Gigacheck

Emanuele Messina, Francesco Risso

2025

Abstract

In this project, we developed and implemented a fault-tolerant matrix multiplication algorithm capable of handling matrices of arbitrary sizes on a GPU. The algorithm incorporates an error detection and correction technique to ensure computational accuracy in the presence of errors, and different buffering strategies were designed to manage memory constraints by splitting large matrices into manageable chunks that fit within GPU memory. We wrote a self-contained benchmark program to evaluate the algorithm performance when varying settings such as input size, available memory and buffering strategies.

Contents

1 Introduction

Matrix multiplication is a fundamental operation in various scientific, engineering, and data-driven domains, including deep learning, scientific simulations, and aerospace systems. These fields often rely on high-performance hardware accelerators, such as GPUs, to handle computationally intensive tasks. However, the complexity and high utilization of modern GPU environments make them susceptible to errors caused by hardware faults, thermal fluctuations, and cosmic rays, particularly in aerospace applications where hardware is exposed to extreme environments. Fault-tolerant computation strategies are critical in such contexts, as undetected errors in matrix operations can lead to model mispredictions in neural networks or potentially catastrophic failures in mission-critical aerospace systems. In this report, we present our implementation of a fault-tolerant matrix multiplication algorithm on CUDA, capable of handling matrices whose size would not allow a single-shot computation of the multiplication inside the GPU, as it would require an excess with respect to the available global memory.

This paper is structured as follows:

- Section 2: background theory regarding matrix multiplication and ABFT.
- Section 3: overview of our algorithm to see how it works in the bigger picture, before going into the details in the subsequent sections.
- Section 4: explanation of how we perform the block level product, error detection, and correction.
- Section 5: illustration of our buffering strategies when dealing with large matrices.
- Section 6: discussion of the performance differences when varying parameters.

2 Background Theory

2.1 ABFT matrix multiplication

The basic idea behind Algorithm-Based Fault Tolerant (ABFT) matrix multiplication is to use checksums to verify the correctness of the computation. The following paragraphs illustrate the steps we perform:

2.1.1 Checksum Calculation

Let A be an $m \times n$ matrix, B be an $n \times q$ matrix, and C be the resulting $m \times q$ matrix from the multiplication $C = AB$. Before performing the matrix multiplication, we calculate the column checksum of A and the row checksum of B:

$$\text{Column checksum of } A: \quad \mathbf{c}_A = \left[\sum_{i=1}^{n} a_{i1}, \sum_{i=1}^{n} a_{i2}, \ldots, \sum_{i=1}^{n} a_{in} \right]$$

$$\text{Row checksum of } B: \quad \mathbf{r}_B = \left[\sum_{j=1}^{n} b_{1j}, \sum_{j=1}^{n} b_{2j}, \ldots, \sum_{j=1}^{n} b_{nj} \right]^T$$

The row checksum of a matrix is a column vector where each element is the sum of the elements in the corresponding row of the matrix. Similarly, the column checksum is a row vector where each element is the sum of the elements in the corresponding column of the matrix.

2.1.2 Augmented Matrices

We then create augmented matrices A_c $(m+1 \times n)$ and B_r $(n \times q+1)$ by appending the column and row checksum vectors, respectively, to the original matrices: Thus we get:

$$A_c = \left[\begin{array}{c} A \\ \hline \mathbf{c}_A \end{array}\right] \quad , \quad B_r = \left[\begin{array}{c|c} B & \mathbf{r}_B \end{array}\right]$$

2.1.3 Multiplication

We multiply A_c and B_r together, yielding

$$C_{cr} = A_c B_r = \left[\begin{array}{c|c} AB & A\,\mathbf{r}_B \\ \hline \mathbf{c}_A\,B & \mathbf{c}_A\,\mathbf{r}_B \end{array}\right]$$

We see that the original product is preserved in the upper left block, while the other blocks contain checksum information.

2.1.4 Checksum Verification

Considering the column and row checksums of the upper left block of the resulting matrix C_{cr}, the following properties hold:

$$\mathbf{c}_{AB} = [\mathbf{c}_A\,B] \quad , \quad \mathbf{r}_{AB} = [A\,\mathbf{r}_B]$$

The proof is trivial.

This is the property that we ultimately exploit to correct errors in the computation, because if the upper block returned to us is corrupted, then we can compute its checksums (we will call them control checksums) and compare them against the peripheral blocks to at least detect the corruption of the result.

In general, the column checksum of the upper blocks of the augmented result is equal to the last row of the augmented result, and the row cheksum of the left blocks of the augmented result is equal to the last column the augmented result. In formulas:

$$\mathbf{c}_{\text{control}} = \mathbf{c}\begin{array}{c|c} AB & A\,\mathbf{r}_B \end{array} = \left[\begin{array}{c|c} \mathbf{c}_A\,B & \mathbf{c}_A\,\mathbf{r}_B \end{array}\right]$$

$$\mathbf{r}_{\text{control}} = \mathbf{r}\begin{array}{c} AB \\ \hline \mathbf{c}_A\,B \end{array} = \left[\begin{array}{c} A\,\mathbf{r}_B \\ \hline \mathbf{c}_A\,\mathbf{r}_B \end{array}\right]$$

As we can see, the last element of each control checksum, is the same, and also corresponds exactly to $C_{cr}\,[m+1, q+1]$. This element is the one that allows us to detect errors in the checksum blocks themselves, generalizing the approach to

the entire augmented result matrix and not just the upper left block containing the original product.

Wherever there is a mismatch between the returned checksum blocks and the associated computed control checksum, we know: first, that there is an error in C_{cr}; and second, one coordinate of the error.

In the case of a single error inside C_{cr}, there would be a single mismatch in both $\mathbf{r}_{\text{control}}$ and $\mathbf{c}_{\text{control}}$. The item indexes on the control checksum vectors give the coordinates of the error inside the result matrix.

If there are multiple errors, they can be collinear or not. By collinear errors we mean errors that share one coordinate, or equivalently stated, they are arranged on the same column or row in the result matrix. Collinear errors can be individually detected and isolated.

The presence of at least two non collinear errors can only be detected, but the exact coordinates of the individual errors cannot be obtained.

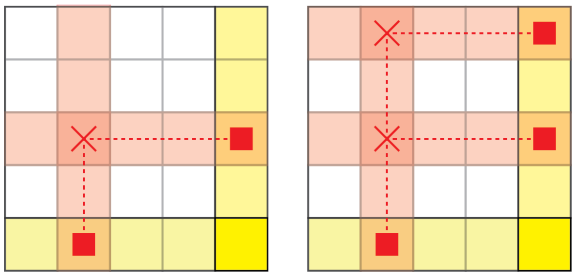


Figure 1: detectable errors cause exactly one mismatch in at least one of the control checksums.

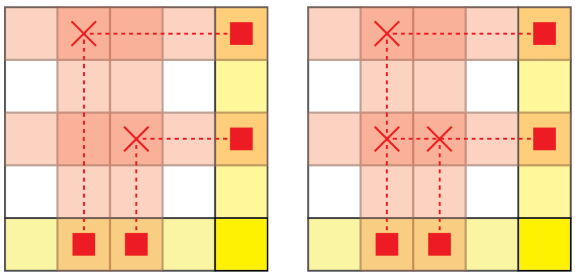


Figure 2: an error set that causes more than one mismatch in both checksums does not allow us to recover the individual error placement inside the matrix, since the configuration is ambiguous.

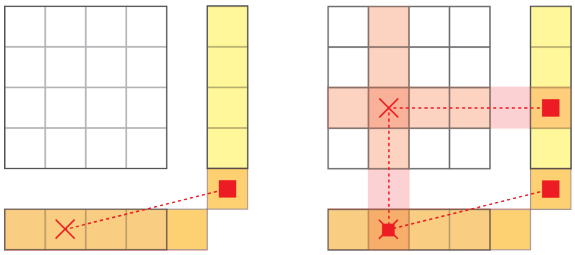


Figure 3: errors in the checksum blocks themselves cause a mismatch in the shared item (the last item), and so can be detected as any other detectable error.

2.1.5 Error Correction

We can only correct detectable errors, i.e. the ones we can isolate from the coordinates obtained by the checksum mismatches.

For a given error C_e of coordinates (i, j) in $C^* := C_{cr}$, we must choose one of the two control checksums and the associated checksum block in C_{cr} to recover the original result value. To do this, we simply enstablish along which axis the errors are collinear, and use the checksum of the opposite axis (the one which contains more than one mismatch) to compute the correction values. For example, if the errors are arranged on a column, the column control checksum will contain a single mismatch while the row control checksum will contain as many mismatches as the error count, thus we have to use the row checksum because it contains distinct correction values for each error. Of course, if there is only a single error present, one control checksum is as good as the other. We can discard errors in the checksum blocks as they don't corrupt the original multiplication result. Actually, though, when using buffering strategies as discussed in Section 5, we need the checksum blocks to be intact. Thus, if we detect errors in the checksum blocks, we can either correct them or we can recalculate them after correcting the errors in the result block. In our implementation we recalculate them for simplicity.

For column-collinear errors, the correction formula is:

$$C_{i,j} = C^*_{i,q+1} - \mathbf{r}_{\text{control}_i} + C_e$$

For row collinear errors, the correction formula is:

$$C_{i,j} = C^*_{m+1,j} - \mathbf{c}_{\text{control}_j} + C_e$$

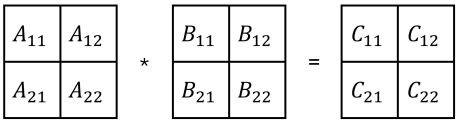
These formulas are quite easy to derive by solving a system of two equations: the equality between a control checksum and the associated checksum block in C^* when no error is present, and the actual control checksum computation equation where the error C_e shows up.

2.2 Strassen algorithm

Strassen algorithm is an alternative to the traditional algorithm for matrix multiplications. It leverages some mathematical properties, to reduce the number of products required to compute a given matrix product.

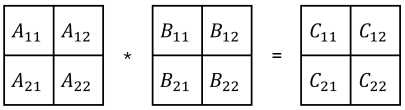
Although it is usually not described like this, the traditional algorithm can be seen as a series of products of submatrices. Figure 4 shows that, when splitting the operands into 4 blocks, 8 multiplications are required. Instead, Strassen algorithm is able to reduce this number to 7 (as visible in figure 5), switching the removed one with a series of less computational intensive sums.

At the beginning we believed that this algorithm could be useful to speed up our program, but then for multiple reasons described below we decided not to use it.



$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$
$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$
$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$
$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

Figure 4: a product of 2x2 matrices with the traditional algorithm



$$M_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) * B_{11}$$
$$M_3 = A_{11} * (B_{12} - B_{22})$$
$$M_4 = A_{22} * (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) * B_{22}$$
$$M_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Figure 5: A product of 2x2 matrices with the Strassen algorithm

2.3 CUDA queues

On top of the parallelization of the threads, CUDA also allows to parallelize different types of operations. In particular, it defines three queues:

- H2D: the queue where the memory transfers from host to device happen
- compute: where all threads are scheduled
- D2H: where the GPU schedules the memory transfers from device to host

If the code is well written, CUDA allows different operations to run concurrently, if they belong to different queues. Ideally, this means for example that a kernel, a `cudaMemcpyHostToDevice`, and a `cudaMemcpyDeviceToHost` could all run concurrently. On top of this, the compute queue can also allow multiple kernel being executed concurrently, if they belong to different streams and hardware resources are enough. Instead, the H2D and D2H queues can only have a single data transfer each at the same time, regardless of streams.

This is however limited by the restrictions imposed by the scheduler:

- An operation is only called if all the previous operations in the same stream are complete
- An operation is only called if all the previous ones in the same queue have been completed
- A blocked operation in a queue blocks all subsequent operations, even if they belong to different streams

- The signal that notifies from the compute queue to the D2H queue that a kernel is finished is delayed to when all sequential threads are finished

This means that calling operations in a breadth-first approach (eg. $kernel_{stream1} \rightarrow kernel_{stream2} \rightarrow D2H_{stream1} \rightarrow D2H_{stream2}$) may lead to $D2H_{stream1}$ having to wait for the completion of $kernel_{stream2}$ before starting. If instead operations are called with a depth-first approach (eg. $kernel_{stream1} \rightarrow D2H_{stream1} \rightarrow kernel_{stream2} \rightarrow D2H_{stream2}$), then the risk does not exist.

In our project, we paid attention to always schedule operations in a depth-first approach.

2.4 Tiled matrix multiplication

The tiled matrix multiplication is an algorithm that implements the traditional matrix multiplication algorithm, by exploiting the shared memory of a GPU to speed up the process.

2.4.1 Without tiling

The standard algorithm without tiling would have independent threads that proceed as follows, for each index k required to compute C_{rc}:

- Load A_{rk} from global memory
- Load B_{kc} from global memory
- Compute $A_{rk} \cdot B_{kc}$
- Add it to a local variable
- When the product is finished, store it to C_{rc}

When multiplying a matrix $m \times n$ by a matrix $n \times q$, this approach requires $2 \cdot m \cdot n \cdot q$ loads from global memory.

2.4.2 With tiling

With the tiling approach, we define *tile* a square region of $T \times T$ cells. Both the result matrix and the operands are split into tiles of the same size. A tile of the result matrix is computed by a block of threads, leveraging shared memory. Different tiles of the result matrix are however computed independently.

Among the thread block C_{RC}, the thread that has to compute C_{rc} (r and c are indices local to the tile) performs the following for each tile t necessary for the computation:

- Loads from global memory to shared memory the value A_{rc} of the tile A_{Rt}
- Loads from global memory to shared memory the value B_{rc} of the tile B_{tC}
- Waits for the other threads to load all their values to shared memory
- Computes and sums to a local variable $A_{rk} \cdot B_{kc}$ for k going from 0 to T (excluded), reading the values from the shared memory
- When the product is finished, stores it to C_{rc}

This way, the number of global memory reads is reduced by a factor of T, thanks to the fact that each thread in the block only loads 2 values per tile. It then relies on the other threads to load the other $2 \cdot (T - 1)$ that it needs.

3 Overview of the Algorithm

Given the matrices A and B, our program processes them via to the following steps:

- Choose if and how they need to be split
- For each block of the final matrix:
  - Set C to a matrix of zeros
  - For each couple of blocks from A and B that is required to compute the current block of C:
    - Load the block of A
    - Load the block of B
    - Add AB to the current C
    - Add errors to C
    - Find and correct the errors on C
  - Unload C to the RAM

All the steps are described more in details in the following paragraphs.

4 Block Product, Error Detection and Correction

4.1 Computing checksums

The checksums are computed by a single kernel, that based on a parameter computes row or column checksums.

For row checksums, a thread block is generated for each row, with a set of threads that initially sum a portion of the row, in a linear way. After that, the different threads of the block organize themselves to sum all the partial sums in a dichotomous way.

Similarly, column checksums generate a thread block for each column, performing the same operations in the orthogonal direction.

After the computation, the kernel is able to store the checksum either in the last row or column of the matrix, or into a separate vector, in order to avoid overriding the previous checksums (the control checksums of C).

4.2 Computing the product

When the matrices are small enough to fit on GPU (or we are working on blocks), the multiplication is realized by means of the standard tiled algorithm for GPU.

Initially, we believed that we could apply one or two recursions of Strassen algorithm to reduce the number of products. Then, we realized that this would make the same error appear in different submatrices: this would make it much harder to correct it, since non-collinear errors are not correctable. For this reason, we decided to drop Strassen algorithm at this smaller size level.

```
# get_element: matrix[blockID, index] or viceversa based on direction

sum = 0

for index in range(matrix_size, step=threads_per_block):
    sum += get_element(index)

shared_mem[threadID] = sum
_sync()

log_reduction()

if(threadID == 0):
    store_result(shared_mem[0])
```

Figure 6: the pseudocode of the kernel that computes the checksums

```
val = 0

for tile in num_tiles:
    shared_A[threadY, threadX] =
        load_value(A, (blockY, tile), (threadY, threadX))

    shared_B[threadY, threadX] =
        load_value(B, (tile, blockX), (threadY, threadX))

    _sync()

    for el in tile_size:
        val += shared_A[threadY, el] * shared_B[el, threadX]

    _sync()

store_add(C, (blockY, blockX), (threadY, threadX), val)
```

Figure 7: the pseudocode of the kernel that computes the matrix multiplication using the tiled approach

4.3 Error addition

In order to demonstrate the error correction capability, some errors can be added after the computation of the product. The number of errors per product can be set by the user, as well as if they must be collinear or not.

The program randomly chooses a set of distinct elements where to add errors, and adds a random delta to the chosen element.

4.4 Error correction

After computing the product and adding the errors, checksums are computed again on the result, both on the rows and the columns. A kernel is then run to find mismatches between the control checksums (the ones obtained as the sum of rows or columns of C) and the ones in C (obtained from the multiplication).

The mismatches are then compared to check if errors are present, and if they are collinear (otherwhise, they are not correctable). If the errors are correctable, they

are corrected by using the control checksums. If they are not correctable, or there are errors on the checksums, then the caller is notified.

```
errs_r = find_mismatches("row checksums", C, control_checksums_row)
errs_c = find_mismatches("col checksums", C, control_checksums_col)

if((len(errs_r) == 0) and (len(errs_c) == 0)):
    return "no errors"

if((len(errs_r) > 1) and (len(errs_c) > 1)):
    return "uncorrectable errors"

for err in (errs_r, errs_c):
    if(err is in checksum):
        notify_recmompute_checksums()
        continue

    correct_error()

return "corrected errors"
```

Figure 8: the pseudocode of the function that detects and corrects the errors (`find_mismatches` is a GPU kernel, while the rest is executed on CPU)

5 Buffering Strategies for Large Matrices

Whenever the matrices are too large for all the allocations to fit on GPU, then they are split into blocks. The algorithm can choose two values:

- `num_split_common_dim`, the number of splits for A's rows and B's columns. When splitting in this direction, a single value of C will be the sum of this amount of products among blocks of A and B.
- `num_split_other_dim`, the number of splits for A's columns and B's rows. This will result in C being made of a square grid of blocks, with this value as edge. The different blocks will be independent from each other.

An example of matrix splitting is provided in figure 9.

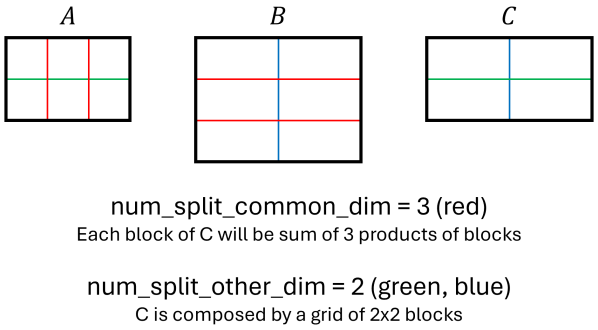


Figure 9: an example of how matrices A and B can be split

In order to choose how to split, the algorithm computes the required memory, and compares it with the available memory. If the available memory is smaller by a factor k, then:

- `num_split_common_dim` is assigned the value $\sqrt{k}$.
- `num_split_other_dim` is assigned the value $\lceil \frac{k}{num\_split\_other\_dim} \rceil$.

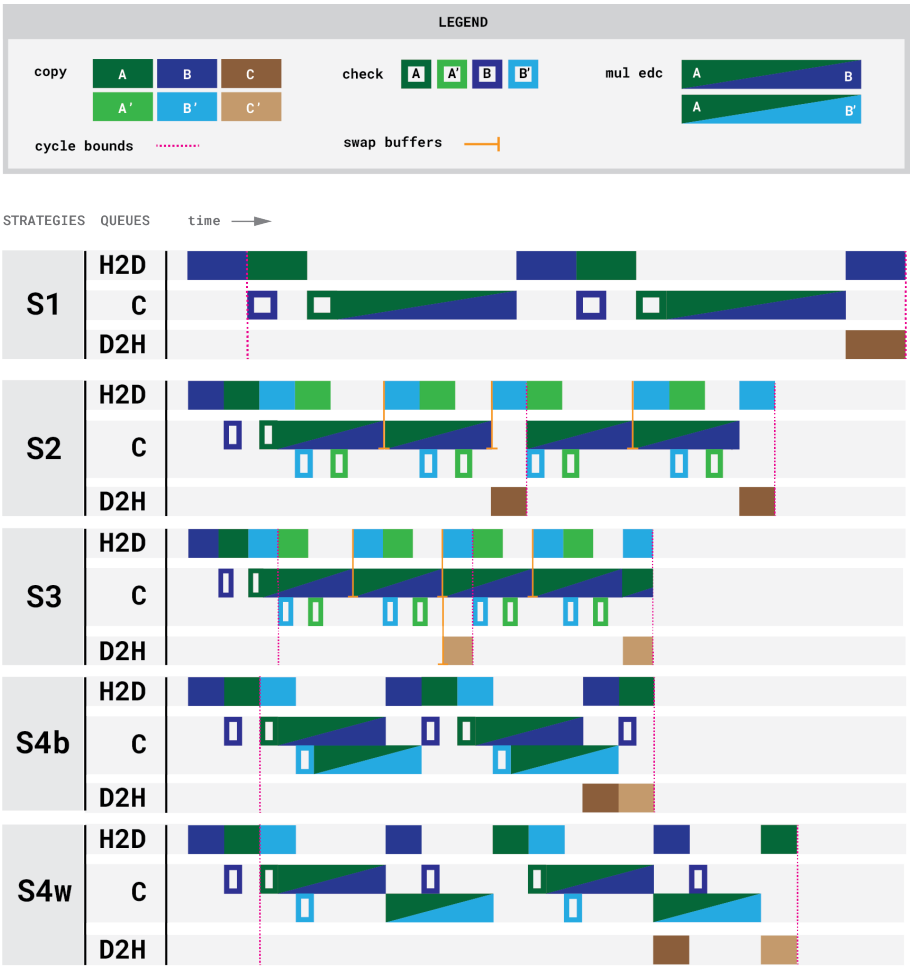


Figure 10: the timing diagram for our strategies, including the best (S4b) and worst (S4w) cases for strategy 4. Timings are scaled based on the block size reduction. The different number of blocks computed represents a computation of a similar number of elements, with S2 and S4 computing about 1.2 times what is computed by S1 and S3

5.1 Strategy 1: no buffering

This is the naive approach: the GPU memory is divided into 3 blocks: A, B and C. At every product, the iteration loads A and B in parallel, then sums the product into C. Whenever all the multiplications for a block of C are done, the block itself is unloaded to the CPU memory, while in the meantime the next A and B can start loading.

A timing diagram of this strategy can be seen in figure 10 (diagram S1).

```
for result_block in result_matrix:
    for tmp_block in num_split_common_dim:
        A = load_block()
        B = load_block()
        _sync()

        C = A*B
        _sync()

    store_block(C)
```

Figure 11: pseudocode of the algorithm for strategy 1

5.2 Strategy 2: double buffer for A and B

When using strategy 1, the multiplication requires A and B to be loaded: this means that while A and B are being transfered to GPU, no computation is being done (except for the checksums, that are however very fast). Similarly, loading the next A and B requires the product to be completed, thus having no memory transfer while computing the product. Effectively, this means that the three CUDA queues are mostly disjoint: the only overlaps are H2D and D2H when copying C (not at every iteration), and H2D and compute while calculating the checksums (for a very short time).

Strategy 2 aims to improve this situation, by dividing the GPU memory in two extra blocks (5 in total, called A, A', B, B' and C). Initially, A and B are pre-loaded. Then, before starting the product, A' and B' start to load the next block, in an asynchronous way. At the same time, C can compute the product on A and B. When both the new loading and the product are finished, the buffers are swapped: C can immediately start multiplying A' and B', while A and B load the next block. This allows to overlap the H2D and compute queues in the time when the "offline" A and B are loading.

Figure 10 (diagram S2) shows the timing relative to this strategy.

```
A = load_block()
B = load_block()
_sync()

for result_block in result_matrix:
    for tmp_block in num_split_common_dim:
        A_alt = load_block()
        B_alt = load_block()

        _sync("copy of C")
        C = A*B
        _sync()

        A = A_alt
        B = B_alt

    store_block(C)

```

Figure 12: pseudocode of the algorithm for strategy 2

5.3 Strategy 3: double buffer for A, B and C

This strategy brings to the extreme the idea behind strategy 2. In strategy 2, multiplications are still forbidden while C is offloading, to avoid overriding not-yet-saved results. Strategy 3 introduces a new buffer, called C', that works similarly to A' and B'. At the beginning, the product is computed on C. Then, when a full block has been computed, C is switched with C', to be able to immediately start the computation, while the offloading of C can go on in the background. This ideally allows the compute stream (the bottleneck of the complex task of matrix multiplications) to run continuously, regardless of the status of the transfer queues.

Differently from strategy 2, however, the gained parallelism only occurs once every `num_split_common_dim` iterations: this fact makes the speedup less impactful, with the risk of it being shadowed by the reduction in block sizes required to fit C' in the GPU memory, that can lead to more multiplications required.

Figure 10 (diagram S3) displays the timing diagram, in the optimal case when the addition of C' does not require more multiplications.

```
A = load_block()
B = load_block()
_sync()

for result_block in result_matrix:
    for tmp_block in num_split_common_dim:
        A_alt = load_block()
        B_alt = load_block()

        C = A*B
        _sync()

        A = A_alt
        B = B_alt

    C_alt = C
    store_block(C_alt)
```

Figure 13: pseudocode of the algorithm for strategy 3

5.4 Strategy 4: double concurrent product

Since the slowest part of the program is the actual computation of the product, we tried to execute more than one in parallel. Under the (uncertain) assumption that two multiplications can be done in parallel, this would really speed up the program. Moreover, if the assumption is false, this idea should not be slower than strategy 3.

Strategy 4 uses the buffers C and C' to compute a block of the final matrix into C, and the block to its right into C', if it exists. Using this method, the two products multiply the same block of A with shifted blocks of C: therefore, we can use just 3 buffers for the operands: A, B and B'. This increases the size of each buffer, and reduces the required memory copies.

For this final strategy we drew the timing diagram for the two cases, as visible in figure 10: if the parallel multiplications hypothesis holds (diagram S4b) or if it is wrong (S4w).

```
for result_block in result_matrix:
    for tmp_block in num_split_common_dim:
        B = load_block()
        A = load_block()
        B_alt = load_block()

        C = A*B
        C_alt = A*B_alt
        _sync()

    store_block(C)
    store_block(C_alt)
    _sync()
```

Figure 14: pseudocode of the algorithm for strategy 4

5.5 Strassen algorithm

We initially discussed about adding an option to use Strassen algorithm at this higher level. Ideally, it could be applied with any of the strategies described above. The problem of this algorithm is that it requires 7 temporary matrices, for which we saw two options: storing all them in GPU memory, or loading and unloading them at need.

In order to compare the two options among them and with the traditional algorithm, we considered the case described in figure 4, where A and B do not fit on GPU memory, but they fit if they are divided into 4 blocks.

When using Strassen algorithm, the product would be as described in figure 5.

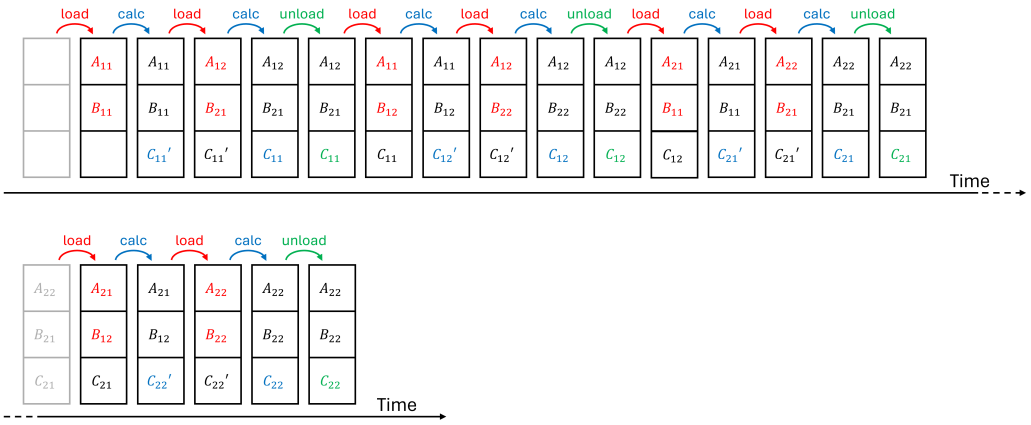


Figure 15: the series of load/calc/unload operations for a 2x2 product with the traditional algorithm. The colors indicate what has been loaded (red), computed (blue), offloaded (green) or left unchanged (black) from the previous step. An apex indicates that the result is partial.

As visible in figure 15, the traditional algorithm requires 20 memory transfers, to load the blocks of A and B, and to offload C when computed. This example assumes that the GPU is divided into 3 blocks of equal size (the 3 squares on top of each other, in the image) If we want to use Strassen algorithm with blocks of the same

size, figure 16 reveals that 45 memory transfers are required, because the program needs to offload temporary sums and matrices to make space for the other values. If instead we decide to have more blocks for the Strassen algorithm, in order to avoid offloading temporary results, we would need 21 memory transfers, as shown in figure 17. This removes one multiplication with respect to the traditional algorithm, at the cost of adding just an extra memory transfer. In principle it would be good, if not for the fact that this approach requires the blocks to have half the size, thus having on average $\sqrt{2}$ more blocks. That would then require more multiplications than what the Strassen algorithm saves, thus making its application not beneficial.

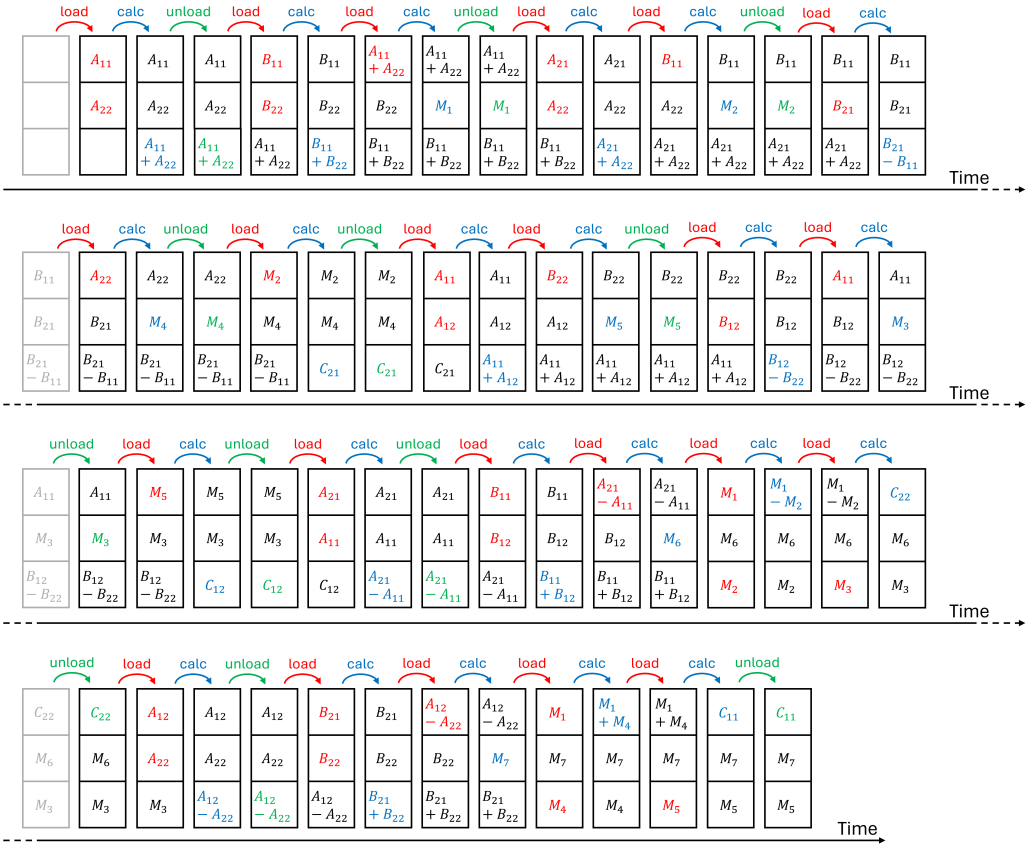


Figure 16: the series of load/calc/unload operations for a 2x2 product with Strassen algorithm, if we want to have the GPU memory divided in just 3 blocks

As a conclusion, we realized that Strassen algorithm was an efficient way to save computation time, but only if the full matrices were fully available on GPU, thus making memory transfers not needed.

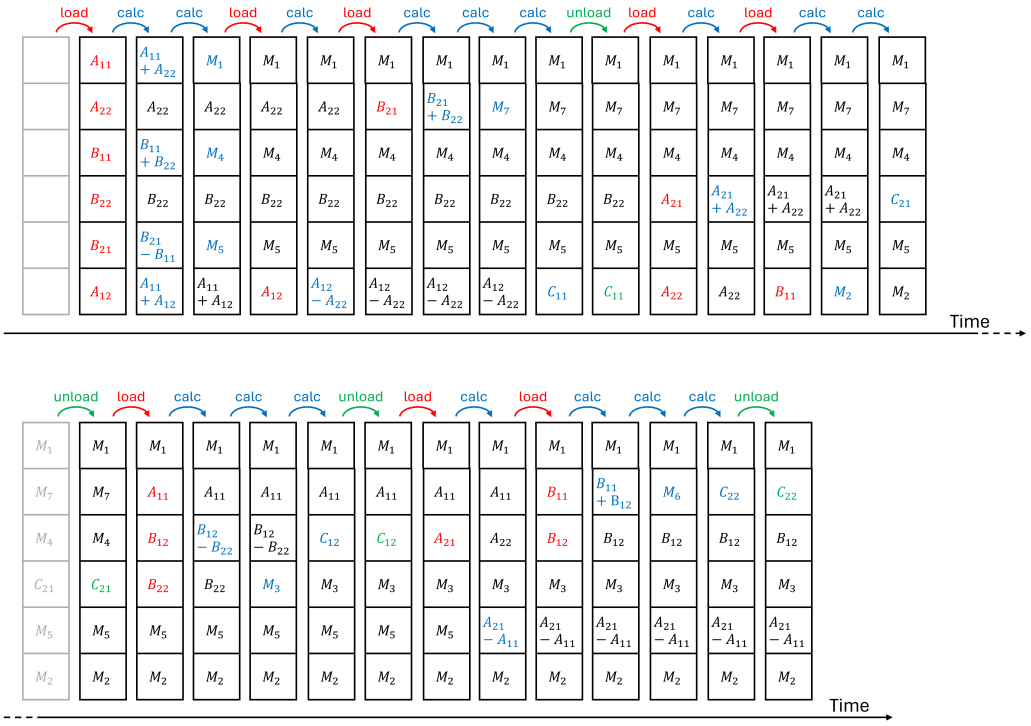


Figure 17: the series of load/calc/unload operations for a 2x2 product with Strassen algorithm, if we want to avoid offloading temporary results

6 Results

We wrote a simple CLI program to evaluate the algorithm under different conditions. The program allows to set the input dimensions, the number of errors to introduce and whether they should be collinear or not, the simulated maximum amount of global memory, and the buffering strategy to use. Several minor debugging parameters can be set, but they are out of the scope of this paper.

We tested the four strategies on a Quadro K620 desktop GPU. An important thing to consider is that display GPUs have a timeout on kernel executions, so we couldn't run tests with very large matrices. In any case, we believe that for bigger matrices the calculated performance should increase asymptotically.

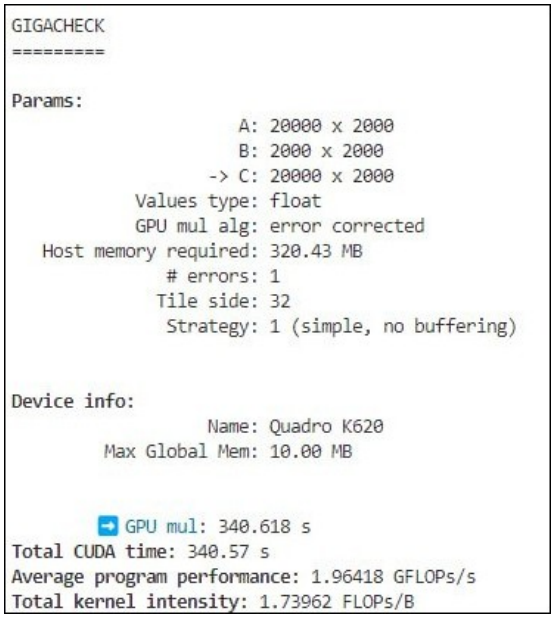
When it came to profiling, the Nsight Compute suite was only available for the newer Ampere architecture, and for this GPU (Maxwell architecture), nvprof didn't have support for performance counters. For this reason we embedded some sort of profiling in the program itself.

As performance metrics, we used the overall execution time of the program (both total CPU and CUDA only), the total floating point arithmetic intensity and the average program performance (calculated as the total floating point operations over the total CUDA time).

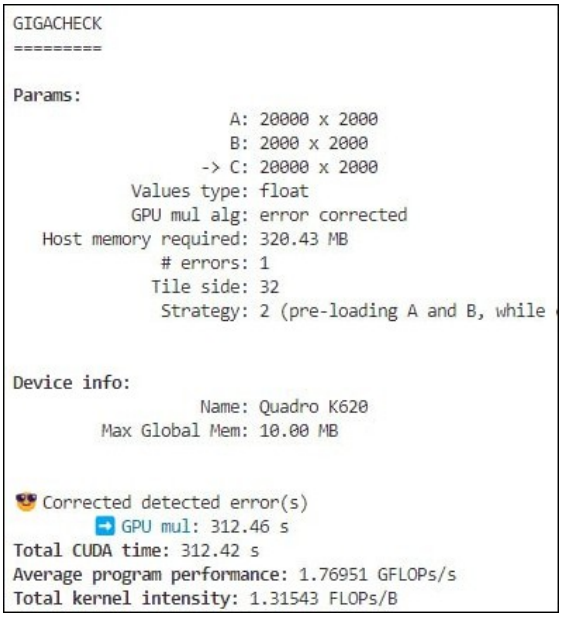
To calculate the total floating point operations and the associated transfers to and from global memory (to then calculate the arithmetic intensity), we manually calculated the floating point operations number and associated global memory trans-

fers count for each kernel we wrote. The program registers each kernel call and adds its float operations and transfers count to their respective counter.

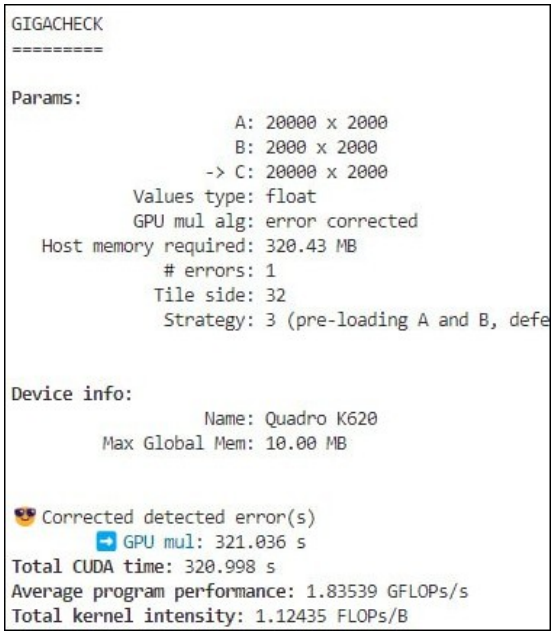
To calculate the effective compute time we use CUDA events to register the start and end of the program (getting the total CUDA time with respect to the default stream). To calculate the average performance we divide the total float operations by the total CUDA time, as a higher value indicate a higher use of the compute queue. To calculate the total intensity we divide the total float operations by the total transfer count.



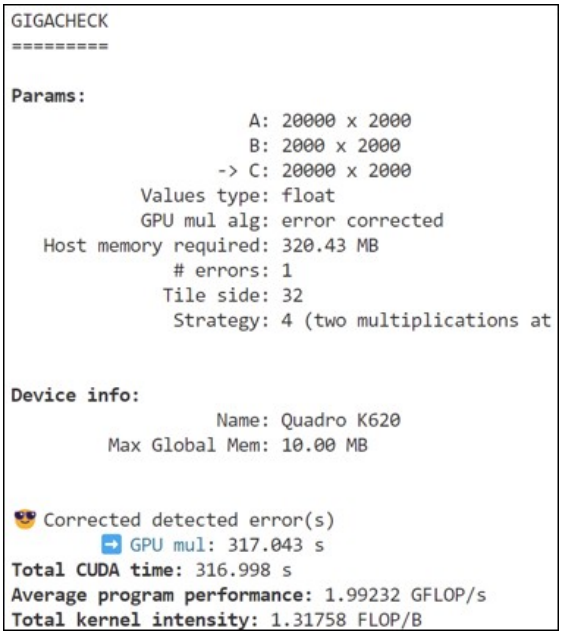
Strategy 1



Strategy 2



Strategy 3



Strategy 4

Figure 18: results for the different strategies

With that said, we hereby present our considerations on the obtained results.

We launched the same multiplication for the four different strategies, with inputs 20000×2000 and 2000×2000 (around 320MB of required memory), with the available global memory constrained to 10MB.

We see that, in comparison to strategy 1, strategy 2 has a slightly worse performance and intensity, but takes less overall time, as conjectured in 10.

Strategy 3 has a slightly worse performance but less time than strategy 1, but higher performance than strategy 2 at a slightly higher time. The intensity is the lowest among all strategies. Strategy 4 has a very similar intensity to strategy 2, but showing the best performance of all strategies as expected from the compute queue utilization maximization hypothesis. However, from the execution time we can tell we are between the best and worst case scenarios for this strategy. We can motivate this due to our imperfect implementation of the depth-first cuda calls scheduling, which doesn't allow an exactly concurrent H2D copy with one of the multiplication kernels.

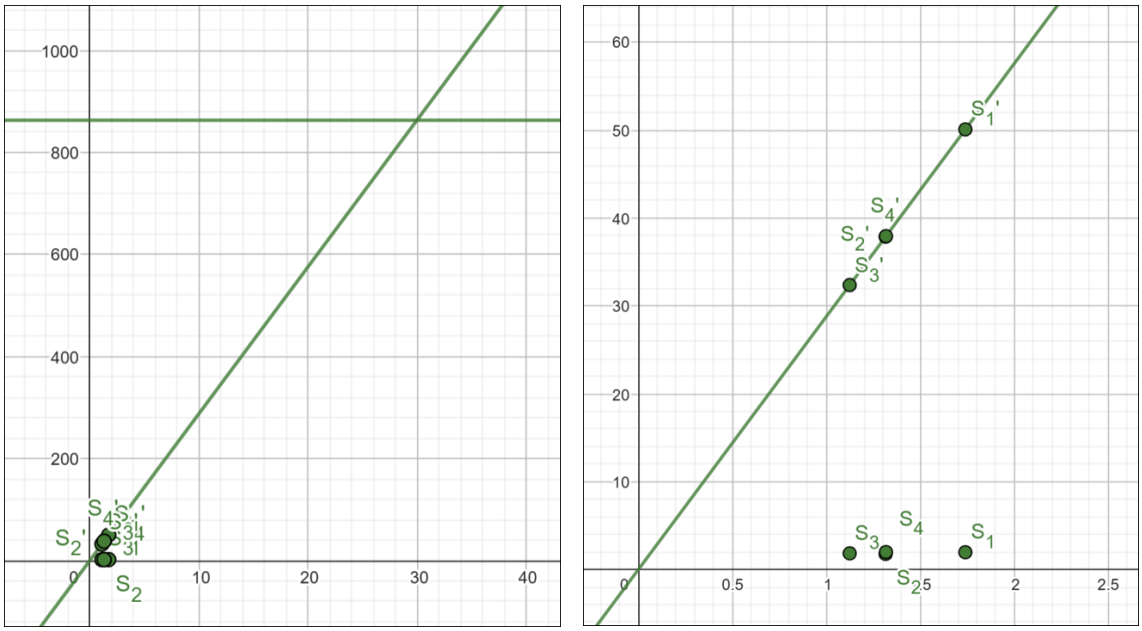
STRATEGIES RANKING

CUDA time	Avg. perf	Tot int.
S2	S4	S1
S4	S1	S4
S3	S3	S2
S1	S2	S3

Figure 19: strategies ranking based on extracted metrics

One could ask why some strategies have higher average performance while taking more time than other strategies with lower performance. We can answer this by observing that the average performance is an indicator of the overall compute queue utilization, being the ratio of the total floating point operations (hardcoded) over the total execution time (including PCIe transfers). A strategy that has higher average performance means either that it is keeping the compute queue occupied more than the copy queues, or that there is an efficient concurrency between copies and computation. However, as we've seen in the Strategies section, a strategy with less gaps between kernel execution must process lower sized blocks, and perform more transfers before processing the whole matrix, thus we can conclude that the overall execution time is a tradeoff between occupancy and block size.

Another thing that could seem counterintuitive is why strategy 1 has the highest intensity but is the slowest overall. We can motivate this by recalling that the intensity is hardcoded as the ratio of the total registered floating point operations count over the total registered bytes amount, so it is not dependent on time. Strategy 1 has to work on bigger blocks since it does not have auxiliary buffers. This enables it to make better use of the memory transfer, by being able to load less times the same blocks. Consequently, it is able to perform the same number of operations with less memory transfers, and that is what raises its intensity. However, the transfers it has to make are bigger and thus slower with respect to less intense strategies, and there are more and larger gaps between kernel executions.



(a) (b)

Figure 20: naive roofline model, (a) complete and (b) zoomed on the area where our program stands

Finally, we plotted our results in the naive roofline model for this GPU in figure 20. We plotted the average performance points (S_x, which reflect the real observed performance), as well as the naive performance (S'_x) we would have according to the naive roofline model, that we obtained multiplying the total registered intensity by the maximum declared bandwidth of the GPU. The naive performance points lie higher on the graph than the observed ones. We observe that the naive performance differs not only in value, but also in the final ranking of the strategies with respect to the real observed performance.