GPU programming
project presentation

Triple Modular Redundancy on CUTLASS

- Davide Giuffrida
- Matteo Fragassi

Overview

- Introduction
- Background
- Proposed Solutions
- Profiling

INTRODUCTION

Underlying topic

- **Problem**

  Under stressful conditions, GPUs may be affected by a variety of **transient** and **permanent faults** that, over time, can lead to failures.

- **Solutions**

  Implement **error detection** and **correction** mechanisms to tackle possible failures.

  Solutions can be very different depending on the targeted fault. For example, memory corruptions are usually detected, and possibly corrected, using error correction codes (ECC) while computational errors can be dealt with in many more ways.

About this project

Focus on **transient** and **permanent faults** that can generate computational errors.

Triple Modular Redundancy (**TMR**) at different levels as protection mechanism to prevent failures.

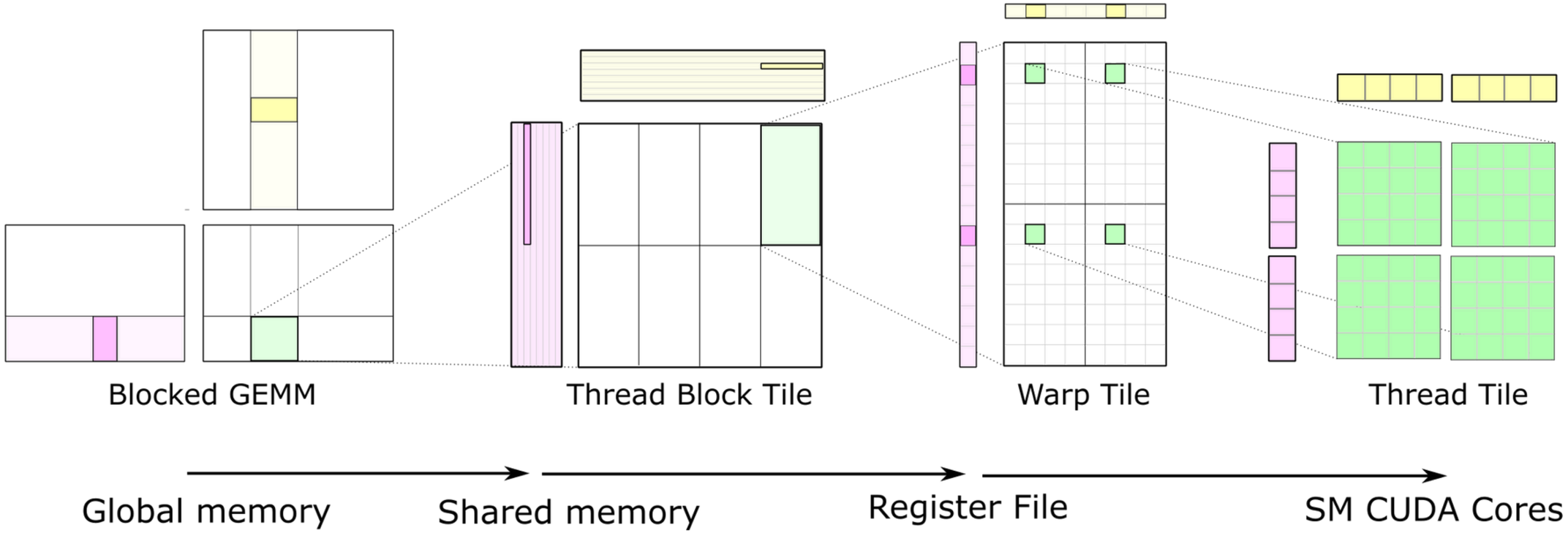
CUTLASS

Nvidia open-source library for high performance matrix-matrix computation (GEMM) using CUDA C++.

BACKGROUND

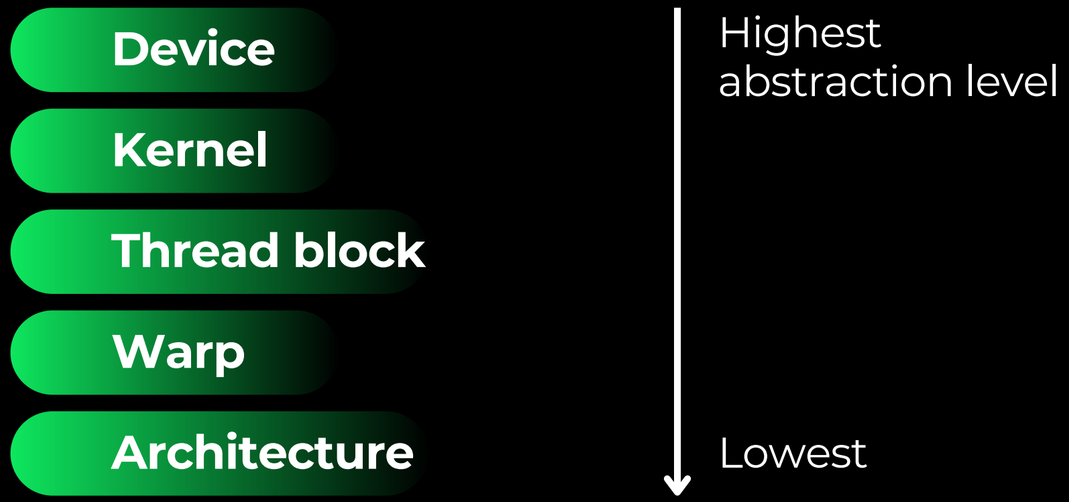
CUTLASS GEMM

A **GEMM operation** performs the computation A x B, where A and B are matrices of respectively M x K and K x N elements.



CUTLASS

The **Cutlass GEMM algorithm** is developed on the entire thread block-warp-thread hierarchy. The organization of the library replicates the logical decomposition of the GEMM operation.



Every proposed solution is integrated in the existing Cutlass structure so the GEMM can be executed by calling the default operators of one of the provided C++ templates.

Triple Modular Redundancy

TMR is an N-modular redundancy algorithm that ensures fault detection/correction by **executing a computation three times** before producing an **output based on a majority-vote** process.

This approach allows to mask errors that affect only one of the copies and to detect errors in multiple copies provided that they are different.

Performance-Reliability trade-off

Higher error frequencies can lead to error accumulation even if the same computation is performed multiple times.

To increase the reliability, some **checkpoints** can be inserted along the computation in order to perform **TMR on partial results**.

PROPOSED SOLUTIONS

Solution #1

TMR at the kernel call level

The solution is completely implemented inside one of the cutlass::gemm::device::Gemm<> templates*, located at the highest level of the library: the *device* level.

Inside the template

1. Initialization of the kernel parameters and instantiation of the additional data structures needed to perform TMR
2. Call of **3 GEMM kernels** that run in parallel on different streams
3. Wait for kernels completion with cudaDeviceSynchcronize( )
4. Majority-vote with **comparison and reduction kernels**.

* The C++ template called in this solution is invoked also in other solutions to start the GEMM execution.

Solution #1

TMR at the kernel call level

This is the **simplest solution** among the proposed ones .

It offers **protection against transient faults**, provided that the error frequency does not create critical fault accumulation, and no valuable protection against permanent faults.

The **performance overhead*** depends on how the GEMM kernels are executed and on the efficiency of the comparison and reduction operations.

Best case overhead ~ 0

Worst case overhead > x2

* Cutlass GEMM kernel used as time reference

Comparison kernel

- Device code

```
__global__ void CompareMatrix_kernel(
   float *dstMatrix,
   float *srcMatrixA,
   float *srcMatrixB,
   int elements) {

   int i = threadIdx.x + blockIdx.x * blockDim.x;
   if(i < elements){
      dstMatrix[i] = (srcMatrixA[i] == srcMatrixB[i])? 1:0;
   }
}
```

- Host code

```
CompareMatrix_kernel<<<grid,block,0,streams[0]>>>
(tmp[0],D[0],D[1],args.problem_size.m()*args.problem_size.n());
```

Reduction kernel

- Device code

```
__global__ void ReduceMatrix_kernel
( float *matrix, int elements, int elem_dist){
   int ja = threadIdx.x * 2 + blockIdx.x * blockDim.x * 2;

   if(elem_dist == 1 || (ja % (elem_dist*2)) == 0)
      if(ja < elements)
         if(ja + elem_dist < elements)
            matrix[ja] = matrix[ja] + matrix[ja + elem_dist];
}
```

- Host code

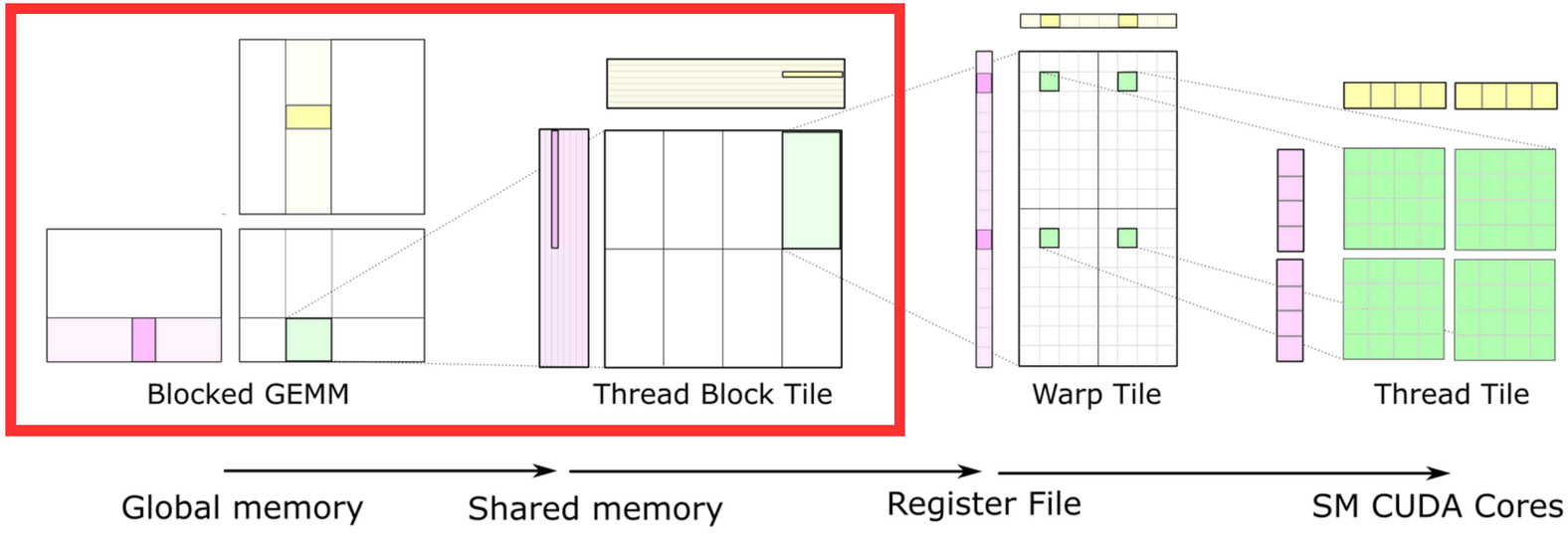
```
for (int i = 0; i < int(log2(args.problem_size.m() * args.problem_size.n())); i++){
   ReduceMatrix_kernel<<<grid_reduce,block_reduce,0,streams[0]>>>(..., pow(2,i));
   // Kernel calls for the other TMR instances (NOT SHOWN)
   cudaDeviceSynchronize();
}
```

Solution #2

TMR at the thread block level

This solution is implemented at the *thread block* level, so the **TMR** is performed **inside the GEMM kernel**.

Inside the **prologue** the global memory fragments are initialized by the (global) memory iterators. The matrix tile of interest is then loaded in shared memory (smem) and the correponding warp tile iterators and fragments are initialized.



Solution #2

TMR at the thread block level

The **core of the solution** starts by initializing the accumulators needed to hold the partial GEMM results of the warp-level TMR instances before iterating over the previously loaded tile.

Example of **matrix tile computation**:

```
// GEMM Mainloop - iterates over the entire K dimension - not unrolled
for (int kblock = 0; kblock < K_dim; kblock += BlockItemsK) {
  // Load A and B tiles from global memory and store to SMEM (not shown)

  // Warp tile structure - iterates over the Thread Block tile
  #pragma unroll
  for (int warp_k = 0; warp_k < BlockItemsK; warp_k += WarpItemsK) {
    // Fetch frag_a and frag_b from SMEM corresponding to k-index (not shown)

    Warp_mma(fragment[0]); Warp_mma(fragment[1]); Warp_mma(fragment[2]);
```

Solution #2

TMR at the thread block level

The actual implementation pre-loads the next matrix tile in the first iteration of the inner loop and moves it to the shared memory in the last iteration.
This allows to **parallelize memory operations and GEMM computation**.

Despite the optimizations the TMR checks introduced after every warp computation create a **significant performance overhead**.

Overall solution #2 offers **better protection against transient faults** than solution #1, but it **cannot detect** errors in case of **permanent faults** since the TMR on each matrix tile is executed on a single SM.

Solution #3

TMR at the thread block and kernel call levels

The basic idea is to call **a number of GEMM kernels** at the *device* level **equal to the number of iterations** that there would be at the *thread block* level **over the dimension K**. This is done for each TMR instance.

This solution combines the approaches of the previous ones, **increasing the complexity of the implementation**.

Even though there are less TMR checkpoints than solution #2, the overhead of multiple kernel launches and comparison and reduction kernels is responsible for **poorer performances**.

Better protection against **transient faults** than solution #1
Better protection against **permanent faults** than solution #2

Solution #3

TMR at the thread block and kernel call levels

Device level

```
// GEMM Mainloop - iterates over the entire K dimension - not unrolled
for (int kblock = 0; kblock < K_dim; kblock += BlockItemsK) {
  cutlass::Kernel<GemmKernel><<<grid, block, smem_size, stream>>>(params_);
```

Thread block level

```
// Warp tile structure - iterates over the Thread Block tile
#pragma unroll
for (int warp_k = 0; warp_k < BlockItemsK; warp_k += WarpItemsK) {
  // Fetch frag_a and frag_b from SMEM corresponding to k-index (not shown)*
  Warp_mma();
```

* At the *thread block* level, the load in shared memory is performed once in the prologue

Solution #4

TMR w/ cooperative groups

This solution is based on **multiple kernel calls** that execute **multiple thread blocks on different SMs** with **cooperative groups**.

Cooperative groups provide **synchronization between different thread blocks** but, in order to work, they need **all thread blocks of the kernel to run concurrently** on the GPU.

For this reason the result matrix is divided in **sub-matrices**, with each sub-matrix computed by a kernel. The sub-matrices can also be divided in **sub-blocks** to speed-up the computation if the number of SMs allows it.

Each sub-block (or **sub-matrix**) **is computed three times** by different thread blocks running concurrently on different SMs.

Solution #4

TMR w/ cooperative groups

Kernel invocation

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

The grid size is redefined according to these constraints

- The **number of thread blocks must lower than or equal to the number of SMs**, to avoid deadlocks during synchronization
- Sub-matrices identification must be done **avoiding to split rows** as much as possible, since the matrices are assumed to be row-major.

The **performances** are explicitly affected by the compute capabilities of the GPU, since the number of kernel calls depends on the number of SMs.

Solution #4

TMR w/ cooperative groups

Mapping thread block on sub-blocks

The allocation of additional thread blocks for the TMR is achieved through the redefinition of the grid

```
dim3(tiled_shape.m() * tile * 3, (tiled_shape.n() + tile - 1) / tile, tiled_shape.k())
```

New thread blocks are added on the x dimension of the grid so it is necessary to change the logic which maps a thread block to the corresponding sub-block

```
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}
```

block_idx_x is replaced with the modulo between the actual id and the original grid size on the x dimension, assigning three threadblocks to the same sub-block.

Solution #4

TMR w/ cooperative groups

Thread block level

The *thread block* level has been modified to divide threads from TMR instances in **master** and **slaves**. Slaves threads copy their intermediate results in global matrices at the end of every iteration over K so that the master is able to access them to perform comparisons.

The grid synchronization primitive **cg::sync(grid)** allows the master to wait for end of the slaves copy.

This primitive is also placed after the master has corrected wrong intermediate results, so that the slaves can reload them from the global matrices and use them during the next iteration over the dimension K.

Profiling results

Execution times

Kernel used	Algorithm steps	Results (ns)
Custom solutions	Kernel level	54.254.158
	Threadblock level	1.866.731.859
	Multiple kernels per block	2.239.077.493
	Cooperative groups	5.419.873.818
Original Gemm Kernel	-	13.527.795
Reference Gemm Kernel	-	1.073.319.824

Thank You