to a problem and is especially useful for small matrices. However, with large matrices, the main loop may become too inefficient because there are too many non-vector loads, resulting in more load instructions and reducing the ratio of FMA to non-FMA instructions. For large matrices, it can be an advantage to prefer vector loads over more specific tile sizes and to do more work overall than if more specialized tile sizes were chosen.

Even Numbers If the schedule generator only tries even numbers as thread tiles, the possibility to create exact tile sizes for small matrices is lost. However, the problem with the non-vector loads is mitigated because vector load of size two can at least be used.

Multiples of 4 If the schedule generator tries to use only numbers that are multiples of 4 as thread tiles, vector loads of size 4 can always be used. However, this can lead to inefficiency with smaller matrices because the thread tiles are too large and therefore too much unnecessary work is performed. This is the approach to be used with CUTLASS.

Multiples of 4 $\cup$ $\{1, 2\}$ Here, the previous three approaches are combined. The schedule generator may try multiples of 4 as well as thread tiles of size 1 and 2 to cover smaller matrices well. This sequence can represent small

and large matrices well and does not suffer from inefficiency if larger thread tiles are chosen.

Powers of Two Until 256-bit This is the same variant as above, but no thread tiles greater than 8 are allowed. We could observe that these are often the best performing configurations and practically all reasonable tiles can be covered by using only these thread tiles. This is also the approach we used in the end, but we also enforced that all thread tiles are square tiles. Note that this does not always give the best decomposition. It may well be that in a specific case one of the other approaches works better. In the end, it all depends on the problem and if it is worth to change more specific tile sizes for a less efficient kernel.

Occupancy

Matrix-matrix multiplication on the GPU requires many registers to store the accumulator and fragments, which greatly limits the occupancy that can be achieved. Often it is advantageous to use smaller tile sizes to increase the occupancy. We observed that it is worthwhile to achieve at least an occupancy of 2 and enforced this by limiting the registers and shared memory per thread block. The occupancy is passed to the compiler via *__launch_bounds__*.

Memory Bound Depending on the GPU and on the kernel configuration, GEMM kernels may be limited by the global memory bandwidth. (See Section A.1 for an example.) If the kernel is limited by the available memory bandwidth, the performance heavily depends on the L2 cache. One way to increase the L2 cache hit rate using the same tile sizes is to reduce the occupancy by using a few more unnecessary registers or shared memory allocations (see [13]).

General

The remaining constraints are described that are imposed on the schedule. Some of them are unavoidable (because they are bound to the hardware), others are not necessary, but make the schedule more efficient.

Register and Shared Memory Usage After the desired occupancy is known, the register and shared memory limit per thread block, warp and a thread can be computed. To each thread 32 additional registers are assigned, which are not used to store the accumulator or fragments, but can be used for general calculations (and as a precaution, since the compiler does not always find the best assignment).

Hardware Limitations All constrains imposed by the hardware are listed.

- Maximum Number of Threads per Block:
  In CUDA at most 1024 threads per block can be started.
- Warp Size:
  The warp size in CUDA is 32. This means the warp tile has to be divided into 32 thread tiles.
- Number of 32-bit Registers per SM:
  How many registers the hardware can provide per SM. (Usually 65536).
- Maximum Number of 32-bit Registers per Thread Block:
  How many registers a thread block can use. The same as in "Number of 32-bit Registers per SM" but not in compute capability 5.3 and 6.2.
- Maximum Number of 32-bit Registers per Thread:
  In CUDA each thread can address a maximum of 255 registers.
- Maximum Amount of Shared Memory per SM:
  How much shared memory the hardware can provide per SM.

Alignments All alignment constraints are presented.

- Thread Block:
  For single precision floating-point numbers, if more than one thread block is used in a given dimension, to ensure vectorized access the thread block has to a multiple of 4. This applies to the M, N and the K dimension.
- Warp Tile:
  For single precision floating-point numbers, if there is more than one warp tile in a given dimension, the following alignment constraints must be met to allow for vectorized access. If the thread tile is greater than or equal to 4, then the warp tile has to be a multiple of 4. If the thread tile is greater than or equal to 2, then the warp tile has to be a multiple of 2. This applies to the M and N dimension.

`LOAD_K` The `LOAD_K` parameter is usually 8 for 32-bit floating-point numbers, but for matrices with a small K dimension, it makes sense to choose it even smaller, otherwise unnecessary work will be performed and no advantage can be taken from software pipelining (Section 2.2.2). The following candidates are considered: 8, 4, 2, 1. The distance is the same as in the vector loads instructions. Once a larger number fits such that software pipelining can be used, it accepted and smaller ones are not considered.

Map Threads Evenly to Global Tiles Although not required by the cuCOSMA kernel, it is usually desired to be able to map the threads evenly

to the tiles of size LOAD_K∗THREADBLOCK_TILE_N and LOAD_K∗THREADBLOCK-_TILE_M (see Fig. 2.2).

3.1.5 Ranking

After the schedule generator has verified that the schedule meets all constraints, it must decide which is the best schedule.

The first thing the schedule generator examines is how many CUDA cores of the GPU the kernel can take advantage of, see Section 3.1.5. If one kernel can use more CUDA cores than another, it is considered as the best new kernel found so far. If two kernels use the resources of the GPU equally well, the communication volume of the kernels is compared (see Section 3.1.5), and the kernel with the smaller volume is taken as the new best kernel. If there are two equally good schedules, the schedule generator prefers the one with smaller SPLIT_K, because the reduction means a small additional effort. If SPLIT_K is the same (happens because of symmetry), the schedule generator prefers the schedule with larger tiles in the N dimension, such that the schedule generator becomes deterministic and the epilogue becomes more efficient.

Use All CUDA Cores of the GPU

The most important property to examine is whether the kernel can use the whole GPU. The main reason for this step is the following: Larger tile sizes are usually more efficient than many smaller ones. However, if more small ones are used, this will result in more thread blocks launched and more CUDA cores used. This is especially important for smaller matrices, which might not occupy all SMs.

To measure this property, the schedule generator calculates how many of the available CUDA cores a kernel can meaningfully use. It calculates how many cores compute at least one element of the output matrix C. For a warp tile of size 16x32 with 4x4 thread tiles, which calculates a matrix multiplication of size 16x16, only 16 threads are meaningfully used because the remaining 16 perform unnecessary work. The schedule generator then selects the schedule where most CUDA cores can participate in a meaningful way. This metric is upper bounded by the number of CUDA cores of the GPU.

Communication Volume

If two kernels use the resources of the GPU equally well, the schedule generator compares the communication volume of the kernels, and the kernel with the smaller volume is taken as the new best kernel.

On GPUs with 3 memory hierarchies: global memory, shared memory and register files, there are multiple types of memory movements. To simplify this, only loading and storing from global memory (as it is the slowest memory) and on loading from shared memory has been considered. The schedule generator puts more emphasis on the global communication volume and therefore always selects the schedule with the smaller global communication volume. If two schedules have the same global communication volume, the shared communication volume is compared and the schedule with smaller shared communication volume is chosen as the new best schedule.

At a closer look, it is also possible to consider the following: The storing in shared memory and registers and loading from registers could be included in the calculations. To write data from global memory to shared memory, it is first stored in registers and then loaded from registers into shared memory. In the new *Ampere* architecture this is no longer the case and it can be written directly into shared memory without any intermediate storage, and asynchronous shared memory loads can be used. Also note that C is loaded directly into the registers thus bypassing shared memory, and A and B are stored into shared memory. In addition, the shared memory accesses in the epilogue were ignored because we assumed that vector stores could always be used.

In MPI, the matrix-matrix multiplication is distributed to the ranks, which are divided in a grid, and communication volume is computed over the area of the boundaries between the ranks. The big difference is that thread blocks in CUDA cannot send messages to each other, but only communicate via global memory, while MPI is based on the principle that ranks communicate with each other.

Global Memory To calculate the total communication volume of global memory, it is computed how much a single thread block causes and then multiplied by the total number of blocks.

To calculate the volume for C, a case distinction is required. If $\beta = 0$ in the equation $C = \alpha \cdot (A * B) + \beta \cdot C$ with $A \in \mathbb{R}^{mxk}$, $B \in \mathbb{R}^{kxn}$, $C \in \mathbb{R}^{mxn}$ and $\alpha, \beta \in \mathbb{R}$, C does not have to be loaded because it can be overwritten, otherwise it has to be loaded first. Therefore,

$$V_{C_{Global}} = \begin{cases} \texttt{THREADBLOCK\_TILE\_M} * \texttt{THREADBLOCK\_TILE\_N} & \beta = 0 \\ 2 * \texttt{THREADBLOCK\_TILE\_M} * \texttt{THREADBLOCK\_TILE\_N} & \beta \neq 0 \end{cases} . \quad (3.1)$$

The formulas for the volumes for A and B are:

$$\begin{aligned} V_{A_{Global}} &= \texttt{THREADBLOCK\_TILE\_M} * \texttt{THREADBLOCK\_TILE\_K} , \\ V_{B_{Global}} &= \texttt{THREADBLOCK\_TILE\_N} * \texttt{THREADBLOCK\_TILE\_K} . \end{aligned} \quad (3.2)$$

The reason why THREADBLOCK_TILE_K is used and not K is because the split K approach needs to be considered. In the end the partial volumes are multiplied with the number of blocks launched as follows

$$V_{Global} = (V_A + V_B + V_C) * total\_thread\_blocks \ . \tag{3.3}$$

Shared Memory To calculate the volume loaded from shared memory to registers a similar approach as above can be used. First calculate at how much volume a warp causes and then multiply this by the number of warps launched. We assume that the broadcasting function works perfectly, i.e. , that a value only needs to be read once and is then distributed throughout the warp. The matrix C does not need to be considered, as it has already been saved directly into the registers.

One warp iterates over the THREADBLOCK_TILE_K dimension and loads the following amount from shared memory

$$\begin{aligned} V_{A_{Shared}} &= \texttt{WARP\_TILE\_M} * \texttt{THREADBLOCK\_TILE\_K} \ , \\ V_{B_{Shared}} &= \texttt{WARP\_TILE\_N} * \texttt{THREADBLOCK\_TILE\_K} \ , \\ V_{Shared} &= (V_A + V_B) * total\_warps \ . \end{aligned} \tag{3.4}$$

Note that the formula has been simplified, since not THREADBLOCK_TILE_K elements are loaded in the K dimension, but

$$\frac{\texttt{THREADBLOCK\_TILE\_K} + \texttt{LOAD\_K} + 1}{\texttt{LOAD\_K}} \tag{3.5}$$

elements. When loading A and B from global memory, boundary checks are present and zeros are written to the places in shared memory, where an out of boundary read in A or B would occur. When loading from shared memory is not necessary to perform boundary checks as the calculation is still correct.

3.2 Kernel

In this section the cuCOSMA kernel written in CUDA C++ is described. Due to limited time, only the multiplication of 32-bit floating-point matrices stored in row-major format was considered.

The kernel features the approaches introduced in Section 2.2. The user can choose the tile sizes, similar to CUTLASS. Compared to CUTLASS which is restricted to thread tiles that are multiples of 4, cuCOSMA can accept more flexible tile sizes, which is a performance advantage in certain situations. cuCOSMA assumes that it knows the matrix dimension at compile time and that each row is correctly aligned to allow for vectorized access. It

was implemented with the following philosophy: Given tile sizes and the matrix dimensions at compile time, it tries to achieve the fastest possible implementation using the specified configuration. The kernel does not try to optimize the given configuration, but leaves it up to the user/schedule generator.

We have tried to use the following invariant to make the code easier to understand, all compile time constants are written in `CAPTIAL LETTERS`.

To see the whole code, see *cuCOSMAV100.cuh*. There are different variations of the code, as different GPUs run faster with different implementations. The version that performed best on the NVIDIA Telsa V100 is presented.

3.2.1 Launch

This section describes how the kernel is launched.

Configuration

To specify the configuration, all variables are written to a *config.h* file. This way, the configuration can be managed centrally with one file and the compiler can put the constants directly into the code, so that they are available at compile time. An example of a configuration file can be found in Listing A.2.

Kernel

How the kernel is launched can be found in Listing A.3.

First the number of launched threads per thread block is calculated at compile time (Line 1). Afterwards, a check is added whether the C matrix needs additional scaling (Line 8). This is necessary if the multiplication is not performed or a split K approach is used with atomics.

Then the number of thread blocks to be started in each dimension is calculated, this varies depending on whether swizzling (Section 2.2.2) is used or not.

Finally, a distinction is made as to how the reduction is to take place if a split K is used. Here cuCOSMA offers two variants. The first one uses atomics, this has the advantage that no additional kernel has to be launched, but has the disadvantage that for `SPLIT_K` > 2 the result is not bit-exact when running the kernel multiple times because the order in which the threads sum up the values is not well-defined. The second variant is an additional reduction kernel. The kernel that performs the multiplication stores the partial results in an additional allocated array and the reduction kernel then performs the reduction over this additional memory and stores the result back to C. This approach has the disadvantage that an additional kernel

start is required, more memory is needed and the data passes through global memory several times, but the result remains bit accurate even with larger `SPLIT_K`.

3.2.2 Implementation

The exact implementation of the kernel is presented, it is divided it into different sections: The Prologue, which sets everything up before the Main Loop, and the Helper Methods methods used by Main Loop.

Prologue

The entire prologue, everything that comes before the main loop, is described.

Kernel signature The Kernel signature can be found in Listing A.4. The important thing to mention here is the *__launch_bounds__* statement on line 2. This gives the compiler the necessary information about how many threads are launched and what occupancy per SM is expected, such that NVCC can optimize correctly.

Assertions The assertion assumed by cuCOSMA can be found in Listing A.5. They specify what kind of tile sizes are supported. In summary, the tiles must be a multiple of each other and if a thread tile reaches a certain size, the warp tile must meet a certain alignment requirement. The alignment is only necessary if there is more than one warp tile in the specific dimension, so it is first calculated how many warp tiles there are in which dimension (line 1-2). Furthermore, a warp tile can only consist of 32 thread tiles.

Warp and Thread Mapping The warp and thread mapping used by cuCOSMA can be found under Listing A.6. First, it is calculated in which warp the thread is and what position a thread has in a warp (Line 7 - 8). Then the warp mapping is calculated (Line 10 - 11). Here it makes no difference if a row-major or column-major layout is chosen. Finally, the thread mapping for all possible configurations of thread tiles (Line 16 - 45) is computed, see Section 2.2.2.

Buffer Setup The setup used for the shared memory buffers and fragments can be found in Listing A.7. First the size of the buffers is computed (Line 1 - 5) and the buffers are initialized as a static shared memory allocation. The fragments and thread tiles can be initialized directly.

We have taken the approach of creating a static buffer for A and B. In addition, we have created an offset for each buffer that describes where

the threads can read from and write to the buffer (Line 10 - 14). In the main loop this offset is alternated using the variable *shared_memory_stage* (see Listing A.10 Line 67 - 82). This is the fastest implementation using a V100 with NVCC 11 we could find.

The swapping of these two buffers takes place in the main loop and the instructions used to accomplish this, reduce the FMA to non-FMA ration. This swapping is, along with address calculation and boundary checks, the limiting factor of a GEMM kernel. Combined with increasing the loop counter, these instructions make up all non-FMA instructions in the main loop that are not loading or storing. Since it is impossible to avoid loading or storing the data and increasing the loop counter, this is one of the keys to achieving peak performance.

Thread Block Remapping As described in Section 2.2.2, the thread block ids are remapped, see Listing A.8. Note that if `SWIZZLE` does not evenly divide `M_TILES`, there is need for a check such that there are no out of bounds memory accesses.

Allowed Vector Loads and Boundary Checks Here, it is calculated if some boundary checks are necessary and what kind of vector loads can be used.

Allowed Vector Loads At compile time it is calculated which vector loads to use, see Listing A.9. In the general case, it would be necessary to check if each row is well aligned. First, it is checked which vector loads are generally available (Line 1 - 9). Then it checked whether vector loads for A may be used in the last iteration (Line 12 - 16). Note that whenever a variable is true, it is allowed to use the corresponding vector load.

Boundary Checks The last iteration over the K dimension is special because boundary checks might be necessary. The conditions are found in Listing A.9 (Line 18 - 19). There are two conditions, as it makes a difference whether split K is used. Note that whenever a variable is true, it is required to make the check.

Main Loop

In this subsection the exact details of how the main loop is implemented are described. The implementation of the main loop can be found in Listing A.10.

In contrast to CUTLASS, which uses template meta-programing and tile iterators, cuCOSMA is written using functions that are marked with *__inline__*. The compiler can insert them into the main loop and optimize them directly

so that there is no jump instruction. These helper functions are described in detail in Section 3.2.2.

The first thing to compute is the K index where to start the computation (Line 2 - 3). This is not 0 in cuCOSMA because it iterates backwards trough the K dimension. This way, the unrolled tile is exactly the one where boundary checks might be necessary and no boundary checks are performed for the K dimension in the main loop itself.

Then, data is loaded from global memory into shared memory for the first time (Line 5 - 15). The template parameters specify what kind of vector loads to use and if certain boundary checks are required. This is the first part of the unrolled loop, the second part can be found on line 86 - 111.

Afterwards, a call to *__syncthreads* (Line 17) makes sure that all data is stored in shared memory before continuing. The counter for the K dimension is decreased for the first time (Line 18).

The outer loop is marked with *#pragma unroll 1*, this tells the compiler not to perform optimizations, that try to reuse data from previous iterations. The outer loop makes steps of the size `LOAD_K`, while the inner loop makes steps of size 1. The inner loop is marked with *#pragma unroll* for maximum performance.

In the inner loop, the following operations are performed, data is loaded from shared memory into the registers and the multiplication is performed. Here, double buffering is used again, but it is not implemented explicitly, but implicitly via the *#pragma unroll* statement, such that the compiler finds the correct instruction order. Two fragments are allocated and the data is loaded into a different one depending on whether K is even or odd (Line 26 - 42). The multiplication is implemented using the same strategy (Line 60 - 64).

To prepare the next iteration of the outer loop, data for the next iteration needs to be loaded from global into shared memory (Line 44 - 58). Note the *false* as a template argument, since no bounds check for the K dimension are necessary, and again a call to *__syncthreads* to make sure that all data is stored in shared memory. Furthermore, the memory offsets need to be adjusted (Line 67 - 82), as described in Section 3.2.2. The *shared_memory_stage* variable alternates between 1 and 0 by using an XOR operation.

The question arises at what location these two operations should be inserted between all the FMA instructions. If the program is written directly is assembly, it would be possible to control it precisely, but in a compiled language, the compiler makes the final decision. We inserted the load from global memory into the last iteration of the inner loop between the loads from shared memory and the block of FMA instructions. The placement of this

block has quite a performance impact, especially the location of the *__syncthreads* plays a crucial role. This could be optimized even more carefully to find the best place for the *__syncthreads* statement and is certainly one of the big advantages of working directly in assembly. The adjustment of the shared memory offset are inserted at the end, as this did not have an impact on performance.

After the completion of the two loops, the inner loop get executed one more time because the outer loop is unrolled by factor 1, but without loading data from global memory and without adjusting the shared memory offsets (Line 86 - 111).

Finally, the matrix C needs to be loaded in case it should be scaled/modified and afterwards the result is saved back to global memory (Line 119 - 124). These two methods have an extra shared memory allocation for the epilogue tile. The size of the shared memory was chosen such that all possible epilogue tiles fit. It is also possible to reuse the shared memory from the main loop, but this way a call to *___syncthreads* can be saved. The function *load_C* is described in Section 3.2.2. The function *store_C* is described in Section 3.2.2.

Helper Methods

In this subsection the helper methods used in the main loop are described in more detail.

load_Global The method *load_Global* is responsible for loading tiles from global memory to shared memory. It calls the methods *load_A_Global* and *load_B_Global*, which respectively load tiles either from the matrix A or B.

load_A_Global This method is only an intermediate step and its assembly cannot be found in the final binary. This method determines at compile time how the matrix A will be loaded. There are 3 possibilities: *load_A_Global_Vector4*, *load_A_Global_Vector2* and *load_A_Global_Single* one for each possible loading instruction of single precision floating-point numbers. It always tries to choose the largest possible vector type, where the threads can be mapped evenly to the tile.

load_B_Global This method is the same as *load_A_Global*, but for matrix B. It uses the following 3 methods: *load_B_Global_Vector4*, *load_B_Global_Vector2* and *load_B_Global_Single*.

load_A_Global_Vector4 This method is used to representatively explain how all methods that load data from global memory from A or B work, since they are all similar.

First, the vector type is defined and the corresponding dimension is shrunk accordingly (Line 7 - 8). The dimension is different for *A* and *B* and depends on how the matrices are stored, as with vector instructions only data that is stored continuously in memory can be loaded. Afterwards, it is calculated how many loads a single thread will perform (Line 11).

Since both matrices are stored in row-major, the threads are also assigned in a row-major way to the global tile (Line 16 - 17). Afterwards the thread computes its corresponding address in global memory (Line 19 - 27) and performs the necessary checks that is does not read or write out of bounds (Line 29 - 40). Then the data is read from global memory (Line 42 - 43) and stored into shared memory (Line 53 - 65).

When saving the global tile from *A* to shared memory there is one issue: The shared memory tile of *A* must be stored in column major format to allow for efficient loading into registers, while the global tile is stored in a row-major format. As seen in Fig. 2.3, a warp loads one column from the *A tile* into registers per iteration. If the *A tile* would be stored in row-major format, the accesses would be far apart in linear memory and bank conflicts would occur, while in column-major format all required values are nicely arranged. This means that the storing to shared memory for *A* cannot be vectorized. For matrix *B* this is not the case as both the global tile and the shared tile are stored in a row-major format.

load_Shared This method is responsible for loading data from shared memory into the register file. It calls the methods *load_A_Shared* and *load_B_Shared* to achieve this.

load_A_Shared This method is used again as an example of how data is loaded from shared memory into register, method *load_B_Shared* is similar.

This method is divided into 3 parts, one for each vector type. First, as much as possible is loaded with the largest type, and then the rest is loaded with the smaller ones if necessary. Note that the decision of what type of loads to use, is decided at compile time. This approach is the key of how cuCOSMA can support practically all variations of thread tiles.

On line 6, it is calculated how often the biggest vector type will be used and on line 8 how many threads are in the specific dimension per warp tile. Then the thread calculates its location in the tile and the corresponding pointer in shared memory (Line 16 - 21), finally it stores the result into to the registers (Line 23 - 25).

The whole procedure is then repeated for the smaller types. There is to notice, that if a greater type was used before, the address for the smaller

ones are different. See Fig. 3.2 for an example of a decomposition of a 56x28 warp tile into 7x7 thread tiles.

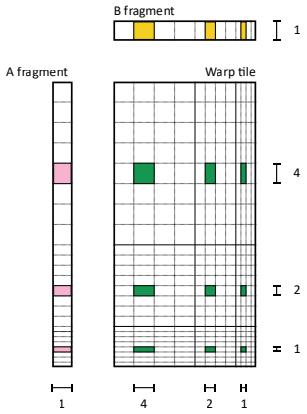


Figure 3.2: The decomposition of a 56x28 warp tile into 7x7 thread tiles.

compute_inner This method assumes that the fragments and the accumulator are stored in registers and performs the actual multiplication as in Fig. 2.6.

load_C This method is responsible for loading C from global memory if necessary and apply any scaling or other modification to the matrices before adding them. There are two possible methods for loading C: *load_C_Single* and *load_C_Vector*.

load_C_Vector and load_C_Single These methods work in the same way as *load_A_Shared*. However, a 2D surface must be processed. The outer method iterates over the columns, while the method with *Row* in the name, then iterates over the rows and applies the scaling with `BETA`. The vector variant then uses vector instructions to store the rows, while the single variant does the shuffling described in Section 2.2.2, but backwards.

store_C This method is responsible for storing the result back to global memory. If no split K approach is used, the vector variant can be used. Otherwise, the single variant is used, which then either performs the reduction using atomics or stores the result in an allocated array. The method *store_C_Single* performs the epilogue shuffling described in Section 2.2.2 for all possible tile sizes.

3.2.3 Reduction Kernel

The reduction kernel launches one thread for each element in C. This thread adds up all `SPLIT_K` values and performs scalings if necessary, see

Section A.6.3. A maximum of 256 threads per thread block is launched to allow more SMs to be used.

3.2.4 hipCOSMA

The CUDA C++ implementation of the kernel got converted to the HIP programming language to run it on AMD GPUs. The whole code was adopted without changes, only the warp size was adjusted from 32 to 64 and the thread tile mapping (Section 2.2.2) was removed.

Chapter 4

Experimental Evaluation

We evaluate the performance of cuCOSMA against the implementations of cuBLAS [1] and CUTLASS [8] and the implementation of hipCOSMA against rocBLAS [26] with various combinations of matrix-matrix multiplications. All multiplications were performed with matrices stored in row-major format and single precision floating-point numbers.

4.1 Evaluation Methods

This section describes how the measurements were made and what kind of multiplications were tested.

4.1.1 Architecture and Implementation Details

The benchmarks were performed using the *Ault* cluster and the XC50 compute nodes of the CSCS (Swiss Natinonal Supercomputing Center). The used nodes adhere to the specification described in Table 4.1.

Each kernel was run 110 times for each matrix-matrix multiplication, with the first 10 iterations being warm-up rounds, where the kernel time was not measured. In the next 100 iterations the kernel time was measured. In addition, after each iteration, a simple kernel was launched to flush the L2 cache.

NVIDIA

All kernels were compiled using NVCC (NVIDIA CUDA Compiler [14]) with the following flags: `-O3 -std=c++11 -gencode arch=compute_70,code=sm_70 -lcublas`. The turbo boost of the accelerators were disabled using `export CUDA_AUTO_BOOST=0`. To measure the performance of the kernels *nvprof* [6] was used with the following options:

Table 4.1: Specification of the Ault05/Ault06 and Ault20 nodes of the Ault cluster and the XC50 compute nodes of the Piz Daint cluster.

	Ault05/Ault06	Ault20	XC50
Sockets	2	2	1
Cores/Socket	18	64	12
Threads	72	256	24
CPU Model	Intel® Xeon® Gold 6140	AMD EPYC 7742	Intel® Xeon® E5-2690 v3
RAM/Node	768 GB	128 GB	64 GB
Accelerator	4 × NVIDIA Tesla V100 (32GB PCIe)	6 × AMD Radeon Instinct MI50 (32GB)	NVIDIA Tesla P100 16GB

`-concurrentkernels off -csv -profile-from-start off -profile--api-trace all -printgputrace`. Note that only the kernel times were measured and not the time taken to set up the kernel or any synchronization between kernels. Especially cuBLAS has a small advantage because the heuristic of cuBLAS to select the best kernel for the problem is executed at runtime, while CUTLASS and cuCOSMA execute the kernel that was compiled. The software version used to perform the benchmarks on NVIDA GPUs can be found in Table 4.2.

Table 4.2: Software versions used to perform the benchmarks on NVIDIA GPUs.

	Ault05/Ault06	XC50
CUDA Driver Version	11.0	10.2
CUDA Runtime Version	10.2	10.2
Driver Version	450.36.06	418.39
CUDA Toolkit (NVCC, cuBLAS)	11.0	10.2
CUTLASS	2.2	2.2

AMD

The kernel were compiled using hipcc 3.5 with clang 11 with the following flags: `-O3 -amdgpu-target=gfx906 -lrocblas`. To measure the performance of the kernels hipEvents were used. The start event was measured

before the GEMM call, while the end event was measured after the call. Note that this time the time to set up the kernel is included in the measurement. The benchmarks were performed using rocm 3.5.1 with driver version 5.6.0.

4.1.2 Matrix Dimensions

We benchmark square, large N, large K and flat matrices, see Table 4.3. For each type of matrix-matrix multiplication a certain set of ω was selected from the range $\min \omega$ to $\max \omega$ and the dimensions were calculated as shown in the table. Note that ω is non evenly spaced, it becomes more sparse the larger it gets.

Table 4.3: The types of matrices we benchmark.

	M	N	K	$\min \omega$	$\max \omega$
Square	ω	ω	ω	1	16384
Large N	ω	ω^2	ω	1	1024
Large K	ω	ω	ω^2	1	1024
Flat	ω^2	ω^2	ω	1	196

Furthermore, we selected some special matrices for which the cuBLAS kernels are not best suited to demonstrate the flexibility of the cuCOSMA kernel.

4.2 Results

We will present the results and analyze them.

4.2.1 Individual Components

In this subsection some individual components of cuCOSMA are analyzed.

SWIZZLE

To find out whether the *SWIZZLE* technique, introduced in Section 2.2.2, provides performance enhancements , a multiplication with the parameters defined in Listing 4.1 was performed, where all parameters are fixed except `SWIZZLE` varies from 1 to 16.

As seen in Fig. 4.1, there is no performance increase using a bigger `SWIZZLE` parameters than 1 on a NVIDIA Tesla V100 for this specific multiplication.

- M: 16384
- N: 16384
- K: 16384
- `THREADBLOCK_TILE_M`: 128
- `THREADBLOCK_TILE_N`: 128
- `THREADBLOCK_TILE_K`: 16384
- `WARP_TILE_M`: 32
- `WARP_TILE_N`: 64
- `THREAD_TILE_N`: 8
- `THREAD_TILE_M`: 8
- `LOAD_K`: 8
- `SPLIT_K`: 1
- `ALPHA`: 1
- `BETA`: 0
- `A_OFFSET`: 4
- `B_OFFSET`: 0
- `ADDITIONAL_OCCUPANCY_WARP`: 4
- `ADDITIONAL_OCCUPANCY_SM`: 2
- `B_OFFSET`: 0
- `SWIZZLE`: 1-16

Listing 4.1: Parameters used for the evaluation of `SWIZZLE`.

However, on the NVIDIA Tesla P100 a performance increase can be seen for the same multiplication, see Fig. 4.2.

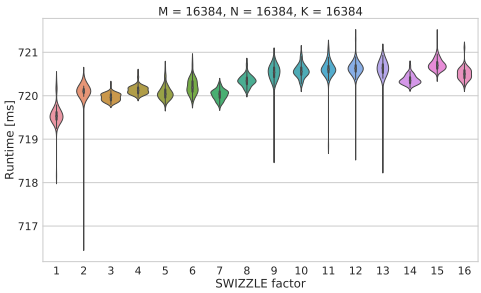


Figure 4.1: Measurements of a 16384x16384 * 16384x16384 multiplication using cuCOSMA on a V100 with NVCC 11.0, where all parameters are fixed according to Listing 4.1 except the `SWIZZLE` varies from 1 to 16.

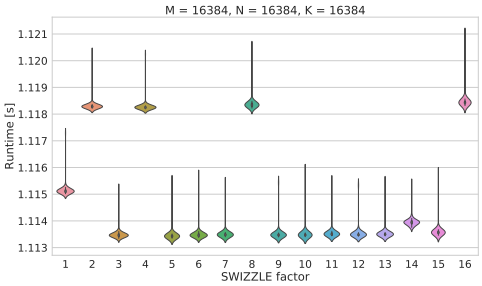


Figure 4.2: Measurements of a 16384x16384 * 16384x16384 multiplication using cuCOSMA on a P100 with NVCC 10.2, where all parameters are fixed according to Listing 4.1 except the `SWIZZLE` varies from 1 to 16.

The NVIDIA Tesla V100 used to perform this tests has a 6 MB L2 cache, while the NVIDIA Tesla P100 has a 4 MB L2 cache. For the NVIDIA Tesla P100, swizzling can enforce that there is more good data in the cache, while the cache on the NVIDIA Tesla V100 already does a good job on its own and swizzling is unnecessarily trying to improve the hit rate by caching too much data. This strategy is unlikely to play an important role in the future, as L2 caches are growing rapidly (NVIDIA Tesla A100 has 40 MB L2 Cache) and the hit rate is already good (Example from Fig. 4.1 with `SWIZZLE` $= 1$

has a L2 hit rate $> 80\%$). However, the L2 cache plays an important role. If one implementation has an even slightly higher L2 hit rate than another, it will probably be faster in the end, even if it needs some more instructions.

Additional plots for square matrices can be found in Section A.2.1.

Atomics vs. Reduction Kernel

To perform the necessary reduction in a split K scenario, the cuCOSMA kernel offers two implementations: one uses atomics to perform the reduction and the other one launches an additional reduction kernel. To measure the difference between these implementations, all large K multiplications were performed once with a reduction kernel and once with atomics. Fig. 4.3 compares the two implementations, the atomic kernel was taken as baseline. The atomic variant is especially useful for smaller matrices and as matrices become larger, the reduction kernel performs better. This can be explained by the fact that atomic instructions are not much slower than normal stores in typical GEMM computation, but for smaller matrices the additional reduction kernel (about 20 μs) is a big overhead. Furthermore, it can happen that depending on how the thread blocks are scheduled, that atomic stores will lead to congestion and therefore there are some outliers, see $\omega = 1000$.

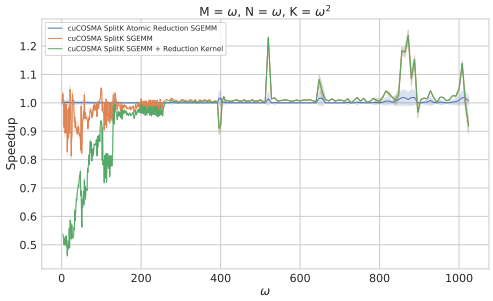


Figure 4.3: Speedup in comparison to cuCOSMA using atomic reduction of large K matrices on a V100 with NVCC 11.0. The kernel were launched using the configuration provided by the schedule generator.

4.2.2 End-to-End Performance

In this subsection the performance of the different implementations is analyzed.

Performance NVIDIA

Here, the overall performance of the kernels on NVIDIA GPUs is compared. A matrix-matrix multiplication was performed using the same 128x128 tile sizes with all 3 competitors.

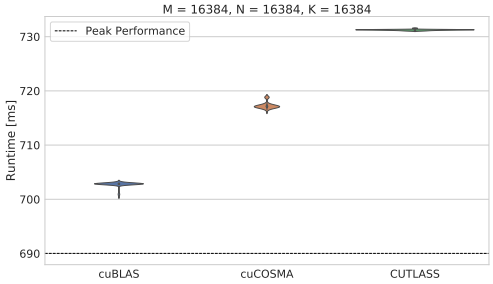


Figure 4.4: Measurements of a 16384x16384 * 16384x16384 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

As seen in Fig. 4.4, cuCOSMA is faster than CUTLASS but still slower than cuBLAS using the same configuration for this multiplication. More plots for some square matrices can be found in Section A.2.2. For the measurements we made, cuCOSMA is overall $1.8\% \pm 3.38$ slower than cuBLAS, while CUTLASS is $4.99\% \pm 3.82$ slower than cuBLAS and **cuCOSMA consistently outperforms CUTLASS**.

If cuBLAS has the right kernel for a problem, it will also have the fastest implementation, although the kernel probably has to be selected manually using *cublasGemmEx* [3]. With the current compiler capabilities, handwritten assembly is more fine-tuned than compiler generated assembly.

Element-wise Functions

While cuBLAS represents the state-of-the-art in GEMM performance, it lacks the possibility to support other epilogues, like for example element-wise functions, which are often used in machine learning. It is not impossible to add this functionality manually to the cuBLAS assembly, but the average user will probably perform this in an additional kernel, while the element-wise functions can easily be added in a compiler generated kernel like cuCOSMA or CUTLASS. The cuBLAS library also supports element-wise functions with the cuBLASLt API [4], but only the ReLu [25] function, others have to be created manually.

To see if cuBLAS is still faster even if an additional kernel is needed, we performed benchmarks with an additional sigmoid kernel (Listing A.36).

Whether cuBLAS with an additional kernel or cuCOSMA is faster depends on the chosen multiplication. For a big 8192x8192 * 8192x8192 multiplication it is worth to start two kernels, as the advantage of the cuBLAS kernel is sufficiently large, see Fig. 4.5. If the K dimension is reduced to 1024, cuCOSMA is faster using an integrated sigmoid function, see Fig. 4.6. The reduction kernel is a big overhead if the K dimension is small compared to the M and N dimension.

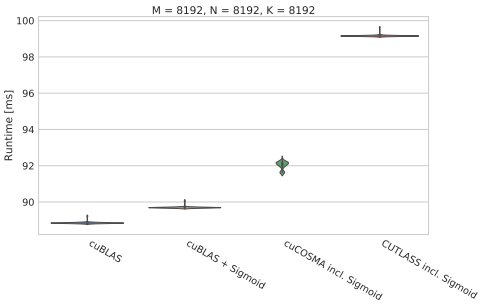


Figure 4.5: Measurements of a 8192x8192 * 8192x8192 multiplication with the sigmoid function on a V100 with NVCC 11.0. All implementations use the 128x128 thread block tile sizes. CUTLASS and cuCOSMA perform the element-wise function in the same kernel as the multiplication, while for cuBLAS another kernel is launched to perform to activation.

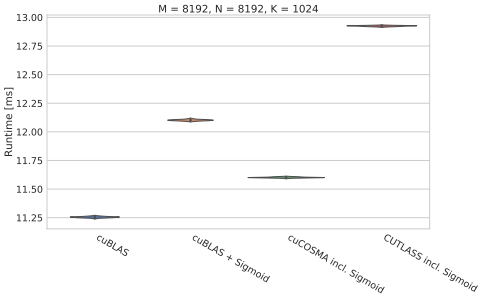


Figure 4.6: Measurements of a 8192x1024 * 1024x8192 multiplication on a V100 with NVCC 11.0. All implementations use the 128x128 thread block tile sizes. CUTLASS and cuCOSMA perform the element-wise function in the same kernel as the multiplication, while for cuBLAS another kernel is launched to perform to activation.

Performance AMD

On NVIDIA implementations it is possible to customize the kernel configuration, this is not possible in rocBLAS. There is only one standard algorithm.

As shown in Fig. 4.7, hipCOSMA can outperform rocBLAS significantly in a large matrix-matrix multiplication. While for a smaller multiplication of size 4096x4096 * 4096*4096 hipCOSMA is only about 1.3× faster than rocblas, see Fig. 4.9. We observed that the relative performance of rocBLAS compared to the peak performance decreases with increasing matrix size, as seen in Fig. 4.7, Fig. 4.8 and Fig. 4.9.

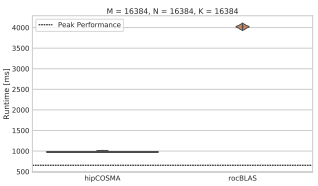


Figure 4.7: Measurements of a 16384x16384 * 16384x16384 multiplication on an AMD Radeon Instinct MI50. The rocBLAS implementation is invoked using rocblas_sgemm. hipCOSMA is invoked with: `THREADBLOCK_TILE_N_M`: 128, `THREADBLOCK_TILE_K`: 8192, `WARP_TILE_N_M`: 64, `THREAD_TILE_N_M`: 8, `LOAD_K`: 8, `SPLIT_K`: 1

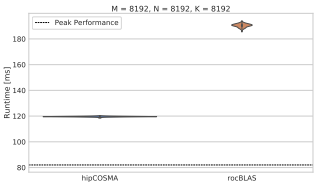


Figure 4.8: Measurements of a 8192x8192 * 8192x8192 multiplication on an AMD Radeon Instinct MI50. The rocBLAS implementation is invoked using rocblas_sgemm. hipCOSMA is invoked with : `THREADBLOCK_TILE_N_M`: 128, `THREADBLOCK_TILE_K`: 8192, `WARP_TILE_N_M`: 64, `THREAD_TILE_N_M`: 8, `LOAD_K`: 8, `SPLIT_K`: 1

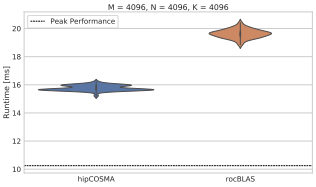


Figure 4.9: Measurements of a 4096x4096 * 4096x4096 multiplication on an AMD Radeon Instinct MI50. The rocBLAS implementation is invoked using rocblas_sgemm. hipCOSMA is invoked with : `THREADBLOCK_TILE_N_M`: 128, `THREADBLOCK_TILE_K`: 8192, `WARP_TILE_N_M`: 64, `THREAD_TILE_N_M`: 8, `LOAD_K`: 8, `SPLIT_K`: 1

4.2.3 Schedule Generator

In this subsection, the results of the matrix-matrix multiplications, as described in Section 4.1.2, are presented.

The cuBLAS kernel is invoked using the *cublasSgemm* method, where a heuristic is used to choose the best kernel. CUTLASS is invoked with its default configuration and cuCOSMA is invoked using the configuration provided by the schedule generator.

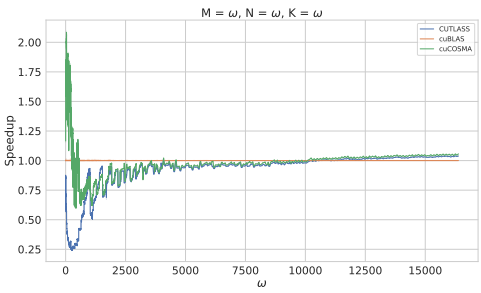


Figure 4.10: Speedup in comparison to cuBLAS of square matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using *cublasSgemm*. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

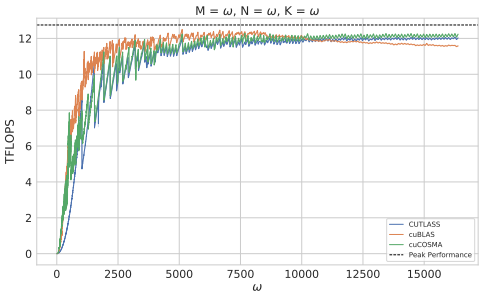


Figure 4.11: Peak performance for square matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using *cublasSgemm*. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

Fig. 4.10 and Fig. 4.11 show the results for square matrices. For square matrices the schedule generator usually chooses good configurations. In the range $\omega = 2000$, it performs slightly worse than cuBLAS, but it can outperform cuBLAS for $\omega > 10000$ and for small matrices.

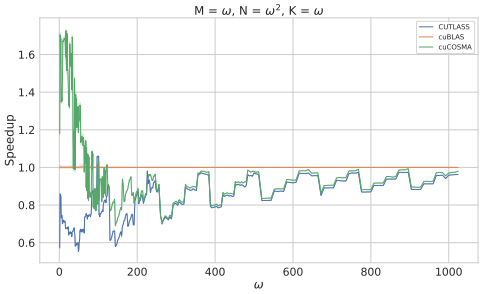


Figure 4.12: Speedup in comparison to cuBLAS of large N matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using *cublasSgemm*. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

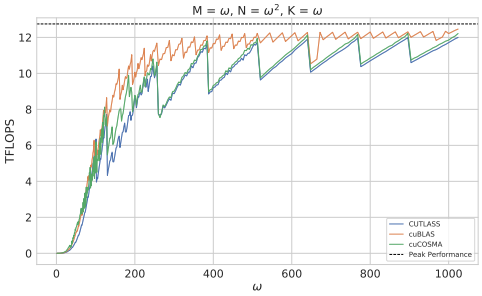


Figure 4.13: Peak performance for large N matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using *cublasSgemm*. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

For large N matrices, see Fig. 4.12 and Fig. 4.13, cuCOSMA can outperform cuBLAS for small matrices, but if the matrices become larger, the schedule generator chooses bad performing configurations.

The results for large K and flat matrices can be found in Section A.2.3. We will not comment on them as the results look similar to those for large N.

For small matrices cuCOSMA can excel because the kernel is more flexible than cuBLAS and the schedule generator can advantage of this by using more CUDA cores to solve the problem faster. After all CUDA cores are used, the schedule generator tries to minimize the communication volume, which usually leads to a worse performance compared to cuBLAS. In these scenarios, cuCOSMA often uses a 3D decomposition using split K, while cuBLAS used one of its smaller 2D kernel. Towards the end, the larger the matrices become, the more cuCOSMA can catch up to or even overtake cuBLAS. However, cuCOSMA using a schedule generator is always faster than CUTLASS, which uses its default configuration.

cuBLAS practically never uses its 128x128 kernel, although it is most efficient for large matrices, but uses the 128x64 kernel instead. The 128x128 kernel can be implemented such that each thread uses 128 registers. In the 128x128 kernel each thread block launches 256 threads, this results in 32768 registers used per thread block, exactly half of what an SM can provide. Therefore the 128x128 kernel is the largest kernel that can still reach an occupancy of 2, as soon as the size of the kernel is further increased, the occupancy is reduced to 1, which has a negative effect on performance.

If one takes a closer look at the results and examines which kernels cuBLAS launches, one can observe that the 128x64 kernel is often the better choice than the 128x128 kernel for large N and flat matrices, while for Large K matrices, the 32x128 kernel without split K outperforms other versions that use a larger kernel but with Split K.

There are two possible explanations why smaller kernels that cause more communication volume perform better than large kernels. Smaller kernel usually achieve a higher occupancy than bigger kernels, it is possible that the increased occupancy has a positive effect on the performance because there is always a warp ready to be scheduled. What seems more likely to us, however, is that smaller kernels have a higher L2 cache hit rate in certain multiplications. We could observe that the fastest kernel for a problem is often the one with the highest L2 hit rate, if all SMs of the GPU are sufficiently occupied and not too small kernels are chosen.

Consequently, we think that the communication volume alone is not the appropriate metric to implement a scheduler generator for matrix-matrix multiplication on the GPU. The L2 cache and the occupancy have signifi-

cant impact on performance and should also be reflected in the schedule generator.

An improved heuristic for selecting tile sizes could work as follows: If the resources of the GPU are well utilized with a 2D schedule, it is not worth switching to a 3D schedule to further reduce the communication volume. Furthermore, if the M or N dimension is small, more weight should be placed on occupancy than on the communication volume.

4.2.4 Special Matrices

This subsection presents some special matrix-matrix multiplications in which cuCOSMA could outperform cuBLAS. The configuration of cuCOSMA is selected manually, while for cuBLAS all kernels are launched to find the fastest one. All kernels from *CUBLAS_GEMM_ALGO0* up to *CUBLAS_GEMM_ALGO23* and *CUBLAS_GEMM_DEFAULT* are considered. The following figures include only the fastest cuBLAS kernel.

Fig. 4.14 shows the performance for a square 128x128 * 128x128 multiplication. While cuBLAS uses its smallest and fastest kernel for this problem, the cuCOSMA kernel can adapt better to this problem and use more CUDA cores to solve the problem faster. Thus, cuCOSMA could achieve a speedup of about 2× compared to cuBLAS.

Fig. 4.15 shows the performance of a multiplication with small M and N but large K dimension. cuBLAS uses a 32x32sliceK+SplitK kernel, which is too large for this problem and performs unnecessary work. The cuCOSMA kernel without slice K can adapt itself better and reach a speedup of about 1.3× compared to cuBLAS.

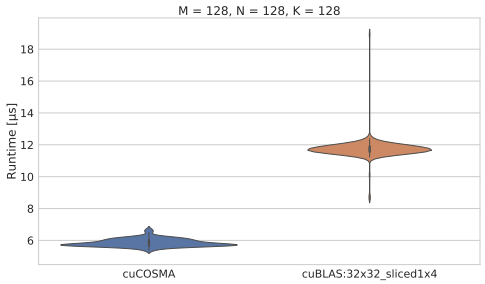


Figure 4.14: Measurements of a 128x128 * 128x128 multiplication on a V100 with NVCC 11.0. cuCOSMA uses the following configuration: `THREADBLOCK_TILE_M`: 16, `THREADBLOCK_TILE_N`: 32 , `THREADBLOCK_TILE_K`: 32, `WARP_TILE_M`: 8, `WARP_TILE_N`: 16, `THREAD_TILE_N`: 2, `THREAD_TILE_M`: 2, `LOAD_K`: 8, `SPLIT_K`: 4, `ATOMIC_REDUCTION`: true, `SWIZZLE`: 1, `ALPHA`: 1, `BETA`: 0.

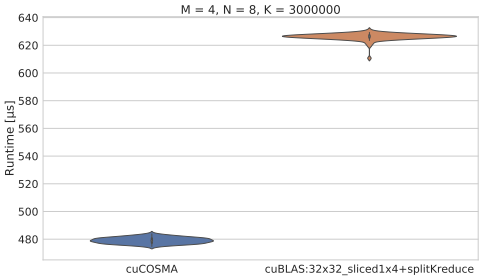


Figure 4.15: Measurements of a 4x3000000 * 3000000x8 multiplication on a V100 with NVCC 11.0. cuCOSMA uses the following configuration: `THREADBLOCK_TILE_M`: 4, `THREADBLOCK_TILE_N`: 8, `THREADBLOCK_TILE_K`: 9375, `WARP_TILE_M`: 4, `WARP_TILE_N`: 8, `THREAD_TILE_N`: 1, `THREAD_TILE_M`: 1, `LOAD_K`: 8, `SPLIT_K`: 320, `ATOMIC_REDUCTION`: true, `SWIZZLE`: 1, `ALPHA`: 1, `BETA`: 0.

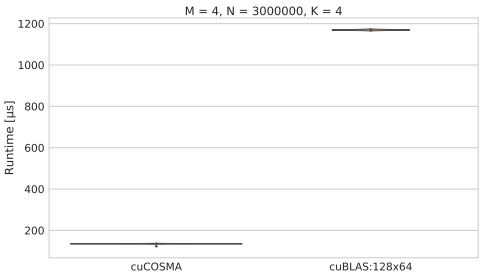


Figure 4.16: Measurements of a 4x4 * 4x3000000 multiplication on a V100 with NVCC 11.0. cuCOSMA uses the following configuration: `THREADBLOCK_TILE_M`: 4, `THREADBLOCK_TILE_N`: 256, `THREADBLOCK_TILE_K`: 4, `WARP_TILE_M`: 4, `WARP_TILE_N`: 256, `THREAD_TILE_N`: 4, `THREAD_TILE_M`: 8, `LOAD_K`: 2, `SPLIT_K`: 1, `ATOMIC_REDUCTION`: true, `SWIZZLE`: 1, `ALPHA`: 1, `BETA`: 0.

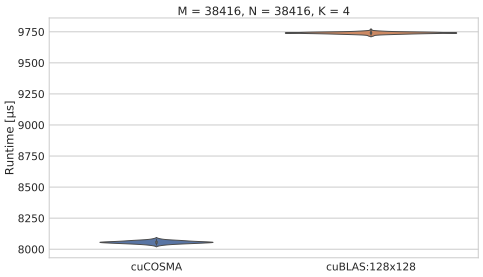


Figure 4.17: Measurements of a 38416x4 * 4x38416 multiplication on a V100 with NVCC 11.0. cuCOSMA uses the following configuration: `THREADBLOCK_TILE_M`: 38416, `THREADBLOCK_TILE_N`: 38416, `THREADBLOCK_TILE_K`: 4, `WARP_TILE_M`: 128, `WARP_TILE_N`: 128, `THREAD_TILE_N`: 32, `THREAD_TILE_M`: 64, `LOAD_K`: 2, `SPLIT_K`: 1, `ATOMIC_REDUCTION`: false, `SWIZZLE`: 1, `ALPHA`: 1, `BETA`: 0.

Fig. 4.16 shows a multiplication with large N but small K and M dimension. The cuCOSMA kernel can adapt better to the problem and can reach a speedup of about $10\times$ compared to cuBLAS.

Fig. 4.17 shows a multiplication with small K but large M and N dimension. Here, the cuBLAS heuristic chooses the best kernel for the problem, but the 128x128 kernel itself is only efficient for larger K dimension. The cuCOSMA kernel can also better to the problem and can reach a speedup of about $1.2\times$ compared to cuBLAS.

In summary, the cuBLAS kernels are only efficient if M and N are greater than 32 and if K is greater than 8. If one of the three dimensions is smaller or the matrix-matrix multiplication itself is small, we expect cuCOSMA to outperform cuBLAS.

Chapter 5

Conclusion

The fastest matrix-matrix multiplication kernels in CUDA are those of cuBLAS, which are written manually in assembly. Although the kernels of cuBLAS cannot cover every problem well and the heuristic for selecting the best kernel for a problem is not perfect, cuBLAS is mostly unbeatable.

Assuming that each matrix row is properly aligned and that the matrix dimensions are available at compile time, we wrote a matrix-matrix multiplication kernel in CUDA C++ that is faster than CUTLASS, but slower than cuBLAS using the same tile sizes. The cuCOSMA kernel supports practically all tile sizes and can provide the fastest implementation in special cases that are not well covered by cuBLAS. Furthermore, hipCOSMA shows significant speedups compared to rocBLAS for large multiplications.

We have tried to integrate the findings of COSMA [31] into a schedule generator to generate tile sizes for the cuCOSMA kernel. However, it showed that the communication volume is not the only metric to measure which tile sizes for a particular problem are good for the whole problem space. This metric shows a positive effect with large multiplications. Thus, other factors also play an important role, such as occupancy and the L2 cache hit rate.

5.1 Future Work

Throughout the implementation of the kernel, we have paid relatively little attention to the L2 cache. In retrospect, however, it turned out to be one of the more important factors for achieving peak performance. We think it is certainly worthwhile to investigate whether SWIZZLE (Section 2.2.2) can be improved and how this technique performs with non-square matrices.

It can already happen today that a GEMM kernel is bandwidth-limited and the performance strongly depends on the L2 cache. This problem will become even more severe in the future as peak performance will be

greatly increased by tensor cores. Tensor cores are special hardware units on the GPU that can compute small GEMMs directly in hardware and achieve much higher peak performance for GEMM computations than normal CUDA cores. It is therefore certainly worth investigating whether in certain cases it is worth reducing the occupancy to achieve a higher L2 hit rate as described in Section 3.1.4. (We speculate that one of the reasons for the disproportionate increase of the L2 cache size in the NVIDIA Tesla A100 (see Table A.1) is that, since FP64 tensor cores were added, NVIDIA wanted to make sure that the DGEMM kernel is not limited by memory bandwidth.)

The cuCOSMA kernel currently only supports single precision floating-point numbers and 32-bit integers as data types, the matrices have to be stored in row-major format. Thus, support for column-major layouts and other data types is missing, as well as the support for tensor cores. Furthermore, the cuCOSMA kernel lacks support for slice K (Section 2.2.2).

The hipCOSMA kernel should be further optimised for AMD GPUs as it does not yet come close to achieving peak performance. For example, it is still to be investigated whether the same warp broadcasting limitation is present in AMD GPUs, that can be found in NVIDIA GPUs (Section 2.2.2).

The presented schedule generator is not yet perfect, there is still room for further enhancements. The improvements referred to in Section 4.2.3 could thus be incorporated.

Appendix A

Appendix

A.1 Example of a kernel limited by memory bandwidth

Assume a NVIDIA RTX 2080 Ti [15] with the following specifications:

- SM Count: 68
- Threads per SM: 64
- Memory Bandwidth: 616 GB/s
- Clock speed: 1635 MHz

We use the following kernel for an infinitely big multiplication:

- M: ∞
- N: ∞
- K: ∞
- `THREADBLOCK_TILE_M`: 64
- `THREADBLOCK_TILE_N`: 64
- `THREADBLOCK_TILE_K`: ∞
- `WARP_TILE_M`: 64
- `WARP_TILE_N`: 32
- `THREAD_TILE_N`: 8
- `THREAD_TILE_M`: 8
- `LOAD_K`: 8

This will result in 64 threads being launched for each thread block. In the main loop of the kernel, there are `THREAD_TILE_N` $*$ `THREAD_TILE_M` $*$

LOAD_K $=$ 512 FMAD instructions, assume that about 20 more cycles are needed in the main loop for the calculation of indices, increasing the loop counter and everything else. In the main loop each thread needs to load 128 bytes from global memory to shared memory (See Equation (A.1)).

$$\frac{(\texttt{THREADBLOCK\_TILE\_M} + \texttt{THREADBLOCK\_TILE\_N}) * \texttt{LOAD\_K}}{threads} * 8 \tag{A.1}$$

The required memory bandwidth by the kernel can be computed as shown in Equation (A.2) and Equation (A.3).

$$\frac{SM\_count * threads\_per\_SM * bytes\_mainloop * clock\_freq}{cycles\_mainloop} \tag{A.2}$$

$$\frac{68 * 64 * 128 * 1.635}{532} \approx 1712 \text{ GB/s} \tag{A.3}$$

The required bandwidth exceeds the provided bandwidth by a lot. In reality, the performance depends on the hit rate of the L2 cache. One solution to alleviate this problem is to choose larger thread bock tile sizes that cause less communication volume.

A.2 Performance Figures

A.2.1 Individual Components

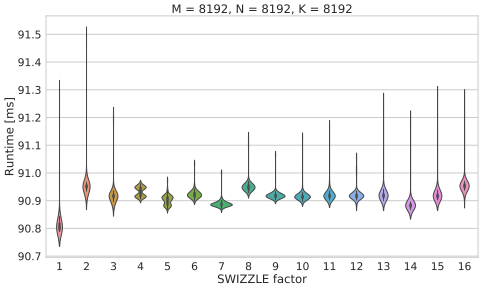


Figure A.1: Measurements of a 8192x8192 * 8192x8192 multiplication using cuCOSMA on a V100 with NVCC 11.0, where all parameters are fixed according to Listing 4.1 except SWIZZLE varies from 1 to 16.

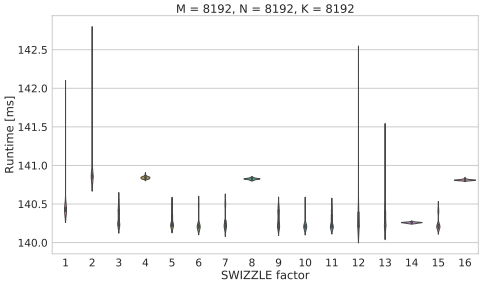


Figure A.2: Measurements of a 8192x8192 * 8192x8192 multiplication using cuCOSMA on a P100 with NVCC 10.1, where all parameters are fixed according to Listing 4.1 except SWIZZLE varies from 1 to 16.

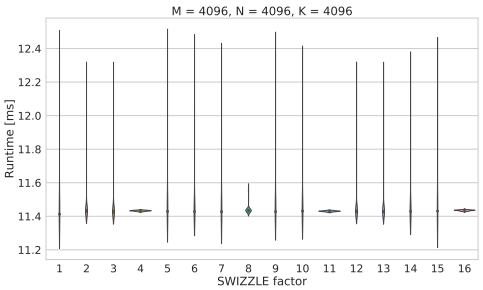


Figure A.3: Measurements of a 4096x4096 * 4096x4096 multiplication using cuCOSMA on a V100 with NVCC 11.0, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

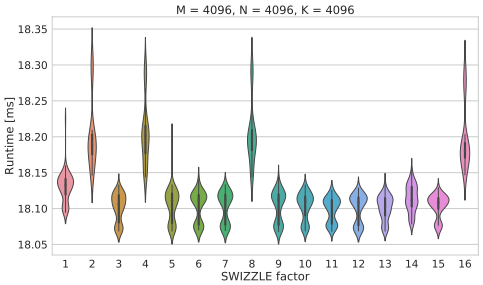


Figure A.4: Measurements of a 4096x4096 * 4096x4096 multiplication using cuCOSMA on a P100 with NVCC 10.1, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

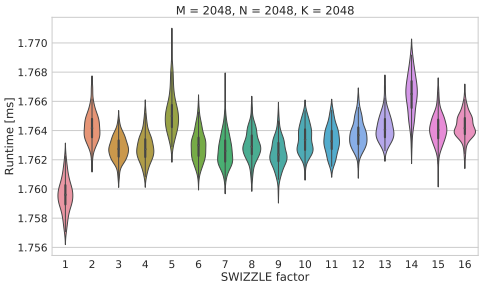


Figure A.5: Measurements of a 2048x2048 * 2048x2048 multiplication using cuCOSMA on a V100 with NVCC 11.0, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

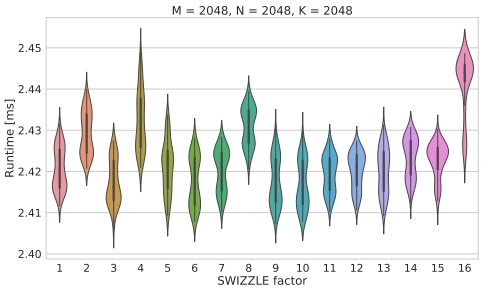


Figure A.6: Measurements of a 2048x2048 * 2048x2048 multiplication using cuCOSMA on a P100 with NVCC 10.1, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

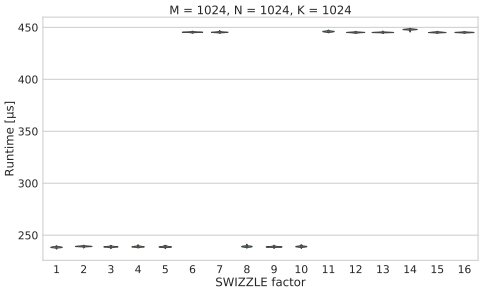


Figure A.7: Measurements of a 1024x1024 * 1024x1024 multiplication using cuCOSMA on a V100 with NVCC 11.0, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

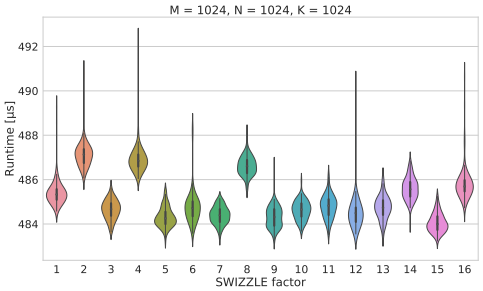


Figure A.8: Measurements of a 1024x1024 * 1024x1024 multiplication using cuCOSMA on a P100 with NVCC 10.1, where all parameters are fixed according to Listing 4.1 except `SWIZZLE` varies from 1 to 16.

A.2.2 Performance NVIDIA

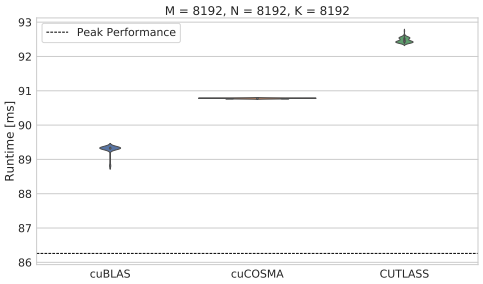


Figure A.9: Measurements of a 8192x8192 * 8192x8192 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

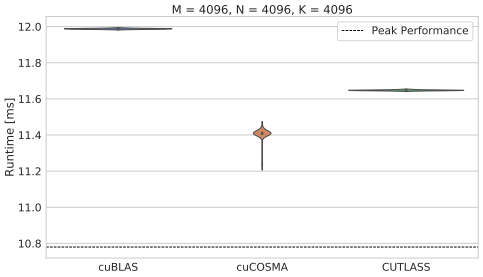


Figure A.10: Measurements of a 4096x4096 * 4096x4096 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

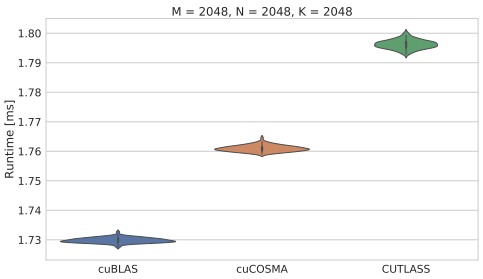


Figure A.11: Measurements of a 2048x2048 * 2048x2048 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

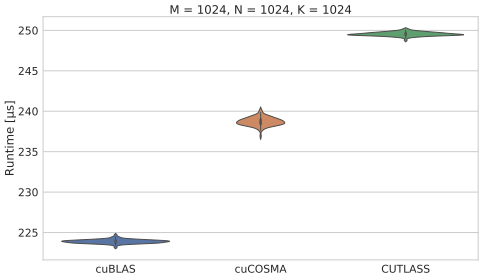


Figure A.12: Measurements of a 1024x1024 * 1024x1024 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

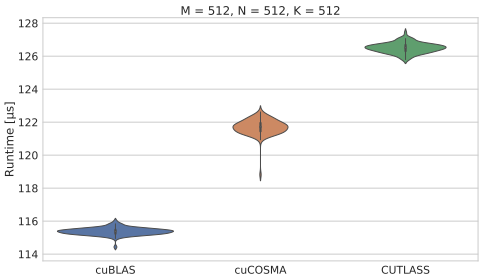


Figure A.13: Measurements of a 512x512 * 512x512 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

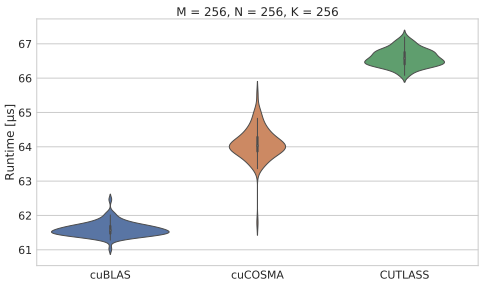


Figure A.14: Measurements of a 256x256 * 256x256 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

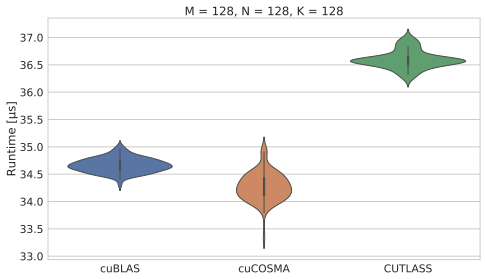


Figure A.15: Measurements of a 128x128 * 128x128 multiplication on a V100 with NVCC 11.0. All implementations use the same tile sizes.

A.2.3 Schedule Generator

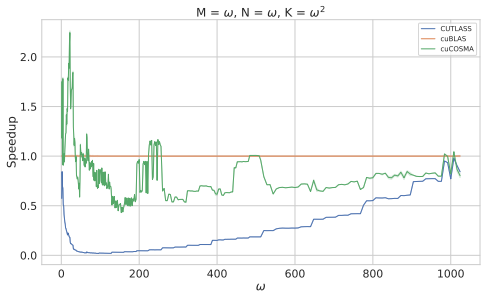


Figure A.16: Speedup in comparison to cuBLAS of large K matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using the *cublasSgemm* method, where a heuristic is used to choose the best kernel. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

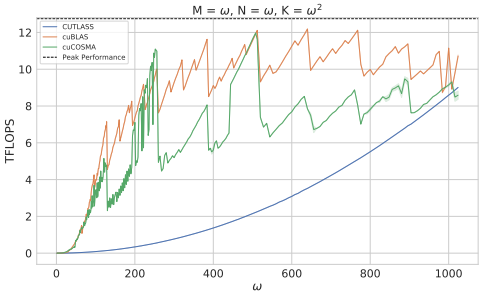


Figure A.17: Peak performance for large K matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using the *cublasSgemm* method, where a heuristic is used to choose the best kernel. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

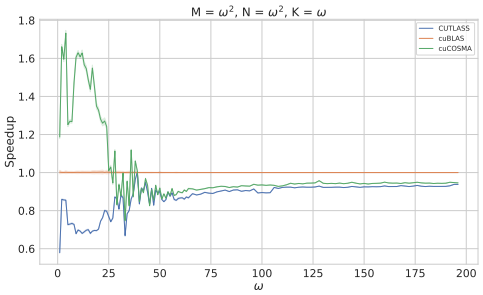


Figure A.18: Speedup in comparison to cuBLAS of flat matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using the *cublasSgemm* method, where a heuristic is used to choose the best kernel. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

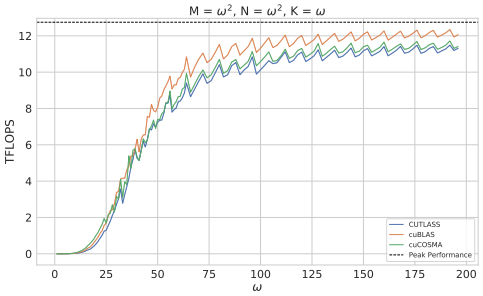


Figure A.19: Peak performance for flat matrices on a V100 with NVCC 11.0. The cuBLAS kernel is invoked using the *cublasSgemm* method, where a heuristic is used to choose the best kernel. CUTLASS is invoked with its default configuration and cuCOSMA uses the configuration provided by the schedule generator.

A.3 Hardware details used by the schedule generator

- The number of streaming multiprocessors
- The number of warps per streaming multiprocessors;
- Maximum amount of shared memory per SM
- Maximum amount of shared memory per thread block
- Number of 32-bit registers per SM
- Maximum number of 32-bit registers per thread block
- Maximum number of 32-bit registers per thread
- Memory bandwidth
- Clock frequency
- L2 cache size
- Maximum number of threads per block
- Compute Capability
- Amount of global memory (Only used to decide if it is even possible to multiply the matrices.)
- Maximum number of resident blocks per SM
- Maximum number of resident warps per SM
- Maximum number of resident threads per SM

A.4 Schedule Generator

```
Schedule best_schedule;

for (int load_k : load_k_possible)
 for (int thread_tile_m : threadtiles_possible)
  for (int thread_tile_n : threadtiles_possbile)
   for (int warp_tile_m = thread_tile_m; warp_tile_m <= registers_per_warp; warp_tile_m += thread_tile_m)
    for (int warp_tile_n = thread_tile_n; warp_tile_n <= registers_per_warp; warp_tile_n += thread_tile_n)
     for (int thread_block_m = warp_tile_m; thread_block_m <= registers_per_thread_block; thread_block_m += warp_tile_m)
      for (int thread_block_n = warp_tile_n; thread_block_n <= registers_per_thread_block; thread_block_n += warp_tile_n)
       for (int split_k = 1; split_k <= SMs * Warps_per_SM * 2; split_k++) {

         Schedule schedule(load_k,thread_tile_m,thread_tile_n,warp_tile_m,warp_tile_n,thread_block_m,thread_block_n,split_k)

         if(!fulfills_constraints(schedule)){
           continue;
         }


         if(schedule > best_schedule){
           best_schedule = schedule;
         }
       }
```

Listing A.1: Schedule generator with 8 for loops.

A.5 Evolution of L2 cache size, memory bandwidth and peak performance of NVIDIAs Tesla series

Table A.1: Evolution of L2 cache size, memory bandwidth and peak performance of NVIDIAs Tesla series

	C870	**C1060**	**M2090**	**K40**	**M40**	**P100**	**V100**	**A100**
Memory bandwidth	76 GB/s	102 GB/s	177 GB/s	288 GB/s	288 GB/s	732 GB/s	900 GB/s	1555 GB/s
Increase		34 %	73 %	62 %	0 %	154 %	23 %	72 %
Performance FP32	345 GF	622 GF	1.3 TF	4.3 TF	6.8 TF	10.6 TF	15.7 TF	19.5 TF
Increase		80 %	109 %	230 %	58 %	55 %	48 %	24 %
Performance FP64		77 GF	666 GF	1.4 TF	0.2 TF	5.3 TF	7.8 TF	9.7 TF
Increase			765 %	110 %	- 85 %	2550 %	47 %	24 %
L2 cache size	96 KB	256 kB	768 KB	1.5 MB	3 MB	4 MB	6 MB	40 MB
Increase		166 %	200 %	100 %	100 %	33 %	50 %	566 %
Peak performance FP64 Tensor Core								19.5 TF
Increase								150 %

Sources: C870: [18], C1060: [17], M2090: [21], K40: [19] [20], M40: [22], P100: [23], V100: [24], A100: [16]

We always used the fastest single GPU model of the respective generation for comparison. The huge increase in memory bandwidth from M40 to P100 can be explained by the new *HBM2* [12] memory technology. The M40 has practically no FP64 cores, which is why it perform poorly in this discipline.

A.6 cuCOSMA Code Listings

A.6.1 Launch

Configuration

```
#define TYPE float
#define VECTORTYPE2 float2
#define VECTORTYPE4 float4
#define M 4096
#define N 4096
#define K 4096
#define THREADBLOCK_TILE_M 128
#define THREADBLOCK_TILE_N 128
#define THREADBLOCK_TILE_K 4096
#define LOAD_K 8
#define WARP_TILE_M 32
#define WARP_TILE_N 64
#define THREAD_TILE_M 8
#define THREAD_TILE_N 8
#define A_OFFSET 4
#define B_OFFSET 0
#define SWIZZLE 1
#define SPLIT_K 1
#define ATOMIC_REDUCTION true
#define ADDITIONAL_OCCUPANCY_SM 2
#define ALPHA 1
#define BETA 0
```

Listing A.2: Example *config.h* file.

Kernel

Table A.2: cosmaSgemm Parameters

A	<type> array of dimensions lda * k.
lda	leading dimension of two-dimensional array used to store the matrix A.
B	<type> array of dimension ldb * n
ldb	leading dimension of two-dimensional array used to store matrix B.
C	<type> array of dimensions ldc * n
ldc	leading dimension of a two-dimensional array used to store the matrix C.

```
#define THREADS ((THREADBLOCK_TILE_M / WARP_TILE_M) * (THREADBLOCK_TILE_N / WARP_TILE_N) * 32)

void cosmaSgemm(
const TYPE *__restrict__ A, const int lda,
const TYPE * __restrict__ B, const int ldb,
TYPE *__restrict__ C, const int ldc) {

  if (BETA != 1 && ((SPLIT_K > 1 && ATOMIC_REDUCTION) || ALPHA == 0)) {


    cublasHandle_t handle;
    cublasCreate(&handle);

    const float factor = BETA;

    cublasSscal(handle, ldc * M, &factor, C, 1);

    cublasDestroy(handle);

    cudaGetLastError();
    cudaDeviceSynchronize();

  }

  if (ALPHA != 0) {

    constexpr int N_TILES = (N + THREADBLOCK_TILE_N - 1) / THREADBLOCK_TILE_N;
    constexpr int M_TILES = (M + THREADBLOCK_TILE_M - 1) / THREADBLOCK_TILE_M;

    constexpr dim3 dimBlock(THREADS, 1, 1);
    dim3 dimGrid;

    if (SWIZZLE != 1) {

      constexpr int N_TILES_SWIZZLE = N_TILES * SWIZZLE;
      constexpr int M_TILES_SWIZZLE = (M_TILES + SWIZZLE - 1) / SWIZZLE;

      dimGrid.x = N_TILES_SWIZZLE;
      dimGrid.y = M_TILES_SWIZZLE;
      dimGrid.z = SPLIT_K;

```

```
42      } else {
43
44        dimGrid.x = N_TILES;
45        dimGrid.y = M_TILES;
46        dimGrid.z = SPLIT_K;
47
48      }
49
50      if (SPLIT_K != 1 && !ATOMIC_REDUCTION) {
51
52        TYPE* C_SPLITK;
53
54        cudaMalloc(&C_SPLITK, sizeof(TYPE) * M * N * SPLIT_K);
55
56        cosmaSgemm_kernel<<<dimGrid, dimBlock>>>(A,lda,B,ldb,C_SPLITK,N);
57
58        cudaGetLastError();
59        cudaDeviceSynchronize();
60
61        cosmaSplitKReduce(C, ldc, C_SPLITK);
62
63        cudaFree(C_SPLITK);
64
65      } else {
66
67        cosmaSgemm_kernel<<<dimGrid, dimBlock>>>(A,lda,B,ldb,C,ldc);
68
69      }
70
71    }
72  }
```

Listing A.3: cuCOSMA kernel launch.

A.6.2 Implementation

Prologue

Kernel signature

```
1  __global__ void
2  __launch_bounds__(THREADS, ADDITIONAL_OCCUPANCY_SM)
3  cosmaSgemm_kernel(const TYPE * __restrict__ A,
4   const int lda, const TYPE * __restrict__ B,
5    const int ldb, TYPE * __restrict__ C, const int ldc) {
6
7  ...
8
9  }
```

Listing A.4: The kernel signature.

Assertions

```
constexpr int M_WARPS = THREADBLOCK_TILE_M / WARP_TILE_M;
constexpr int N_WARPS = THREADBLOCK_TILE_N / WARP_TILE_N;

constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

static_assert(THREAD_TILE_N < 4 || WARP_TILE_N % 4 == 0 || N_WARPS == 1,
 "Threadtile smaller 4 or Warptile mod 4 for vector access");
static_assert(THREAD_TILE_M < 4 || WARP_TILE_M % 4 == 0 || M_WARPS == 1,
 "Threadtile smaller 4 or Warptile mod 4 for vector access");
static_assert(THREAD_TILE_N < 2 || WARP_TILE_N % 2 == 0 || N_WARPS == 1,
 "Threadtile smaller 2 or Warptile mod 2 for vector access");
static_assert(THREAD_TILE_M < 2 || WARP_TILE_M % 2 == 0 || M_WARPS == 1,
 "Threadtile smaller 2 or Warptile mod 2 for vector access");

static_assert(WARP_TILE_N % THREAD_TILE_N == 0,
 "Threadtile needs to divde warptile");
static_assert(WARP_TILE_M % THREAD_TILE_M == 0,
 "Threadtile needs to divde warptile");
static_assert(THREADBLOCK_TILE_M % WARP_TILE_M == 0,
 "Warptilde needs to divide Threadblocktile");
static_assert(THREADBLOCK_TILE_N % WARP_TILE_N == 0,
 "Warptilde needs to divide Threadblocktile");
static_assert(N_THREADS * M_THREADS == 32, "Warp has 32 Threads");
```

Listing A.5: Static assertions, what kind of tile sizes we allow.

Warp and Thread Mapping

```
constexpr int M_WARPS = THREADBLOCK_TILE_M / WARP_TILE_M;
constexpr int N_WARPS = THREADBLOCK_TILE_N / WARP_TILE_N;

constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

const int WarpId = threadIdx.x / 32;
const int threadId = threadIdx.x % 32;

const int WarpIdx = WarpId % N_WARPS;
const int WarpIdy = WarpId / N_WARPS;

int LaneIdx;
int LaneIdy;

if (N_THREADS == 1) {

  LaneIdx = 0;
  LaneIdy = threadId;

} else if (N_THREADS == 2) {

  LaneIdx = (((threadId & 0x60) >> 4) | (threadId & 1));
  LaneIdy = ((threadId >> 1) & (M_THREADS - 1));

} else if (N_THREADS == 4) {

  LaneIdx = (((threadId & 0x30) >> 3) | (threadId & 1));
  LaneIdy = ((threadId >> 1) & (M_THREADS - 1));

} else if (N_THREADS == 8) {

  LaneIdx = (((threadId & 0x18) >> 2) | (threadId & 1));
  LaneIdy = ((threadId >> 1) & (M_THREADS - 1));

} else if (N_THREADS == 16) {

  LaneIdx = (((threadId & 0x1c) >> 1) | (threadId & 1));
  LaneIdy = ((threadId >> 1) & (M_THREADS - 1));

} else if (N_THREADS == 32) {

  LaneIdx = threadId;
  LaneIdy = 0;
}
```

Listing A.6: Warp and Thread Mapping.

Buffer Setup

```
constexpr int A_SHARED_SIZE = (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K;
constexpr int A_SHARED_BUFFER = 2 * A_SHARED_SIZE;

constexpr int B_SHARED_SIZE = LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET);
constexpr int B_SHARED_BUFFER = 2 * B_SHARED_SIZE;

__shared__ TYPE A_Shared[A_SHARED_BUFFER];
__shared__ TYPE B_Shared[B_SHARED_BUFFER];

int B_Shared_Offset_0 = 0;
int B_Shared_Offset_1 = B_SHARED_SIZE;

int A_Shared_Offset_0 = 0;
int A_Shared_Offset_1 = A_SHARED_SIZE;

int shared_memory_stage = 1;

register TYPE Thread_Tile[THREAD_TILE_M * THREAD_TILE_N];

register TYPE A_register_0[THREAD_TILE_M];
register TYPE A_register_1[THREAD_TILE_M];

register TYPE B_register_0[THREAD_TILE_N];
register TYPE B_register_1[THREAD_TILE_N];

#pragma unroll
for (int i = 0; i < THREAD_TILE_M; ++i) {
#pragma unroll
  for (int j = 0; j < THREAD_TILE_N; ++j) {

    Thread_Tile[i * THREAD_TILE_N + j] = 0.0;

  }
}
```

Listing A.7: Shared memory buffers and register fragments setup

Thread Block Remapping

```
  int block_idx_x;
  int block_idx_y;

  if (SWIZZLE != 1) {

    block_idx_x = blockIdx.x / SWIZZLE;
    block_idx_y = (blockIdx.y * SWIZZLE) + (blockIdx.x % SWIZZLE);

    constexpr int M_TILES = (M + THREADBLOCK_TILE_M - 1) / THREADBLOCK_TILE_M;

    if (M_TILES % SWIZZLE != 0 && block_idx_y >= TILE_SHAPE_M) {
      return;
    }

  } else {

    block_idx_x = blockIdx.x;
    block_idx_y = blockIdx.y;

  }
```

Listing A.8: Thread Block Mapping.

Allowed Vector Loads and Boundary Checks

```
constexpr bool A_VECTOR_4 = (LOAD_K % 4 == 0)
                            && (SPLIT_K == 1 || THREADBLOCK_TILE_K % 4 == 0);
constexpr bool A_VECTOR_2 = (LOAD_K % 2 == 0)
                            && (SPLIT_K == 1 || THREADBLOCK_TILE_K % 2 == 0);

constexpr bool B_VECTOR_4 = THREADBLOCK_TILE_N % 4 == 0
                            && ((N % THREADBLOCK_TILE_N) % 4 == 0);
constexpr bool B_VECTOR_2 = THREADBLOCK_TILE_N % 2 == 0
                            && ((N % THREADBLOCK_TILE_N) % 2 == 0);

constexpr bool A_VECTOR_4_LAST = A_VECTOR_4
                                 && (THREADBLOCK_TILE_K % LOAD_K) % 4 == 0
                                 && (SPLIT_K == 1 || ( K % THREADBLOCK_TILE_K) % 4 == 0);
constexpr bool A_VECTOR_2_LAST = A_VECTOR_2
                                 && (THREADBLOCK_TILE_K % LOAD_K) % 2 == 0
                                 && (SPLIT_K == 1 || ( K % THREADBLOCK_TILE_K) % 2 == 0);

constexpr bool K_CHECK = (K % THREADBLOCK_TILE_K != 0 && SPLIT_K > 1);
constexpr bool THREADBLOCK_TILE_K_CHECK = THREADBLOCK_TILE_K % LOAD_K != 0;
```

Listing A.9: How to calculated what kind of vector loads are allowed.

Main Loop

```
constexpr int K_START = (((THREADBLOCK_TILE_K + LOAD_K - 1) / LOAD_K) - 1)
             * LOAD_K;
int cta_k = K_START;

load_Global<
A_VECTOR_4_LAST, A_VECTOR_2_LAST,
B_VECTOR_4, B_VECTOR_2,
K_CHECK, THREADBLOCK_TILE_K_CHECK
>(
&A_Shared, &B_Shared,
A, B,
lda, ldb,
cta_k,
block_idx_x, block_idx_y,
A_Shared_Offset_0, B_Shared_Offset_0);

__syncthreads();
cta_k -= LOAD_K;

#pragma unroll 1
for (; cta_k >= 0; cta_k -= LOAD_K) {

#pragma unroll
  for (int K = 0; K < LOAD_K; k++) {

    if (k % 2 == 0) {
      load_Shared(
      &A_Shared, &A_register_0,
      &B_Shared, &B_register_0,
      k,
      WarpIdx, WarpIdy,
      LaneIdx, LaneIdy,
      A_Shared_Offset_0, B_Shared_Offset_0);
    } else {
      load_Shared(
      &A_Shared, &A_register_1,
      &B_Shared, &B_register_1,
      k,
      WarpIdx, WarpIdy,
      LaneIdx, LaneIdy,
      A_Shared_Offset_0, B_Shared_Offset_0);
    }

    if (k == LOAD_K - 1) {
        load_Global<
        A_VECTOR_4, A_VECTOR_2,
        B_VECTOR_4, B_VECTOR_2,
        (THREADBLOCK_TILE_K * SPLIT_K - K > LOAD_K), false
        >(
        &A_Shared, &B_Shared,
        A, B,
        lda, ldb,
```

```
53          cta_k,
54          block_idx_x, block_idx_y,
55          A_Shared_Offset_1, B_Shared_Offset_1);
56
57        __syncthreads();
58      }
59
60      if (k % 2 == 0) {
61        compute_inner(&A_register_0, &B_register_0, &Thread_Tile);
62      } else {
63        compute_inner(&A_register_1, &B_register_1, &Thread_Tile);
64      }
65    }
66
67    if (shared_memory_stage == 1) {
68      B_Shared_Offset_0 = B_SHARED_SIZE;
69      B_Shared_Offset_1 = 0;
70
71      A_Shared_Offset_0 = A_SHARED_SIZE;
72      A_Shared_Offset_1 = 0;
73
74    } else {
75      B_Shared_Offset_0 = 0;
76      B_Shared_Offset_1 = B_SHARED_SIZE;
77
78      A_Shared_Offset_0 = 0;
79      A_Shared_Offset_1 = A_SHARED_SIZE;
80    }
81
82    shared_memory_stage ^= 1;
83  }
84
85  #pragma unroll
86  for (int K = 0; K < LOAD_K; k++) {
87
88    if (k % 2 == 0) {
89      load_Shared(
90      &A_Shared, &A_register_0,
91      &B_Shared, &B_register_0,
92      k,
93      WarpIdx, WarpIdy,
94      LaneIdx, LaneIdy,
95      A_Shared_Offset_0, B_Shared_Offset_0);
96    } else {
97      load_Shared(
98      &A_Shared, &A_register_1,
99      &B_Shared, &B_register_1,
100     k,
101     WarpIdx, WarpIdy,
102     LaneIdx, LaneIdy,
103     A_Shared_Offset_0, B_Shared_Offset_0);
104   }
105
106   if (k % 2 == 0) {
```

```
107       compute_inner(&A_register_0, &B_register_0, &Thread_Tile);
108     } else {
109       compute_inner(&A_register_1, &B_register_1, &Thread_Tile);
110     }
111   }
112
113
114   __shared__ TYPE C_Shared[M_WARPS * N_WARPS * 192];
115
116   load_C(
117   Thread_Tile, C, ldc,
118   WarpIdx, WarpIdy, LaneIdx, LaneIdy,
119   block_idx_x, block_idx_y, &C_Shared);
120
121   store_C(
122   Thread_Tile, C, ldc,
123   WarpIdx, WarpIdy, LaneIdx, LaneIdy,
124   block_idx_x, block_idx_y, &C_Shared);
```

Listing A.10: Main Loop

Helper Methods

Loading data from global memory to shared memory

load_Global

Table A.3: load_Global Parameters

A_useVector4	Specifies if we are allowed to use float4 loads for loading A
A_useVector2	Specifies if we are allowed to use float2 loads for loading A
B_useVector4	Specifies if we are allowed to use float4 loads for loading B
B_useVector2	Specifies if we are allowed to use float2 loads for loading B
K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
A_Shared	The shared memory to store the tile, column major
B_Shared	The shared memory to store the tile, row major.
A	Global A, row major
B	Global B, row major
lda	Leading dimension of A
ldb	Leading dimension of B
cta_k	Start k-index of current tile
block_idx_x	The blockId in the x dimension of the current block, it has not to be equal to blockIdx.x because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1  template<
2  bool A_useVector4, bool A_useVector2,
3  bool B_useVector4, bool B_useVector2,
4  bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK
5  >
6  __device__ __inline__ void load_Global(
7  TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
8  TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
9  const TYPE* __restrict__ A, const TYPE* __restrict__ B,
10 const int lda, const int ldb,
11 const int cta_k,
12 const int block_idx_x, const int block_idx_y,
13 const int A_Shared_Offset, const int B_Shared_Offset) {
14
15   // Load A into shared memory
16   load_A_Global<
17   A_useVector4, A_useVector2,
18   K_CHECK, THREADBLOCK_TILE_K_CHECK
19   >(
20   A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
21
22   // Load B into shared memory
```

```
23    load_B_Global<
24    B_useVector4, B_useVector2,
25    K_CHECK, THREADBLOCK_TILE_K_CHECK
26    >(B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);
27  }
```

Listing A.11: load_Global

load_A_Global

Table A.4: load_A_Global Parameters

useVector4	Specifies if we are allowed to use float4 loads
useVector2	Specifies if we are allowed to use float2 loads
K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
A_Shared	The shared memory to store the tile, column major
A	Global A, row major
lda	Leading dimension of A
cta_k	Start k-index of current tile
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1   template<
2   bool useVector4, bool useVector2,
3   bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
4   __device__ __inline__ void load_A_Global(
5   TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
6   const TYPE* __restrict__ A,  const int lda,
7   const int cta_k, const int block_idx_y, const int A_Shared_Offset) {
8
9     if ((THREADBLOCK_TILE_M * (LOAD_K / 4)) % THREADS == 0 && useVector4) {
10
11      load_A_Global_Vector4<
12      K_CHECK, THREADBLOCK_TILE_K_CHECK
13      >(A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
14
15    } else if ((THREADBLOCK_TILE_M * (LOAD_K / 2)) % THREADS == 0
16      && useVector2) {
17
18      load_A_Global_Vector2<
19      K_CHECK, THREADBLOCK_TILE_K_CHECK
20      >(A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
21
22    } else if ((THREADBLOCK_TILE_M * LOAD_K) % THREADS == 0) {
23
24      load_A_Global_Single<
25      K_CHECK, THREADBLOCK_TILE_K_CHECK>(
```

```
26      A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
27
28    } else if (useVector4) {
29
30      load_A_Global_Vector4<
31      K_CHECK, THREADBLOCK_TILE_K_CHECK>(
32      A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
33
34    } else if (useVector2) {
35
36      load_A_Global_Vector2<
37      K_CHECK, THREADBLOCK_TILE_K_CHECK>(
38      A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
39
40    } else {
41
42      load_A_Global_Single<
43      K_CHECK, THREADBLOCK_TILE_K_CHECK>(
44      A_Shared, A, lda, cta_k, block_idx_y, A_Shared_Offset);
45
46    }
47  }
```

Listing A.12: load_A_Global

load_A_Global_Vector4

Table A.5: load_A_Global_Vector4 Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
A_Shared	The shared memory to store the tile, column major
A	Global A, row major
lda	Leading dimension of A
cta_k	Start k-index of current tile
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1  template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
2  __device__ __inline__ void load_A_Global_Vector4(
3  TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
4  const TYPE* __restrict__ A,  const int lda,
5   const int cta_k, const int block_idx_y, const int A_Shared_Offset) {
6
7  constexpr int VECTORCOUNT = 4;
8  constexpr int LOAD_K_VECTOR = LOAD_K / 4;
9
10 constexpr int TIMES = (THREADBLOCK_TILE_M * LOAD_K_VECTOR + THREADS - 1)
11              / THREADS;
```

```
12
13  #pragma unroll
14  for (int i = 0; i < TIMES; i++) {
15
16    const int shared_j = (i * THREADS + threadIdx.x) % LOAD_K_VECTOR;
17    const int shared_i = (i * THREADS + threadIdx.x) / LOAD_K_VECTOR;
18
19    const auto global_i = block_idx_y * THREADBLOCK_TILE_M + shared_i;
20
21    int global_j;
22    if (SPLIT_K == 1) {
23      global_j = cta_k + shared_j * VECTORCOUNT;
24    } else {
25      global_j = blockIdx.z * THREADBLOCK_TILE_K +
26             cta_k + shared_j * VECTORCOUNT;
27    }
28
29    // If the threads do not evenly divide the whole tile,
30    // we need to make this check.
31    if ((THREADBLOCK_TILE_M * LOAD_K_VECTOR % THREADS == 0 ||
32     (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_M * LOAD_K_VECTOR)) {
33
34      VECTORTYPE4 a;
35
36      // If the tiles are not perfect multiples we need to make this checks.
37      if ((M % THREADBLOCK_TILE_M == 0 || global_i < M)
38          && (!K_CHECK || global_j < K)
39          && (!THREADBLOCK_TILE_K_CHECK ||
40              cta_k + shared_j * VECTORCOUNT < THREADBLOCK_TILE_K)) {
41
42        const TYPE* global_pointer = &A[global_i * lda + global_j];
43        a = reinterpret_cast<const VECTORTYPE4*>(global_pointer)[0];
44
45      } else {
46
47        a.x = 0.0;
48        a.y = 0.0;
49        a.z = 0.0;
50        a.w = 0.0;
51      }
52
53      /* We need to store A in this non vectorized way, because global A
54       is in row major format and shared A is column major format.
55       We cannot store shared A in row major format because
56       we would not be able to load from shared memeory to the
57       registers in an efficient way. */
58      (*A_Shared)[A_Shared_Offset + shared_i +
59       (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 0)] = a.x;
60      (*A_Shared)[A_Shared_Offset + shared_i +
61       (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 1)] = a.y;
62      (*A_Shared)[A_Shared_Offset + shared_i +
63       (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 2)] = a.z;
64      (*A_Shared)[A_Shared_Offset + shared_i +
65      (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 3)] = a.w;
```

```
66    }
67  }
```

Listing A.13: load_A_Global_Vector4

load_A_Global_Vector2

Table A.6: load_A_Global_Vector2 Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
A_Shared	The shared memory to store the tile, column major
A	Global A, row major
lda	Leading dimension of A
cta_k	Start k-index of current tile
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1  template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
2  __device__ __inline__ void load_A_Global_Vector2(
3  TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
4  const TYPE* __restrict__ A,  const int lda,
5   const int cta_k, const int block_idx_y, const int A_Shared_Offset) {
6
7  constexpr int VECTORCOUNT = 2;
8  constexpr int LOAD_K_VECTOR = LOAD_K / 2;
9
10 constexpr int TIMES = (THREADBLOCK_TILE_M * LOAD_K_VECTOR + THREADS - 1) / THREADS;
11
12 #pragma unroll
13 for (int i = 0; i < TIMES; i++) {
14
15 const int shared_j = (i * THREADS + threadIdx.x) % LOAD_K_VECTOR;
16 const int shared_i = (i * THREADS + threadIdx.x) / LOAD_K_VECTOR;
17
18 const auto global_i = block_idx_y * THREADBLOCK_TILE_M + shared_i;
19
20 int global_j;
21
22 if (SPLIT_K == 1) {
23   global_j = cta_k + shared_j * VECTORCOUNT;
24 } else {
25   global_j = blockIdx.z * THREADBLOCK_TILE_K + cta_k + shared_j * VECTORCOUNT;
26 }
27
28 // If the threads do not evenly divide the whole tile, we need to make this check.
29 if ((THREADBLOCK_TILE_M * LOAD_K_VECTOR % THREADS == 0
30  || (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_M * LOAD_K_VECTOR)) {
31
```

```
32    VECTORTYPE2 a;
33
34    // If the tiles are not perfect multiples we need to make this checks.
35    if ((M % THREADBLOCK_TILE_M == 0 || global_i < M) && (!K_CHECK || global_j < K)
36      && (!THREADBLOCK_TILE_K_CHECK
37        || cta_k + shared_j * VECTORCOUNT < THREADBLOCK_TILE_K)) {
38
39      const TYPE* global_pointer = &A[global_i * lda + global_j];
40      a = reinterpret_cast<const VECTORTYPE2*>(global_pointer)[0];
41
42    } else {
43
44      a.x = 0;
45      a.y = 0;
46    }
47
48    /* We need to store A in this non vectorized way, because global A
49     is in row major format and shared A is column major format.
50     We cannot store shared A in row major format because
51     we would not be able to load from shared memeory to the
52     registers in an efficient way. */
53
54    (*A_Shared)[A_Shared_Offset + shared_i +
55     (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 0)] = a.x;
56    (*A_Shared)[A_Shared_Offset + shared_i +
57     (THREADBLOCK_TILE_M + A_OFFSET) * (shared_j * VECTORCOUNT + 1)] = a.y;
58    }
59  }
```

Listing A.14: load_A_Global_Vector2

load_A_Global_Single

Table A.7: load_A_Global_Single Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
A_Shared	The shared memory to store the tile, column major
A	Global A, row major
lda	Leading dimension of A
cta_k	Start k-index of current tile
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1  template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
2  __device__ __inline__ void load_A_Global_Single(
3  TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
4  const TYPE* __restrict__ A,  const int lda,
5   const int cta_k, const int block_idx_y, const int A_Shared_Offset) {
```

```
6
7   constexpr int TIMES = (THREADBLOCK_TILE_M * LOAD_K + THREADS - 1) / THREADS;
8
9   #pragma unroll
10  for (int i = 0; i < TIMES; i++) {
11
12  const int shared_j = (i * THREADS + threadIdx.x) % LOAD_K;
13  const int shared_i = (i * THREADS + threadIdx.x) / LOAD_K;
14
15  const int global_i = block_idx_y * THREADBLOCK_TILE_M + shared_i;
16
17  int global_j;
18
19  if (SPLIT_K == 1) {
20    global_j = cta_k + shared_j;
21  } else {
22    global_j = blockIdx.z * THREADBLOCK_TILE_K + cta_k + shared_j;
23  }
24
25  // If the threads do not evenly divide the whole tile, we need to make this check.
26  if ((THREADBLOCK_TILE_M * LOAD_K % THREADS == 0 ||
27    (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_M * LOAD_K)) {
28    TYPE a;
29    // If the tiles are not perfect multiples we need to make this checks.
30    if ((M % THREADBLOCK_TILE_M == 0 || global_i < M) && (!K_CHECK || global_j < K)
31      && (!THREADBLOCK_TILE_K_CHECK || cta_k + shared_j < THREADBLOCK_TILE_K)) {
32
33      a = A[global_i * lda + global_j];
34
35    } else {
36      a = 0;
37    }
38
39    (*A_Shared)[A_Shared_Offset + shared_i +
40         (THREADBLOCK_TILE_M + A_OFFSET) * shared_j] = a;
41
42    }
43  }
```

Listing A.15: load_A_Global_Single

load_B_Global

Table A.8: load_B_Global Parameters

useVector4	Specifies if we are allowed to use float4 loads
useVector2	Specifies if we are allowed to use float2 loads
K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
B_Shared	he shared memory to store the tile, row major
B	Global B, row major
ldb	Leading dimension of B
cta_k	Start k-index of current tile
block_idx_x	The blockId in the x dimension of the current block, it has not to be equal to blockIdx.x because we can manually remap it

```
template<
bool useVector4, bool useVector2,
 bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
__device__ __inline__ void load_B_Global(
TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
const TYPE* __restrict__ B,  const int ldb,
const int cta_k, const int block_idx_x, const int B_Shared_Offset) {

  if (((THREADBLOCK_TILE_N / 4) * LOAD_K) % THREADS == 0 && useVector4) {
    load_B_Global_Vector4<
    K_CHECK, THREADBLOCK_TILE_K_CHECK
    >(
    B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);

  } else if (((THREADBLOCK_TILE_N / 2) * LOAD_K) % THREADS == 0 && useVector2) {
    load_B_Global_Vector2<
    K_CHECK, THREADBLOCK_TILE_K_CHECK
    >(
    B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);

  } else if ((THREADBLOCK_TILE_N * LOAD_K) % THREADS == 0) {
    load_B_Global_Single<
    K_CHECK, THREADBLOCK_TILE_K_CHECK
    >(
    B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);

  } else if (useVector4) {
    load_B_Global_Vector4<
    K_CHECK, THREADBLOCK_TILE_K_CHECK
    >(
    B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);

  } else if (useVector2) {
    load_B_Global_Vector2<
    K_CHECK, THREADBLOCK_TILE_K_CHECK
    >(
```

```
37      B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);
38
39    } else {
40      load_B_Global_Single<
41      K_CHECK, THREADBLOCK_TILE_K_CHECK
42      >(
43      B_Shared, B, ldb, cta_k, block_idx_x, B_Shared_Offset);
44    }
45
46  }
```

Listing A.16: load_B_Global

load_B_Global_Vector4

Table A.9: load_B_Global_Vector4 Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
B_Shared	he shared memory to store the tile, row major
B	Global B, row major
ldb	Leading dimension of B
cta_k	Start k-index of current tile
block_idx_x	The blockId in the x dimension of the current block, it has not to be equal to blockIdx.x because we can manually remap it

```
1   template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
2   __device__ __inline__ void load_B_Global_Vector4(
3   TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
4   const TYPE* __restrict__ B,  const int ldb,
5    const int cta_k, const int block_idx_x, const int B_Shared_Offset) {
6
7   constexpr int VECTORCOUNT = 4;
8
9   constexpr int THREADBLOCK_TILE_N_VECTOR = THREADBLOCK_TILE_N / VECTORCOUNT;
10
11  constexpr int TIMES = (THREADBLOCK_TILE_N_VECTOR * LOAD_K + THREADS - 1) / THREADS;
12
13  #pragma unroll
14  for (int i = 0; i < TIMES; i++) {
15
16    const int shared_j = (i * THREADS + threadIdx.x) % THREADBLOCK_TILE_N_VECTOR;
17    const int shared_i = (i * THREADS + threadIdx.x) / THREADBLOCK_TILE_N_VECTOR;
18
19    int global_i;
20
21    if (SPLIT_K == 1) {
22      global_i = cta_k + shared_i;
23    } else {
```

```
24      global_i = blockIdx.z * THREADBLOCK_TILE_K + cta_k + shared_i;
25    }
26
27    const auto global_j = block_idx_x * THREADBLOCK_TILE_N + shared_j * VECTORCOUNT;
28
29    // If the threads do not evenly divide the whole tile, we need to make this check.
30    if ((THREADBLOCK_TILE_N_VECTOR * LOAD_K % THREADS == 0 ||
31     (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_N_VECTOR * LOAD_K)) {
32
33      // If the tiles are not perfect multiples we need to make this checks.
34      if ((!K_CHECK || global_i < K) && (N % THREADBLOCK_TILE_N == 0 || global_j < N)
35          && (!THREADBLOCK_TILE_K_CHECK || cta_k + shared_i < THREADBLOCK_TILE_K)) {
36
37        const TYPE* global_pointer = &B[global_i * ldb + global_j];
38        VECTORTYPE4 a2 = reinterpret_cast<const VECTORTYPE4*>(global_pointer)[0];
39
40        reinterpret_cast<VECTORTYPE4*>(B_Shared)[B_Shared_Offset / 4 +
41         shared_i * THREADBLOCK_TILE_N_VECTOR + shared_j] = a2;
42
43      } else {
44
45        VECTORTYPE4 zero;
46        zero.x = 0;
47        zero.y = 0;
48        zero.z = 0;
49        zero.w = 0;
50
51        reinterpret_cast<VECTORTYPE4*>(B_Shared)[B_Shared_Offset / 4 +
52          shared_i * THREADBLOCK_TILE_N_VECTOR + shared_j] = zero;
53
54      }
55    }
56  }
```

Listing A.17: load_B_Global_Vector4

load_B_Global_Vector2

Table A.10: load_B_Global_Vector2 Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
B_Shared	he shared memory to store the tile, row major
B	Global B, row major
ldb	Leading dimension of B
cta_k	Start k-index of current tile
block_idx_x	The blockId in the x dimension of the current block, it has not to be equal to blockIdx.x because we can manually remap it

```
template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
__device__ __inline__ void load_B_Global_Vector2(
TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
const TYPE* __restrict__ B,  const int ldb,
 const int cta_k, const int block_idx_x, const int B_Shared_Offset) {

constexpr int VECTORCOUNT = 2;

constexpr int THREADBLOCK_TILE_N_VECTOR = THREADBLOCK_TILE_N / VECTORCOUNT;

constexpr int TIMES = (THREADBLOCK_TILE_N_VECTOR * LOAD_K + THREADS - 1) / THREADS;

#pragma unroll
for (int i = 0; i < TIMES; i++) {

  const int shared_j = (i * THREADS + threadIdx.x) % THREADBLOCK_TILE_N_VECTOR;
  const int shared_i = (i * THREADS + threadIdx.x) / THREADBLOCK_TILE_N_VECTOR;

  int global_i;

  if (SPLIT_K == 1) {
    global_i = cta_k + shared_i;
  } else {
    global_i = blockIdx.z * THREADBLOCK_TILE_K + cta_k + shared_i;
  }

  const auto global_j = block_idx_x * THREADBLOCK_TILE_N + shared_j * VECTORCOUNT;

  // If the threads do not evenly divide the whole tile, we need to make this check.
  if ((THREADBLOCK_TILE_N_VECTOR * LOAD_K % THREADS == 0 ||
   (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_N_VECTOR * LOAD_K)) {

    // If the tiles are not perfect multiples we need to make this checks.
    if ((!K_CHECK || global_i < K) && (N % THREADBLOCK_TILE_N == 0 || global_j < N)
        && (!THREADBLOCK_TILE_K_CHECK || cta_k + shared_i < THREADBLOCK_TILE_K)) {

      const TYPE* global_pointer = &B[global_i * ldb + global_j];
      VECTORTYPE2 a2 = reinterpret_cast<const VECTORTYPE2*>(global_pointer)[0];

      reinterpret_cast<VECTORTYPE2*>(B_Shared)[B_Shared_Offset / 2 +
       shared_i * THREADBLOCK_TILE_N_VECTOR + shared_j] = a2;

    } else {

      VECTORTYPE2 zero;
      zero.x = 0;
      zero.y = 0;

      reinterpret_cast<VECTORTYPE2*>(B_Shared)[B_Shared_Offset / 2 +
       shared_i * THREADBLOCK_TILE_N_VECTOR + shared_j] = zero;

    }
  }
}
```

Listing A.18: load_B_Global_Vector2

load_B_Global_Single

Table A.11: load_B_Global_Single Parameters

K_CHECK	Defines whether or not we need to check if we read out of bounds (< K)
THREADBLOCK_TILE_K_CHECK	Defines whether or not we need to check if we read out of bounds (< THREADBLOCK_TILE_K)
B_Shared	he shared memory to store the tile, row major
B	Global B, row major
ldb	Leading dimension of B
cta_k	Start k-index of current tile
block_idx_x	The blockId in the x dimension of the current block, it has not to be equal to blockIdx.x because we can manually remap it

```
template<bool K_CHECK, bool THREADBLOCK_TILE_K_CHECK>
__device__ __inline__ void load_B_Global_Single(
TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
const TYPE* __restrict__ B,  const int ldb,
 const int cta_k, const int block_idx_x, const int B_Shared_Offset) {
constexpr int TIMES = (THREADBLOCK_TILE_N * LOAD_K + THREADS - 1) / THREADS;

#pragma unroll
for (int i = 0; i < TIMES; i++) {

  const int shared_j = (i * THREADS + threadIdx.x) % THREADBLOCK_TILE_N;
  const int shared_i = (i * THREADS + threadIdx.x) / THREADBLOCK_TILE_N;

  int global_i;

  if (SPLIT_K == 1) {
    global_i = cta_k + shared_i;
  } else {
    global_i = blockIdx.z * THREADBLOCK_TILE_K + cta_k + shared_i;
  }

  const auto global_j = block_idx_x * THREADBLOCK_TILE_N + shared_j;

  // If the threads do not evenly divide the whole tile, we need to make this check.
  if ((THREADBLOCK_TILE_N * LOAD_K % THREADS == 0 ||
   (i * THREADS + threadIdx.x) < THREADBLOCK_TILE_N * LOAD_K)) {
    TYPE a;

      // If the tiles are not perfect multiples we need to make this checks.
    if ((!K_CHECK || global_i < K) && (N % THREADBLOCK_TILE_N == 0 || global_j < N)
        && (!THREADBLOCK_TILE_K_CHECK || cta_k + shared_i < THREADBLOCK_TILE_K)) {

      a = B[global_i * ldb + global_j];
```

```
34
35      } else {
36          a = 0;
37      }
38
39      (*B_Shared)[B_Shared_Offset + shared_i * THREADBLOCK_TILE_N + shared_j] = a;
40    }
41  }
```

Listing A.19: load_B_Global_Single

Loading data from shared memory into the register file

load_Shared

Table A.12: load_Shared Parameters

A_Shared	The shared memory to store the tile, column major
A_register	Registers to store A
B_Shared	The shared memory to store the tile, row major
B_register	Registers to store B
k	Current k index to load
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
A_Shared_Offset	Offset used to access A_Shared due to double buffering
B_Shared_Offset	Offset used to access B_Shared due to double buffering

```
1   __device__ __inline__ void load_Shared(
2   TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
3   TYPE (* __restrict__ A_register)[THREAD_TILE_M],
4   TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
5   TYPE (* __restrict__ B_register)[THREAD_TILE_N],
6   const int k, const int WarpIdx, const int WarpIdy,
7   const int LaneIdx, const int LaneIdy,
8   const int A_Shared_Offset, const int B_Shared_Offset) {
9
10    load_A_Shared(A_Shared, A_register, k, WarpIdy, LaneIdy, A_Shared_Offset);
11
12    load_B_Shared(B_Shared, B_register, k, WarpIdx, LaneIdx, B_Shared_Offset);
13  }
```

Listing A.20: load_Shared

load_A_Shared

Table A.13: load_A_Shared Parameters

A_Shared	The shared memory to store the tile, column major
A_register	Registers to store A
k	Current k index to load
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
A_Shared_Offset	Offset used to access A_Shared due to double buffering

```
__device__ __inline__ void load_A_Shared(
const TYPE (* __restrict__ A_Shared)[2 * (THREADBLOCK_TILE_M + A_OFFSET) * LOAD_K],
TYPE (* __restrict__ A_register)[ THREAD_TILE_M],
const int k, const int WarpIdy, const int LaneIdy, const int A_Shared_Offset) {

  constexpr int TIMES = THREAD_TILE_M / 4;

  constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

  const int Shared_j = k;

// We use as many float4 loads as we can
#pragma unroll
  for (int i = 0; i < TIMES; i++) {

    const int Shared_i = WarpIdy * WARP_TILE_M + i * M_THREADS * 4 + LaneIdy * 4;

    const TYPE* shared_mem_pointer = &(*A_Shared)[A_Shared_Offset + Shared_i +
      (THREADBLOCK_TILE_M + A_OFFSET) * Shared_j];

    const VECTORTYPE4 a = reinterpret_cast<const VECTORTYPE4*>(shared_mem_pointer)[0];

    TYPE* register_ptr = &(*A_register)[i * 4];

    reinterpret_cast<VECTORTYPE4*>(register_ptr)[0] = a;
  }

// If there is a rest greater equal 2, we can use one more float 2 load
  if (THREAD_TILE_M % 4 >= 2) {

    const int Shared_i = WarpIdy * WARP_TILE_M + TIMES * M_THREADS * 4 + LaneIdy * 2;

    const TYPE* shared_mem_pointer = &(*A_Shared)[A_Shared_Offset + Shared_i +
      (THREADBLOCK_TILE_M + A_OFFSET) * Shared_j];

    const VECTORTYPE2 a = reinterpret_cast<const VECTORTYPE2*>(shared_mem_pointer)[0];

    TYPE* register_ptr = &(*A_register)[TIMES * 4];

    reinterpret_cast<VECTORTYPE2*>(register_ptr)[0] = a;
  }

// And use one single load in the end, if there is still some rest
```

```
44    if (THREAD_TILE_M % 2 > 0) {
45
46      constexpr int ADDITIONAL_OFFSET_SHARED = (THREAD_TILE_M % 4 >= 2) ? M_THREADS * 2 : 0;
47
48      constexpr int ADDITIONAL_OFFSET_REGISTER = (THREAD_TILE_M % 4 >= 2) ? 2 : 0;
49
50      const int Shared_i = WarpIdy * WARP_TILE_M + TIMES * M_THREADS * 4 +
51        LaneIdy + ADDITIONAL_OFFSET_SHARED;
52
53      (*A_register)[TIMES * 4 + ADDITIONAL_OFFSET_REGISTER] =
54      (*A_Shared)[A_Shared_Offset + Shared_i + (THREADBLOCK_TILE_M + A_OFFSET) * Shared_j];
55
56    }
57  }
```

Listing A.21: load_A_Shared

load_B_Shared

Table A.14: load_B_Shared Parameters

B_Shared	The shared memory to store the tile, row major
B_register	Registers to store B
k	Current k index to load
WarpIdx	The WarpId in the x dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
B_Shared_Offset	Offset used to access B_Shared due to double buffering

```
1   __device__ __inline__ void load_B_Shared(
2   TYPE (* __restrict__ B_Shared)[2 * LOAD_K * (THREADBLOCK_TILE_N + B_OFFSET)],
3   TYPE (* __restrict__ B_register)[ THREAD_TILE_N],
4    const int k, const int WarpIdx, const int LaneIdx, const int B_Shared_Offset) {
5
6     constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
7
8     constexpr int TIMES = THREAD_TILE_N / 4;
9
10    const int Shared_i = k;
11
12  // We use as many float4 loads as we can
13  #pragma unroll
14    for (int i = 0; i < TIMES; i++) {
15
16      const int Shared_j = WarpIdx * WARP_TILE_N + LaneIdx * 4 + i * N_THREADS * 4;
17
18      const TYPE* shared_mem_pointer = &(*B_Shared)[B_Shared_Offset +
19        Shared_i * (THREADBLOCK_TILE_N + B_OFFSET) + Shared_j];
20
21      const VECTORTYPE4 a = reinterpret_cast<const VECTORTYPE4*>(shared_mem_pointer)[0];
22
23      TYPE* register_ptr = &(*B_register)[i * 4];
24
```

```
25      reinterpret_cast<VECTORTYPE4*>(register_ptr)[0] = a;
26    }
27
28  // If there is a rest greater equal 2, we can use one more float 2 load
29    if (THREAD_TILE_N % 4 >= 2) {
30
31      const int Shared_j = WarpIdx * WARP_TILE_N + LaneIdx * 2 + TIMES * N_THREADS * 4;
32
33      const TYPE* shared_mem_pointer = &(*B_Shared)[B_Shared_Offset +
34        Shared_i * (THREADBLOCK_TILE_N + B_OFFSET) + Shared_j];
35
36      const VECTORTYPE2 a = reinterpret_cast<const VECTORTYPE2*>(shared_mem_pointer)[0];
37
38      TYPE* register_ptr = &(*B_register)[TIMES * 4];
39
40      reinterpret_cast<VECTORTYPE2*>(register_ptr)[0] = a;
41    }
42
43  // And use one single load in the end, if there is still some rest
44    if (THREAD_TILE_N % 2 > 0) {
45
46      constexpr int ADDITIONAL_OFFSET_SHARED = (THREAD_TILE_N % 4 >= 2) ? N_THREADS * 2 : 0;
47
48      constexpr int ADDITIONAL_OFFSET_REGISTER = (THREAD_TILE_N % 4 >= 2) ? 2 : 0;
49
50      const int Shared_j = WarpIdx * WARP_TILE_N + LaneIdx +
51        TIMES * N_THREADS * 4 + ADDITIONAL_OFFSET_SHARED;
52
53      (*B_register)[TIMES * 4 + ADDITIONAL_OFFSET_REGISTER] =
54      (*B_Shared)[B_Shared_Offset + Shared_i * (THREADBLOCK_TILE_N + B_OFFSET) + Shared_j];
55    }
56
57  }
```

Listing A.22: load_B_Shared

Table A.15: compute_inner Parameters

A_register	Values needed from A.
B_register	Values needed from B.
Thread_Tile	The accumulator used to accumulate the result.

compute_inner

```
1  __device__ __inline__ void compute_inner(
2  const TYPE (* __restrict__ A_register)[ THREAD_TILE_M],
3  const TYPE (* __restrict__ B_register)[ THREAD_TILE_N],
4  TYPE (*Thread_Tile)[THREAD_TILE_M * THREAD_TILE_N]) {
5
6  #pragma unroll
7    for (int i = 0; i < THREAD_TILE_M; ++i) {
8  #pragma unroll
9      for (int j = 0; j < THREAD_TILE_N; ++j) {
```

```
10
11        TYPE a = (*A_register)[i];
12        TYPE b = (*B_register)[j];
13
14        (*Thread_Tile)[i * THREAD_TILE_N + j] += a * b;
15      }
16    }
17  }
```

Listing A.23: compute_inner

Loading C from global memory

load_C

Table A.16: load_C Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
Shared	The shared memory to perform the epilogue shuffle

```
1   __device__ __inline__ void load_C(
2   TYPE * __restrict__ Thread_Tile, const TYPE * __restrict__ C, const int ldc,
3   const int WarpIdx, const int WarpIdy,
4   const int LaneIdx, const int LaneIdy,
5   const int block_idx_x, const int block_idx_y,
6   TYPE (* __restrict__ Shared)[
7   (THREADBLOCK_TILE_M / WARP_TILE_M) * (THREADBLOCK_TILE_N / WARP_TILE_N) * 192]) {
8
9     if ((ALPHA != 1.0 && SPLIT_K == 1) ||
10    (ALPHA != 1.0 && SPLIT_K != 1 && ATOMIC_REDUCTION)) {
11
12  #pragma unroll
13      for (int i = 0; i < THREAD_TILE_M * THREAD_TILE_N; i++) {
14        Thread_Tile[i] *= ALPHA;
15      }
16    }
17
18    if (BETA != 0.0 && SPLIT_K == 1) {
19
20      load_C_Vector(
```

```
21      Thread_Tile, C, ldc,
22      WarpIdx, WarpIdy, LaneIdx, LaneIdy,
23      block_idx_x, block_idx_y);
24    }
25
26  }
```

Listing A.24: load_C

load_C_Single

Table A.17: load_C_Single Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
Shared	The shared memory to perform the epilogue shuffle

```
1   __device__ __inline__ void load_C_Single(
2   TYPE * __restrict__ Thread_Tile, const TYPE * __restrict__ C, const int ldc,
3   const int WarpIdx, const int WarpIdy,
4   const int LaneIdx, const int LaneIdy,
5   const int block_idx_x, const int block_idx_y) {
6
7     constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;
8
9     const int global_i_upleft = block_idx_y *
10       THREADBLOCK_TILE_M + WarpIdy * WARP_TILE_M;
11
12    constexpr int M_TIMES = THREAD_TILE_M / 4;
13
14  #pragma unroll
15    for (int i = 0; i < M_TIMES; i++) {
16
17  #pragma unroll
18      for (int ii = 0; ii < 4; ii++) {
19
20        const int global_i = global_i_upleft
21          + LaneIdy * 4 + i * M_THREADS * 4 + ii;
22
23
24          const int Threadtile_i = i * 4 + ii;
25
```

```
26          load_C_OneRow_Single(
27            Thread_Tile, C, ldc,
28            WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
29      }
30    }
31
32    if (THREAD_TILE_M % 4 >= 2) {
33
34      for (int ii = 0; ii < 2; ii++) {
35
36        const int global_i = global_i_upleft
37          + LaneIdy * 2 + M_TIMES * M_THREADS * 4 + ii;
38
39
40          const int Threadtile_i = M_TIMES * 4 + ii;
41
42          load_C_OneRow_Single(
43            Thread_Tile, C, ldc,
44            WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
45      }
46    }
47
48    if (THREAD_TILE_M % 2 > 0) {
49
50      constexpr int ADDITIONAL_OFFSET_GLOBAL =
51        (THREAD_TILE_M % 4 >= 2) ? M_THREADS * 2 : 0;
52
53      constexpr int ADDITIONAL_OFFSET_REGISTER =
54        (THREAD_TILE_M % 4 >= 2) ? 2 : 0;
55
56      const int global_i = global_i_upleft + LaneIdy +
57        M_TIMES * M_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
58
59        const int Threadtile_i = M_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
60
61        load_C_OneRow_Single(
62          Thread_Tile, C, ldc,
63          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
64    }
65  }
```

Listing A.25: load_C_Single

load_C_Vector

Table A.18: load_C_Vector Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
__device__ __inline__ void load_C_Vector(
TYPE * __restrict__ Thread_Tile, const TYPE * __restrict__ C, const int ldc,
const int WarpIdx, const int WarpIdy,
const int LaneIdx, const int LaneIdy,
 const int block_idx_x, const int block_idx_y) {

  constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

  const int global_i_upleft =
     block_idx_y * THREADBLOCK_TILE_M + WarpIdy * WARP_TILE_M;

  constexpr int M_TIMES = THREAD_TILE_M / 4;

#pragma unroll
  for (int i = 0; i < M_TIMES; i++) {

#pragma unroll
    for (int ii = 0; ii < 4; ii++) {

      const int global_i =
       global_i_upleft + LaneIdy * 4 + i * M_THREADS * 4 + ii;

      if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {

        const int Threadtile_i = i * 4 + ii;

        load_C_OneRow_Vector(
          Thread_Tile, C, ldc,
          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
      }
    }
  }

  if (THREAD_TILE_M % 4 >= 2) {

    for (int ii = 0; ii < 2; ii++) {

      const int global_i =
```

```
39          global_i_upleft + LaneIdy * 2 + M_TIMES * M_THREADS * 4 + ii;
40
41        if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {
42
43          const int Threadtile_i = M_TIMES * 4 + ii;
44
45          load_C_OneRow_Vector(
46            Thread_Tile, C, ldc,
47            WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
48        }
49      }
50    }
51
52    if (THREAD_TILE_M % 2 > 0) {
53
54      constexpr int ADDITIONAL_OFFSET_GLOBAL =
55       (THREAD_TILE_M % 4 >= 2) ? M_THREADS * 2 : 0;
56
57      constexpr int ADDITIONAL_OFFSET_REGISTER =
58       (THREAD_TILE_M % 4 >= 2) ? 2 : 0;
59
60      const int global_i = global_i_upleft + LaneIdy +
61        M_TIMES * M_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
62
63      if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {
64
65        const int Threadtile_i = M_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
66
67        load_C_OneRow_Vector(
68          Thread_Tile, C, ldc,
69          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
70      }
71    }
72  }
```

Listing A.26: load_C_Vector

load_C_OneRow_Vector

Table A.19: load_C_OneRow_Vector Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
global_i	The row to load
Threadtile_i	The row in the accumulator where to store the loaded row
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
__device__ __inline__ void load_C_OneRow_Vector(
TYPE * __restrict__ Thread_Tile, const TYPE * __restrict__ C, const int ldc,
const int WarpIdx,const int LaneIdx,
const int global_i, const int Threadtile_i, const int block_idx_x) {

  constexpr int N_TIMES = THREAD_TILE_N / 4;
  constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
  const int global_j_upleft =
   block_idx_x * THREADBLOCK_TILE_N + WarpIdx * WARP_TILE_N;

// We use as many float4 loads as we can
#pragma unroll
  for (int j = 0; j < N_TIMES; j++) {

    const int global_j = global_j_upleft + LaneIdx * 4 + j * N_THREADS * 4;

    if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {

      const TYPE* global_pointer = &C[global_i * ldc + global_j];

      TYPE* a = &Thread_Tile[Threadtile_i * THREAD_TILE_N + j * 4];

      VECTORTYPE4 a2 = reinterpret_cast<const VECTORTYPE4*>(global_pointer)[0];

      reinterpret_cast<VECTORTYPE4*>(a)[0] += BETA * a2;
    }
  }

// If there is a rest greater equal 2, we can use one more float 2 load
  if (THREAD_TILE_N % 4 >= 2) {

    const int global_j =
     global_j_upleft + LaneIdx * 2 + N_TIMES * N_THREADS * 4;

    if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {

      const TYPE* global_pointer = &C[global_i * ldc + global_j];

      TYPE* a = &Thread_Tile[Threadtile_i * THREAD_TILE_N + N_TIMES * 4];

      VECTORTYPE2 a2 = reinterpret_cast<const VECTORTYPE2*>(global_pointer)[0];

      reinterpret_cast<VECTORTYPE2*>(a)[0] += BETA * a2;
    }
  }

// And use one single load in the end, if there is still some rest
  if (THREAD_TILE_N % 2 > 0) {

    constexpr int ADDITIONAL_OFFSET_GLOBAL =
     (THREAD_TILE_N % 4 >= 2) ? N_THREADS * 2 : 0;

    constexpr int ADDITIONAL_OFFSET_REGISTER =
     (THREAD_TILE_N % 4 >= 2) ? 2 : 0;
```

```
55
56      const int global_j = global_j_upleft + LaneIdx
57       + N_TIMES * N_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
58
59      if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {
60
61        Thread_Tile[Threadtile_i * THREAD_TILE_N + N_TIMES * 4
62         + ADDITIONAL_OFFSET_REGISTER] += BETA * C[global_i * ldc + global_j];
63      }
64    }
65  }
```

Listing A.27: load_C_OneRow_Vector

load_C_OneRow_Single

Table A.20: load_C_OneRow_Single Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
global_i	The row to load
Threadtile_i	The row in the accumulator where to store the loaded row
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
THREAD_TILE_Y_HEIGHT	The height of the current thread tile in the y dimension
Shared	The shared memory to perform the epilogue shuffle

```
1   _device__ __inline__ void load_C_OneRow_Single(
2   TYPE * __restrict__ Thread_Tile, const TYPE * __restrict__ C, const int ldc,
3   const int WarpIdx, const int LaneIdx, const int LaneIdy,
4   const int global_i_upleft, const int Threadtile_i,
5   const int block_idx_x, const int THREAD_TILE_Y_HEIGHT,
6   TYPE (* __restrict__ Shared)[
7   (THREADBLOCK_TILE_M / WARP_TILE_M) * (THREADBLOCK_TILE_N / WARP_TILE_N) * 192]) {
8
9     constexpr int N_TIMES = THREAD_TILE_N / 4;
10
11    constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
12    constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;
13    constexpr int SHARED_MEM_PER_WARP = 192;
14
15    const int threadId = threadIdx.x % 32;
16
17    const int WarpId = threadIdx.x / 32;
18
19  #pragma unroll
20    for (int j = 0; j < N_TIMES; j++) {
```

```
21
22      constexpr int EPILOGUE_N = N_THREADS * 4; // Size N of the epilogue tile
23
24      const int epilogue_i_write = LaneIdy; // i index in the epilogue tile
25      const int epilogue_j_write = LaneIdx * 4; // j index in the epilogue tile
26
27      // M_THREADS: 1, EPILOGUE_OFFSET: 0
28      // M_THREADS: 2, EPILOGUE_OFFSET: 16
29      // M_THREADS: 4, EPILOGUE_OFFSET: 8
30      // M_THREADS: 8, EPILOGUE_OFFSET: 4
31      // M_THREADS: 16, EPILOGUE_OFFSET:2
32      // M_THREADS: 32, EPILOGUE_OFFSET: 1
33      constexpr int EPILOGUE_OFFSET = (M_THREADS == 1) ? 0 :
34                      (M_THREADS == 2) ? 16 :
35                      (M_THREADS == 4) ? 8 :
36                      (M_THREADS == 8) ? 4 :
37                      (M_THREADS == 16) ? 4 : 0;
38
39  #if __CUDA_ARCH__ >= 700
40      __syncwarp();
41  #endif
42
43      if (EPILOGUE_N <= 32) {
44
45        const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
46        const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
47
48        const int global_i = global_i_upleft +
49        threadIdIdy_epilogue * (4 * THREAD_TILE_Y_HEIGHT);
50        const int global_j = block_idx_x * THREADBLOCK_TILE_N
51        + WarpIdx * WARP_TILE_N + threadIdIdx_epilogue + j * N_THREADS * 4;
52
53        TYPE a0, a1, a2, a3;
54
55        if (SPLIT_K == 1) {
56          if ((M % THREADBLOCK_TILE_M == 0 ||
57           global_i + 0 * THREAD_TILE_Y_HEIGHT < M)
58            && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
59            a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
60          }
61          if ((M % THREADBLOCK_TILE_M == 0 ||
62           global_i + 1 * THREAD_TILE_Y_HEIGHT < M)
63            && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
64            a1 = C[(global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
65          }
66          if ((M % THREADBLOCK_TILE_M == 0 ||
67           global_i + 2 * THREAD_TILE_Y_HEIGHT < M)
68           && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
69            a2 = C[(global_i + 2 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
70          }
71          if ((M % THREADBLOCK_TILE_M == 0 ||
72           global_i + 3 * THREAD_TILE_Y_HEIGHT < M)
73            && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
74            a3 = C[(global_i + 3 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
```

```
75          }
76        }
77
78        (*Shared)[SHARED_MEM_PER_WARP * WarpId
79        + (threadIdIdy_epilogue * 4 + 0) * (EPILOGUE_N + EPILOGUE_OFFSET)
80         + threadIdIdx_epilogue] = a0;
81        (*Shared)[SHARED_MEM_PER_WARP * WarpId
82        + (threadIdIdy_epilogue * 4 + 1) * (EPILOGUE_N + EPILOGUE_OFFSET)
83         + threadIdIdx_epilogue] = a1;
84        (*Shared)[SHARED_MEM_PER_WARP * WarpId
85        + (threadIdIdy_epilogue * 4 + 2) * (EPILOGUE_N + EPILOGUE_OFFSET)
86         + threadIdIdx_epilogue] = a2;
87        (*Shared)[SHARED_MEM_PER_WARP * WarpId
88        + (threadIdIdy_epilogue * 4 + 3) * (EPILOGUE_N + EPILOGUE_OFFSET)
89        + threadIdIdx_epilogue] = a3;
90
91      } else if (EPILOGUE_N == 64) {
92
93        const int threadIdIdx_epilogue_1 = threadId;
94        const int threadIdIdx_epilogue_2 = threadId + 32;
95
96        const int global_i = global_i_upleft;
97
98        const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N
99         + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_1;
100       const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N
101        + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_2;
102
103       TYPE a0, a1, a2, a3;
104
105       if (SPLIT_K == 1) {
106         if ((M % THREADBLOCK_TILE_M == 0 ||
107          (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
108           && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
109           a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1];
110         }
111         if ((M % THREADBLOCK_TILE_M == 0 ||
112          (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
113           && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
114           a1 = C[(global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1];
115         }
116         if ((M % THREADBLOCK_TILE_M == 0 ||
117          (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
118           && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
119           a2 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2];
120         }
121         if ((M % THREADBLOCK_TILE_M == 0 ||
122          (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
123          && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
124           a3 = C[(global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2];
125         }
126
127       }
128
```

```
129         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
130         0 * (EPILOGUE_N + EPILOGUE_OFFSET)
131          + threadIdIdx_epilogue_1] = a0;
132         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
133          1 * (EPILOGUE_N + EPILOGUE_OFFSET)
134           + threadIdIdx_epilogue_1] = a1;
135         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
136          0 * (EPILOGUE_N + EPILOGUE_OFFSET)
137          + threadIdIdx_epilogue_2] = a2;
138         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
139          1 * (EPILOGUE_N + EPILOGUE_OFFSET)
140           + threadIdIdx_epilogue_2] = a3;
141
142       }
143
144       else if (EPILOGUE_N == 128) {
145
146         const int threadIdIdx_epilogue_1 = threadId;
147         const int threadIdIdx_epilogue_2 = threadId + 32;
148         const int threadIdIdx_epilogue_3 = threadId + 64;
149         const int threadIdIdx_epilogue_4 = threadId + 96;
150
151         const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N
152         + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_1;
153         const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N
154         + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_2;
155         const int global_j_3 = block_idx_x * THREADBLOCK_TILE_N
156         + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_3;
157         const int global_j_4 = block_idx_x * THREADBLOCK_TILE_N
158          + WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_4;
159
160         const int global_i = global_i_upleft;
161
162         TYPE a0, a1, a2, a3;
163
164         // Store the values into C
165         if (SPLIT_K == 1) {
166           if ((M % THREADBLOCK_TILE_M == 0 ||
167           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
168           && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
169             a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1];
170           }
171           if ((M % THREADBLOCK_TILE_M == 0 ||
172            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
173             && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
174             a1 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2];
175           }
176           if ((M % THREADBLOCK_TILE_M == 0 ||
177           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
178            && (N % THREADBLOCK_TILE_N == 0 || global_j_3 < N)) {
179             a2 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_3];
180           }
181           if ((M % THREADBLOCK_TILE_M == 0 ||
182            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
```

```
183             && (N % THREADBLOCK_TILE_N == 0 || global_j_4 < N)) {
184             a3 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_4];
185           }
186
187         }
188
189         (*Shared)[SHARED_MEM_PER_WARP * WarpId
190         + 0 * (EPILOGUE_N + EPILOGUE_OFFSET) +  threadIdIdx_epilogue_1] = a0;
191         (*Shared)[SHARED_MEM_PER_WARP * WarpId
192          + 0 * (EPILOGUE_N + EPILOGUE_OFFSET) +  threadIdIdx_epilogue_2] = a1;
193         (*Shared)[SHARED_MEM_PER_WARP * WarpId
194          + 0 * (EPILOGUE_N + EPILOGUE_OFFSET) +  threadIdIdx_epilogue_3] = a2;
195         (*Shared)[SHARED_MEM_PER_WARP * WarpId
196         + 0 * (EPILOGUE_N + EPILOGUE_OFFSET) +   threadIdIdx_epilogue_4] = a3;
197
198     }
199
200 #if __CUDA_ARCH__ >= 700
201     __syncwarp();
202 #endif
203     const int Threadtile_j = j * 4;
204
205     TYPE* a_ptr = &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
206
207     TYPE* shared_ptr = &(*Shared)[SHARED_MEM_PER_WARP * WarpId +
208     epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilogue_j_write];
209     reinterpret_cast<VECTORTYPE4*>(a_ptr)[0] +=
210     BETA * reinterpret_cast<VECTORTYPE4*>(shared_ptr)[0];
211
212   }
213
214   if (THREAD_TILE_N % 4 >= 2) {
215
216     constexpr int EPILOGUE_N = N_THREADS * 2;
217
218     const int epilogue_i_write = LaneIdy;
219     const int epilogue_j_write = LaneIdx * 2;
220
221     // M_THREADS: 1, EPILOGUE_OFFSET: 0
222     // M_THREADS: 2, EPILOGUE_OFFSET: 16
223     // M_THREADS: 4, EPILOGUE_OFFSET: 8
224     // M_THREADS: 8, EPILOGUE_OFFSET: 4
225     // M_THREADS: 16, EPILOGUE_OFFSET:0
226     // M_THREADS: 32, EPILOGUE_OFFSET: 1
227     constexpr int EPILOGUE_OFFSET = (M_THREADS == 1 || M_THREADS == 16) ? 0 :
228                     (M_THREADS == 32) ? 0 : (M_THREADS == 4) ? 8 :
229                     (M_THREADS == 2) ? 16 : 4;
230
231 #if __CUDA_ARCH__ >= 700
232     __syncwarp();
233 #endif
234
235     if (EPILOGUE_N <= 32) {
236
```

```
237         const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
238         const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
239
240         const int global_i = global_i_upleft +
241         threadIdIdy_epilogue * (2 * THREAD_TILE_Y_HEIGHT);
242         const int global_j = block_idx_x * THREADBLOCK_TILE_N
243         + WarpIdx * WARP_TILE_N + threadIdIdx_epilogue + N_TIMES * N_THREADS * 4;
244
245         TYPE a0, a1;
246
247         if (SPLIT_K == 1) {
248           if ((M % THREADBLOCK_TILE_M == 0 ||
249            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
250             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
251             a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
252           }
253           if ((M % THREADBLOCK_TILE_M == 0 ||
254           (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
255            && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
256             a1 = C[(global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
257           }
258
259         }
260
261         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
262         (threadIdIdy_epilogue * 2 + 0) * (EPILOGUE_N + EPILOGUE_OFFSET) +
263         threadIdIdx_epilogue] = a0;
264         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
265         (threadIdIdy_epilogue * 2 + 1) * (EPILOGUE_N + EPILOGUE_OFFSET) +
266          threadIdIdx_epilogue] = a1;
267
268       } else if (EPILOGUE_N == 64) {
269
270         const int global_i = global_i_upleft;
271
272         const int threadIdIdx_epilogue_1 = threadId;
273         const int threadIdIdx_epilogue_2 = threadId + 32;
274
275         const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N +
276         WarpIdx * WARP_TILE_N + N_TIMES * N_THREADS * 4 + threadIdIdx_epilogue_1;
277         const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N +
278          WarpIdx * WARP_TILE_N + N_TIMES * N_THREADS * 4 + threadIdIdx_epilogue_2;
279
280         TYPE a0, a2;
281
282         if (SPLIT_K == 1) {
283           if ((M % THREADBLOCK_TILE_M == 0 ||
284            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
285            && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
286             a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1];
287           }
288           if ((M % THREADBLOCK_TILE_M == 0 ||
289           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
290            && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
```

```
291             a2 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2];
292           }
293         }
294
295         // Load the values
296         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
297         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_1] = a0;
298         (*Shared)[SHARED_MEM_PER_WARP * WarpId +
299          0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_2] = a2;
300
301         // Store the values into C
302
303       }
304   #if __CUDA_ARCH__ >= 700
305       __syncwarp();
306   #endif
307
308       const int Threadtile_j = N_TIMES * 4;
309
310       TYPE* a_ptr = &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
311
312       TYPE* shared_ptr = &(*Shared)[SHARED_MEM_PER_WARP * WarpId
313        + epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilogue_j_write];
314       reinterpret_cast<VECTORTYPE2*>(a_ptr)[0] +=
315       BETA * reinterpret_cast<VECTORTYPE2*>(shared_ptr)[0];
316
317     }
318
319     if (THREAD_TILE_N % 2 > 0) {
320
321       constexpr int EPILOGUE_N = N_THREADS;
322
323       const int epilogue_i_write = LaneIdy;
324       const int epilogue_j_write = LaneIdx;
325
326       // M_THREADS: 1, EPILOGUE_OFFSET: 0
327       // M_THREADS: 2, EPILOGUE_OFFSET: 0
328       // M_THREADS: 4, EPILOGUE_OFFSET: 16
329       // M_THREADS: 8, EPILOGUE_OFFSET: 0
330       // M_THREADS: 16, EPILOGUE_OFFSET:0
331       // M_THREADS: 32, EPILOGUE_OFFSET: 0
332       constexpr int EPILOGUE_OFFSET = (M_THREADS == 4) ? 16 : 0;
333
334   #if __CUDA_ARCH__ >= 700
335       __syncwarp();
336   #endif
337
338       const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
339       const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
340
341       constexpr int ADDITIONAL_OFFSET_GLOBAL =
342       (THREAD_TILE_N % 4 >= 2) ? N_THREADS * 2 : 0;
343
344       const int global_i = global_i_upleft +
```

```
345     threadIdIdy_epilogue * (1 * THREAD_TILE_Y_HEIGHT);
346     const int global_j = block_idx_x * THREADBLOCK_TILE_N +
347     WarpIdx * WARP_TILE_N + threadIdIdx_epilogue + N_TIMES * N_THREADS * 4
348         + ADDITIONAL_OFFSET_GLOBAL;
349
350     TYPE a0;
351
352     if (SPLIT_K == 1) {
353       if ((M % THREADBLOCK_TILE_M == 0 ||
354        (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
355         && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
356         a0 = C[(global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j];
357       }
358     }
359
360     (*Shared)[SHARED_MEM_PER_WARP * WarpId +
361     (threadIdIdy_epilogue * 1 + 0) * (EPILOGUE_N + EPILOGUE_OFFSET)
362     + threadIdIdx_epilogue] = a0;
363
364 #if __CUDA_ARCH__ >= 700
365     __syncwarp();
366 #endif
367
368     constexpr int ADDITIONAL_OFFSET_REGISTER =
369     (THREAD_TILE_N % 4 >= 2) ? 2 : 0;
370
371     const int Threadtile_j = N_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
372
373     Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j] += BETA
374         * (*Shared)[SHARED_MEM_PER_WARP * WarpId +
375         epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilogue_j_write];
376
377   }
378
379 }
```

Listing A.28: load_C_OneRow_Single

Storing C to global memory

store_C

Table A.21: store_C Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
Shared	The shared memory to perform the epilogue shuffle

```
__device__ __inline__ void store_C(
const TYPE * __restrict__ Thread_Tile, TYPE * __restrict__ C, const int ldc,
const int WarpIdx, const int WarpIdy,
const int LaneIdx, const int LaneIdy, const int block_idx_x, const int block_idx_y) {

  if (SPLIT_K == 1) {

    store_C_Vector(
    Thread_Tile, C, ldc,
    WarpIdx, WarpIdy, LaneIdx, LaneIdy, block_idx_x, block_idx_y);

  } else {

    store_C_Single(
    Thread_Tile, C, ldc,
    WarpIdx, WarpIdy, LaneIdx, LaneIdy, block_idx_x, block_idx_y,Shared);
  }
}
```

Listing A.29: store_C

store_C_Single

Table A.22: store_C_Single Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
Shared	The shared memory to perform the epilogue shuffle

```
__device__ __inline__ void store_C_Single(
const TYPE * __restrict__ Thread_Tile, TYPE * __restrict__ C, const int ldc,
const int WarpIdx, const int WarpIdy,
const int LaneIdx, const int LaneIdy,
 const int block_idx_x, const int block_idx_y) {

  constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

  const int global_i_upleft =
   block_idx_y * THREADBLOCK_TILE_M + WarpIdy * WARP_TILE_M;

  constexpr int M_TIMES = THREAD_TILE_M / 4;

#pragma unroll
  for (int i = 0; i < M_TIMES; i++) {

#pragma unroll
    for (int ii = 0; ii < 4; ii++) {

      const int global_i = global_i_upleft + LaneIdy * 4 + i * M_THREADS * 4 + ii;

        const int Threadtile_i = i * 4 + ii;

        store_C_OneRow_Single(
          Thread_Tile, C, ldc,
          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
    }
  }

  if (THREAD_TILE_M % 4 >= 2) {

    for (int ii = 0; ii < 2; ii++) {

      const int global_i =
       global_i_upleft + LaneIdy * 2 + M_TIMES * M_THREADS * 4 + ii;

```

```
37
38          const int Threadtile_i = M_TIMES * 4 + ii;
39
40          store_C_OneRow_Single(
41            Thread_Tile, C, ldc,
42            WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
43      }
44    }
45
46    if (THREAD_TILE_M % 2 > 0) {
47
48      constexpr int ADDITIONAL_OFFSET_GLOBAL =
49       (THREAD_TILE_M % 4 >= 2) ? M_THREADS * 2 : 0;
50
51      constexpr int ADDITIONAL_OFFSET_REGISTER =
52       (THREAD_TILE_M % 4 >= 2) ? 2 : 0;
53
54      const int global_i =
55       global_i_upleft + LaneIdy + M_TIMES * M_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
56
57
58        const int Threadtile_i = M_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
59
60        store_C_OneRow_Single(
61          Thread_Tile, C, ldc,
62          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
63    }
64  }
```

Listing A.30: store_C_Single

store_C_Vector

Table A.23: store_C_Vector Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
WarpIdy	The WarpId in the y dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
LaneIdy	The LaneId in the y dimension of the current thread
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
block_idx_y	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1  __device__ __inline__ void store_C_Vector(
2  const TYPE * __restrict__ Thread_Tile, TYPE * __restrict__ C,
3  const int ldc, const int WarpIdx, const int WarpIdy,
4  const int LaneIdx, const int LaneIdy, const int block_idx_x, const int block_idx_y) {
```

```
5
6     constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;
7
8     const int global_i_upleft = block_idx_y * THREADBLOCK_TILE_M + WarpIdy * WARP_TILE_M;
9
10    constexpr int M_times = THREAD_TILE_M / 4;
11
12  #pragma unroll
13    for (int i = 0; i < M_times; i++) {
14
15  #pragma unroll
16      for (int ii = 0; ii < 4; ii++) {
17
18        const int global_i = global_i_upleft + LaneIdy * 4 + i * M_THREADS * 4 + ii;
19
20        if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {
21
22          const int Threadtile_i = i * 4 + ii;
23
24          store_C_OneRow_Vector(
25          Thread_Tile, C, ldc,
26          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
27        }
28      }
29    }
30
31    if (THREAD_TILE_M % 4 >= 2) {
32
33      for (int ii = 0; ii < 2; ii++) {
34
35        const int global_i = global_i_upleft + LaneIdy * 2 + M_times * M_THREADS * 4 + ii;
36
37        const int Threadtile_i = M_times * 4 + ii;
38
39        if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {
40
41          store_C_OneRow_Vector(
42          Thread_Tile, C, ldc,
43          WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
44        }
45      }
46    }
47
48    if (THREAD_TILE_M % 2 > 0) {
49
50      constexpr int ADDITIONAL_OFFSET_GLOBAL = (THREAD_TILE_M % 4 >= 2) ? M_THREADS * 2 : 0;
51
52      constexpr int ADDITIONAL_OFFSET_REGISTER = (THREAD_TILE_M % 4 >= 2) ? 2 : 0;
53
54      const int global_i =
55       global_i_upleft + LaneIdy + M_times * M_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
56
57      const int Threadtile_i = M_times * 4 + ADDITIONAL_OFFSET_REGISTER;
58
```

```
59      if (M % THREADBLOCK_TILE_M == 0 || global_i < M) {
60
61        store_C_OneRow_Vector(
62        Thread_Tile, C, ldc,
63        WarpIdx, LaneIdx, global_i, Threadtile_i, block_idx_x);
64      }
65    }
66  }
```

Listing A.31: store_C_Vector

store_C_OneRow_Vector

Table A.24: store_C_OneRow_Vector Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
global_i	The row to load
Threadtile_i	The row in the accumulator where to store the loaded row
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it

```
1   __device__ __inline__ void store_C_OneRow_Vector(
2   const TYPE * __restrict__ Thread_Tile, TYPE * __restrict__ C, const int ldc,
3   const int WarpIdx, const int LaneIdx,
4   const int global_i, const int Threadtile_i, const int block_idx_x) {
5
6     constexpr int N_TIMES = THREAD_TILE_N / 4;
7     constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
8     const int global_j_upleft =
9      block_idx_x * THREADBLOCK_TILE_N + WarpIdx * WARP_TILE_N;
10
11  #pragma unroll
12    for (int j = 0; j < N_TIMES; j++) {
13
14      const int global_j = global_j_upleft + LaneIdx * 4 + j * N_THREADS * 4;
15
16      if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {
17
18        TYPE* global_pointer = &C[global_i * ldc + global_j];
19
20        const int Threadtile_j = j * 4;
21
22        const TYPE* a = &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
23
24        const VECTORTYPE4 a2 = reinterpret_cast<const VECTORTYPE4*>(a)[0];
25
26        reinterpret_cast<VECTORTYPE4*>(global_pointer)[0] = a2;
```

```
27      }
28    }
29
30    if (THREAD_TILE_N % 4 >= 2) {
31
32      const int global_j = global_j_upleft + LaneIdx * 2 + N_TIMES * N_THREADS * 4;
33
34      if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {
35
36        TYPE* global_pointer = &C[global_i * ldc + global_j];
37
38        const int Threadtile_j = N_TIMES * 4;
39
40        const TYPE* a = &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
41
42        const VECTORTYPE2 a2 = reinterpret_cast<const VECTORTYPE2*>(a)[0];
43
44        reinterpret_cast<VECTORTYPE2*>(global_pointer)[0] = a2;
45      }
46    }
47
48    if (THREAD_TILE_N % 2 > 0) {
49
50      constexpr int ADDITIONAL_OFFSET_GLOBAL =
51       (THREAD_TILE_N % 4 >= 2) ? N_THREADS * 2 : 0;
52
53      constexpr int ADDITIONAL_OFFSET_REGISTER =
54       (THREAD_TILE_N % 4 >= 2) ? 2 : 0;
55
56      const int global_j =
57       global_j_upleft + LaneIdx + N_TIMES * N_THREADS * 4 + ADDITIONAL_OFFSET_GLOBAL;
58
59      const int Threadtile_j = N_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
60
61      if (N % THREADBLOCK_TILE_N == 0 || global_j < N) {
62
63        C[global_i * ldc + global_j] =
64         Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
65      }
66    }
67  }
```

Listing A.32: store_C_OneRow_Vector

store_C_OneRow_Single

Table A.25: store_C_OneRow_Single Parameters

Thread_Tile	The accumulator used to accumulate the result.
C	Global C, row major
ldc	Leading dimension of C
WarpIdx	The WarpId in the x dimension of the current thread
LaneIdx	The LaneId in the x dimension of the current thread
global_i	The row to load
Threadtile_i	The row in the accumulator where to store the loaded row
block_idx_x	The blockId in the y dimension of the current block, it has not to be equal to blockIdx.y because we can manually remap it
THREAD_TILE_Y_HEIGHT	The height of the current thread tile in the y dimension
Shared	The shared memory to perform the epilogue shuffle

```
__device__ __inline__ void store_C_OneRow_Single(
const TYPE * __restrict__ Thread_Tile, TYPE * __restrict__ C, const int ldc,
const int WarpIdx, const int LaneIdx, const int LaneIdy,
const int global_i_upleft, const int Threadtile_i,
const int block_idx_x,
TYPE (* __restrict__ Shared)[
(THREADBLOCK_TILE_M / WARP_TILE_M) * (THREADBLOCK_TILE_N / WARP_TILE_N) * 192],
 const int THREAD_TILE_Y_HEIGHT) {

  constexpr int N_TIMES = THREAD_TILE_N / 4;

  constexpr int N_THREADS = WARP_TILE_N / THREAD_TILE_N;
  constexpr int M_THREADS = WARP_TILE_M / THREAD_TILE_M;

  const int split_K_OFFSET = (SPLIT_K != 1) ? (blockIdx.z * M * N) : 0;

  constexpr int SHARED_MEM_PER_WARP = 192;

  const int threadId = threadIdx.x % 32;

  const int WarpId = threadIdx.x / 32;

  {

    constexpr int EPILOGUE_N = N_THREADS * 4;

    const int epilogue_i_write = LaneIdy;
    const int epilogue_j_write = LaneIdx * 4;

    // M_THREADS: 1, EPILOGUE_OFFSET: 0
    // M_THREADS: 2, EPILOGUE_OFFSET: 16
    // M_THREADS: 4, EPILOGUE_OFFSET: 8
    // M_THREADS: 8, EPILOGUE_OFFSET: 4
    // M_THREADS: 16, EPILOGUE_OFFSET:2
    // M_THREADS: 32, EPILOGUE_OFFSET: 1
    constexpr int EPILOGUE_OFFSET = (M_THREADS == 1) ? 0 :
```

```
37                      (M_THREADS == 2) ? 16 :
38                      (M_THREADS == 4) ? 8 :
39                      (M_THREADS == 8) ? 4 :
40                      (M_THREADS == 16) ? 2 : 1;
41
42  #pragma unroll
43      for (int j = 0; j < N_TIMES; j++) {
44
45        const int Threadtile_j = j * 4;
46
47        const TYPE* a_ptr =
48          &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
49        VECTORTYPE4 a_val =
50          reinterpret_cast<const VECTORTYPE4*>(a_ptr)[0];
51
52        TYPE* shared_ptr =
53          &(*Shared)[SHARED_MEM_PER_WARP * WarpId +
54            epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilogue_j_write];
55        reinterpret_cast<VECTORTYPE4*>(shared_ptr)[0] = a_val;
56
57  #if __CUDA_ARCH__ >= 700
58        __syncwarp();
59  #endif
60
61        if (EPILOGUE_N <= 32) {
62
63          const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
64          const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
65
66          // Load the values
67          const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId
68          + (threadIdIdy_epilogue * 4 + 0) * (EPILOGUE_N + EPILOGUE_OFFSET)
69           + threadIdIdx_epilogue];
70          const TYPE a1 = (*Shared)[SHARED_MEM_PER_WARP * WarpId
71          + (threadIdIdy_epilogue * 4 + 1) * (EPILOGUE_N + EPILOGUE_OFFSET)
72           + threadIdIdx_epilogue];
73          const TYPE a2 = (*Shared)[SHARED_MEM_PER_WARP * WarpId
74          + (threadIdIdy_epilogue * 4 + 2) * (EPILOGUE_N + EPILOGUE_OFFSET)
75           + threadIdIdx_epilogue];
76          const TYPE a3 = (*Shared)[SHARED_MEM_PER_WARP * WarpId
77          + (threadIdIdy_epilogue * 4 + 3) * (EPILOGUE_N + EPILOGUE_OFFSET)
78           + threadIdIdx_epilogue];
79
80          const int global_i = global_i_upleft +
81           threadIdIdy_epilogue * (4 * THREAD_TILE_Y_HEIGHT);
82          const int global_j = block_idx_x * THREADBLOCK_TILE_N +
83          WarpIdx * WARP_TILE_N + threadIdIdx_epilogue + j * N_THREADS * 4;
84
85          // Store the values into C
86
87          if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
88            if ((M % THREADBLOCK_TILE_M == 0 ||
89             global_i + 0 * THREAD_TILE_Y_HEIGHT < M)
90              && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
```

```
91              C[split_K_OFFSET +
92              (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a0;
93            }
94            if ((M % THREADBLOCK_TILE_M == 0 ||
95             global_i + 1 * THREAD_TILE_Y_HEIGHT < M)
96              && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
97              C[split_K_OFFSET +
98              (global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a1;
99            }
100           if ((M % THREADBLOCK_TILE_M == 0 ||
101            global_i + 2 * THREAD_TILE_Y_HEIGHT < M)
102             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
103             C[split_K_OFFSET +
104             (global_i + 2 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a2;
105           }
106           if ((M % THREADBLOCK_TILE_M == 0 ||
107            global_i + 3 * THREAD_TILE_Y_HEIGHT < M)
108             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
109             C[split_K_OFFSET +
110             (global_i + 3 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a3;
111           }
112         } else {
113           if ((M % THREADBLOCK_TILE_M == 0 ||
114            global_i + 0 * THREAD_TILE_Y_HEIGHT < M)
115            && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
116             atomicAdd(&C[(global_i +
117              0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a0);
118           }
119           if ((M % THREADBLOCK_TILE_M == 0 ||
120            global_i + 1 * THREAD_TILE_Y_HEIGHT < M)
121             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
122             atomicAdd(&C[(global_i +
123             1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a1);
124           }
125           if ((M % THREADBLOCK_TILE_M == 0 ||
126            global_i + 2 * THREAD_TILE_Y_HEIGHT < M)
127             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
128             atomicAdd(&C[(global_i +
129             2 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a2);
130           }
131           if ((M % THREADBLOCK_TILE_M == 0 ||
132            global_i + 3 * THREAD_TILE_Y_HEIGHT < M)
133             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
134             atomicAdd(&C[(global_i +
135             3 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a3);
136           }
137         }
138
139       } else if (EPILOGUE_N == 64) {
140
141         const int threadIdIdx_epilogue_1 = threadId;
142         const int threadIdIdx_epilogue_2 = threadId + 32;
143
144         // Load the values
```

```
145         const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
146         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_1];
147         const TYPE a1 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
148         1 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_1];
149         const TYPE a2 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
150         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_2];
151         const TYPE a3 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
152          1 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_2];
153
154         const int global_i = global_i_upleft;
155
156         const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N + WarpIdx *
157         WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_1;
158         const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N + WarpIdx *
159         WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_2;
160
161         // Store the values into C
162         if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
163           if ((M % THREADBLOCK_TILE_M == 0 ||
164            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
165             && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
166             C[split_K_OFFSET + (global_i + 0 * THREAD_TILE_Y_HEIGHT)
167              * ldc + global_j_1] = a0;
168           }
169           if ((M % THREADBLOCK_TILE_M == 0 ||
170           (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
171           && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
172             C[split_K_OFFSET + (global_i + 1 * THREAD_TILE_Y_HEIGHT)
173              * ldc + global_j_1] = a1;
174           }
175           if ((M % THREADBLOCK_TILE_M == 0 ||
176           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
177            && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
178             C[split_K_OFFSET + (global_i + 0 * THREAD_TILE_Y_HEIGHT)
179              * ldc + global_j_2] = a2;
180           }
181           if ((M % THREADBLOCK_TILE_M == 0 ||
182            (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
183             && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
184             C[split_K_OFFSET + (global_i + 1 * THREAD_TILE_Y_HEIGHT)
185              * ldc + global_j_2] = a3;
186           }
187
188         } else {
189           if ((M % THREADBLOCK_TILE_M == 0 ||
190            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
191             && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
192             atomicAdd(&C[(global_i + 0 * THREAD_TILE_Y_HEIGHT)
193              * ldc + global_j_1], a0);
194           }
195           if ((M % THREADBLOCK_TILE_M == 0 ||
196            (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
197            && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
198             atomicAdd(&C[(global_i + 1 * THREAD_TILE_Y_HEIGHT)
```

```
199               * ldc + global_j_1], a1);
200           }
201           if ((M % THREADBLOCK_TILE_M == 0 ||
202            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
203             && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
204             atomicAdd(&C[(global_i + 0 * THREAD_TILE_Y_HEIGHT)
205              * ldc + global_j_2], a2);
206           }
207           if ((M % THREADBLOCK_TILE_M == 0 ||
208           (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
209            && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
210             atomicAdd(&C[(global_i + 1 * THREAD_TILE_Y_HEIGHT)
211              * ldc + global_j_2], a3);
212           }
213         }
214
215       } else if (EPILOGUE_N == 128) {
216
217         const int threadIdIdx_epilogue_1 = threadId;
218         const int threadIdIdx_epilogue_2 = threadId + 32;
219         const int threadIdIdx_epilogue_3 = threadId + 64;
220         const int threadIdIdx_epilogue_4 = threadId + 96;
221
222         const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
223         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_1];
224         const TYPE a1 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
225         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_2];
226         const TYPE a2 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
227          0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_3];
228         const TYPE a3 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
229         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_4];
230
231         const int global_i = global_i_upleft;
232
233         const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N +
234          WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_1;
235         const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N +
236          WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_2;
237         const int global_j_3 = block_idx_x * THREADBLOCK_TILE_N +
238          WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_3;
239         const int global_j_4 = block_idx_x * THREADBLOCK_TILE_N +
240         WarpIdx * WARP_TILE_N + j * N_THREADS * 4 + threadIdIdx_epilogue_4;
241
242         // Store the values into C
243         if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
244           if ((M % THREADBLOCK_TILE_M == 0 ||
245            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
246             && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
247             C[split_K_OFFSET +
248             (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1] = a0;
249           }
250           if ((M % THREADBLOCK_TILE_M == 0 ||
251            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
252            && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
```

```
253             C[split_K_OFFSET +
254             (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2] = a1;
255           }
256           if ((M % THREADBLOCK_TILE_M == 0 ||
257            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
258             && (N % THREADBLOCK_TILE_N == 0 || global_j_3 < N)) {
259             C[split_K_OFFSET +
260             (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_3] = a2;
261           }
262           if ((M % THREADBLOCK_TILE_M == 0 ||
263           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
264            && (N % THREADBLOCK_TILE_N == 0 || global_j_4 < N)) {
265             C[split_K_OFFSET +
266             (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_4] = a3;
267           }
268
269         } else {
270           if ((M % THREADBLOCK_TILE_M == 0 ||
271           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
272            && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
273             atomicAdd(&C[(global_i +
274              0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1], a0);
275           }
276           if ((M % THREADBLOCK_TILE_M == 0 ||
277           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
278            && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
279             atomicAdd(&C[(global_i +
280             0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2], a1);
281           }
282           if ((M % THREADBLOCK_TILE_M == 0 ||
283            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
284             && (N % THREADBLOCK_TILE_N == 0 || global_j_3 < N)) {
285             atomicAdd(&C[(global_i +
286             0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_3], a2);
287           }
288           if ((M % THREADBLOCK_TILE_M == 0 ||
289            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
290             && (N % THREADBLOCK_TILE_N == 0 || global_j_4 < N)) {
291             atomicAdd(&C[(global_i +
292              0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_4], a3);
293           }
294         }
295
296       }
297
298 #if __CUDA_ARCH__ >= 700
299       __syncwarp();
300 #endif
301
302     }
303   }
304   {
305
306     constexpr int EPILOGUE_N = N_THREADS * 2;
```

```
307
308     const int epilogue_i_write = LaneIdy;
309     const int epilogue_j_write = LaneIdx * 2;
310
311 // M_THREADS: 1, EPILOGUE_OFFSET: 0
312 // M_THREADS: 2, EPILOGUE_OFFSET: 1
313 // M_THREADS: 4, EPILOGUE_OFFSET: 8
314 // M_THREADS: 8, EPILOGUE_OFFSET: 4
315 // M_THREADS: 16, EPILOGUE_OFFSET:0
316 // M_THREADS: 32, EPILOGUE_OFFSET: 1
317     constexpr int EPILOGUE_OFFSET = (M_THREADS == 1 || M_THREADS == 16) ? 0 :
318                     (M_THREADS == 2 || M_THREADS == 32) ? 1 :
319                     (M_THREADS == 4) ? 8 : 4;
320 //    constexpr int EPILOGUE_OFFSET = 5;
321
322     if (THREAD_TILE_N % 4 >= 2) {
323
324       const int Threadtile_j = N_TIMES * 4;
325
326       const TYPE* a_ptr = &Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
327       VECTORTYPE2 a_val = reinterpret_cast<const VECTORTYPE2*>(a_ptr)[0];
328
329       TYPE* shared_ptr = &(*Shared)[SHARED_MEM_PER_WARP * WarpId +
330       epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilogue_j_write];
331       reinterpret_cast<VECTORTYPE2*>(shared_ptr)[0] = a_val;
332
333 #if __CUDA_ARCH__ >= 700
334       __syncwarp();
335 #endif
336
337       if (EPILOGUE_N <= 32) {
338
339         const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
340         const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
341
342         // Load the values
343         const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
344         (threadIdIdy_epilogue * 2 + 0) *
345         (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue];
346         const TYPE a1 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
347         (threadIdIdy_epilogue * 2 + 1) *
348         (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue];
349
350         const int global_i = global_i_upleft +
351          threadIdIdy_epilogue * (2 * THREAD_TILE_Y_HEIGHT);
352         const int global_j = block_idx_x * THREADBLOCK_TILE_N +
353          WarpIdx * WARP_TILE_N + threadIdIdx_epilogue + N_TIMES * N_THREADS * 4;
354
355         // Store the values into C
356
357         if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
358           if ((M % THREADBLOCK_TILE_M == 0 ||
359            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
360             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
```

```
361             C[split_K_OFFSET +
362             (global_i + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a0;
363           }
364           if ((M % THREADBLOCK_TILE_M == 0 ||
365            (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
366             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
367             C[split_K_OFFSET +
368             (global_i + 1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a1;
369           }
370
371         } else {
372           if ((M % THREADBLOCK_TILE_M == 0 ||
373           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
374           && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
375             atomicAdd(&C[(global_i +
376             0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a0);
377           }
378           if ((M % THREADBLOCK_TILE_M == 0 ||
379            (global_i + 1 * THREAD_TILE_Y_HEIGHT) < M)
380             && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
381             atomicAdd(&C[(global_i +
382             1 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a1);
383           }
384
385         }
386
387       } else if (EPILOGUE_N == 64) {
388
389         const int threadIdIdx_epilogue_1 = threadId;
390         const int threadIdIdx_epilogue_2 = threadId + 32;
391
392         // Load the values
393         const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
394         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_1];
395         const TYPE a2 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
396         0 * (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue_2];
397
398         const int global_i = global_i_upleft;
399
400         const int global_j_1 = block_idx_x * THREADBLOCK_TILE_N +
401         WarpIdx * WARP_TILE_N + N_TIMES * N_THREADS * 44 + threadIdIdx_epilogue_1;
402         const int global_j_2 = block_idx_x * THREADBLOCK_TILE_N +
403          WarpIdx * WARP_TILE_N + N_TIMES * N_THREADS * 4 + threadIdIdx_epilogue_2;
404
405         // Store the values into C
406         if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
407           if ((M % THREADBLOCK_TILE_M == 0 ||
408            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
409            && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
410             C[split_K_OFFSET + (global_i
411             + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1] = a0;
412           }
413           if ((M % THREADBLOCK_TILE_M == 0 ||
414           (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
```

```
415             && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
416              C[split_K_OFFSET + (global_i
417              + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2] = a2;
418            }
419          } else {
420            if ((M % THREADBLOCK_TILE_M == 0 ||
421            (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
422             && (N % THREADBLOCK_TILE_N == 0 || global_j_1 < N)) {
423              atomicAdd(&C[(global_i +
424              0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_1], a0);
425            }
426            if ((M % THREADBLOCK_TILE_M == 0 ||
427             (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
428              && (N % THREADBLOCK_TILE_N == 0 || global_j_2 < N)) {
429              atomicAdd(&C[(global_i +
430              0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j_2], a2);
431            }
432          }
433
434        }
435
436      }
437  #if __CUDA_ARCH__ >= 700
438      __syncwarp();
439  #endif
440    }
441  //
442    if (THREAD_TILE_N % 2 > 0) {
443
444      constexpr int EPILOGUE_N = N_THREADS;
445
446      const int epilogue_i_write = LaneIdy;
447      const int epilogue_j_write = LaneIdx;
448
449  // M_THREADS: 1, EPILOGUE_OFFSET: 0
450  // M_THREADS: 2, EPILOGUE_OFFSET: 0
451  // M_THREADS: 4, EPILOGUE_OFFSET: 22
452  // M_THREADS: 8, EPILOGUE_OFFSET: 0
453  // M_THREADS: 16, EPILOGUE_OFFSET:0
454  // M_THREADS: 32, EPILOGUE_OFFSET: 0
455      constexpr int EPILOGUE_OFFSET = (M_THREADS == 4) ? 22 : 0;
456
457      constexpr int ADDITIONAL_OFFSET_REGISTER = (THREAD_TILE_N % 4 >= 2) ? 2 : 0;
458
459      const int Threadtile_j = N_TIMES * 4 + ADDITIONAL_OFFSET_REGISTER;
460
461      const TYPE a_val = Thread_Tile[Threadtile_i * THREAD_TILE_N + Threadtile_j];
462
463      (*Shared)[SHARED_MEM_PER_WARP * WarpId + epilogue_i_write * (EPILOGUE_N + EPILOGUE_OFFSET) + epilog
464
465  #if __CUDA_ARCH__ >= 700
466      __syncwarp();
467  #endif
468
```

```
469     const int threadIdIdx_epilogue = threadId % EPILOGUE_N;
470     const int threadIdIdy_epilogue = threadId / EPILOGUE_N;
471
472     const TYPE a0 = (*Shared)[SHARED_MEM_PER_WARP * WarpId +
473     (threadIdIdy_epilogue * 1 + 0) *
474     (EPILOGUE_N + EPILOGUE_OFFSET) + threadIdIdx_epilogue];
475
476     constexpr int ADDITIONAL_OFFSET_GLOBAL =
477     (THREAD_TILE_N % 4 >= 2) ? N_THREADS * 2 : 0;
478
479     const int global_i = global_i_upleft +
480     threadIdIdy_epilogue * (1 * THREAD_TILE_Y_HEIGHT);
481     const int global_j = block_idx_x * THREADBLOCK_TILE_N +
482     WarpIdx * WARP_TILE_N + threadIdIdx_epilogue +
483     N_TIMES * N_THREADS * 4
484         + ADDITIONAL_OFFSET_GLOBAL;
485
486     if (SPLIT_K == 1 || !ATOMIC_REDUCTION) {
487       if ((M % THREADBLOCK_TILE_M == 0 ||
488        (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
489        && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
490        C[split_K_OFFSET + (global_i
491        + 0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j] = a0;
492       }
493     } else {
494       if ((M % THREADBLOCK_TILE_M == 0 ||
495        (global_i + 0 * THREAD_TILE_Y_HEIGHT) < M)
496        && (N % THREADBLOCK_TILE_N == 0 || global_j < N)) {
497        atomicAdd(&C[(global_i +
498        0 * THREAD_TILE_Y_HEIGHT) * ldc + global_j], a0);
499       }
500     }
501 #if __CUDA_ARCH__ >= 700
502     __syncwarp();
503 #endif
504
505   }
506 }
```

Listing A.33: store_C_OneRow_Single

A.6.3 SplitKReduction

Table A.26: cosmaSplitKReduce Parameters

C	The orignial C matrix
ldc	leading dimension of a two-dimensional array used to store the matrix C.
C_SPLIT_K	The intermediate storage to perform the split K reduction

```
1 void cosmaSplitKReduce(TYPE * __restrict__ C, const int ldc,
2  TYPE * __restrict__ C_SPLIT_K) {
3
```

```
4     int threads = std::min(256, M * N);
5     int blocks = (M * N + threads - 1) / threads;
6
7     cosmaSplitKReduce_Kernel<<<blocks, threads>>>(C, ldc,C_SPLIT_K);
8
9   }
```

Listing A.34: cosmaSplitKReduce

Table A.27: cosmaSplitKReduce Kernel Parameters

C	The orignial C matrix
ldc	leading dimension of a two-dimensional array used to store the matrix C.
C_SPLIT_K	The intermediate storage to perform the split K reduction

```
1   __global__ void cosmaSplitKReduce_Kernel(TYPE * __restrict__ C,
2   const int ldc, TYPE * __restrict__ C_SPLIT_K) {
3
4     const int id = blockIdx.x * blockDim.x + threadIdx.x;
5
6     if (id >= M * N) {
7       return;
8     }
9
10    TYPE val = 0;
11
12  #pragma unroll
13    for (int z = 0; z < SPLIT_K; z++) {
14      val += C_SPLIT_K[z * M * N + id];
15    }
16
17    const int x = id % N;
18    const int y = id / N;
19
20    TYPE c_val = C[y * ldc + x];
21
22    C[y * ldc + x] = BETA * c_val + ALPHA * val;
23
24  }
```

Listing A.35: cosmaSplitKReduce Kernel

A.6.4 Sigmoid Kernel

There is a trade-off to be made. In order to perform the matrix multiplication as fast as possible, it is worth to pad each row accordingly for coalesced loading and storing. This, however, can make the kernel that performs the element-wise function, less efficient. The kernel presented is for the case that the rows do not have to be padded. We performed some measurement and and verified that this kernel works best launched with 256 threads per

thread block and 400 thread blocks (<<<*256,400*>>>) on a V100 for large matrices.

Table A.28: sigmoid_kernel Parameters

array	The original C matrix

```
__global__ void sigmoid_kernel(float * __restrict__ array) {
  const int stride = gridDim.x * blockDim.x;
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
#pragma unroll
  for (int i = tid; i < M * N / 4; i += stride) {
    float4 a = reinterpret_cast<float4*>(array)[i];

    a.x = 1.0f / (1.0f + expf(-a.x));
    a.y = 1.0f / (1.0f + expf(-a.y));
    a.z = 1.0f / (1.0f + expf(-a.z));
    a.w = 1.0f / (1.0f + expf(-a.w));

    reinterpret_cast<float4*>(array)[i] = a;
  }
}
```

Listing A.36: Sigmoid Activation

List of Figures

List of Tables

List of Listings

Bibliography

[1] cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.

[2] cublasGemmAlgo_t. https://docs.nvidia.com/cuda/cublas/index.html#cublasgemmalgo_t.

[3] cublasGemmEx. https://docs.nvidia.com/cuda/cublas/index.html#cublas-GemmEx.

[4] cuBLASLt API. https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasLt-api.

[5] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] CUDA Profiling. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[7] cudaGetDeviceProperties. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g1bf9d625a931d657e08db2b4391170f0.

[8] CUTLASS. https://github.com/NVIDIA/cutlass.

[9] CUTLASS: Fast Linear Algebra in CUDA C++. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

[10] CUTLASS Threadblock Swizzle. https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/gemm/threadblock/threadblock_swizzle.h.

[11] Features and Technical Specifications. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

[12] High Bandwidth Memory. https://de.wikipedia.org/wiki/High_Bandwidth_Memory.

[13] NervanaSystems SGEMM. https://github.com/NervanaSystems/maxas/wiki/SGEMM#reading-from-shared.

[14] NVIDIA CUDA Compiler. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[15] NVIDIA RTX 2080 Ti. https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti/.

[16] NVIDIA Tesla A100. https://www.nvidia.com/de-de/data-center/a100/.

[17] NVIDIA Tesla C1060. https://www.techpowerup.com/gpu-specs/tesla-c1060.c1539.

[18] NVIDIA Tesla C870. https://www.techpowerup.com/gpu-specs/tesla-c870.c1542.

[19] NVIDIA Tesla K40. https://www.microway.com/hpc-tech-tips/nvidia-tesla-k40-atlas-gpu-accelerator-kepler-gk110b-up-close/.

[20] NVIDIA Tesla K40. https://www.gpuzoo.com/GPU-NVIDIA/Tesla_K40.html.

[21] NVIDIA Tesla M2090. https://www.techpowerup.com/gpu-specs/tesla-m2090.c1537.

[22] NVIDIA Tesla M40. https://www.techpowerup.com/gpu-specs/tesla-m40.c2771.

[23] NVIDIA Tesla P100. https://www.nvidia.com/de-de/data-center/tesla-p100/.

[24] NVIDIA Tesla V100. https://www.nvidia.com/de-de/data-center/tesla-v100/.

[25] Rectifier. https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[26] rocBLAS. https://github.com/ROCmSoftwarePlatform/rocBLAS.

[27] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, USA, 1969. AAI7010025.

[28] Jaeyoung Choi, David W. Walker, and Jack J. Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[29] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, 10 2012.

[30] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 326–333, New York, NY, USA, 1981. Association for Computing Machinery.

[31] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019. `http://spcl.inf.ethz.ch/Publications/.pdf/mmm-tr.pdf`.

[32] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[33] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, USA, 1995.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

cuCOSMA: Near Optimal Matrix-Matrix Multiplication in CUDA

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Walo	Neville

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
01.07.2020	N.Walo

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.