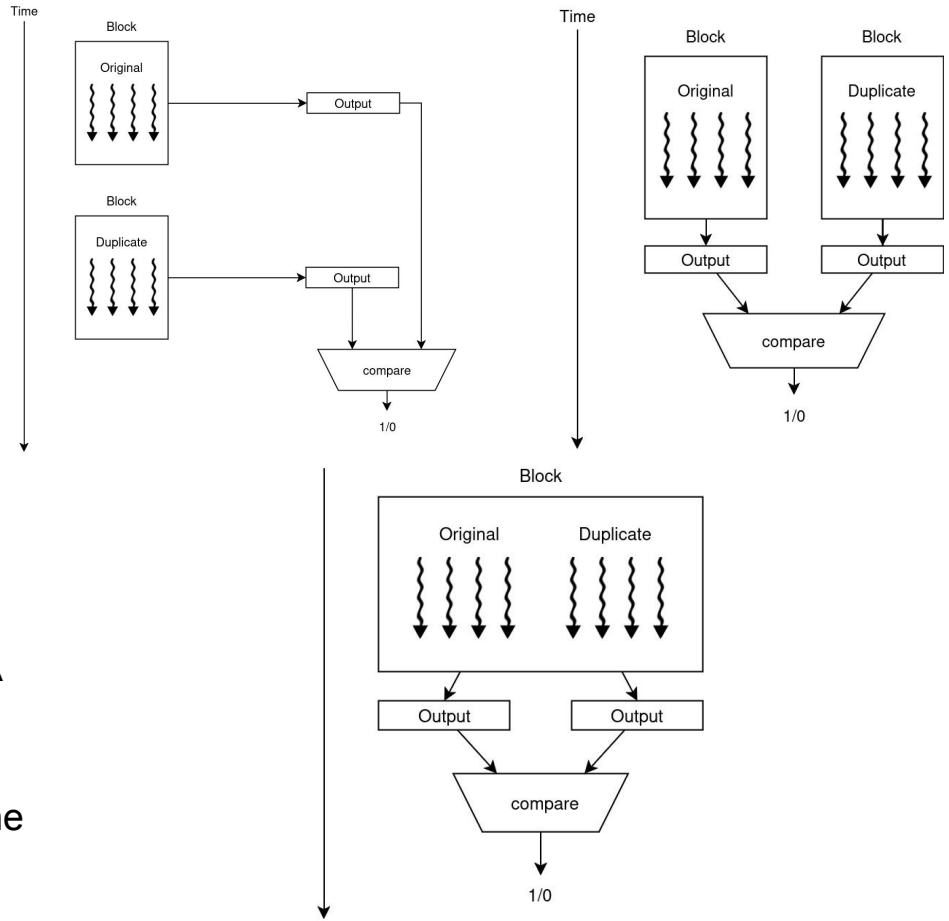
POLITECNICO
MILANO 1863

Protecting GPU applications via CUSPIS

Francesco Galbiati 10728028,
Tiya Jreige 10958178

Context: CUSPIS

- Library that implements execution redundancy for CUDA kernels
- Makes available 3 kinds of policies:
  - Redundant Kernels
  - Redundant Blocks
  - Redundant Threads
- Memory allocations and kernel launch handled with dedicated wrappers of CUDA functions
- Redundant executions are compared by the library



Images taken from https://github.com/HEAPLab/ASPIS/tree/master/cuspis/figures

Project Objectives

- Goal:
  - Implement redundancy mechanisms on a CUDA-based molecular dynamics benchmark.

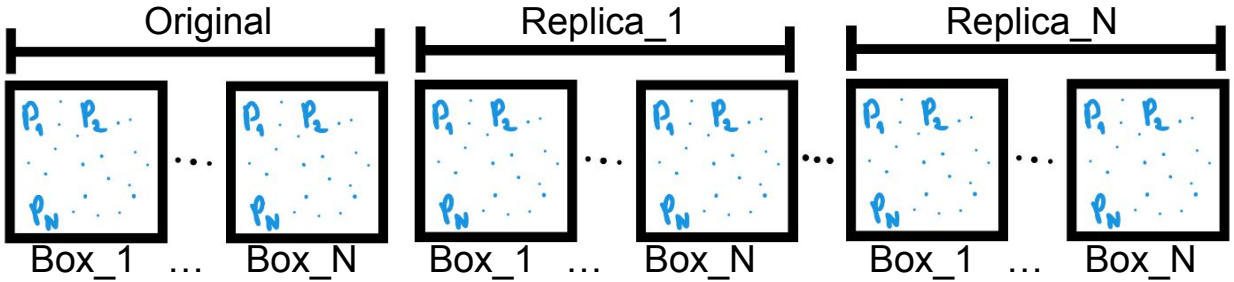
- Benchmark
  - Simulates particle interaction in a 3D space
  - Divide the space into boxes

- Contribution:
  - Fault-tolerant GPU computations using CUSPIS library
  - Support of duplicated and triplicated executions

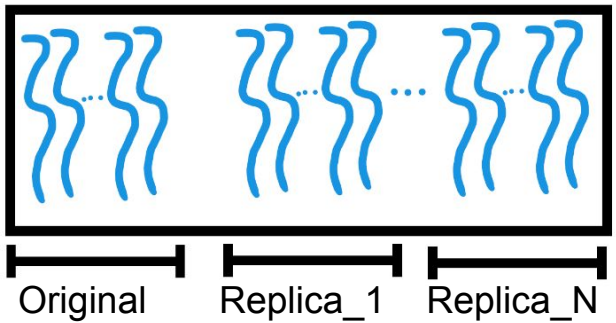
Problems

- The main problems encountered during the project were of two natures:
  - Problems regarding memory addresses:
    - Shared memory in thread redundancy was not enough for more than one replica
    - Indexes read in the replicated area of memory pointed to non-replicated memory locations
    - Several variables were accessed by thread or block index, generating conflicts in the case of replicas
  - Incompatibility between redundancy policies:
    - The different redundancy policies required different behaviours in some lines of code to correctly replicate the execution

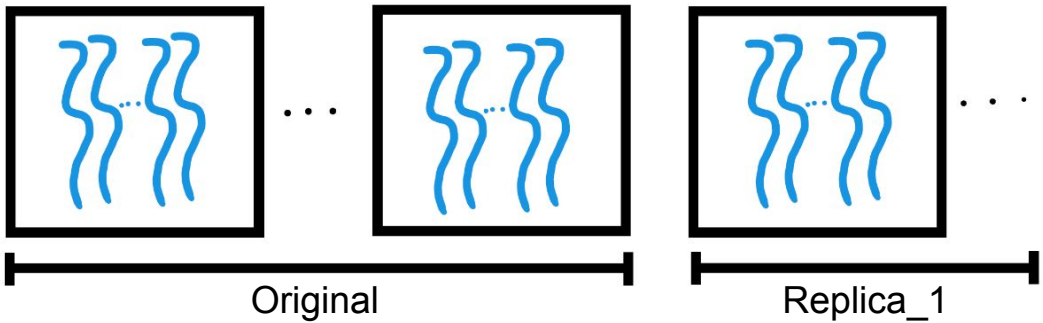
Solution



Thread Redundancy



Block Redundancy



Solution

- Offset to access the first replicated particle with respect to the block/thread replica
  - `int block_replica = bx / d_dim_gpu.number_boxes + tx / NUMBER_THREADS;`
- Access to the appropriate box for the thread replica
  - `int thread_replica = tx / NUMBER_THREADS;`
- Pointer to the first particle in a box:
  - `first_i = d_box_gpu[bx + thread_replica * d_dim_gpu.number_boxes].offset + ( block_replica * NUMBER_PAR_PER_BOX * d_dim_gpu.number_boxes);`
  - `first_j = d_box_gpu[pointer + thread_replica * d_dim_gpu.number_boxes].offset + ( block_replica * NUMBER_PAR_PER_BOX * d_dim_gpu.number_boxes);`

Solution

- Modulo ensures that replicated threads start at an offset of 0 from the first particle in their box
  - ```
    wtx = wtx % NUMBER_THREADS;
    while (wtx < NUMBER_PAR_PER_BOX) {
        rA_shared[wtx + NUMBER_PAR_PER_BOX * thread_replica] =  rA[wtx];
        wtx = wtx + NUMBER_THREADS;
    }
    wtx = tx;
    ```
  - ```
    wtx = wtx % NUMBER_THREADS;
    while (wtx < NUMBER_PAR_PER_BOX) {
        rB_shared[wtx + NUMBER_PAR_PER_BOX * thread_replica] =  rB[wtx];
        // Copying data relates to particles
        wtx = wtx + NUMBER_THREADS;
    }
    wtx = tx;
    ```

Solution

- The “j” index was updated to access the appropriate neighbour replicas

```
wtx = wtx % NUMBER_THREADS;
while (wtx < NUMBER_PAR_PER_BOX) {
    for (j = NUMBER_PAR_PER_BOX * thread_replica; j <
    NUMBER_PAR_PER_BOX + (thread_replica * NUMBER_PAR_PER_BOX); j++)
    {
        // Shared memory is accessed with an index of
        // wtx + NUMBER_PAR_PER_BOX * thread_replica
        // for performing computations
    }
}
```

Concrete Outcomes

The concrete outcomes can be found in this GitHub repository:

- The final product is under the name of `cuspis_lava_kbtr.cu`
- The repository also contains other files necessary to run the benchmark as well as the original CUDA version of it
- The project can be run either locally or on Google Collab using the Tesla T4 runtime

Learning outcomes:

- More familiar with CUDA and with GPU programming in general
- Explored an application of the concept of redundancy which was not part of the AOS or ES courses
- Soft skills relative to team work (work balance, communication and discussion skills)

Conclusions

- We managed to port the requested benchmark in CUSPIS and to obtain successful and correct executions
- Some limitations include further memory optimizations of some edge cases that occur according to the policies to be used
- Ultimately, we believe the final product of our work is ready to be used in a fault injection context