GIGACHECK

ABFT Matrix Multiplication of any size in CUDA

Emanuele Messina, Francesco Risso

2025

Motivation

Matrix multiplication is fundamental in scientific, engineering, and data-driven domains, including deep learning, scientific simulations, and aerospace systems.

These fields rely on high-performance hardware accelerators like GPUs.

Modern GPU environments are susceptible to **errors from hardware faults**, thermal fluctuations, and cosmic rays, especially in aerospace applications.

Fault-tolerant computation strategies are critical because undetected errors can lead to model mispredictions or catastrophic failures.

Our project

We implemented a **fault-tolerant matrix multiplication algorithm on CUDA.**

The algorithm is designed to handle matrices whose size would not allow a single-shot computation inside the GPU due to **memory constraints.**

The algorithm incorporates an **error detection and correction** technique to ensure computational accuracy in the presence of errors.

We designed **different buffering strategies** to manage memory constraints by **splitting large matrices into manageable blocks.**

We wrote a **self-contained benchmark program** to evaluate the algorithm's performance with varying settings.

Error Detection and Correction

Augmented Matrices

Column checksum for A

$$\mathbf{c}_A = \left[\sum_{i=1}^{n} a_{i1}, \sum_{i=1}^{n} a_{i2}, \ldots, \sum_{i=1}^{n} a_{in}\right] \longrightarrow A_c = \left[\begin{array}{c} A \\ \hline \mathbf{c}_A \end{array}\right]$$

Row checksum for B

$$\mathbf{r}_B = \left[\sum_{j=1}^{n} b_{1j}, \sum_{j=1}^{n} b_{2j}, \ldots, \sum_{j=1}^{n} b_{nj}\right]^T \longrightarrow B_r = \left[\begin{array}{c|c} B & \mathbf{r}_B \end{array}\right]$$

Original Product

Checksum Blocks

Augmented Product

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

GigaChecksum

Control checksums

Augmented Product

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

Column control

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

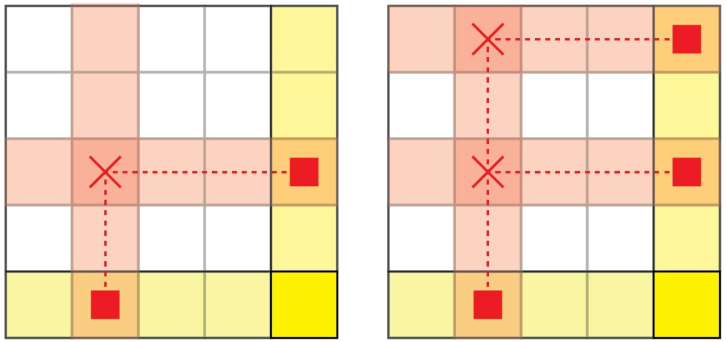
Row control

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

Verified iff original product is intact
➜ no errors

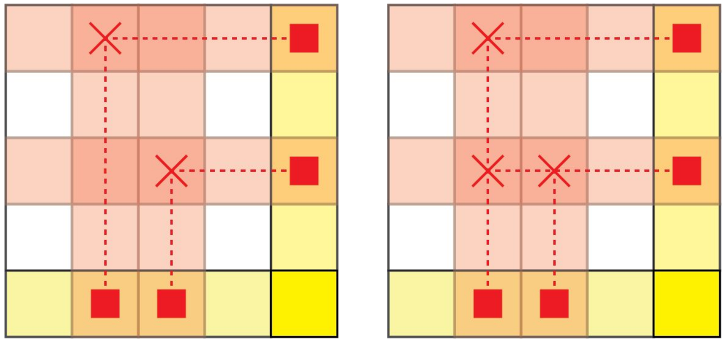
Control checksum mismatches with checksum blocks

Correctable errors



Either a single error, or collinear errors: error coordinates recoverable from mismatch indices

Uncorrectable errors



Mismatch indices describe an area: multiple errors configuration possible, individual error coordinates not recoverable

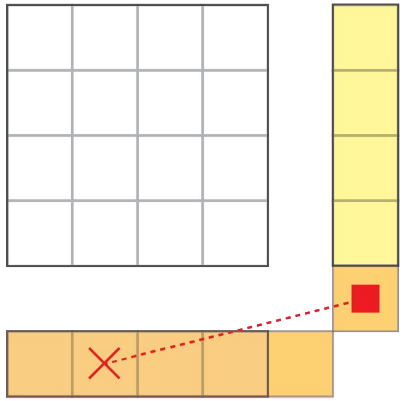
Errors in checksum blocks

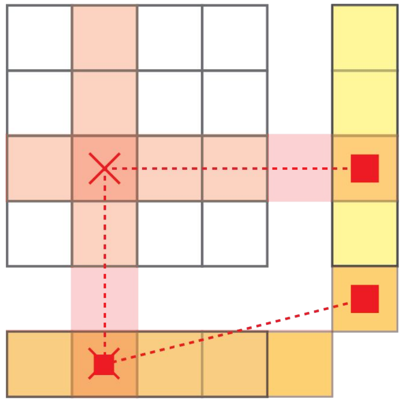
Same principle as product errors.

Error in a checksum block causes a mismatch in the gigacheck of the control checksum of the other checksum block.

Recoverable with the original gigacheck in correctable configurations.

Gigacheck corruption (rare) requires special handling ➜ not implemented.





Correction formulas

Row checksum block correction element

Column-collinear errors

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

Detected error value

Row-collinear errors

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

Column checksum block correction element

Overview of the algorithm

Overview of the algorithm

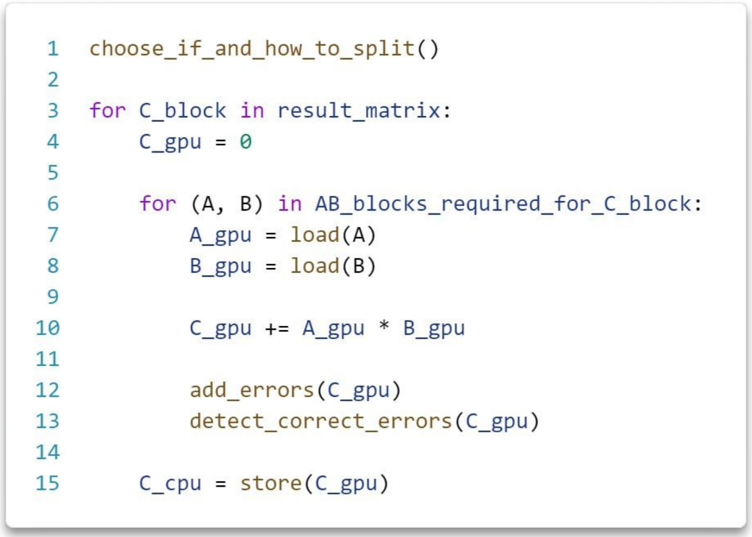
We suppose to have big matrices, that may not fit on GPU memory.

Therefore, we may have to split them.

After that, the product must be computed in blocks, with each block of C being composed as a sum of products of A and B blocks.

The error detection and correction is performed at the “small” multiplication level.

However, since the partial C are summed, the next error corrections also detect errors on the sum.



```
choose_if_and_how_to_split()

for C_block in result_matrix:
    C_gpu = 0

    for (A, B) in AB_blocks_required_for_C_block:
        A_gpu = load(A)
        B_gpu = load(B)

        C_gpu += A_gpu * B_gpu

        add_errors(C_gpu)
        detect_correct_errors(C_gpu)

    C_cpu = store(C_gpu)
```

Choosing how to split matrices

Choosing how to split matrices

The algorithm can choose two values:

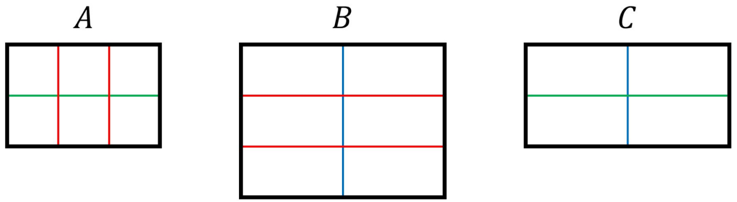
- `num_split_common_dim`, in the direction of the product (columns for A, rows for B)
- `num_split_other_dim`, in the opposite direction

Initially, the amount of memory required for hosting the full matrices is computed.

Then it is used to compute the “overflowing factor” k.

`num_split_common_dim` = √k

`num_split_other_dim` = ⌈k/`num_split_common_dim`⌉



num_split_common_dim = 3 (red)

Each block of C will be sum of 3 products of blocks

num_split_other_dim = 2 (green, blue)

C is composed by a grid of 2x2 blocks

Multiplying blocks that fit on GPU

Two approaches: traditional (tiled) vs Strassen

Tiled multiplication algorithm

Uses the traditional algorithm, with complexity O(n^3).

Leverages the idea of tiles for using the shared memory.

With square tiles of size TxT, the algorithm can reduce the loads from global memory by a factor of T.

$$\begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \ast \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

$$\begin{aligned} C_{11} &= A_{11} * B_{11} + A_{12} * B_{21} \\ C_{12} &= A_{11} * B_{12} + A_{12} * B_{22} \\ C_{21} &= A_{21} * B_{11} + A_{22} * B_{21} \\ C_{22} &= A_{21} * B_{12} + A_{22} * B_{22} \end{aligned}$$

Strassen algorithm

It is able to reduce complexity to about O(n$^{2.8074}$).

This is thanks to the fact that it transforms one multiplication into a series of sums.

Usually applied recursively once or twice on large matrices, to reduce their size with less multiplications.

After that, the traditional algorithm is used on the smaller blocks.

$$\begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

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

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Strassen vs traditional (tiled) algorithm

Ideally, Strassen is faster.

However, every error introduced may be repeated on two blocks.

If this creates non-collinear errors, they are not correctable.

We risk transforming a correctable error into an uncorrectable one.

In order to avoid this, we chose to only use the traditional (tiled) algorithm for multiplying the blocks that fit on GPU.

The kernels

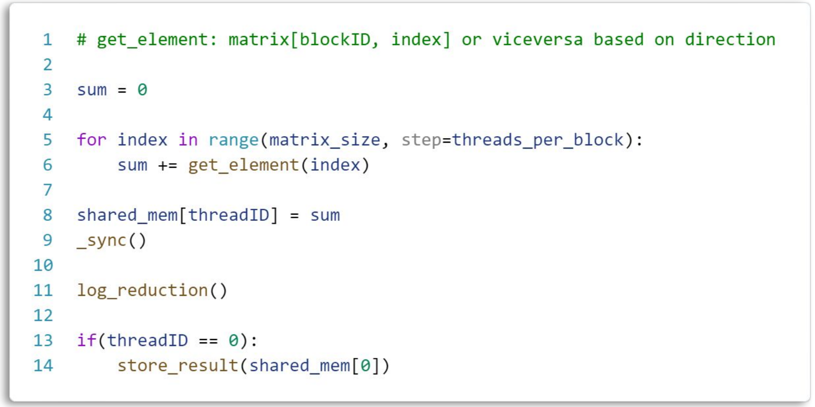
Computing Checksums

Parametric:

- col/row checksum
- checksum block (in place) / control checksum

1 (kernel) block per col/row (1D)

Reduction sum over shared mem



```
# 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])
```

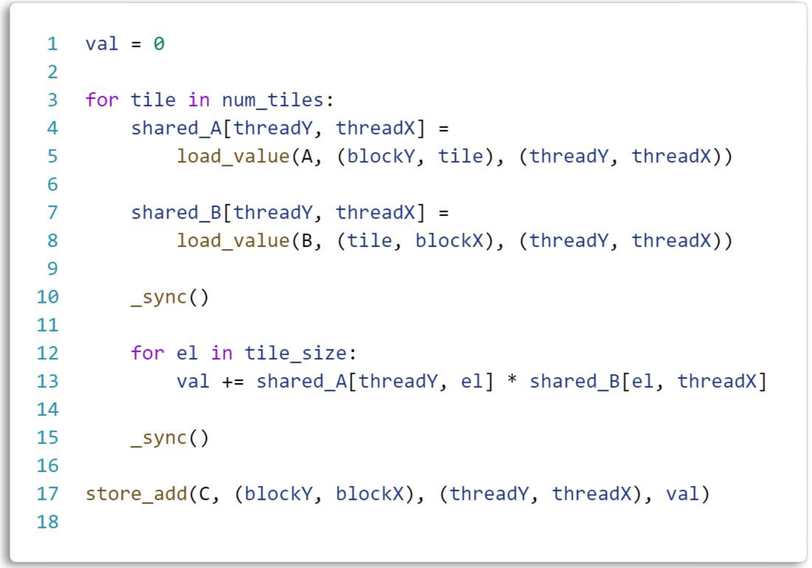
Block product and error injection

A and B are multiplied and return C

Random errors are injected in C according to chosen configuration

Then the EDC module takes action to detect and correct the errors if possible

This happens for every block multiplication



```
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)

```

Find mismatches

Parametric: compare col/row checksums

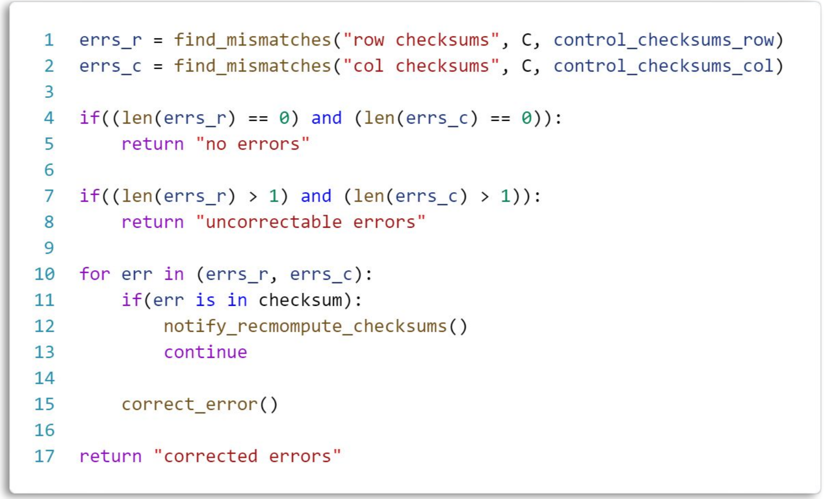
Each (kernel) block checks a portion of the vector

Mismatch found if difference higher than a thresh

Atomic increment of errors count

Triggers early stop for all the threads if too many errors found with error flag

EDC module efficiently analizes the returned indices to decide correctability and extracts the coordinates



```
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"
```

CUDA Hardware Queues

CUDA Hardware Queues

Streams don't matter among copies, they are **serialized anyway** in the copy queues: H2D, D2H.

A scheduled **operation in a queue has to wait** for the completion of **all the preceding operations in the same stream**.

A scheduled **operation in a queue has to wait** for the completion of **all the other tasks in the same queue**, except for kernels.

Kernels of different streams can run concurrently in the Compute queue if resources are available.

Kernels emit signals to the copy queues **when they finish**. **If kernels are issued sequentially** in the code, **the signal is delayed** until every kernel has finished, **blocking the copy queue**.

For a GPU that supports both H2D and D2H queues (the majority nowadays), **a depth first scheduling approach works best** because **kernels calls are interleaved with copy calls**, so the signal is not delayed.

Buffering strategies

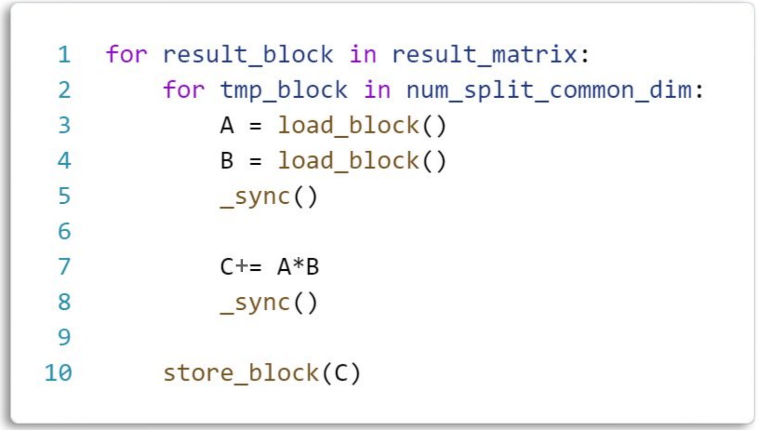
Strategy 1

This is the “naive approach”.

The GPU memory is divided into 3 blocks, called A, B and C.

At every iteration, A and B are loaded, and then the rolling update on C is computed.

Whenever C is ready, it is stored.



```
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)
```

Problems with strategy 1

PROBLEM

The CUDA queues are almost disjoint:

- H2D and D2H can overlap when C is being unloaded, while A and B are being loaded.
  - This only happens once every `num_split_common_dim` iterations
- H2D and compute can overlap when B is being transferred and A is having the checksums computed.
  - This is quite a short period

In the remaining time, they do not overlap.

POSSIBLE SOLUTION

Two new blocks can be introduced, A' and B'.

Initially, the first blocks are loaded into A and B.

Then, the algorithm can compute C = AB, while at the same time it can load the next block into A' and B'.

When the multiplication is finished, the buffers can be swapped to immediately start the next product.

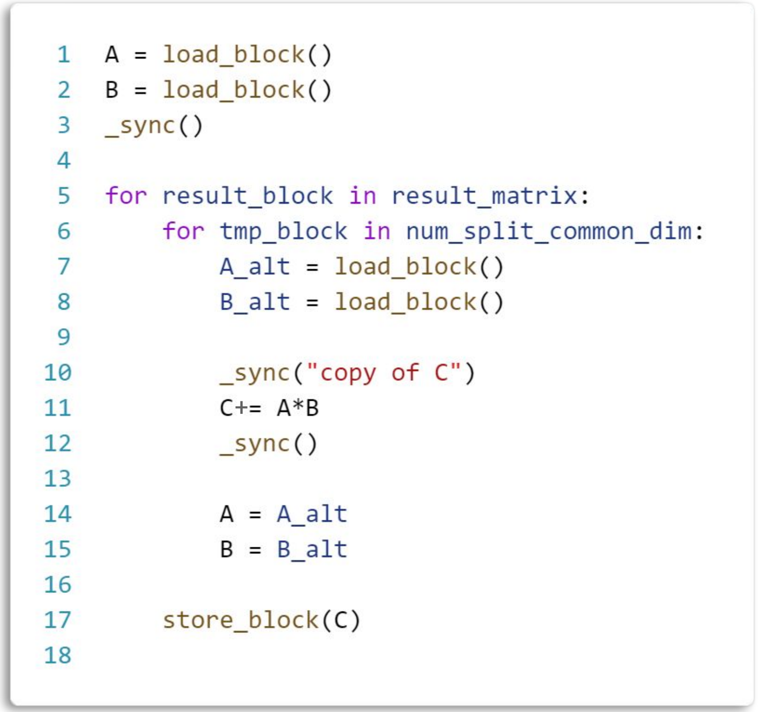
Strategy 2

It solves the problem of strategy 1, by using a double buffering idea for A and B.

Since it has 5 blocks instead of 3, each block has a size which is smaller by a factor 3/5 with respect to the first strategy blocks.

This makes all steps faster to compute.

It however increases the number of steps that are required.



```
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)

```

Problems with strategy 2

PROBLEM

Every `num_split_common_dim` iterations, the storing of C is required.

While this is happening, A' and B' can start loading.

However, the compute queue cannot be running (except for computing the checksums), since C cannot be overwritten until it is fully unloaded.

This makes the compute queue (which is the main bottleneck) not fully utilized.

POSSIBLE SOLUTION

Bring to the extreme the strategy 2, by adding an extra block for also swapping C.

Consecutive blocks of the final matrix are computed alternatively on C and C'.

This way, the new block can start the multiplication while the previous one is still unloading, since they work on separate buffers.

This strategy tries to (theoretically) remove any gap from the compute queue.

Strategy 3

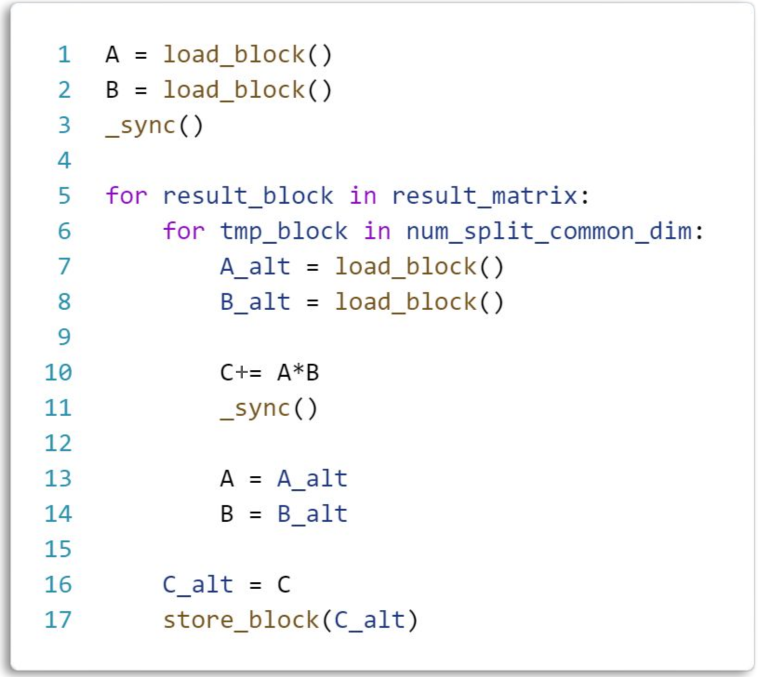
It solves the problem of strategy 2, by using a double buffering idea also for C.

It now requires 6 blocks on GPU: each one will be half the size of the block of the naive version.

This makes each step slightly faster than strategy 2, but potentially more steps are required.

We have to note that the buffer exchange C does not happen at every iteration: this will make the speedup happen only on some cycles.

This may lead to have the speedup improvement shadowed by the reduction of the block size.



```
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)
```

Another problem of all strategies

PROBLEM

Let S_{RC} be the set of blocks of A and B that are required to compute a block C_{RC}

The various S_{RC} are not disjoint if only one coordinate is changed.

Previously, all strategies would load the full S_{RC} for every block of C, thus loading every block of A and B multiple times.

POSSIBLE SOLUTION

Have two buffers C and C', that can compute adjacent blocks in parallel.

In particular, we decided that C' will compute the block to the right of C, if it exists.

This way, the two multiplications can share the same buffer for the block of A.

They however need two separate buffers B and B' for the second operand matrix.

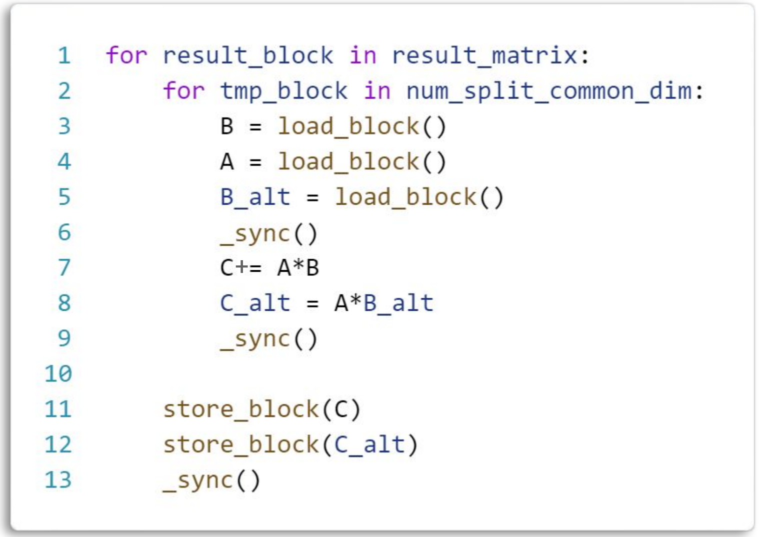
Strategy 4

It extends strategy 1, by computing two multiplications in parallel.

The number (and size) of the buffers is the same as in strategy 2.

The compute queue is not completely full as in strategy 3.

However, if we assume that the two multiplications can be executed simultaneously, then this strategy should be even faster than the third.



```
for result_block in result_matrix:
    for tmp_block in num_split_common_dim:
        B = load_block()
        A = load_block()
        B_alt = load_block()
        _sync()
        C+= A*B
        C_alt = A*B_alt
        _sync()

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

Problems with strategy 4

PROBLEM

Strategy 4 is built on top of strategy 1, without the improvements of strategies 2 and 3.

As such, it still has a reduced overlap in the CUDA queues.

POSSIBLE SOLUTION

We could add 5 more buffers, to bufferize A, B, B’, C and C’

We however believe that this big fragmentation would introduce more drawbacks than benefit, therefore we decided not to introduce this “strategy 5”

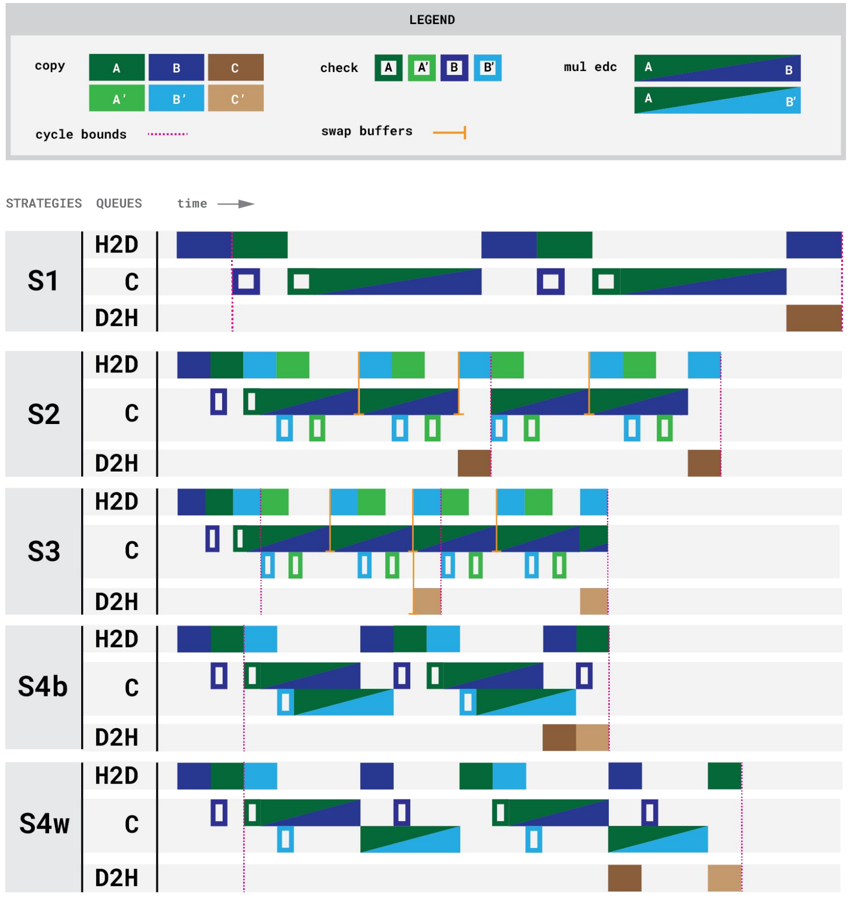
Strategies comparison

Strategies comparison (1)

This diagram displays what theoretically happens over time in the different queues.

Each strategy is shown, and strategy 4 is depicted both in the best case scenario (S4b, where the hypothesis of concurrent multiplications holds) and in the worst case scenario (S4w).

The diagrams are scaled with respect to the block size, since transferring and operating on smaller blocks should be faster.



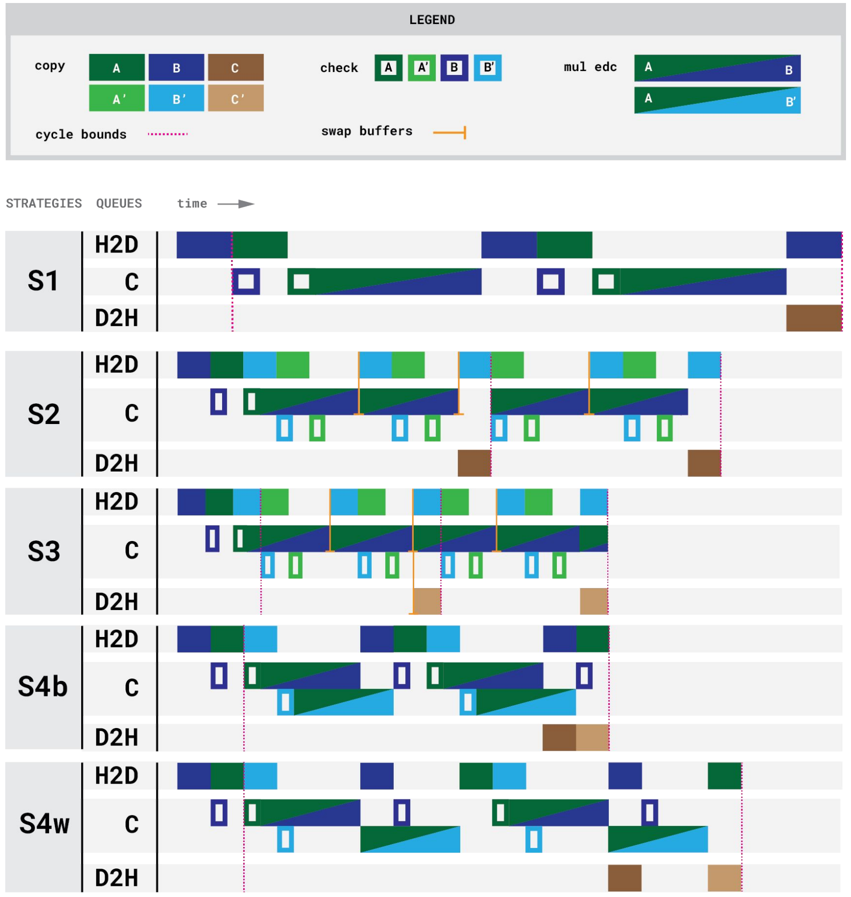
Strategies comparison (2)

The diagram describes the computation of a portion of C, assuming A is Mx2 and B is 2xN.

Strategy 1 shows the time required to compute a block, while the others depict the time required to compute approximately the same amount of data.

In particular, strategy 3 computes the same amount of data as strategy 1, while strategies 2 and 4 compute 1.2 (= 6/5) times more data, due to their block size.

Theoretically, the best strategy would be S4 if the hypothesis holds, otherwise it would be S3.



Strassen for scheduling large products

Strassen for scheduling large products

Strassen algorithm is used to break a matrix product into smaller ones.

As such, we could use it to divide large matrices into manageable ones.

It however requires some temporary matrices that are needed for multiple blocks of C.

In order to manage this, we either had to do many load/unload operations, or we had to increase the number of blocks in GPU.

$$\begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

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

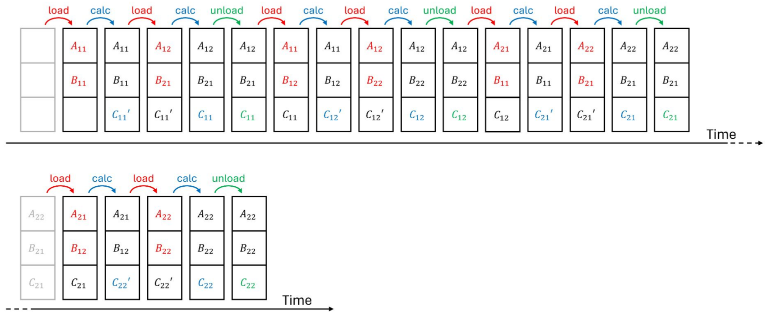
$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Memory transfers: traditional approach

Assume we have a product C = AB, where the full matrices cannot fit in GPU, but they can fit if they are split in four parts.

The traditional multiplication approach (executed on blocks instead of values) would require 20 memory accesses:

- 8 loads for blocks of A
- 8 loads for blocks of B
- 4 unloads for the blocks of the final matrix

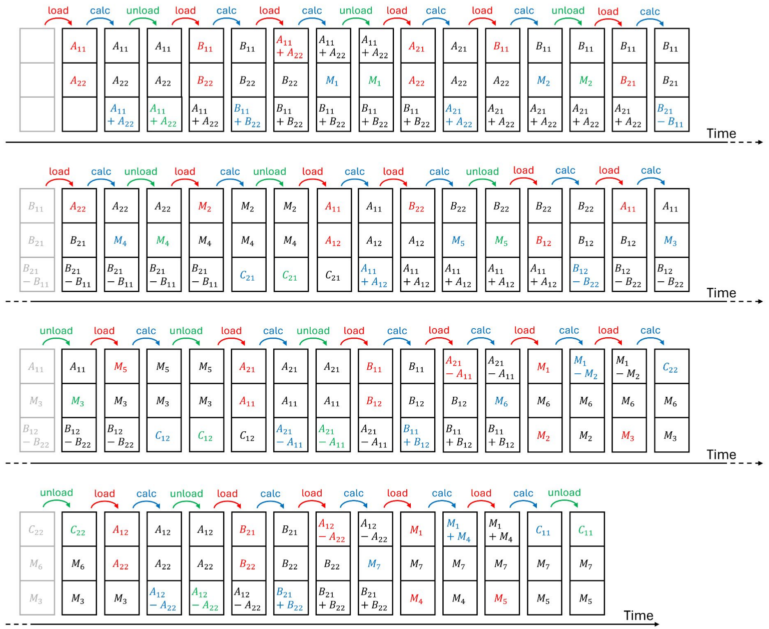


Memory transfers: Strassen (still with 3 GPU blocks)

If instead we use Strassen algorithm, we have some temporary blocks that need to be saved and then re-loaded.

In total, we need 45 memory transfers:

- 12 loads of blocks of A
- 10 loads of blocks of B
- 3 loads of temporary sum blocks
- 3 unloads of temporary sum blocks
- 8 loads of the temporary matrices M
- 5 unloads of the temporary matrices M
- 4 unloads for the blocks of the final matrix

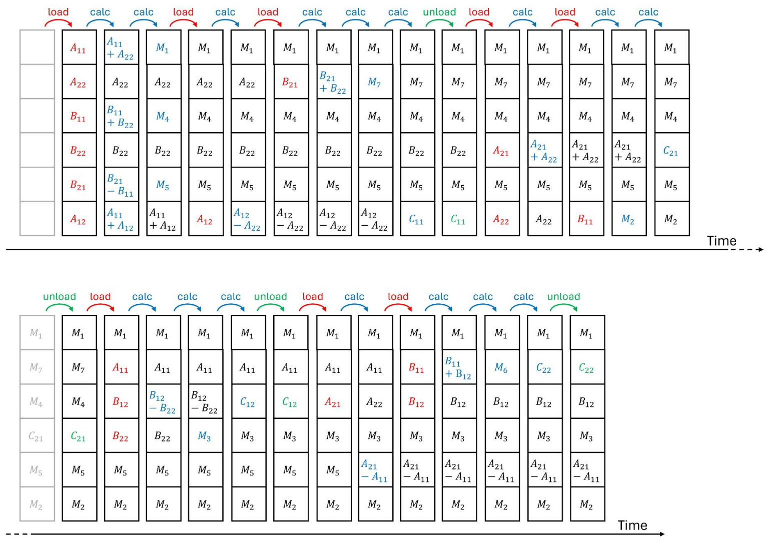


Memory transfers: Strassen (with more GPU blocks)

We can however decide to allocate as many GPU blocks as required to only unload the final matrix.

This way, using 6 blocks, we only need 21 memory transfers:

- 8 loads for blocks of A
- 9 loads for blocks of B
- 4 unloads for the blocks of the final matrix



Strassen for scheduling large products - conclusions

Strassen algorithm with many transfers is not helpful, since it adds a **substantial amount of memory copies**, that would worsen the performance of our program.

Using **Strassen with more blocks** apparently seems beneficial, since it saves a product at the sole cost of an extra block transfer.

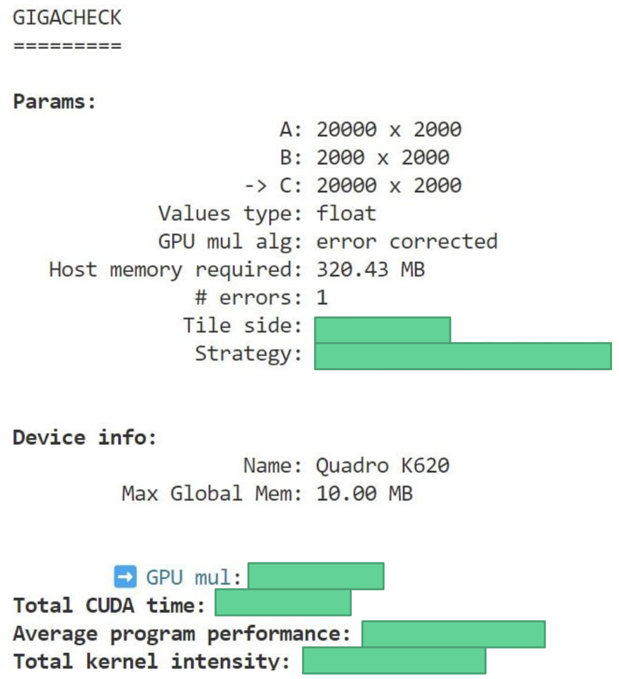
However, having double the amount of blocks makes them half in size, thus requiring about √2 times more blocks: this **may lead to** an addition of **more multiplications than what is saved**.

From this evaluation, we conclude that **Strassen algorithm is beneficial only if the matrices fully fit on the GPU memory**, since the first approach would not have any drawbacks.

As such, **we opted not to use Strassen also at this higher level**.

Results

Test config



```
GIGACHECK
=========

Params:
                     A: 20000 x 2000
                     B: 2000 x 2000
                  -> C: 20000 x 2000
           Values type: float
           GPU mul alg: error corrected
  Host memory required: 320.43 MB
              # errors: 1
             Tile side:
              Strategy:


Device info:
                  Name: Quadro K620
        Max Global Mem: 10.00 MB


        ➡ GPU mul:
Total CUDA time:
Average program performance:
Total kernel intensity:
```

Tile sizes: [8 , 16 , 32]

Strategies: [1 , 2 , 3 , 4]

GPU mul: total CPU time

Total CUDA time: CUDA only time

Avg. program performance: total_flops / total_cuda_time

Total kernel intensity: total_flops / total_globmem_transfers

Metrics: T8

T8	CUDA time [s]	avg. Perf [MFLOP/s]	tot. Intensity [FLOP/B]
S1	425.299	47.6468	2.00946
S2	427.144	47.496	2.00998
S3	435.955	47.4014	2.02076
S4	432.724	46.8542	1.97979

Performance depends on matrix size and max allowed memory, as blocks become larger and there is a **tradeoff between occupancy and latency**.

T8 has the worst times.

These matrices are too large to reap the benefits of the strategies, **tile size is too low, too many redundant transfers into shared mem**.

S1 works on bigger matrix blocks: **less ram transfers and kernel launches → best time**.

Intensity and performance is the same for all strategies.

Metrics: T16

T16	CUDA time [s]	avg. Perf [MFLOP/s]	tot. Intensity [FLOP/B]
S1	339.68	29.9695	3.99276
S2	340.821	29.8575	3.97513
S3	349.419	29.7976	3.96947
S4	345.798	29.4188	3.91152

Higher intensities mean **more operations with less global mem** to shared mem / registers **redundant transfers.**

T16 has **lower times, but still not best**, still too many shared mem transfers.

Metrics: T32

T32	CUDA time [s]	avg. Perf [MFLOP/s]	tot. Intensity [FLOP/B]
S1	341.157	15.8381	6.89697
S2	312.891	16.5527	6.83216
S3	321.4	16.5818	6.42872
S4	317.496	16.3275	6.62367

T32 (maximum block size) has the **best times overall**: probably best occupancy for large matrices (blocks)
➔ less transfers into shared mem.

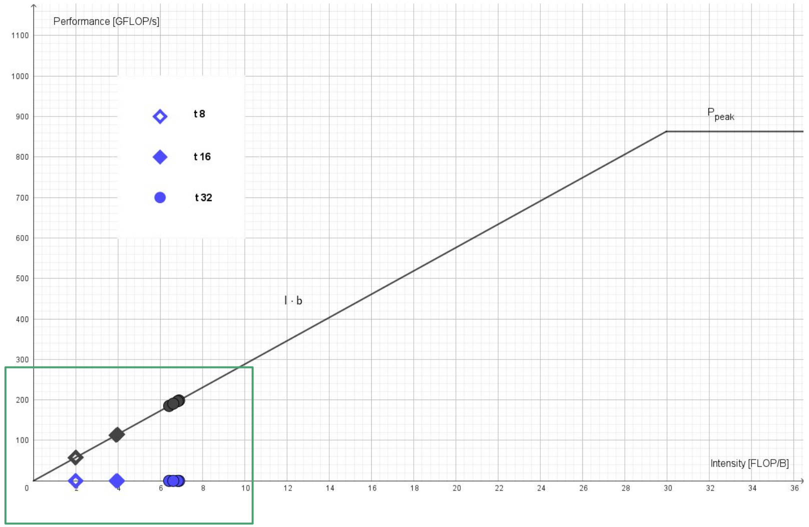
High performance means compute queue occupied more than copy queues, or efficient compute-copy concurrency. But strategies with **less gaps** has to process **smaller blocks**
➔ **tradeoff**.

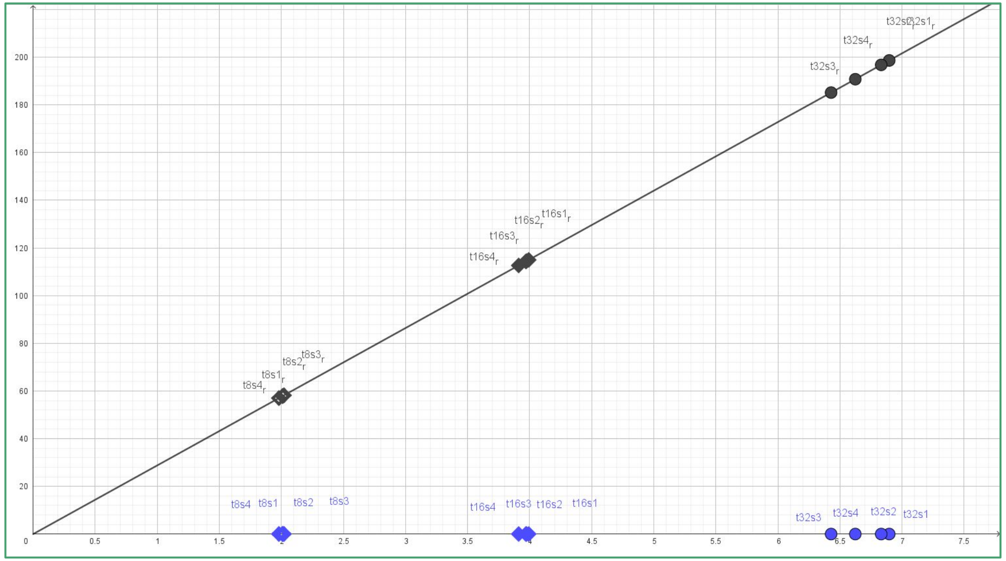
S1 is the slowest as hypothesized, here memory transfers are the bottleneck.

S4 is similar to S3, confirming the maximum utilization hypothesis.

S2 is actually the fastest: on the long run it beats S3 because it calculates more items per iteration, even if the single iteration is slower for a 2x2 split matrix.

Naive Roofline Model


Performance [GFLOP/s]
t 8
t 16
t 32
P_peak
I · b
Intensity [FLOP/B]


t32s2_r
t32s1_r
t32s4_r
t32s3_r
t16s2_r
t16s1_r
t16s3_r
t16s4_r
t8s3_r
t8s2_r
t8s1_r
t8s4_r
t8s4
t8s1
t8s2
t8s3
t16s4
t16s3
t16s2
t16s1
t32s3
t32s4
t32s2
t32s1
200
180
160
140
120
100
80
60
40
20
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5
7
7.5