Simulating an Ising Model on GPU

Parallel and Distributed Systems

Christina Koutsou
github: @kchristin22

1 About

The purpose of this project is to implement a cellular automaton in order to simulate an Ising model on a GPU and observe its states after specified iterations. Multiple versions have been written to compare the different features offered by CUDA.

2 Algorithm and Implementation details

The Ising Model consists of a grid of spins whose values are determined following the formula:

$$sign(G[i-1,j]+G[i,j-1]+G[i,j]+G[i+1,j]+G[i,j+1])$$

and the lattice is treated like an toroidal. In the code, the -1 spin corresponds to 0 and the input and output pointers are swapped before each iteration.

The different versions implemented are:

- Sequential (V0): This version is the only CPU implementation, mainly used to evaluate the correctness of the GPU versions. Due to compilation of the project with `nvcc`, OpenMP could not be used for the parallelization of this code.
- Each thread evaluates one element (V1)
- Each thread evaluates a block of elements (V2)
- Multiple threads evaluate a block of elements (V3)

In the versions where multiple threads operate within a block, consecutive threads access contiguous memory addresses enabling coalesced memory transactions.

Given that each element's computation relies on its neighboring elements, GPU blocks cannot be treated as independent entities. Consequently, implementations fall into two categories based on whether the element count exceeds the maximum parallelizable grid size. More specifically:

1. Single kernel is launched for all iterations
2. One kernel is launched for each moment (general)

The first approach requires synchronization of all running threads before proceeding to the next iteration. While CUDA offers a standard function for the synchronization of all threads within a block, the need for a more flexible definition of thread groups to be synced gave birth to the concept of `cooperative groups`. By designating the entire grid as such a group, the otherwise independent blocks can synchronize seamlessly. If the device does not support this feature, a manual barrier using an atomic counter is also included. This counter ensures that blocks wait in a while loop until all have reached a specific point in the program. Evidently though, it's optimal to distribute the work to as little blocks as possible by utilizing the maximum number of threads available per block.

Despite the efficacy of launching a single kernel for all the iterations, this method is suitable for only processing limited amount of data. On the other hand, invoking a kernel for each iteration allows sequential block execution if necessary. The cost of kernel launches can be concealed as it can be performed in parallel to a previous kernel's execution in the same stream and since the

kernel calls are executed sequentially in a single stream, the host and the device are synced only after all the iterations have been performed. In order to further battle the cost of kernel launching at each iteration, `graphs` are employed to benefit from the single creation and initialization of the kernel calls beforehand for rapid invocation. The need for a CPU (Host) node requiring costly synchronization with the device is eliminated by replacing input and output pointer swapping with a second kernel that copies the output to the input and clears the output vector in preparation for the next iteration. Moreover, as an alternative to using `Event` signals across streams to signify the dependency of the second kernel to the first one, both the kernels are added to the same stream to ensure sequential execution and capture this relationship. The choice of capturing only a single iteration offsets the overhead of the graph's initialization due to the limited use of resources and the need to re-use the same graph multiple times.

To further leverage parallelization opportunities and optimize memory access, in the version where each block consists of a single thread and, hence, there's room for more to be utilized, another implementation has been added. In this approach, a separate `stream` is created for each value that needs to be added to an element's output. In other words, five threads collaboratively operate on a single element, atomically adding the equivalent neighbor's value when it is equal to 1. Facilitating this operation, a function pointer is passed as an argument to the kernel, allowing it to calculate the offset of the output array. Afterwards, the default stream is employed to transfer the output to the input and clear the output. Notably, this step doesn't necessitate Event signaling or CPU-Device synchronization, as the default stream is inherently synchronous with all other streams. The corresponding graph representation of this version has also been implemented.

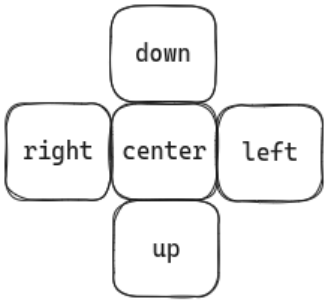

Figure 2.1: *All streams add their value to the center*

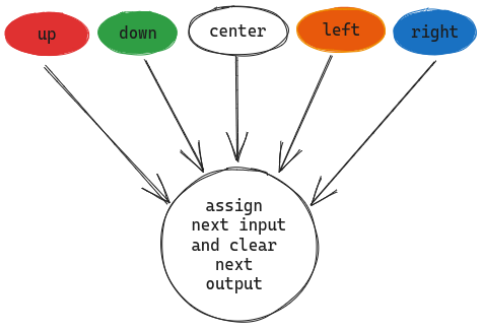

Figure 2.2: *Node dependencies in the graph representation*

Lastly, CUDA `Async` functions have been employed to bypass synchronizing the entire device after each call, thus accelerating GPU processes. Furthermore, these operations are executed within the default stream, which, as previously mentioned, is synchronous to other streams. Consequently, if another stream is utilized subsequently, it will execute sequentially in relation to the prior function calls in the default stream.

3 Benchmarks

To perform the benchmarks the `nanobench` tool was used. Each version is run for 5 epochs with at least 10 iterations each to stabilize the results, except for the versions where many threads are used within a block (V1 and V3) which require a minimum of 100 iterations. The number of iterations and epochs can be fine-tuned with respect to the median error percentage.

The tests were run on Aristotle HPC and specifically on its gpu partition.

Firstly, we compare all versions to each other for the same input size (n is the size of a single dimension, hence the input's ize is n^2) and for a single iteration.

	n = 100, b = 10, t = 10	**n = 1000, b = 1000, t = 10**
V0	521 us	41430 us
V1	33 us	- (too many blocks in cooperative launch)
V2	925 us	1877 us
V3	138 us	703 us
V1 general	34 us	362 us
V2 general	874 us	1664 us
V2 multiple streams	1369 us	7592 us
V3 general	137 us	585 us
V1 general-graph	55 us	389 us
V2 general-graph	984 us	1838 us
V2 multiple streams-graph	1600 us	7939 us
V3 general-graph	217 us	748 us

Table 3.1: *Comparison of median execution times of different versions for n = 100 and n = 1000*

It is evident that the overhead of transferring data from CPU to GPU is only justified when leveraging a substantial number of threads (as observed in V1). This may become even more pronounced when synchronization isn't necessary before proceeding to the next iteration, as exhibited in the slightly superior performance of the general versions over a single launch of the kernel and the utilization of cooperative groups. This fact is also preferable as the latter cannot be widely applied due to the limited number of threads supported. Furthermore, it's worth highlighting the scalability demonstrated by the CUDA versions as the problem size increased. For instance, while maintaining the same analogy of number of elements and blocks, a mere 100x increase in array size resulted in execution time spikes of 11x and 7x for the V1 general and graph versions respectively, contrasting the 80x increase observed in the sequential CPU code.

The graph versions show promising potential, given that these benchmarks were conducted for a single iteration. The slowest performers were those employing multiple streams to operate on a single element, implying that the overhead of thread manipulation and the necessary atomic operations outweighed the benefits. Additionally, versions accumulating shared memory among threads within the same block demonstrated significant advantage, but it would be more interesting to compare them to a version that uses the same amount of resources but employs the global memory instead. Moreover, the limited size of the shared memory demands the usage of more blocks which is not advisable.

The diagrams below depict the correlation between a specified input and the quantity of blocks and threads per block. Specifically, as the number of threads per block increases, the execution time decreases, contrasting with the behavior of blocks, whose optimal value is 10x less than the input size. This observation reinforces the previous hypothesis favoring a higher thread count within the same block over increasing the number of blocks. Additionally, when the resources utilized by

the V3 general version match the ones of V1, it manages to outperform the V1 versions, which relied on global memory. Since both versions require fetching and storing data back to the global memory at each iteration, it is assumed that the overhead of performing computations within shared memory is insignificant. Due to neighboring blocks requiring the computed data saved in the shared memory for their own computations, the aforementioned overhead is unavoidable and hence, further performance improvement is limited by this constraint.

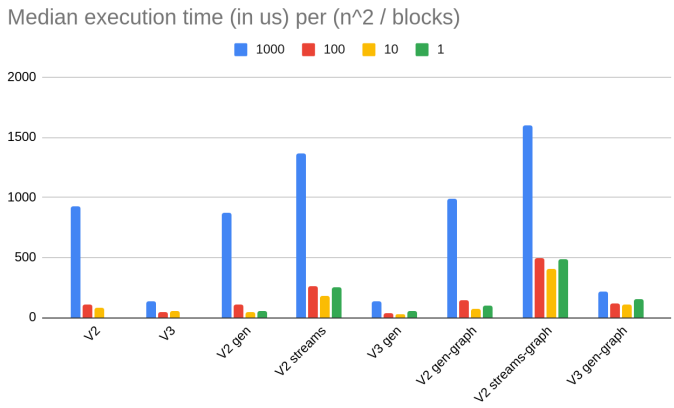

Figure 3.1: *Study of optimal relationship between input size and number of blocks (10 threads per block, execution time in us)*

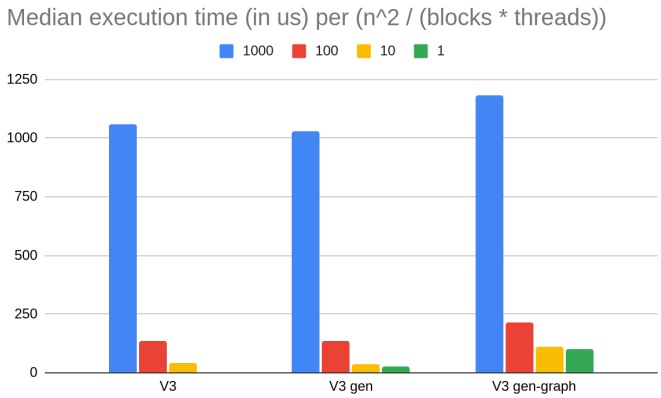

Figure 3.2: *Study of optimal relationship between input size and number of running threads in the grid (b = 10, execution time in us)*

In Figure 3.3, the relationship of the general versions and their graph versions is displayed across varying numbers of iterations. Specifically, it is evident that in the majority of versions, the curves tend to stabilize as the number of iterations increases, consistently favoring the versions that do not utilize CUDA `graphs`. Nevertheless, it is likely that with a great increase of iterations these execution times might converge or even tilt in favor of the graph-utilizing versions. An exception to this trend is observed in the multiple streams version, where the advantage of using graphs is apparent even in as few as 100 iterations, hinting that a problem with more nodes than one or two, as is the case with the rest of the graph versions, could benefit from the use of graphs.

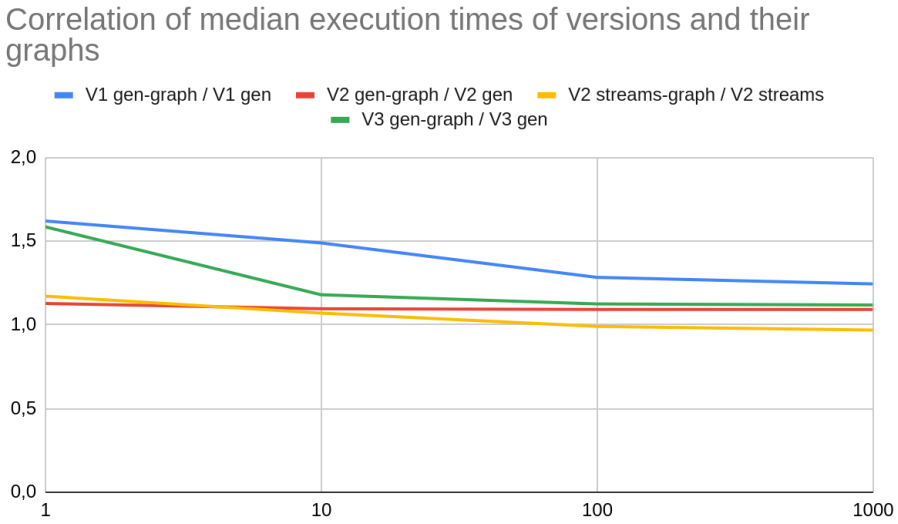

Figure 3.3: *Overhead of graph versions as iterations increase*

4 Extras

A few additional ideas worth exploring include:

- Consider using a sparse matrix representation for the input array, without the need for a Value vector as the non-zero elements' value is always equal to 1
- Add check for circles in states of the model to end the program sooner
- Consider predicting the value of certain elements from the first iteration
- Use occupancy calculators offered by CUDA to determine the optimal number of blocks for a given problem
- Record `Events` inside device code to use for synchronization among blocks (this feature is currently not developed)

5 Citations

- CUDA C Programming Guide
- NVIDIA Forum: __threadfence_block() vs __threadfence()?
- CUDA Streams presentation
- CUDA graphs
- CUDA asynchronous programming presentation
- NVIDIA Forum: Multi-streams with CUDA Graphs
- Stream-ordered memory allocator in CUDA