Protecting GPU applications via CUSPIS

Francesco Galbiati Tiya Jreige

September 10, 2025

Contents

1 Project data

- Project supervisor(s): Davide Baroffio
- Describe in this table the group that is delivering this project:

Last and first name	Person code	Email address
Francesco Galbiati	10728028	francesco4.galbiati@mail.polimi.it
Tiya Jreige	10958178	tiya.jreige@mail.polimi.it

- Describe here how development tasks have been subdivided among members of the group:
  - We worked together on the project, without splitting it in parts, mainly because of the monolithic nature of the task.
- Links to the project source code: https://github.com/Francesco4Galbiati/AOS_Project_2025_Galbiati_Jreige

2 Project description

This project aims to implement redundancy mechanisms on a CUDA-based benchmark that models molecular dynamics. The benchmark simulates interactions between particles in a 3D space by dividing the space into boxes of particles and focusing the computation between one of the boxes, called the home box, and its neighbouring boxes.

The main contribution of the project is to make the GPU computations of the benchmark fault tolerant, by implementing redundancy mechanisms using the CUSPIS library. This library provides a set of APIs that encapsulate CUDA functions to support duplicated and triplicated execution. The CUSPIS library provides replication at three levels:

- **Kernel-Level Redundancy**: The entire kernel is launched many times via different streams.
- **Block-Level Redundancy**: There is spatial replication at the block level, which means that redundant blocks work on the same data.
- **Thread-Level Redundancy**: Replication occurs at the thread level. Therefore, the number of threads is replicated inside the blocks where the replicated threads execute the same instruction independently.

The Advanced Operating System course provided a strong foundation that helped us understand the system-level behaviour of the benchmark, particularly its core working mechanism. It also introduced us to key concepts that were applicable during the project implementation. These include the following:

- **Resource Management**: When replicating executions, it is essential to appropriately allocate the resources to the replicas, in order to ensure correct execution. For instance, we need to ensure that each replica has its own resources, e.g. its own registers, its own share of the shared memory, and is appropriately allocated SM resources to ensure independent execution without interference.
- **Concurrent Execution**: There is a need to ensure safe parallel execution. For instance, the use of "__syncthreads()" function in the code ensures that all the threads inside a block have all reached the same point of the execution before any of them can proceed. One example of this synchronisation is applied in the simulation after copying the data from global memory to shared memory, ensuring that all threads within a block have finished the data transfer before proceeding with the computation that relies on the data in the shared memory.

2.1 Design and implementation

This section provides an explanation of the implementation of redundancy mechanisms in the CUDA-based benchmark using the CUSPIS library APIs.

The first change applied to the benchmark's code is the inclusion of the CUSPIS library `"CUSPIS.cuh"` and the definition of the macro `NUM_REPLICAS CUSPIS::NUM_REPLICAS`.

Following that, the GPU memory setup was modified to support redundant executions. Therefore, standard CUDA functions for memory allocation were replaced with the CUSPIS APIs wrappers that handle memory allocation with respect to the number of replicas. The modified `gpu_memory_setup()` function is available here. The same was applied for memory deallocation as shown here. In addition, the shared memory in the kernel was updated to allocate memory based on the number of replicas as shown here.

In the CUPIS library, we have that:

- Invoking the `cuspisMalloc(devPtr, size)` wrapper is equivalent to invoking `cudaMalloc(devPtr, size * NUM_REPLICAS)`, as shown here, and the appropriate deallocation wrapper is shown here.
- Invoking `cuspisMemcpyToDevice(...)` is equivalent to looping over the number of replicas and invoking `cudaMemcpy(..., cudaMemcpyHostToDevice)` to copy the host content in the different replicas, where each replica starts at the `destination_pointer + data_size * the_loop_iteration_index`. The wrapper function is shown here.
- Invoking the `cuspisMemcpyToHost(...)` wrapper is equivalent to invoking `cudaMemcpy(...,cudaMemcpyDeviceToHost)`, in addition to comparing the result of the replicated executions in case the number of replicas > 1. The implementation of this wrapper is shown here.

Then in the `main(...)` function, we updated the GPU setup, in the following way `cudaMemset(..., dim_cpu.space_mem * NUM_REPLICAS)` as shown here. In addition, we are launching the kernel using the CUSPIS kernel wrapper that supports redundant execution by specifying the redundant policy to be applied during the execution. The kernel is launched using `kernel_b(..., CUSPIS::CUSPIS_Redundant_Policy)` as shown here.

After applying these setups in order to launch the kernel with the Kernel redundant policy, we need to specify the redundant policy as `CUSPIS::cuspisRedundantKernel` when launching the kernel. Then we need to update the number of streams in here to 1,2, or 3, depending on the number of redundant kernel executions that we would like to execute. In this way, the kernel is set to be launched with kernel redundant execution and the CUSPIS `modify()` function takes care of shifting to the replicated location in memory before launching the next redundant kernel execution.

Now, in order to launch the kernel with redundant threads or blocks, we need to further modify the kernel code. In the original code without redundancy, we had that each box in the 3D space is associated with a block, and the particles of this box are executed in parallel by different threads in the appropriate block. To support redundancy, the following modifications were applied to the kernel:

- The following condition `if( bx < d_dim_gpu.number_boxes * NUM_REPLICAS)` was modified here, by multiplying it by `NUM_REPLICAS`, to ensure that all boxes, including the replicated ones, will be associated with a block since `bx` represents the block ID.
- Then the variable `first_i = d_box_gpu[bx].offset`, which represents the starting index of the first particle in a box, was modified to the version shown here.
  - The indexing of `d_box_gpu[...]` was modified to support thread redundancy, since it already supports block redundancy, because each block is associated with a box. In case of thread redundancy, `thread_replica`, declared here, represents which replica of the original thread is being executed, ensuring that replicated threads access replicated boxes at the correct addresses, applying an offset to the original data location.
  - The `d_box_gpu[...].offset` of replicas matches the one of the original box, and not the actual first particle location of each replica box, since they are exact replicas of the data of the original box. Therefore, an additional offset is added, based on the thread or block replica index, represented as `block_replica` declared here, to point to the first replicated particle.
- Then the loop responsible for copying data from global memory to the shared memory of each block was modified as follows here. Each block corresponds to a box, and each thread within a block copies the data of one or more particle, located at a certain offset from the starting address `ra` which points to first particle's data in that box. In case the number of threads $\geq$ total number of particles in a box, each thread will copy the data of a single particle from the box. Each thread, therefore, accesses the appropriate particle's data at an offset equal to the thread index modulo the original total number of threads to ensure that replicated threads start at an offset of 0 from `ra` in their replicated boxes. To also ensure that each thread copies the data in a different location in shared memory, the data is copied starting at an offset equal to the number of particles per box multiplied by the thread replica index. However, when the number of threads $\leq$ the number of particles in a box, each thread is responsible for copying the data of more than one particle spaced by the total number of threads until the data of all particles are copied.

  Now we need to copy the data associated with the neighbours of the current box in shared memory. The bound of the loop that goes over the neighbours of a given box was modified as shown here, and the pointer to the neighbour box was also updated as shown here to ensure that the correct box is accessed, in replicated execution. For copying the data of the neighbours particles, the same modifications were done as the ones of the home box, as shown here.

  For the computation section between each of the home box particles and the particles of the neighbouring boxes, modifications were applied to ensure correct memory access in the presence of redundant execution. For instance, the replicated threads access the data of the replicated boxes at an offset equal to the thread or block replica index multiplied by the number of particles per box from the original data, for both the home and neighbouring boxes, as shown here. In global memory, we need to ensure that the results are written back to boxes starting at offset 0 from the address specified at `fa` so the following modifications shown here were applied.

3 Project outcomes

3.1 Concrete outcomes

The results of our work can be consulted in the GitHub repository linked at the start of this document. The repository itself contains the following files:

- `cuspis_lava_kbtr.cu`: this file contains the lava benchmark adapted to be used with the CUSPIS library, including the CUDA kernel, the main and utility functions and the setup parameters.

- `cuda_lava.cu`: this is the original benchmark to be run with CUDA, it can be used for comparing the results of the CUSPIS adaptation with the original code.
- `half.hpp`: C++ header file containing the definition of the half precision floating point numbers, it's necessary to compile and run both the CUDA and the CUSPIS versions of the lava benchmark
- `compile_cuda.sh`: shell script that compiles the CUDA code from `cuspis_lava_kbtr.cu` into code runnable with CUSPIS.
- `cuspis/cuspis.cuh`: the source code of the CUSPIS library, necessary to compile CUDA into CUSPIS.

These files can be downloaded and run as they are, either in a local or cloud environment (such as Google Colab).

Our version of the benchmark compiles and runs without error, providing the correct results to the user in the form of a binary file. It should be noted, however, that CUDA often struggles with high-precision floating-point notations. It sometimes happens that a few (generally one or two) values of the results are off by 0.00001 between the CUSPIS and the CUDA version; we do not impute this difference to a coding error by us, but rather to an occurrence due to the handling of the floating point notation by CUDA.

3.2 Learning outcomes

Thanks to this project we had the opportunity to deepen our understanding on the topics of GPU programming with CUDA, which is an increasingly relevant tool in the modern-day scientific landscape that allows for more efficient computations with respect to classical architectures, and the challenges this area of research entails, by forcing us to reason in a massively concurrent way, rather than in a sequential one, to which we have been more exposed during our studies at Politecnico di Milano.

Another topic we expanded our knowledge of is the different kinds of redundancy possible in a complex scenario, we believe this topic is not limited only to the GPU or Operating Systems areas, but spans other seemingly unrelated topics such as distributed computations and system architecture. By undertaking this project we could challenge ourselves and experiment with the integration of different kinds of redundancy to better understand the strength of each one.

Ultimately, we were also able to test our teamwork skills and our capabilities in confronting and discussing ideas during meetings.

3.3 Existing knowledge

Pre-existing knowledge on the topics of the project was mainly composed of university courses (both curricular and extracurricular) that we took during our degrees, in particular (aside from Advanced Operating Systems):

- Francesco Galbiati: Embedded Systems, Advanced Computer Architectures, Advanced Algorithms and Parallel Programming, GPU101 (Passion in Action)
- Tiya Jrige: Embedded systems, Hardware architecture for embedded edge AI, Hardware accelerators, computer graphics, Advanced Computer Architectures

3.4 Problems encountered

The main issues we encountered during the project were due to our unfamiliarity with the benchmark itself, since it's complicated in its structure and not well-documented. The parts we struggled with the most were the ones where the requirements for the different kinds of redundancy overlapped, and we needed to think about how to reconcile the different solutions we had thought of for each one, especially in some corner cases concerning the shared memory that were not immediately apparent.

4 Honor Pledge

We pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents my/our original production, unless otherwise cited.

We also understand that this project, if successfully graded, will fulfil part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2025.

Group Students' signatures