Implementation of Lattice Bolzmann Method on GPU

Benjamin BOURBON
HPC-AI - Mines ParisTech

April 20, 2023

I Introduction

I.1 Context

The objective of this project is to take an existing code over, written in Python on the Lattice Bolzmann Method in order to accelerate computations by using a GPU. For this reason, two Python libraries have been chosen for this report:

- Cupy is an open-source library for accelerated computations on GPU with Python. CuPy uses CUDA libraries, including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL for using the whole GPU architecture.
- Numba allows to write your own GPU algorithms completly with Python, and supports CUDA for NVIDIA and ROCm for AMD. Moreover, it translates Python functions to optimize them during execution time by using the standard compilation library LLVM.

Also, new libraries are emerging such as *cuNumeric* supported by *Legate* but due to lack of time, this study focuses only on *CuPy* and *Numba*.

I.2 Lattice Bolzmann Method

The Lattice Bolzmann Method is described by 8 main functions:

- `bounce_back`
- `collision`
- `equilibrium`
- `inflow`
- `macroscopic`
- `outflow`
- `streaming_step`
- `update_fin`

Each of these functions plays a necessary role in the method. Most of them use matrices with $9 \times n_x \times n_y$ dimensions. It is relevant to use a GPU for accelerating computations.

I.3 Implementation with Numba

The implementation with Numba requires a detailed understanding of how a GPU works and the integration of GPU kernels. To ensure reliability, a test file has been designed to verify the results. The results obtained by NumPy are saved in *pickle* files. Then, after each kernel calculation, the GPU result is returned to the CPU to compare the results with those of NumPy.

Simply run:

```
python alltests.py -p # generates the pickle files
python alltests       # displays the test results for the Numba kernels
```

II Implementation with CuPy

The huge advantage of CuPy is that most of NumPy's functions are implemented. So, switching from NumPy to CuPy is almost as simple as changing the import so that we replace *numpy* with *cupy*. But this is still much less efficient than creating our own kernels. Tests have been implemented, as for *Numba*:

```
python alltests.py -p # generate pickle files
python alltests -c    # display test results for CuPy kernels
```

III Computational Kernels

Each computational kernel is a translation of the original NumPy function. Some computational kernels use constant arrays. In other words, if we take the variable v, it is a constant and static array. It is shared by all threads in the *equilibrium* function to have easier and therefore faster access during thread computations.

For computational kernels to be efficient, we also need to consider how to distribute the workload across the different threads. In other words, we need to scale the block size and the GPU grid size. To do this, here are the two functions that address this issue:

```
def dispatch(m, n):
    r = min(floor(log((m * n) // 216, 2)), 10)
    inf, sup = (m, n) if m < n else (n, m)
    c = log(sup, 2) - log(inf, 2)
    a, b = min(10, int(floor(r / 2 - c))), min(10, int(ceil(r / 2 + c)))
    tx, ty = (2 ** a, 2 ** b) if m < n else (2 ** b, 2 ** a)
    blockspergrid_x = int(m // tx + bool(m % tx))
    blockspergrid_y = int(n // ty + bool(n % ty))
    blockspergrid = (blockspergrid_x, blockspergrid_y)
    threadsperblock = (int(max(1, tx)), int(max(1, ty)))
    return threadsperblock, blockspergrid


def dispatch1D(n):
    t = threadsperblock = int(2 ** min(10, floor(log(max(1, n // 216), 2))))
    blockspergrid = int(n // t + bool(n % t))
    return threadsperblock, blockspergrid
```

The GPUs we're working on are equipped with 108 SMs (Streaming Multiprocessors). Each of them can handle 2048 threads at a time. A block can contain a maximum of 1024 threads. Therefore, the grid must be at least $108 \times \frac{2048}{1024} = 216$ blocks in order to fully utilize the GPU. In other words, for any grid that is a multiple of 216, we theoretically ensure that the GPU is fully loaded.

We consider the most difficult case: a two-dimensional grid with two-dimensional blocks. What is proposed above in the `dispatch` function is a way to distribute the workload for two-dimensional computations. We are looking for a grid that is a multiple of 216: $(m \times n) \perp 216$. We want the number of threads to be an integer power of 2: $\lfloor \log_2((m \times n) \perp 216) \rfloor$. But we are limited by the GPU capacity ($2^{10} = 1024$): $r = \min(\lfloor \log_2((m \times n) \perp 216) \rfloor, 10)$.

Thus, we know that r is an integer power of 2. But since we have to work in two dimensions, we need to distribute the value of r evenly. That's why we then look for the proposition between the dimensions in powers of 2: $c = \log_2(\text{sup}) - \log_2(\text{inf})$. By dividing r by 2, we obtain a balanced but unrepresentative distribution of the proposition between the two input dimensions. So we calculate the new powers in integer values:

$$\begin{cases} p_{\text{inf}} & = \lfloor \frac{r}{2} - c \rfloor \\ p_{\text{sup}} & = \lceil \frac{r}{2} + c \rceil \end{cases}$$

We impose that these powers do not exceed 10: $a = \min(p_{\text{inf}}, 10)$ and $b = \min\left(p_{\text{sup}}, 10\right)$. We can then calculate the block dimensions and the grid dimensions.

This reasoning is the same with a single dimension, except that we do not have a balanced distribution of loads according to the input dimensions since we have a single input dimension. In practice, we are not limited to small dimensions so since the input dimensions are large, often the block size is $(32, 32)$.

IV Using the Nsight Compute Profiler

To facilitate GPU performance analysis, the Nsight Compute Profiler provides an interface that describes each core: how resources were used, the computation time, and whether the GPU was properly utilized based on the grid size.

Another way to analyze performance is to record computation times alone. But obviously, this remains much more basic. Here's a snippet of code that allows you to obtain the computation time of a kernel:

```
start = cuda.event()
start.record()
equilibrium[BPG2D, TPB2D](d_rho, d_u, d_v, d_t, d_feq, INTNX, INTNY)
cuda.synchronize()
stop = cuda.event()
stop.record()
print("Elapsed Time:", cuda.event_elapsed_time(start, stop))
```

It's important to note, however, that the profiler isn't a tool to be taken literally, but rather as a guide to improvement. Indeed, we'll see from analyzing the results that just because the block grid is small doesn't necessarily mean computation times are negatively impacted. Quite the contrary, for kernels with 1D blocks and a 1D grid, having a weak grid improves execution time.

V Results

This study does not aim to evaluate GPU results and compare them to CPU results (execution time, memory management, percentage of architecture power utilization, etc.). Indeed, given that a CPU architecture is not designed for the same type of application as a GPU architecture, it goes without saying that comparing these two architectures is meaningless. Rather, the aim is to try to understand GPU performance based on the implemented computational kernels.

We recall that 8 computational kernels were implemented and that the following were recorded:

- execution time
- computation throughput as a percentage
- memory throughput as a percentage
- cache throughput (L1 memory)
- cache throughput (L2 memory)
- occupancy reached

Kernel name	Execution Duration	Compute Throughput	Memory Throughput	L1 Cache Throughput	L2 Cache Throughput
macroscopic	7.67ms	6.79%	90.08%	90.52%	51.71%
equilibrium	2.34ms	19.05%	88.02%	88.49%	58.85%
streaming_step	2.18ms	57.37%	85.31%	85.75%	83.36%
collision	4.56ms	6.29%	70.96%	71.52%	64.97%
bounce_back	345.09µs	16.49%	71.9%	72.6%	65.08%
inflow	17.95µs	2.03%	3.47%	0.96%	4.09%
update_fin	10.37µs	0.73%	5.26%	2.19%	7.31%
outflow	9.31µs	0.72%	3.12%	2.13%	4.58%

Table 1: Results obtained from the Nsight Compute profiler on the different Numba type computing cores

Kernel name	Execution Duration	Compute Throughput	Memory Throughput	L1 Cache Throughput	L2 Cache Throughput
macroscopic	8.05ms	6.79%	91.09%	91.77%	51.15%
streaming_step	2.14ms	25.37%	86.8%	87.24%	82.5%
equilibrium	2.33ms	19.11%	88.36%	88.97%	59.31%
collision	4.74ms	4.9%	67.45%	67.7%	62.43%
bounce_back	340.45µs	12.2%	72.74%	73.92%	65.38%
update_fin	10.56µs	0.65%	6.21%	2.28%	7.87%
inflow	10.82µs	0.82%	5.15%	1.98%	7.88%
outflow	9.7µs	0.52%	3.44%	2.26%	4.86%

Table 2: Results obtained from the Nsight Compute profiler on the different Cupy type computing cores

In addition, here are the different occupations achieved for each of the computing cores:

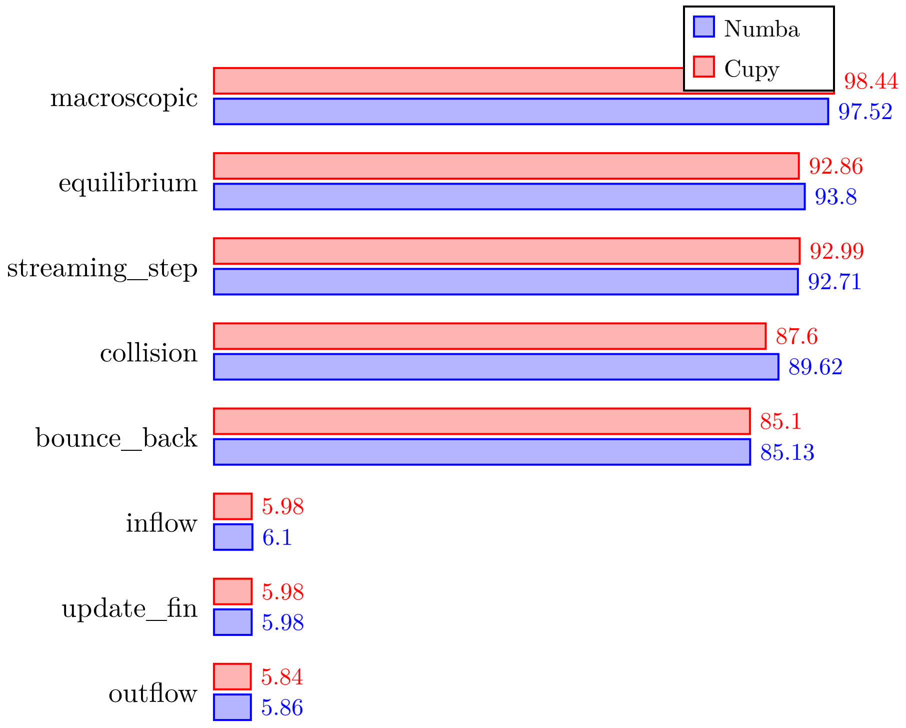

Figure 1: Occupancy reached (in %) for the different computing cores

In the Table 1 table and the Table 2 table, we can see similar results in terms of computation time. *CuPy* manages computation throughput differently. This is quite clearly seen in the `equilibrium` computation kernel, with a difference of 38.26%. The `macroscopic` computation kernel is the slowest of all. We will see how to improve it later. The computation throughput column shows how kernel executions are managed by the *Streaming Multiprocessors* as groups of 32 *threads* called *warps*. The last three columns illustrate memory management.

In the Figure 1, we can compare how *Numba* and *CuPy* manage occupancy. We cannot visualize an imbalance. It is therefore up to the user to carefully write their computational kernel and find the right sizing for the number of threads per block and the grid size. A rather contradictory point is the deduction of the *Nsight Compute* profiler's recommendations based on the data collected during the computations. Indeed, if we take the *outflow* computational kernel, the theoretical occupancy is 100%. We are far from this objective, but by reducing the number of threads and increasing the grid size, we can go from approximately 6% to approximately 30%. However, we lose speed by going from 9.31 μs to 11.23 μs. The difference is not significant for a few microseconds, but it is still significant.

VI Optimization

As explained in the last section, *macroscopic* does not produce good results. Let's try to understand why: Here's the implementation with *Numba*:

```
@cuda.jit
def macroscopic(fin, v, rho, u, nx, ny):
    row, col = cuda.grid(2)
    if row < nx and col < ny:
        # Initialization at zero
        rho[row, col] = float64(0.0)
        u[0, row, col] = float64(0.0)
        u[1, row, col] = float64(0.0)

        # Computation of rho
        for i in range(9):
            rho[row, col] += fin[i, row, col]

        # Computation of u
        for i in range(9):
            u[0, row, col] += v[i, 0] * fin[i, row, col]
            u[1, row, col] += v[i, 1] * fin[i, row, col]
        u[0, row, col] /= rho[row, col]
        u[1, row, col] /= rho[row, col]
```

This implementation is the naive one. However, we can see several problems:

- the variable v is not modified: it is a small static variable that can be stored in each thread's memory.
- there are 10 writes and 9 reads for the calculation of *rho*
- there are 22 writes and 18 reads for the calculation of u

These elements take a lot of time and can be reduced considerably. Here is a new implementation:

```
@cuda.jit
def macroscopic(fin, v, rho, u, nx, ny):
    cv = cuda.const.array_like(v) # shared array between all threads
    row, col = cuda.grid(2)
    if row < nx and col < ny:
        trho = float64(0.0)
        tu0 = float64(0.0)
        tu1 = float64(0.0)
        for i in range(9):
            fvalue = fin[i, row, col]
            trho += fvalue
            tu0 += cv[i, 0] * fvalue
            tu1 += cv[i, 1] * fvalue

        rho[row, col] = trho
        u[0, row, col] = tu0 / trho
        u[1, row, col] = tu1 / trho
```

We move from 7.67 ms to 1.5 ms. This means we have a speedup of approximately 5.11. Furthermore, the computational throughput increases to 21.82% (6.79% for the unoptimized version).

It's also relevant to estimate the bandwidth. For the unoptimized version, we can count 3 writes for null values followed by 9 writes of *rho* followed by 20 writes of *u*. Similarly, we have 18 reads for *rho* and 36 reads for *u*. We're working with numbers of 4 bytes. The estimated bandwidth is calculated using the following formula:

$$\text{bandwidth} = \frac{(B_r + B_w) \times 10^{-9}}{t}$$

where B_r is the number of bytes read, B_w is the number of bytes written, and t is the time in seconds.

We can thus estimate the bandwidth at 635 GB/s. Using the optimized version, there is a 9-repetition loop containing 3 reads, and at the loop exit, there are 3 writes. We obtain a bandwidth of 1132 GB/s. Given that the theoretical bandwidth is around 1500 GB/s, the bandwidth of the optimized version represents 75.5% of the theoretical bandwidth.

Based on current results and existing implementation of other kernels, it is more difficult to improve them.

VII Conclusion

This project provided an opportunity to try out libraries like Numba and CuPy and see their effectiveness. Not only are these libraries written for Python, a high-level programming language, but they also achieve very good performance for a language that isn't known for its speed. This opens up the possibility of future developments in machine learning and deep learning libraries to achieve high-performance computing, such as those currently available on the market, PyTorch or Tensorflow, for example.

This project enabled the development of computational kernels based on existing code. It also revealed that despite its syntax being very similar to NumPy, simply changing the functions implemented with NumPy to CuPy does not allow for the creation of code that could be described as a "prototype". Indeed, writing computational kernels remains ideal for high-performance computing but limits their flexibility to their targeted use.

Next, using a profiler such as Nsight Compute allows for performance measurements useful for improving computational kernels. However, this type of tool is not the Holy Grail either. It can provide users with good improvement guidelines in most cases, but can also be misleading.

However, certain aspects, such as thread synchronization and data sharing across threads, were not addressed. The simple reason is lack of time. This idea of improvement would significantly improve performance over what has already been achieved, but it would also require a review of the structure of the computational kernels and the use of indices within them. We can, however, cite the example of multiplying two matrices together, which, in its naive implementation, does not share data or perform thread synchronization. Nonetheless, in the optimized implementation, we must store part of the data as arrays shared between each thread, perform thread synchronization, perform the matrix calculation followed by thread synchronization, and finally store and return the result. This implementation reduces memory access and thus improves performance.

Finally, the use of the *cuNumeric* library with *Legate* cannot be explored due to lack of time.