Politecnico di Torino

GPU Programming

Final project

Triple Modular Redundancy in CUTLASS

Students:
Matteo Fragassi (s317636)
Davide Giuffrida (s310265)

Contents

1 Introduction **1**

2 CUTLASS background **3**
2.1 Introduction . . . 3
2.2 GEMM algorithm . . . 3
2.3 CUTLASS hierarchy . . . 4
2.3.1 The *device* section . . . 4
2.3.2 The *kernel* section . . . 5
2.3.3 The *threadblock* section . . . 6
2.3.4 The *warp* section . . . 8
2.3.5 The *arch* section . . . 8
2.3.6 A brief discussion on library modifications . . . 8

3 TMR strategies trade-offs **10**
3.1 Introduction . . . 10
3.2 TMR for error detection and correction . . . 10
3.3 TMR against transient and permanent faults . . . 11

4 Proposed solutions **12**
4.1 Introduction . . . 12
4.2 TMR at the kernel level . . . 12
4.2.1 Advantages and disadvantages . . . 14
4.3 TMR at the threadblock level: multiple computations of the same matrix block on the same SM . . . 15
4.3.1 Advantages, disadvantages and possible variations . . . 17
4.4 TMR at the threadblock level: multiple computations of the same matrix block on different SM through multiple kernel calls . . . 18
4.4.1 Advantages and disadvantages . . . 21
4.5 TMR at the threadblock level: multiple computations of the same matrix block on different processors through cooperative groups . . . 22
4.5.1 Advantages and disadvantages . . . 28

5 Profiling **29**
5.1 Solution 1 . . . 29
5.2 Solution 2 . . . 30
5.3 Solution 3 . . . 30
5.4 Solution 4 . . . 31

List of Figures

2.1 The GEMM algorithm hierarchy: matrix (a), thread block (b), warp (c) and thread (d) level 4

CHAPTER 1

Introduction

Error detection and diagnosis have become difficult problems in the last decades due to the growing complexity of the developed systems. According to the definition, hardware-related errors are caused by faults which affect the circuitry, whose functionalities can be undermined either temporarily (transient faults) or definitively (permanent faults). Most applications require proper solutions to spot errors which may be caused by transient faults, since they are particularly common when a system is stressed by high workloads or harsh environmental conditions, like in the aerospace domain. The most common way to compensate for errors is to introduce some redundancy by repeating the same computation multiple times. With this approach, it is possible to rely on majority voting to determine the actual output. The error tolerance of these methods is given by the number of times the computation is repeated. In our work, we will focus on Triple Modular Redundancy (TMR) which, as the name suggests, relies on repeating the computation three times and then performing a majority voting on the three results. The possible outcomes are listed below:

- The three outputs are the same

  There is no error, so the result can be assumed to be correct. The outcome would be identical in case the same error is present on the three copies, but this is a case which occurs with a negligible probability, so it is almost impossible in practice.

- Only two outputs are the same

  In this case we can assume that the outlier is the one affected by the error, so we can output the result of the two agreeing copies.

- The three outputs differ

  There is more than one error, so we are able to detect it but not to correct it. The only way to obtain a meaningful result is to repeat the computation.

The repetition can be characterized both in time, by executing the three copies one after the other, and in space, by exploiting parallel computation on devices that allow it. Since we work on GPUs, the spatial parallelism can be exploited to some extent, depending on the GPU structure and computational capabilities. A more in-depth analysis can also be performed on the granularity of the parallelism. In particular, it is possible to tune the number of "checkpoints" in which we divide the computation, such that the "majority voting"-based check is performed once after every checkpoint. Setting the number of checkpoints (i.e. the granularity redundancy) is a critical task because it introduces a trade-off between added overhead and latency between error genesis and error detection.

This report exposes the work performed to introduce TMR at different levels of CUTLASS, an open-source library to perform efficient matrix-matrix multiplications (GEMM) [1]. Strengths and

weaknesses of four solutions are going to be discussed, together with ways to tweak the previously mentioned parameters. For what concerns the hardware, the attention is mainly focused on the tensor cores, some special units which are accessed through particular PTX instructions. The structure of the report is the following one:

- **Background** 2

  Brief presentation of the structure of CUTLASS where we also highlight the code sections modified to introduce the TMR. In this section, the configuration of the GEMM kernel that allows us to use the tensor cores will be shown.

- **TMR Trade-offs** 3

  In this chapter, we will detail the trade-offs imposed by the frequency of the majority votes.

- **Proposed solutions** 4

  Discussion of the various solutions, deepened by the code showcase and the presentation of advantages and disadvantages.

- **Profiling** 5

  Profiling information on the proposed solutions.

CHAPTER 2

CUTLASS background

2.1 Introduction

As already mentioned, CUTLASS is a library which allows for optimized GEMM computation with CUDA. This is a problem that is common to many fields, like graphics and Machine Learning, so introducing some fault tolerance mechanism in the existing structure may be fundamental to correctly perform matrix multiplications in critical conditions. The library is highly modular, which means that every step of the computation can be customised to be performed in a particular way or by particular hardware structures in the GPU. This chapter is devoted to briefly explain the actual algorithm used by CUTLASS to perform the multiplication, to highlight the library hierarchy and provide a configuration for the top level kernel in order to make use of the tensor cores.

2.2 GEMM algorithm

The GEMM algorithm is quite complex, since it works on the entire blocks-warp-threads hierarchy to divide the input/output matrices in smaller blocks as the hardware abstraction level decreases. The dimensions of the operand matrices are M x K (first operand) and K x N (second operand). Image 2.1 will be used as a reference during the description of the different levels of the hierarchy:

- Initially, the output matrix is divided in blocks (e.g. the green one in the leftmost figure), each of which is computed by a *threadblock*. This operation is performed by accumulating the results obtained by multiplying the pink and yellow blocks. In particular, we start by multiplying the pink and yellow blocks which occupy respectively the leftmost part of the pink band and the topmost part of the yellow band, then we move the two blocks scanning the two bands. This kind of computation is performed over a number of iterations that is given by:

$$N_{iter} = \frac{K}{pink_{hor}} = \frac{K}{yellow_{ver}} \tag{2.1}$$

  $pink_{hor}$ and $yellow_{ver}$ are sized the same and they are respectively the horizontal side of the pink block and the vertical side of the yellow one. Pink and yellow blocks are divided between the threads so that each thread will fetch some tiles from global memory and store them into shared memory in the first part of the computation. After fetching each one the tile that was assigned to it, the threads need to synchronize to wait for the whole pink and yellow blocks to be present in shared memory.

- Each block is divided in k sub-blocks (8 in figure b of 2.1) and each of them is assigned to a warp. Proper mapping of threads to the blocks is done through `ThreadblockSwizzle`, a library

module that will be detailed later in the report. The division is proposed in the second figure of the image, where it is shown how pink and yellow blocks are divided in sub-blocks too. Again, each thread in the warp is in charge of copying some of the tiles in the pink and yellow sub-blocks from a bigger memory to a smaller and thus faster one. In this case, the elements are moved from the shared memory to the appropriate register file.

- Each warp block is divided between the threads that are part of that warp, such that each thread is assigned n (in this case 4) green tiles (as detailed in the third figure). The pink and yellow tiles corresponding to the green ones are sent as input to the tensor cores, which works in cooperation for all threads in a warp. As before, in order to make the tensor cores work (their PTX instructions accept only inputs from the threads local RF) we need to enforce cooperation between the threads to move data from the warp-level RF to the local ones. The actual warp level behavior can be considered for a further extension of our TMR, but we decided to stop at the invocation of the warp level MMA without going deeper.

Later we will show in the code where these steps are performed, together with the additions that are necessary to implement TMR functionalities.

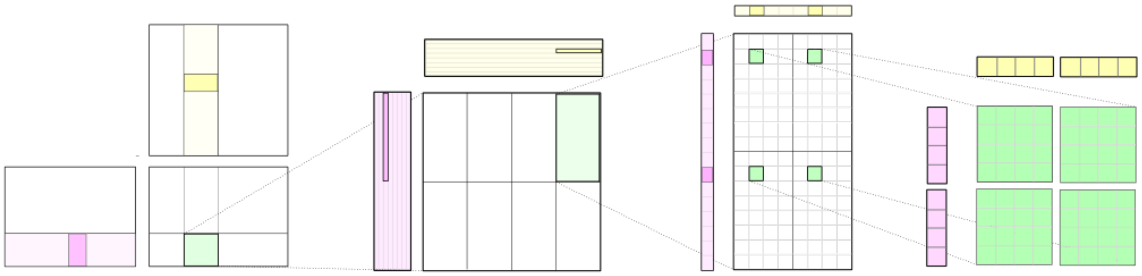

Figure 2.1: The GEMM algorithm hierarchy: matrix (a), thread block (b), warp (c) and thread (d) level

2.3 CUTLASS hierarchy

The library is highly modular, since every variation over the classic GEMM problem can be customised by adding specific flags to the GEMM operator declaration in the host code. The next sections describe the most important parts of the library to understand the proposed solutions.

2.3.1 The *device* section

This part of the library is fully contained inside the `device/` folder. It is executed on the host and performs the interfacing with the device, mainly by properly formatting the kernel call. The whole code in the folder is executed from the `Gemm` class, which is instantiated in each example with different template parameters. The important ones can be found in the lines of code reported below:

```
using CutlassGemm = cutlass::gemm::device::Gemm<
float,        // Data-type of A matrix
RowMajor,  // Layout of A matrix
float,        // Data-type of B matrix
RowMajor,  // Layout of B matrix
float,        // Data-type of C matrix
RowMajor, // Layout of C matrix
```

```
  float, // Data-type of D matrix
  cutlass::arch::OpClassTensorOp, // how to perform the last step
  cutlass::arch::Sm80, // the Cuda architecture
  cutlass::gemm::GemmShape<128, 128, 16>, // TB size
  cutlass::gemm::GemmShape<64, 64, 16>, // warp size
  cutlass::gemm::GemmShape<16, 8, 8>, // Tensor core op size
  cutlass::epilogue::thread::LinearCombination<float, 128 /cutlass::
    sizeof_bits<float>::value, float,float>, //epilogue
  cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<>, // the
    ThreadblockSwizzle
  2>; // the number of software pipeline stages
```

The first parameters define the matrices layouts and data types, including the ones related to C and D matrices: the C matrix is used to perform accumulation, while D is the one where the actual result is stored. As it is possible to notice, the actual operation is a Multiply And Accumulate (MAC), instead of a simple GEMM. The accumulation is exploited in one of the proposed solutions where the computation is divided in multiple kernels executed in series, so it is necessary to accumulate the partial results. The tensor cores are selected to perform the low-level computation with the `cutlass::arch::OpClassTensorOp` flag. These cores exist only in some of the latest Cuda architectures, so we need to pass an additional parameter to the template specifying the underlying architecture name, `cutlass::arch::Sm80` in this case. The `cutlass::gemm::GemmShape<>` parameters account for the definition of the green shapes in image 2.1, allowing to partially customize how the matrix is divided. Since the blocks are handled by some specific GPU units, not every value can be provided to this template. In particular, we have to respect limitations in terms of maximum number of threads in a block, in a warp and the tensor core PTX instruction shape (in our case it is M16 x N8 x K8). The last parameters are used to define some objects needed to perform the computation. `ThreadblockSwizzle` defines how the matrix is scanned and produces the dim3 fields, as we will see in more details in section 4.5. `epilogue` copies all the results to the global memory region that represents the matrix D and performs accumulation over C if needed. The default value of the number of software pipeline stages is 2. This means that the computation over the previously fetched pink and yellow tiles (i.e. a partial result) is done in parallel with the fetch of the next tiles. In this case, the `mmaPipelined` class will be instantiated for threadblock level computation, while, if this number is bigger than 2, then `mmaMultistage` will be selected instead. Inside the `gemm.h` file all these template parameters are used to define the quantities that are going to be used inside the kernel, passing them as arguments during the actual kernel call. When the `Gemm` class is instantiated, some additional arguments are passed through the `args` structure, which mainly contains the actual matrices and their sizes. Inside the `Gemm` class template parameters and `args` parameters are put together to build an instance of class `Params`, which contains the final parameters for the kernel. They are passed during the kernel call in the following way:

```
cutlass::Kernel<GemmKernel><<<grid, block, smem_size, stream>>>(
    params_);
```

It is important to specify that the actual kernel call depends on the actual implementation, since some TMR strategies require calling the kernel more than once.

2.3.2 The *kernel* section

This section is contained inside the `kernel/` folder and describes the first steps to be done by each thread after the start of the kernel. These steps are still very high-level and they mainly consist in instantiating the iterators used to scan the matrices in global memory (each thread fetches a different part of the matrices), calling the actual threadblock-level Matrix Multiply and Accumulate

operator (MMA) and then executing the epilogue when the threadblock operator concludes. Proper synchronization between the threads is done inside the low-level operators, as we will see afterward. This part has been modified only marginally in each strategy, since it doesn't include any kind of computation. The only part that is worth to be taken into account is the one where the positions of the pink and yellow blocks are computed starting from the block idx, since this part directly interacts with the `ThreadblockSwizzle`. The bulk of this interaction is done in the reported piece of code:

```
cutlass::gemm::GemmCoord threadblock_tile_offset =
    threadblock_swizzle.get_tile_offset(params.swizzle_log_tile);
```

`params.swizzle_log_tile` is 0 by default, and the function called is the following one:

```
CUTLASS_DEVICE
  static GemmCoord get_tile_offset(int log_tile, GemmCoord tiled_shape
   ) {
    int block_idx_x = RematerializeBlockIdxX();
    int block_idx_y = RematerializeBlockIdxY();
    int block_idx_z = RematerializeBlockIdxZ();

    // assign redudant threads to the same matrix positions as the
   original ones
    return GemmCoord{(block_idx_x >> log_tile), (block_idx_y <<
   log_tile) + (block_idx_x & ((1 << (log_tile)) - 1)), block_idx_z};
  }
```

This function is shown because it has been modified in one of the TMR implementations in order to change the way in which threadblocks are mapped to the original matrix. To be more specific, this is needed to introduce redundancy at the block level, since different blocks have to be associated with the same sub-matrix.

2.3.3 The *threadblock* section

`mma_pipelined.h` and `mma_multistage.h`, both devoted to threadblock-level computation, are located in this section. As anticipated before 2.3.1, the only difference between the two files is the number of software pipeline stages they implement. For the sake of simplicity, we focused only on `mma_pipelined.h`, but the same work could be implemented on `mma_multistage.h` as well. The structure of `mma_pipelined.h` can be summed up as follows:

- A **prologue**

  Each thread loads a tile from the first pink and yellow blocks (figure a in 2.1) from global memory, parsing the two matrices through the iterators defined in the kernel section, and copies these two tiles in shared memory. In this step and in the following ones, parts of the matrices that are fetched from memory cooperatively during one iteration are called *fragments*. Basically, they are tiles of a certain level of the hierarchy (threadblock, warp, thread) composed by fragments of the next level. For example, taking image 2.1 as a reference, the green block in figure (b) is a "threadblock fragment" that will be cooperatively computed by a certain set of warps. After this prologue we need to synchronize all threads, because, for the next part, the whole matrix needs to be in shared memory.

- The actual **threadblock-level computation**

  This part is structured as follows:

```
1  // fetching sequences for warp_frag_A[0] and warp_frag_B[0] from
       shared memory, not shown for brevity. These two fragments
       reside in warp-level register files.
2  for (; gemm_k_iterations > 0; --gemm_k_iterations) {
3      for (int warp_mma_k = 0; warp_mma_k < Base::
       kWarpGemmIterations; ++warp_mma_k) {
4        if (warp_mma_k == Base::kWarpGemmIterations - 1) {
5          // Write fragments to shared memory
6          this->smem_iterator_A_.store(transform_A_(tb_frag_A));
7          this->smem_iterator_B_.store(transform_B_(tb_frag_B));
8          // Wait until we have at least one completed global fetch
       stage
9          gmem_wait();
10         // Advance smem read and write stages
11         advance_smem_stages();
12       }
13       this->warp_tile_iterator_A_.set_kgroup_index((warp_mma_k +
       1) % Base::kWarpGemmIterations);
14       this->warp_tile_iterator_B_.set_kgroup_index((warp_mma_k +
       1) % Base::kWarpGemmIterations);
15       // load the second fragments while you perform computation
       on the first ones (the two operations overlap)
16       this->warp_tile_iterator_A_.load(warp_frag_A[(warp_mma_k +
       1) % 2]);
17       this->warp_tile_iterator_B_.load(warp_frag_B[(warp_mma_k +
       1) % 2]);
18       ++this->warp_tile_iterator_A_;
19       ++this->warp_tile_iterator_B_;
20       if (warp_mma_k == 0) {
21         // Load fragments from global memory
22         tb_frag_A.clear();
23         iterator_A.load(tb_frag_A);
24         ++iterator_A;
25         tb_frag_B.clear();
26         iterator_B.load(tb_frag_B);
27         ++iterator_B;
28         // Avoid reading out of bounds if this was the last loop
       iteration
29         iterator_A.clear_mask(gemm_k_iterations <= 2);
30         iterator_B.clear_mask(gemm_k_iterations <= 2);
31       }
32       warp_mma(
33         accum, warp_frag_A[warp_mma_k % 2], warp_frag_B[warp_mma_k
       % 2], accum);
34     }
35 }
36
```

The two for loops, at lines 2 and 3, are needed to perform iteration over the pink and yellow blocks, respectively in figure (a) and (b) of image 2.1. As anticipated, fetching sequences for the

next tiles and computation over the current ones are done in parallel, both at the threadblock level and at the warp level. The threadblock-level fetch is performed in the conditional statement at line 20, where the new blocks are loaded from global memory, and it is concluded at line 4, where the loaded values are stored in shared memory. In addition, a `_synchthreads()` is performed to wait for the whole blocks to be available in shared memory. The newly fetched blocks will be used in the next iteration of the outer loop. The warp-level fetch is performed as a part of lines 16 and 17, where the fragments are loaded from shared memory into the warp-level RF. Once again, these fragments are the ones that will be used in the next inner loop iteration, not the ones to be used in the current one, since the `warp_mma()` call on line 32 receives `warp_frag_A[warp_mma_k \% 2]`. These are the mechanisms that allow us to implement a 2-stages software pipeline. This section has been modified multiple times, in order to change the workload to be executed at the threadblock/warp level. For example, in case we wanted to implement time-based redundancy we could repeat the same warp-level MMA call thrice, keeping three copies of the accumulators and comparing them after the three calls.

2.3.4 The *warp* section

This part of the library, contained in the `warp/` folder, is in charge of dividing the warp workload over different threads and then calling the architectural MMA operator which has been defined in the `Gemm` class configuration. It could be either SIMT based or tensor cores based. As we can see from figure (d) in image 2.1, each thread is in charge of computing multiple green tiles, so there is the need to iterate on two indices to cover all the tiles, both horizontally and vertically. In case tensor cores are used to implement the architectural-level Gemm operator, each tile will consist in matrix sized in accordance to the values accepted by the tensor cores PTX instructions. The whole warp-level Gemm operator simply consists in two loops which iterate over the tiles and call the architectural-level operator for each tile.

2.3.5 The *arch* section

This last section is the one in charge of performing the lowest-level computation on the fragments associated with each thread. As hinted in 2.3.4, the operator involved in this computation depends on how the `Gemm` class was configured from host code. The main available operations are two: the first one based on SIMT parallelism and the second one on tensor cores. SIMT is the kind of parallelism that is normally exploited on GPUs, consisting in multiple executions of the same instruction in parallel in the same clock cycle. In this solution the MMA is simply computed by software, relying on the ALU accessible by each thread to perform multiplications and additions. The tensor core based solution is different, since it relies on units that are present in a limited number inside each SM. The interface with the cores is given by particular PTX instructions, which are said to be executed "cooperatively" by all threads in a warp due to the availability of 1/2 tensor cores per SM. This approach leads to faster results, since the tensor cores are units specialized to perform directly in hardware all the operations necessary for a MMA.

2.3.6 A brief discussion on library modifications

Introducing the TMR requires a considerable amount of changes, especially if we plan to reschedule the workload at the threadblock level in order to increase the number of threadblocks and assign some of them to the same matrix block, to introduce redundancy. Some of the additions included in the library are showcased here, without detailing too much the particular TMR implementation they are related to:

- If we plan to introduce **redundancy at the kernel call level** we may need to modify the *device* layer, introducing multiple kernel calls with different parameters and then comparing the results produced by each call with the other ones.
- In case we intend to introduce more **frequent checks** we may need to introduce checkpoints at lower levels, thus making it necessary to modify the structure at the threadblock layer or even at a lower one. A possible modification of this kind could consist in repeating the `warp_mma` operator call thrice and then performing the comparison. As a result, we would spot errors sooner with respect to a solution which relies on redundancy at higher levels, but we would increase the overhead.
- Scheduling **multiple threadblocks to perform the same tasks** can be needed in case we want to fully exploit the GPU capabilities, provided that we have some communication mechanisms to allow for inter-block synchronization when the results need to be compared. This approach requires modifying the dim3 to allow for a bigger number of threadblocks to be scheduled and it introduces the need to map multiple blocks to the same region, thus making it necessary to modify the `ThreadblockSwizzle` class. Threadblock based parallelism can be useful in case we want to excite multiple tensor cores, because of the limited number available on each SM.

CHAPTER 3

TMR strategies trade-offs

3.1 Introduction

This chapter is devoted to explain some approaches to integrate TMR in the library, at which depth the redundancy can be inserted and which are the pros and cons for each possible implementation. Moreover, some hints are provided on how these solutions may be affected by the GPU specifications, in particular by the number of SMs and the memory availability. The actual code will be shown later.

3.2 TMR for error detection and correction

The main objective when using TMR based solution is not only to spot an error when it arises, but also to correct it if possible. Obviously, TMR error correction capabilities are valid when an error is present in the computation of one of the copies, since the other two will still agree on the same result and win the majority vote. If the computation lasts a lot before the check is performed, the risk of errors being produced in more than one copy grows. Let us consider, for example, a task which lasts 10 ns executed in three instances at the same time on a machine which exhibits one error every 5 ns. We assume that the error is transient and that the checks are performed only at the end of the 10 ns parallel execution, so that there are no checkpoints. In this case, we could potentially end up in a situation in which one error arises in the first copy during the first 5 ns and another error appears in the second copy during the remaining 5 ns. Such configuration would lead to a failure due to errors arising in more than one copy. The situation would be different if a checkpoint was introduced in the middle, because it would have allowed to correct the first error. The flow would have changed as follows:

1. The three instances would have performed computation in parallel until the 5 ns mark.
2. At 5 ns, a check would have run to determine the correct value by performing majority voting over the three intermediate results. The correct result would have been assigned to all the three copies, to allow them to build on that result during the second part of the computation. This check would have corrected the intermediate result produced by the first copy, the one which was affected by the error.
3. the copies would have continued to compute until the end, when an another check would have been run to determine the final result. This second check would have corrected the result coming from the second copy, affected by the error born during the last 5 ns window.

We can see that introducing intermediate checkpoints helps to compensate for high frequency errors, because it allows us to correct them sooner avoiding error accumulation. However, this does not come for free because the more checkpoints we add the higher the overhead will be. The optimal solution resides in a trade-off between the overhead and the error correction capabilities, which is strongly affected by the error frequency of each particular application.

In the previous example the three instances were executed in parallel, but it is also possible to execute them in series. If we plan to do so, it is necessary to make sure that the error frequency is such that only one error will be produced during the window of time that spans over the three computations. In the following chapters, solutions for TMR which rely on serialized executions and on parallel ones are going to be shown.

3.3 TMR against transient and permanent faults

TMR can be used to correct both transient and permanent faults, but a different allocation of resources is needed in the two cases. GPU related issues will be discussed later, since we will be dealing with them specifically in the following chapters.

A GPU consists of multiple streaming multiprocessors (SMs), where each one of them could be affected by a fault that can cause an error. Dealing with transient faults does not introduce any kind of constraint, because it is enough to repeat the computation multiple times and the fault effect will disappear. Permanent faults are more difficult to handle, because they require to perform the computation another time on a different multiprocessor. There are not many ways to do so in CUDA, because the allocation of kernels over SMs is decided by a combination of runtime and hardware scheduler. The only way for the user to control allocation is to rely on *cooperative groups* [3], a feature that makes it possible to perform threadblock-level synchronization. This poses some significant constraints, since it works only when the blocks to be synchronized are running on the GPU at the same time. If this condition is not verified then a deadlock may happen, like when two blocks that require synchronization are scheduled to be run in series on the same SM. As a result, only a number of blocks lower than the number of SMs can be scheduled in the kernel grid if we plan to use cooperative groups. It is important to underline that limiting the size of the grid, i.e. the number of blocks that are launched concurrently, decreases the number of blocks per kernel. This forces to possibly launch many more kernels, thus to introduce a non negligible overhead. The approach that exploits cooperative groups is still the only one among the proposed ones that allows us to fully compensate for permanent faults, so it has been included for completeness.

CHAPTER 4

Proposed solutions

4.1 Introduction

After providing the necessary background on CUTLASS and TMR, we are going to present the four developed solutions, highlighting for each of them the advantages and disadvantages. The following discussion covers different levels of the library and provides useful insights according to the different needs a user may require. We will focus on the code, explaining the changes made to the snippets shown in section 2.3. The complete code for each solution is available in the GitHub repository of the project [2].

4.2 TMR at the kernel level

The easiest solution to implement is the one which works on the top level of the library: *device*. As we know from section 2.3.1, this level sets up the parameters and performs the actual kernel call. The solution consists in repeating the same kernel call by supplying to the kernel different matrices to use as destinations and comparing the results produced by the three instances. Since the three instances are independent, they can concurrently run on separate CUDA streams. The modifications are applied to `device/gemm.h` and they are structured as follows:

```
1    for(int i=0;i<TMR;i++){
2      args_arr[i] = Arguments({args.problem_size.m(), args.
   problem_size.n(), args.problem_size.k()},  // Gemm Problem
   dimensions
3                {args.ref_A.const_ref().data(),args.problem_size.m()},
    // Tensor-ref for source matrix A
4                {args.ref_B.const_ref().data(),args.problem_size.k()},
     // Tensor-ref for source matrix B
5                {args.ref_C.data(),args.problem_size.m()},    // Tensor-
   ref for source matrix C
6                {D[i], args.problem_size.m()},    // Tensor-ref for
   destination matrix D (may be different memory than source C matrix
   )
7                {1,1});
8      status[i] = initialize(args_arr[i], workspace, streams[i]);
9      if (status[i] == Status::kSuccess) {
10       status[i] = run(stream);
```

```
11      }
12      if (status[i] != Status::kSuccess) {
13        return Status::kErrorInternal;
14      }
15    }
16
17    cudaDeviceSynchronize();
18    CompareMatrix_kernel<<<grid,block,0,streams[0]>>>(tmp[0],D[0],D
      [1],args.problem_size.m()*args.problem_size.n());
19    CompareMatrix_kernel<<<grid,block,0,streams[1]>>>(tmp[1],D[1],D
      [2],args.problem_size.m()*args.problem_size.n());
20    CompareMatrix_kernel<<<grid,block,0,streams[2]>>>(tmp[2],D[0],D
      [2],args.problem_size.m()*args.problem_size.n());
21
22    cudaDeviceSynchronize();
23
24    for (int i = 0; i < int(log2(args.problem_size.m() * args.
      problem_size.n())); i++){
25      ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams[0]>>>(
      tmp[0],args.problem_size.m()*args.problem_size.n(),pow(2,i));
26      ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams[1]>>>(
      tmp[1],args.problem_size.m()*args.problem_size.n(),pow(2,i));
27      ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams[2]>>>(
      tmp[2],args.problem_size.m()*args.problem_size.n(),pow(2,i));
28      cudaDeviceSynchronize();
29    }
30
31    cudaDeviceSynchronize();
32
33    // the matrices are moved to the corresponding host_tmp and host_D
      through a cudaMemcpy deviceToHost (NOT SHOWN)
34
35    // actual checks
36    if(*host_tmp[0] == args.problem_size.m()*args.problem_size.n()){
37      res = 0; // or 1, it's the same
38    }else if(*host_tmp[1] == args.problem_size.m()*args.problem_size.n
      ()){
39      res = 1; // or 2, it's the same
40    }else if(*host_tmp[2] == args.problem_size.m()*args.problem_size.n
      ()){
41      res = 0; // or 2, it's the same
42    }
43    if(res == 10){
44      return Status::kRedundancyError;
45    }
```

The loop at line 1 is the core of this implementations, since it performs different operations:

- Instantiation of the arguments for each kernel call. In fact, each kernel will work on a different matrix of the D array, as we can see at line 6. Moreover, the two scalar constants `alpha` and `beta` (line 7) are multiplied to each element of the matrices AB and C respectively. In all the

proposed solutions, these parameters are hardcoded to 1 because we were not able to access them to replicate their values into the `args` copies. In general, they can be modified in order to assign different weights to each element of the sum. In particular, it is possible to completely remove the sum by setting beta to 0. Nevertheless, `beta` should not be changed in solution 4.4, because, in that case, the partial results are accumulated on the C matrix along the multiple kernel calls executed sequence.

- Initialization of the `Params` structure that is passed to the kernel. This structure depends on both `args` fields and template parameters.
- Call the kernel through the `run()` function. The dim3 instantiation is done inside this function depending on the template parameters. Since the grid size is the same as the default implementation, the `run()` function will not be covered for now.

After launching the three kernels a `cudaDeviceSynchronize()` is needed to ensure their completion. At the end of the computation, the results will be available in the D matrices. The three `CompareMatrix_kernel` calls from line 18 produce the three boolean matrices `tmp[0]`, `tmp[1]` and `tmp[2]` that respectively hold the results of the comparisons between $D[0]$ and $D[1]$, $D[1]$ and $D[2]$, $D[2]$ and $D[0]$. The matrices are sized exactly as the D ones (M x N), and each element can be 0 or 1 depending on whether the elements of the compared matrices are different or not. If all the elements are the same, then the sum of all elements in the `tmp[i]` matrix should be equal to M x N. The kernels launched from line 25 on do exactly this check. The algorithm is based on *matrix reduction* and works like explained below:

- At each iteration $(M*N)/2$ threads are scheduled to perform the sum between adjacent elements in the array representation of the matrix, saving the result in the leftmost element of the couple (`tmp[i][0], tmp[i][2], tmp[i][4], tmp[i][6]...`).
- After each iteration, half the threads are not used anymore and the remaining ones add up the adjacent results produced during the previous step, saving once again the result in the leftmost element of the couple (`tmp[i][0], tmp[i][4], tmp[i][8]...`). Even though this solution introduces divergence in the thread execution, divergent threads conclude immediately so they do not cause a considerable overhead.
- The execution goes on like this until we are left with only one element in the first position of the matrix/array.

After doing so we need to copy the tmp and the D matrices in the host space. This is done by allocating the `host_tmp` and `host_D` matrices and copying the results inside them through a `cudaMemcpy()`. The checks are then performed over the first elements of the host_tmp matrices to assess which results are correct.

4.2.1 Advantages and disadvantages

This solution is valid because of its simplicity since it does not require any kind of modification inside the kernel itself. Furthermore, because kernels are mapped to different streams, they can be executed in parallel if the allocation of resources to the SMs makes it possible to do so. Even if the execution is serialized, the solution is still able to guarantee error correction if the errors manifest themselves with a frequency of one every three kernel calls. No kind of protection is offered against permanent faults, because in case there was one then all the three kernels would produce an incorrect result, provided that the entire GPU was completely used by all kernels. A possible serial execution of the kernels is responsible for a decrease in performances in almost every scenario, mainly because of the launching sequences overhead. In case the error frequency increases too much it is necessary to use checkpoints, which can be implemented only by modifying the internal layers of the library.

4.3 TMR at the threadblock level: multiple computations of the same matrix block on the same SM

The first step in implementing checkpoints involves repeating multiple times the computation of a single threadblock and then comparing the results. The repetition is done on the same SM, so there is no need to change the grid size. In this case it is necessary to introduce modifications in deeper levels of the library, to be more specific in the two loops showed in section 2.3.3 inside `threadblock/mma_pipelined.h`. The new structure is the following one:

```
// declare additional accumulators to store the results of the three iterations
FragmentC accum_array[3];
// declare helper variables
int tocopy;
accum_array[0].clear();
accum_array[1].clear();
accum_array[2].clear();
tocopy.clear();
for (; gemm_k_iterations > 0; --gemm_k_iterations) {

    for (int warp_mma_k = 0; warp_mma_k < Base::kWarpGemmIterations; ++warp_mma_k) {

      if (warp_mma_k == Base::kWarpGemmIterations - 1) {

        // Write fragments to shared memory
        this->smem_iterator_A_.store(transform_A_(tb_frag_A));

        this->smem_iterator_B_.store(transform_B_(tb_frag_B));

        // Wait until we have at least one completed global fetch stage
        gmem_wait();

        // Advance smem read and write stages
        advance_smem_stages();
      }

      this->warp_tile_iterator_A_.set_kgroup_index((warp_mma_k + 1) % Base::kWarpGemmIterations);
      this->warp_tile_iterator_B_.set_kgroup_index((warp_mma_k + 1) % Base::kWarpGemmIterations);

      // load the second fragments while you perform computation on the first ones

      this->warp_tile_iterator_A_.load(warp_frag_A[(warp_mma_k + 1) % 2]);
      this->warp_tile_iterator_B_.load(warp_frag_B[(warp_mma_k + 1) % 2]);
```

```
34
35      ++this->warp_tile_iterator_A_;
36      ++this->warp_tile_iterator_B_;
37
38      if (warp_mma_k == 0) {
39
40        // Load fragment from global A
41        tb_frag_A.clear();
42        iterator_A.load(tb_frag_A);
43        ++iterator_A;
44
45        // Load fragment from global B
46        tb_frag_B.clear();
47        iterator_B.load(tb_frag_B);
48        ++iterator_B;
49
50        // Avoid reading out of bounds if this was the last loop
   iteration
51        iterator_A.clear_mask(gemm_k_iterations <= 2);
52        iterator_B.clear_mask(gemm_k_iterations <= 2);
53      }
54
55      warp_mma(
56        accum_array[0],
57        warp_frag_A[warp_mma_k % 2],
58        warp_frag_B[warp_mma_k % 2],
59        accum_array[0]);
60
61      warp_mma(
62        accum_array[1],
63        warp_frag_A[warp_mma_k % 2],
64        warp_frag_B[warp_mma_k % 2],
65        accum_array[1]);
66
67      warp_mma(
68        accum_array[2],
69        warp_frag_A[warp_mma_k % 2],
70        warp_frag_B[warp_mma_k % 2],
71        accum_array[2]);
72    }
73
74
75  // compare the three results to check if they are the same
76  for (int i = 0; i < int(accum.kStorageElements); ++i) {
77    if(accum_array[0].raw_data()[i] == accum_array[1].raw_data()[i]){
78      tocopy = 0;
79    } else if (accum_array[0].raw_data()[i] == accum_array[2].raw_data
   ()[i]){
80      tocopy = 0;
```

```
81      } else if (accum_array[1].raw_data()[i] == accum_array[2].raw_data
        ()[i]){
82         tocopy = 1;
83      } else {
84         // propagate an error and return it as an output of the cuda
        call
85      }
86      // perform the actual copy operation
87      accum.data()[i] = accum_array[tocopy].data()[i];
88      if (tocopy != 0) accum_array[0].data()[i] = accum_array[tocopy].
        data()[i];
89      if (tocopy != 1) accum_array[1].data()[i] = accum_array[tocopy].
        data()[i];
90      if (tocopy != 2) accum_array[2].data()[i] = accum_array[tocopy].
        data()[i];
91    }
92  }
```

As anticipated, there is the need to define new fragments to host the results of the repeated computations over the same green tile (check image 2.1 as reference). These fragments are defined starting from line 2 and they are initialized to zero. Another fragment `tocopy` is allocated to host the index of the copy which produced the correct result for each element of the fragment. The core of the loop resides at line 55, where the three `warp_mma()` calls are performed in series, storing their results in a different fragment of `accum_array`. These fragments are used as accumulators, since multiple iterations of the inner loop are performed before comparing them. When inner loop iterations terminate, a loop (line 76) is used to compare the results in the three fragments element by element. Depending on the result of the comparison, an element of tocopy is set to identify which matrix has to be used as the source of the correct value for that element. After the comparisons, the accum fragment (i.e. the one returned by the function) is updated with the correct values (line 87). There is still the need to correct the wrong values in the fragments that are part of the accum_array: if we do not correct these fragments, then the `warp_mma()` calls accumulating on these fragments would accumulate over a wrong result. The correction is performed in the loop at line 88, where the tocopy elements are checked one by one and, depending on them, the fragment elements are updated.

4.3.1 Advantages, disadvantages and possible variations

By introducing checkpoints at the threadblock level, this solution potentially reduces the latency between error detection and correction. In order to quantify its effectiveness we should take into account the following cases:

- The kernel grid consists in a number of blocks that is lower than the number of SMs

  In this case the checkpoints efficiency is reduced, because the time required for computing a single block almost coincides with the one needed to execute the whole kernel. This means that the checkpoint coincides with the check that we performed at the end in the previous solution, provided that the three kernels are able to run in parallel. However, the scenario where the kernels run in parallel (in the first solution) and the number of blocks is lower than the number of SM only happens under rare circumstances. Overall this solution can be considered to be an improvement in almost every regard with respect to the previous one.

- The kernel grid consists in a number of blocks higher than the number of SMs

In this case the new solution is always better because a single SM will be used to compute multiple blocks. As a result, having a checkpoint in correspondence of the end of a block does not coincide with having a checkpoint at the end of the kernel.

This technique does not compensate for permanent faults as the first one, because redundant computations will be executed on the same SM of the base computation. Actually, this solution makes things even worse if we are dealing with permanent faults, because the three results may be altered in the same way.

Another powerful feature of this solution resides in the tocopy fragment. Because it has the same size of the other fragments in accum_array, it is possible to source different elements of the fragments from different fragments of accum_array by using the elements of tocopy to tell which is the instance that produced the correct value for that element. This makes the solution resistant to multiple transient faults that may happen during different instances of the computation, provided that they do not affect the same element of the fragments. Nevertheless, this is a very rare occurrence because it would require a transient fault which exhibits a degree of regularity.

The proposed approach is mostly good, with its only serious vulnerability residing in potentially consistent errors due to permanent faults. In the following sections we will propose other solutions, trying to find compromises between checkpoint frequency and resilience to permanent faults.

4.4 TMR at the threadblock level: multiple computations of the same matrix block on different SM through multiple kernel calls

This solution is a compromise between the previous ones, since it combines the same checkpoint structure of section 4.3 with the possibility to compute the same block on different multiprocessors by running the three instances as part of different kernels, as in section 4.2. The basic approach is to scheduling three kernels for each couple of yellow and pink blocks in figure 2.1. As a result, the number of kernels will be:

$$N_{kernels} = 3N_{iter}$$

N_{iter} is defined in equation 2.1.

Each kernel produces a partial result, so their outputs should be sent as inputs to the kernels that will run on the same stream. As in the solution described in section 4.2, there is one stream per copy to be computed, but in this case there are multiple kernel launches on the same stream. Intermediate results from different streams are compared as soon as they are completed, so the comparison and reduction kernels also need to be executed N_{iter} times. The implementation of this solution require to appropriately modify two files:

- `device/gemm.h`: to organize the kernel calls.
- `threadblock/mma_pipelined.h`: to eliminate the loop over N_{iter}.

The structure of `threadblock/mma_pipelined.h` is the same of the first version, aside for the missing loop on N_{iter} (i.e. the outer loop). `device/gemm.h` has been modified as shown below:

```
1     // define the new args_int structures
2     for (int i = 0; i < TMR; i++){
3       args_int[i] = Arguments({args.problem_size.m(), args.
    problem_size.n(), args.problem_size.k()},  // Gemm Problem
    dimensions
4       {args.ref_A.const_ref().data(),args.problem_size.m()},    //
    Tensor-ref for source matrix A
```

```
5       {args.ref_B.const_ref().data(),args.problem_size.k()},    //
    Tensor-ref for source matrix B
6       {args.ref_C.const_ref().data(),args.problem_size.m()},    //
    Tensor-ref for source matrix C
7       {device_D[i], args.problem_size.m()},    // Tensor-ref for
    destination matrix D (may be different memory than source C matrix
    )
8       {1,1});
9     }
10    for (int iter = 0; iter < total_gemm_k_iterations; iter++){
11      for (int tmr_inst = 0; tmr_inst < TMR; tmr_inst++){
12        status = initialize(args_int[tmr_inst], iter, workspace,
    streams[tmr_inst]);
13
14        if (status == Status::kSuccess) {
15          status = run(streams[tmr_inst]);
16        }
17      }
18      cudaDeviceSynchronize();
19      // copy matrices to host and print them
20      for (int i = 0; i < TMR; i++)
21        cudaMemcpy(host_D[i], device_D[i], args.problem_size.m() *
    args.problem_size.n() * sizeof(float), cudaMemcpyDeviceToHost);
22      if (args.problem_size.m()*args.problem_size.n() >= 512){
23        grid = dim3((int)ceil(args.problem_size.m()*args.problem_size.
    n()/512),1,1);
24        block = dim3(512, 1, 1); // 512 maximum block size (in terms
    of number of threads)
25      } else {
26        grid = dim3(1,1,1);
27        block = dim3(args.problem_size.m()*args.problem_size.n(), 1,
    1); // 512 maximum block size (in terms of number of threads)
28      }
29      if (args.problem_size.m()*args.problem_size.n() >= 1024){
30        grid_reduce = dim3((int)ceil(args.problem_size.m()*args.
    problem_size.n()/1024),1,1);
31        block_reduce = dim3(512, 1, 1);
32      } else {
33        grid_reduce = dim3(1,1,1);
34        block_reduce = dim3(args.problem_size.m()*args.problem_size.n
    ()/2, 1, 1); // 512 maximum block size (in terms of number of
    threads)
35      }
36      // compare the results through the optimized kernels
37      for (int i = 0; i < TMR; i++)
38      CompareMatrix_kernel<<<grid,block,0,streams[i]>>>(tmp[i],
    device_D[i],device_D[(i+1) % TMR],args.problem_size.m()*args.
    problem_size.n());
39      // wait for the comparison kernels to finish
```

```
40      cudaDeviceSynchronize();
41      // Reduction
42      // BEWARE: DO NOT USE the CUDA implementation of pow() (i.e.
    inside the kernel), THERE ARE ROUNDING ERRORS
43      for (int i = 0; i < int(log2(args.problem_size.m() * args.
    problem_size.n())); i++){
44        ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams
    [0]>>>(tmp[0],args.problem_size.m()*args.problem_size.n(),pow(2,i)
    );
45        ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams
    [1]>>>(tmp[1],args.problem_size.m()*args.problem_size.n(),pow(2,i)
    );
46        ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams
    [2]>>>(tmp[2],args.problem_size.m()*args.problem_size.n(),pow(2,i)
    );
47        cudaDeviceSynchronize();
48      }
49      // copy the results to host_D (NOT SHOWN)
50      // actual checks
51      if(*host_D[0] == args.problem_size.m()*args.problem_size.n()){
52        res = 0; // or 1, it's the same
53      }else if(*host_D[1] == args.problem_size.m()*args.problem_size.n
    ()){
54        res = 1; // or 2, it's the same
55      }else if(*host_D[2] == args.problem_size.m()*args.problem_size.n
    ()){
56        res = 0; // or 2, it's the same
57      }
58      if(res == DEFAULT_VAL){
59        return Status::kRedundancyError;
60      }
61      // copy the intermediate result in the other device matrices,
    since their results may be incorrect
62      for (int i = 0; i < TMR; i++)
63        if (i != res)
64          cudaMemcpy(device_D[i], device_D[res], args.problem_size.m()
     * args.problem_size.n() * sizeof(float), cudaMemcpyDeviceToDevice
    );
65      // copy D matrices in the corresponding C ones
66      for (int i = 0; i < TMR; i++)
67        cudaMemcpy(device_C[i], device_D[i], args.problem_size.m() *
    args.problem_size.n() * sizeof(float), cudaMemcpyDeviceToDevice);
68      for (int i = 0; i < TMR; i++)
69        args_int[i] = Arguments({args.problem_size.m(), args.
    problem_size.n(), args.problem_size.k()},  // Gemm Problem
    dimensions
70        {args.ref_A.const_ref().data(),args.problem_size.m()},    //
    Tensor-ref for source matrix A
71        {args.ref_B.const_ref().data(),args.problem_size.k()},    //
```

```
    Tensor-ref for source matrix B
72          {device_C[i],args.problem_size.m()},    // Tensor-ref for
    source matrix C
73          {device_D[i], args.problem_size.m()},    // Tensor-ref for
    destination matrix D (may be different memory than source C matrix
    )
74          {1,1});
75      }
```

It is evident that this solution borrows some concepts from the first one, since there is the need to process kernel results in the same way. As usual, the first step is to define the arguments to be passed to kernels running on the CUDA streams (line 2), keeping into account that each stream is devoted to compute a copy of the final matrix. After the first kernel of each stream is executed, the accumulation matrix (C) needs to be changed because multiple kernels are going to run in series and the following iterations need to accumulate on partial results produced by the previous ones. For this reason, the first accumulation is done using the original matrix C, while the following ones will rely on the partial D produced in the previous step (line 68). It should now be clear why the argument `beta` needs to be set to 1 for this solution to work, as explained in section 4.2. The partial D matrix of each stream is copied into the new accumulation matrix `device_C[i]` in the for loop at line 66). Because in this solution the N_{iter} iterations are executed at the kernel call level, there is the need to wrap the kernel calls loop in an outer loop (line 10). At each iteration, three kernels will be scheduled to be run on three different streams and once they are done, the comparison and reduction kernels are launched. Executing these kernels requires to adjust the grids, because we need a different amount of threads to be scheduled. This adjustment should also be done in the solution presented in section 4.2, but it has not been shown in the previous discussion for brevity. The main idea behind this operation is to check the problem size against one threshold, depending on the kernel to be run (comparison or reduction):

- **Comparison**

  When the comparison kernel is run, one thread per matrix element is needed for a total of $M * N$ threads. The maximum number of threads per threadblock is 512, so we need to compare $M * N$ with this value: if it is larger, then we schedule multiple maximally sized blocks (line 22), otherwise we schedule a single properly sized block (line 25).

- **Reduction**

  The considerations done for the comparison operation can be applied to grid_reduce, the grid size used for the reduction kernel. In this case, however, we need only a number of threads that is half the matrix size, since reductions are done over couples of elements. This means that we can process matrices twice as big with the same number of blocks as before, so the threshold on the matrix elements can be raised to 1024.

After copying the results to the host space and performing the final checks, the wrong result matrices are replaced with the correct one. Such correction is performed through the for loop at line 62, where device_D[res] (the correct matrix) is copied in all the other device_D[i] matrices. As anticipated, after each iteration among the N_{iter} ones, we need to change the matrix involved in the accumulation. This is done at line 68.

4.4.1 Advantages and disadvantages

As mentioned before, this solution poses as a compromise between the first two, since it tries to offer additional resistance to permanent faults w.r.t. the second solution by sacrificing some performances.

The importance of this approach resides in being a stepping stone toward the possibility to execute multiple instances of the same block on different multiprocessors, combining it with a checkpoint mechanism. The key disadvantage is the non negligible overhead added by the repeated kernel calls, which should account for the comparison and reduction kernels as well. Another critical point resides in the parallelism offered by multiple streams since, if the matrix is too big, the GPU occupation rises, preventing an actual parallel execution of the kernels scheduled on different streams. This does not only mean that the kernels will be serialized, but it also makes it difficult to understand whether the copies are really executed on different SMs or not. This problem is solved in the next solution that will guarantee execution of different instances on different SMs by relying on cooperative kernels, despite being responsible for a similar overhead in time.

4.5 TMR at the threadblock level: multiple computations of the same matrix block on different processors through cooperative groups

The last solution we are going to propose is by far the most complex one, since it relies on both the synchronization between different blocks of a kernel through *cooperative groups* and the redistribution of the workload on multiple kernels. The need to have additional kernels stems from the fact the cooperative groups require a number of blocks for each kernel that is lower than the number of SMs. In case this condition was not satisfied, some threadblocks could hang during the synchronization with other blocks that were scheduled to be executed later on the same SM. Each kernel execution computes a certain number of blocks of the D matrix, executing all the three copies of the computation that are part of TMR algorithm on different threadblocks. This is the only way to be fully sure that the three instances are run on different multiprocessors, since this is a requirement of the synchronization protocol. One threadblock over the three that perform the same computation is labelled as the "master" because it is the one that compares the three results. Performing this operation requires to access a global memory area, since this is the only memory level that is shared by multiple SMs, so it negatively affects the performances. Due to the constraints imposed by cooperative kernel calls, the kernel grids must be reshaped so, as hinted before, multiple threadblocks on the same grid must be mapped to the same tile of matrix D, because of the TMR. The modifications are going to be explained strating with the `device/gemm.h` file, which now contains the logic to divide the workload in multiple kernels:

```
1  cudaGetDeviceProperties(&deviceProp, 0);
2  SM_no = deviceProp.multiProcessorCount;
3  int inst_shape = InstructionShape::kM * InstructionShape::kN;
4  cutlass::gemm::GemmCoord grid_shape = threadblock_swizzle.
       get_tiled_shape(
5    args.problem_size,
6    {ThreadblockShape::kM, ThreadblockShape::kN, ThreadblockShape::kK},
7    args.split_k_slices);
8
9  // define the grid for the problem, depending on the matrix size
10 dim3 grid = threadblock_swizzle.get_grid_shape(grid_shape);
11
12 // check if the matrix is too big for a cooperative launch
13 if (grid.x * grid.y * grid.z > SM_no) {
14
15   // schedule multiple kernel calls on the same stream
16   // compute the number of row blocks per each kernel. This value is
       obtained under the hypothesis that we have to avoid
```

```
17  // as much as possible to divide over the columns, since it requires
      refactoring the matrix to keep the row major layout.
18  int row_blocks_no = floor(((float)SM_no) / (grid.y * 3));
19
20  if (row_blocks_no == 0)
21    row_blocks_no = 1;
22
23  if (row_blocks_no > grid.x)
24    row_blocks_no = grid.x;
25
26  // compute the number of columns blocks per each kernel
27  int column_blocks_no = grid.y / ceil(((float)grid.y) / (((float)
     SM_no) / 3));
28  if (column_blocks_no > 8)
29    column_blocks_no = 8;
30
31  // allocate matrices on the device to host B/C/D submatrices (
     integrity checks are omitted)
32  result = AllocateMatrix(&B_sub, GemmKernel::kThreadCount * args.
     problem_size.k() * column_blocks_no, inst_shape);
33  result = AllocateMatrix(&C_sub, GemmKernel::kThreadCount *
     column_blocks_no * row_blocks_no, inst_shape);
34  result = AllocateMatrix(&D_sub, GemmKernel::kThreadCount *
     column_blocks_no * row_blocks_no, inst_shape);
35
36  // allocate matrices on the host, to retrieve the results (NOT SHOWN
     )
37
38  // allocate D[0],D[1] submatrices for TMR copies
39  result = AllocateMatrix(&D[0], GemmKernel::kThreadCount *
     column_blocks_no * row_blocks_no, inst_shape);
40
41  result = AllocateMatrix(&D[1], GemmKernel::kThreadCount *
     column_blocks_no * row_blocks_no, inst_shape);
42
43  // iterate on the column blocks
44  for (int i = 0; i < grid.y / column_blocks_no; i++){
45
46    // populate B submatrix
47    for (int k = 0; k < args.problem_size.k(); k++){
48      cudaMemcpy(B_sub + k * column_blocks_no * GemmKernel::
     kThreadCount, args.ref_B.non_const_ref().data() + k * args.
     problem_size.n() + i * column_blocks_no * GemmKernel::kThreadCount
     , GemmKernel::kThreadCount * column_blocks_no * sizeof(float),
     cudaMemcpyDeviceToDevice);
49    }
50
51    for (int j = 0; j < grid.x / (3 * row_blocks_no); j++){
52
```

```
53      // populate C submatrix
54      // k1 is the index which keeps count of the number of blocks on
    the rows
55      for (int k1 = 0; k1 < row_blocks_no; k1++) {
56          // k2 is the index which keeps count of the line inside the
    currently selected block k1
57          for (int k2 = 0; k2 < GemmKernel::kThreadCount; k2++){
58            cudaMemcpy(C_sub + k1 * column_blocks_no * GemmKernel::
    kThreadCount * GemmKernel::kThreadCount + k2 * column_blocks_no *
    GemmKernel::kThreadCount, args.ref_C.data() + i * GemmKernel::
    kThreadCount * column_blocks_no + j * GemmKernel::kThreadCount *
    args.problem_size.n() * row_blocks_no + k1 * GemmKernel::
    kThreadCount * args.problem_size.n() + k2 * args.problem_size.n(),
     GemmKernel::kThreadCount * column_blocks_no * sizeof(float),
    cudaMemcpyDeviceToDevice);
59          }
60      }
61      // override args value with the new matrix dimension and the
    submatrices to be passed as parameters
62      arg_int = Arguments({GemmKernel::kThreadCount * row_blocks_no,
    GemmKernel::kThreadCount * column_blocks_no , args.problem_size.k
    ()},  // Gemm Problem dimensions
63          {&args.ref_A.const_ref().data()[j * args.problem_size.k() *
    row_blocks_no * GemmKernel::kThreadCount],args.problem_size.k()},
      // Tensor-ref for source matrix A
64          {B_sub,column_blocks_no * GemmKernel::kThreadCount},    //
    Tensor-ref for source matrix B
65          {C_sub,column_blocks_no * GemmKernel::kThreadCount},    //
    Tensor-ref for source matrix C
66          {D_sub, column_blocks_no * GemmKernel::kThreadCount},    //
    Tensor-ref for destination matrix D (may be different memory than
    source C matrix)
67          {1,1});
68
69      // launch the initialization phase and the kernel itself
70      status = initialize(arg_int, D, workspace, stream);
71
72      if (status == Status::kSuccess) {
73        status = run(stream);
74      }
75
76      // copy back the D submatrix in the corresponding location
    inside D
77      for (int k1 = 0; k1 < row_blocks_no; k1++){
78          // k2 is the index which keeps count of the line inside the
    currently selected block k1
79          for (int k2 = 0; k2 < GemmKernel::kThreadCount; k2++){
80            cudaMemcpy(args.ref_D.data() + i * GemmKernel::
    kThreadCount * column_blocks_no + j * GemmKernel::kThreadCount *
```

```
    args.problem_size.n() * row_blocks_no + k1 * GemmKernel::
    kThreadCount * args.problem_size.n() + k2 * args.problem_size.n(),
     D_sub + k1 * column_blocks_no * GemmKernel::kThreadCount *
    GemmKernel::kThreadCount + k2 * column_blocks_no * GemmKernel::
    kThreadCount, GemmKernel::kThreadCount * column_blocks_no * sizeof
    (float), cudaMemcpyDeviceToDevice);
81          }
82       }
83     }
84   }
85 } else {
86     // do everything in a single kernel call (already shown in
    previous solutions)
87 }
```

The majority of the code consists in the redefinition of the grid size, which depends on the matrix size and the SMs count on the GPU. In order to understand how many kernels are needed to compute the matrix multiplication, we need to compare the number of SMs with the grid size. The comparison between the overall grid size and the number of SMs is done at line 13 and the outcomes can be the following ones:

- The grid is bigger than the number of streaming multiprocessors

  In this case there is the need to divide the matrix in sub-matrices and execute them in different kernels.

- The grid is smaller than the number of streaming multiprocessors

  In this case the execution can be performed as detailed in previous methods, keeping the default grid size.

The division of the matrix has been done in a way such that the number of splits over the columns is minimized. This approach allows us to avoid rearranging matrix elements as much as possible, since the matrix layout is in row major so splitting on the columns would mean to separate a row in two parts. The number of sub-blocks the vertical dimension is divided in is computed at line 18 according to the following formula:

$$row\_blocks\_no = \left\lfloor \frac{SM\_no}{3 * grid.y} \right\rfloor$$

It simply takes the number of SMs and divide it by the y size of the grid (the number of sub-blocks on the horizontal side) in order to compute the new x grid size. At the end of the computation the number of sub-blocks on the horizontal side multiplied by the number of sub-blocks on the vertical side should be equal to $n_{SM}/3$. Here, we assume that the y grid size used to compute `row_blocks_no` will be exactly as the original one, so not as `column_blocks_no`, i.e. the actual y dimension of the grid in the kernel. As already explained, this is done in order to avoid affecting the y size as much as possible. The whole result is divided by 3, because we need to keep into account that thrice the threadblocks will actually be scheduled for each kernel due to the TMR. The number of sub-blocks of the horizontal dimension is given by the formula at line 27:

$$column\_blocks\_no = \frac{grid.y}{\left\lceil \frac{3 * grid.y}{SM\_no} \right\rceil}$$

In this case, the new y grid size is computed by adjusting the old one to make the total number of threadblocks fit the number of multiprocessors. After computing the new kernel grid size, the sub-matrices that will be used for sub-blocks computation are declared (line 31). B, C and D sub-blocks

are allocated with a sizing that allows for the computation of the number of threadblocks scheduled in a single kernel call. There is no need to allocate an A matrix sub-block, because A is parsed by rows, so it is enough to provide a pointer to the first row among the ones of the sub-block to select a sub-matrix. TMR copies of the D matrix are allocated as well (line 38) and they act exactly as `D_sub`, the computed D sub-matrix, but for the TMR copies. After these declarations, the actual computation is performed. It is done again with two nested for loops, but this time they are used to iterate over the sub-blocks in the horizontal dimension (i.e. over the columns at line 44) and in the vertical one (i.e. over the rows at line 51). For every iteration over the columns the B sub-matrix has to be rebuilt (line 47), because we are selecting sub-blocks which corresponds to different groups of columns in the matrix. Since this operation is time consuming, iterating on the column sub-blocks in the outer loop is the solution which optimizes performances. Inside the inner for loop, the C sub-matrix is populated too and, in order to do so, we need the indication of both the column and the row sub-blocks to be selected. The intersection between the currently selected group of rows and the currently selected group of columns identify the C sub-matrix, as well as the D matrix, over which the computation is performed for the current kernel call. The formulae used to address this part are not explained, but they can be easily built by identifying how threadblocks and threads are scheduled to manipulate each element in the sub-matrices. After identifying the correct C sub-matrix, the `args` structure needs to be updated with the new problem size:

$$problem\_size = (block\_size_x * row\_blocks\_no, block\_size_y * column\_blocks\_no, K)$$

The M dimension is given by the number of rows in each sub-block multiplied by the number of sub-blocks over the rows in the current sub-matrix, the N dimension is given by the number of columns per sub-block multiplied by the number of sub-blocks over the columns in the current sub-matrix, while K is kept the same because we are still iterating over a whole row for matrix A and a whole column for matrix B. Since K is the same, the kernel will completely build some of the matrix blocks, making it seem like if there were no checkpoint mechanism at play. Actually, the checkpoints over N_{iter} are placed inside `threadblock/mma_pipelined.h` as we will see later. After the preparations are done, the initialization sequence and the kernel itself are ready to be launched. When the execution is concluded, the results are fetched from the D_sub sub-matrix and they are copied back in the corresponding location of the original D matrix (line 77). The value in D_sub are always correct, because corrections are performed inside the kernel after each one of the N_{iter} checkpoints depending on the values of D_sub, D[0] and D[1]. At the end of the outer for loop, the results will be ready in matrix D.

The new kernel invocation is done in this way:

```
1 cudaLaunchCooperativeKernel((void *)cutlass::Kernel<GemmKernel>, grid,
      block, kernelArgs, smem_size, stream);
```

The grid is defined as discussed before while `kernelArgs` is a wrapper for `params_`.
`cudaLaunchCooperativeKernel()` is in charge of performing some feasibility checks to assess if the grid is compatible with the conditions needed for cooperative groups to work properly before launching the kernel. In case these checks fail, a `cudaError` will be thrown.

As anticipated, there is the need to map multiple threads to the same matrix region in order to allow different threadblocks, corresponding to different TMR instances, to compute the same matrix elements. The association sub-block to threadblock is done through the function `get_tile_offset`, which uses the threadblocks ids to compute the coordinate of the first element in the D sub-block associated to the threadblock. As a part of the new grid definition, newly added threadblocks for the TMR have been concatenated on the x dimension (the vertical one, along the rows). Threadblocks are mapped to the same sub-block in a modular way, which means that the threadblock with block_idx_x = 0, the threadblock with block_idx_x = N_{blocks} and the one with block_idx_x = $2N_{blocks}$ will be mapped to the same sub-block. This is evident starting at line 7 of the following snippet:

```
static GemmCoord get_tile_offset(int log_tile, GemmCoord tiled_shape)
   {
    int block_idx_x = RematerializeBlockIdxX();
    int block_idx_y = RematerializeBlockIdxY();
    int block_idx_z = RematerializeBlockIdxZ();

    // assign redundant threads to the same matrix positions as the
   original ones
    return GemmCoord{((block_idx_x % tiled_shape.m()) >> log_tile),
   //
                     (block_idx_y << log_tile) + ((block_idx_x %
   tiled_shape.m()) & ((1 << (log_tile)) - 1)),
                     block_idx_z};
}
```

The only thing that is missing is the inner part of the kernel. This part is more or less the same as the default `threadblock/mma_pipelined.h` file, with the addition of checkpoints and grid synchronization through cooperative groups primitives. Here are shown only the parts added to the original library that have to be executed at the end of each outer loop iteration (i.e. the one on N_{iter}):

```
// write results to the destination matrix
  // compute the index where to write
  int index = grid_size.n() * tile_offset.m() * blockDim.x +
   tile_offset.n() * blockDim.x  + threadIdx.x;

  // perform the actual write over destination, only if you are not
   master thread
  if (!is_master)
    for (int i = 0; i < accum.kStorageElements; i++)
      destination[index * accum.kStorageElements + i] = accum.data()[i
   ];

  // synchronize all threads in the grid
  cg::sync(grid);

  // perform checks in the master threads to assess which thread
   produced the right results
  int mas_copy1 = 1;
  int mas_copy2 = 1;
  int copy1_copy2 = 1;
  if (is_master){
    for (int i = 0; i < accum.kStorageElements; i++){
      if (accum.data()[i] != D1[index * accum.kStorageElements + i])
        mas_copy1 = 0;
      if (accum.data()[i] != D2[index * accum.kStorageElements + i])
        mas_copy2 = 0;
      if (D1[index * accum.kStorageElements + i] != D2[index * accum.
   kStorageElements + i])
        copy1_copy2 = 0;
    }
  }
```

```
27
28  // correct the wrong results
29  if (is_master){
30    if (mas_copy1 == 1)
31      // adjust D2
32      for (int i = 0; i < accum.kStorageElements; i++)
33        D2[index * accum.kStorageElements + i] = accum.data()[i];
34    else if (mas_copy2 == 1)
35      // adjust D1
36      for (int i = 0; i < accum.kStorageElements; i++)
37        D1[index * accum.kStorageElements + i] = accum.data()[i];
38    else if (copy1_copy2 == 1)
39      // adjust master
40      for (int i = 0; i < accum.kStorageElements; i++)
41        accum.data()[i] = D2[index * accum.kStorageElements + i];
42  }
43  // synchronization point to force the non master threads to wait for
      the master
44  cg::sync(grid);
45  // load the new results back in the local accumulators
46  if (!is_master)
47     for (int i = 0; i < accum.kStorageElements; i++)
48       accum.data()[i] = destination[index * accum.kStorageElements +
     i];
```

As already mentioned, each threadblock is assigned either a *master* or a *slave* role. The threads that belong to slave threadblocks are in charge of copying the intermediate results from their local accumulators to a global destination matrix (D[0] or D[1]), in order to make them accessible to the master which will performs the TMR checks (line 6). After this operation a first synchronization point is necessary (line 11) because slave and master threads do not run on the same SM. Once all threads in the three threadblocks reach this point, the master will perform the checks over the results (line 17). Depending on the outcomes, there may be the need for the master to correct some wrong results (line 29). Once everything is done, another synchronization (line 44) is needed because the slave threads can resume working only when the intermediate results are ready. After passing this point, slave threads should reload the intermediate results back into their local accumulators (line 46).

4.5.1 Advantages and disadvantages

This solution is the most complete and complex among the proposed ones. It allows for a good response both to transient faults, because of the repeated execution, and to permanent faults, since it is the only one that guarantees execution of the TMR copies on different multiprocessors. The major limitation of this solution is related to the need to schedule massive amounts of kernels in case the matrix size becomes too big. Unfortunately, relying on cooperative groups is the only way to perform synchronization at the threadblock level. This is something that is needed to implement proper checkpoints in a solution with the described approach. Because of its power in terms of error correction capabilities, however, this solution allows to produce error-free results in critical computations. Obviously, the price to pay is the potential performance limitations. If the system needs real-time computations, it is possible to use a more lightweight solutions like the second one, which offers a good protection against transient faults but is highly vulnerable to permanent ones.

CHAPTER 5

Profiling

This chapter is devoted to the presentation of the profiling results of each solution. These data are going to be compared with the original Cutlass GEMM and a naive GEMM kernel (reference GEMM kernel). The size of the matrix used for the benchmark is $(4096, 4096, 4096)$ since it allows to show the significant performance differences between the CUTLASS kernel and the reference kernel. Some tables that summarize the most significant results are provided in the following sections.

5.1 Solution 1

Kernel used	Algorithm steps	Number of calls	Intermediate results (ns)	Results (ns)
TMR compliant GEMM kernel	GEMM Kernel	3	44.340.332	54.254.158
	Comparison Kernel	3	1.450.583	
	Reduction kernel	72	8.463.243	
Original Gemm Kernel	-	1	13.527.795	
Reference Gemm Kernel	-	1	1.073.319.824	

Since this solution consists in a simple repetition of the CUTLASS kernel call, the overall duration amounts to around thrice the duration of the original kernel. The additional overhead is mainly caused by the reduction kernel, which is strongly sub-optimal since it relies on multiple kernel calls. It would be possible to optimize it either by using cooperative groups or by performing multiple kernel calls with maximally sized threadblocks. However both these techniques would still be responsible for multiple kernel calls in case large matrices are manipulated. This solution is the one with the smallest overhead and with the most limited correction capabilities due to the absence of checkpoints.

5.2 Solution 2

Kernel used	Algorithm steps	Number of calls	Intermediate results (ns)	Results (ns)
TMR compliant GEMM kernel	-	1	-	1.866.731.859
Original Gemm Kernel	-	1	-	13.527.795
Reference Gemm Kernel	-	1	-	1.071.913.830

Due to the necessity to perform checks inside the kernel, this solution is responsible for a significant increase in the kernel duration. It is possible to partially compensate for this increase by reducing the amount of checkpoints, however this would negatively affect the correction capabilities. The best checkpoint granularity has to be defined depending on the fault frequency, whose effect on the computation could be studied by relying on some proper fault injection tool. Reducing the overhead associated with the comparisons could be very difficult, because they rely on for-loops which can potentially introduce divergent behavior in case of faults. The powerful feature of this solution is related to the possibility to adapt it to different fault frequencies, making it able to compensate for almost any kind of transient fault and for multiple faults. The only criticality is the scenario where a thread is affected by the same error during the computation of the same element over two of the TMR instances. However, the probability of this event is almost zero.

5.3 Solution 3

Kernel used	Algorithm steps	Number of calls	Intermediate results (ns)	Results (ns)
TMR compliant GEMM kernel	GEMM Kernel	384	969.370.680	2.239.077.493
	Comparison Kernel	384	185.807.116	
	Reduction kernel	9.216	1.083.899.697	
Original Gemm Kernel	-	1	-	13.527.795
Reference Gemm Kernel	-	1	-	1.073.465.191

The third solution affects considerably both the duration of the GEMM kernel, mainly because of the need to call it multiple times, and the overhead, due to reduction kernel calls. Because of this, the solution is the one that could benefit the most from an improvement in the reduction kernel structure. In terms of performances, it is in the middle between the second and the fourth one, adding some improvements in terms of permanent fault detection with respect to the former, but not reaching complete protection as the latter. Also in this case, decreasing the number of checkpoints could affect the performances in a positive way, since it reduces the number of kernel calls.

5.4 Solution 4

Kernel used	Algorithm steps	Number of calls	Intermediate results (ns)	Results (ns)
TMR compliant GEMM kernel	-	1	-	5.419.873.818
Original Gemm Kernel	-	1	-	13.527.795
Reference Gemm Kernel	-	1	-	1.070.939.856

This solution sacrifices performances even further to guarantee potentially the same amount of protection against transient faults as the second solution while hardening the algorithm against permanent faults too. The price to pay is another increase in execution time, which is again due to the repeated kernel invocations. This time it is even harder to perform optimization, because the number of kernels to call depends only on the matrix size and the number of streaming multiprocessors, which are related to the problem formulation and to the device respectively.

Bibliography

[1] Nvidia, *CUTLASS*. Reference: `https://github.com/NVIDIA/cutlass`

[2] Project repository. Reference: `https://github.com/Davide-Giuffrida/CUTLASS.git`

[3] Nvidia, *Cooperative Groups: Flexible CUDA Thread Programming*. Reference: `https://developer.nvidia.com/blog/cooperative-groups/`