UNIVERSITÀ DEGLI STUDI DI MILANO

GPU Computing Project

Bitonic Sort using Numba on NVIDIA GPUs

Author:

Manuele Lucchi

08659A

Academic Year 2022/2023

1 Abstract

This documents describe the performances of an existing algorithm, the Bitonic Sort, implemented using Python and Numba, comparing it with a serial approach and CUDA C implementation.

Contents

2 Introduction

The Bitonic Mergesort [1] is an algorithm created in 1968 by Ken Batcher[2]. The elements to choose for the comparison are indipendent from the value (it's not data-dependant) therefore is well suited for parallel processing.
Over the years there are been many implementations using low level languages such as C and even high level such as Java or Python, but regarding the parallel variant, only C/C++ examples can be found on the web (using OpenMP [3] and CUDA C [4]) The CUDA community developed an abstraction layer over the CUDA runtime to be able to execute CUDA kernels from python with little-to-no overhead using Numba [5].
Numba is not a CUDA specific framework, it's a generic parallelization abstraction that allows to compile Python functions into machine code, while integrating with numpy [6] to have access to a powerful tensor manipulation library; but of course its primary target is CUDA.
The scope of the paper is to implement the algorithm using Numba and compare it with different implementations

3 Algorithm

The algorithm is based on the concept of Bitonic Sequence [1], a sequence with $x_0 \leq ... \leq x_k \geq x... \geq x_{n-1}$ for some k, $0 \leq k \leq n$, meaning that there are two subsequences sorted in opposite directions.
By using a sorting network, we can create a Bitonic Sequence from any sequence and then merging them to obtain the final sorted sequence, so the algorithm has two phases.

This algorithm has an asymptotic complexity of $O(nlog(n)^2)$, the same of the odd-even mergesort and shellsort [7]

3.1 Step 1: Bitonic Sort

To create a Bitonic Sequence we need to build a sorting network. This step has $log_2 n$ phases (with the last one being the second step), each one with i stages, where i is the current phase starting from 1. By sorting each pair of elements in the sequence in different directions pairwise (using the so called ”comparers”), we obtain a sequence full of bitonic subsequences. We can then at each phase double the size of these subsequences and half their number.
As we can see in picture one, at each stage of a phase the elements in the comparers get closer and in each phase a bitonic subsequence is sorted. This structure is known as butterfly network [8]

3.2 Step 2: Bitonic Merge

The last step is a variation of the first one, where we only have a single bitonic sequence and sort the two subsequences with the comparers oriented in the same direction, resulting in the sorted sequence.

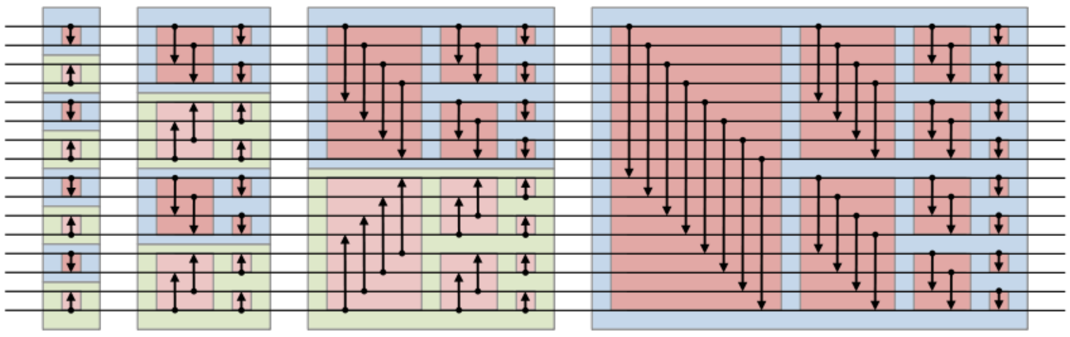

Figure 1: The structure of a bitonic sorting network

4 Implementation

The scope of the implementation is to use Numba to measure its efficiency compared to serial python algorithms and a standard CUDA C++ version.

The other versions are inspired by the course material and by the Wikipedia iterative implementation, with some changes to support a generic sorting direction (and of course, translated to python).

The Python code consists of 3 functions

4.1 The host code

A function named *bitonic_sort* that accepts the initial data, its size and the direction. This function loops two indices, doubling the outer one each external iteration and halfing the inner one each internal iteration. This represents the $p = log_2 n$ phases and $s = log_2 p$ stages for each phase. Inside the two loops, the kernel function is called forwarding the data, the direction and passing the current phase and stage.

4.2 The kernel code

The *global* code, called by the host and executed in the device decides which pair of elements to compare is represented by the function *bitonic_kernel.*
It's decorated with the *@cuda.jit* decorator, that signales Numba that it's a function to compile to CUDA instructions. This function checks if the **bitwise XOR** between the phase and the stage is bigger than the current thread global index, then executes the *compare_and_swap* function swapping the two indices compared in the previous conditions as arguments based on if the **bitwise AND** is equal to 0 or not.

4.3 The device code

The last function has the purpose of swapping the elements based on their positions and the sorting direction.
More specifically, if the requested direction (represented by a $0-1$ valued integer) equals the direction of the two elements, the pair is swapped This function could be further improved by adding the support for the atomic operation **Compare**

and Swap [9], that could reduce Warp Divergence by swapping conditions with mathematical calculations.

5 Benchmark and Profiling

5.1 Environment

The benchmark and profiling environment is represented by a computer with an AMD Ryzen 7 5800H CPU (8 Cores, 16 Threads), 16 GB of RAM and an NVIDIA GeForce RTX 3050 Ti for Laptops. Basic specifications comprise **20 SMs** for a total of **2560 CUDA cores**, 20 RT cores and 80 Tensor cores. Clock speeds up to 1695 MHz boost at 80W, or as low as 1035 MHz for the 35W configuration, giving us quite a wide power range of 35 to 80W. It has 4GB of GDDR6 memory on a 128-bit bus, clocked at 12 Gbps. The **Compute Capability** Version is 8.6.

5.2 JIT Compilation

The CUDA runtime executes functions, called kernels, compiled for NVIDIA GPUs instruction set using the NVCC Compiler [10] and this is a different approach compared to Python, which is a **runtime interpreted language**.
To solve this issue in the fastest way without changing paradigm was to compile **Just In Time** the python kernel into machine code, **caching it** and then executing it calling the CUDA Runtime Library. This of course comes with a delay in the first execution of a kernel (that needs to be compiled) and since the benchmarks would be cached except for the first one, the JIT compilation time has been considered separately. On the kernel function defined for the Bitonic Sort, the compilations on the environment takes around **0.3 seconds**, so it's noticeable if you want to compare with a sequential approach or a CUDA C version with a cold start.

5.3 Serial and Parallel approach

The first benchmark is between two serial approaches and the Numba implementation, with all three being in Python.

To have some meaningful data, the benchmark had multiple size of the array, starting from $n = 2^8$ to $n = 2^14$, doubling each time.

As we can see from the following image, the Numba implementation time maintains a negligible time while both the recursive and iterative serial approaches went above the second for the last one, with the recursive being slight better.

This has to be expected since the parallel version is ideally $O(log(n)^2)$ on each core, so noticeably faster than a sequential version. This is of course not completely true in reality since the GPU has not an infinite number of cores and with a further increase on the input size we'll see in the next section a rapid increase on time, but still way faster than the serial version.

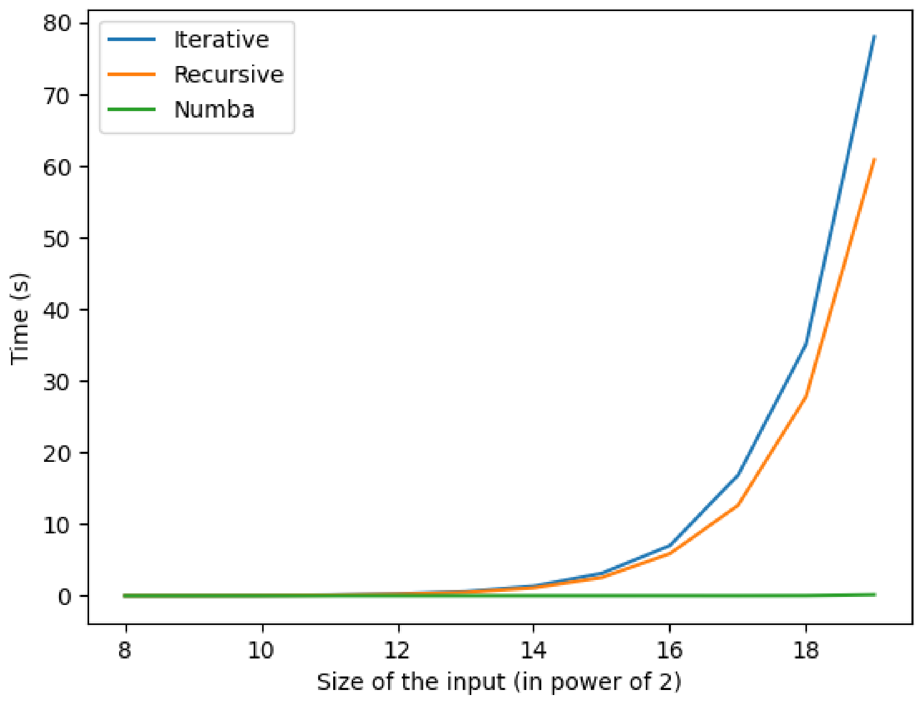

Figure 2: Time comparison between different implementations

The following table shows in details the results for each size

Size (Power of 2)	**CPU Iterative**	**CPU Recursive**	**GPU**
8	0.008 ms	0.007 ms	0.001 ms
9	0.017 ms	0.015 ms	0.002 ms
10	0.049 ms	0.042 ms	0.002 ms
11	0.104 ms	0.086 ms	0.003 ms
12	0.247 ms	0.200 ms	0.004 ms
13	0.592 ms	0.471 ms	0.004 ms
14	1.281 ms	1.076 ms	0.004 ms
15	3.007 ms	2.401 ms	0.004 ms
16	6.738 ms	5.577 ms	0.005 ms
17	15.256 ms	12.094 ms	0.005 ms
18	33.877 ms	26.759 ms	0.022 ms
19	76.762 ms	62.243 ms	0.174 ms

5.4 CUDA C and Numba

The second and last benchmark compares the Numba and Python implementation with the CUDA C one.
The code is basically the same but in different languages, so the theoretical performances should be the same. As we can see from the image below, the CUDA C implementation, while it's faster, it is just for a constant coefficient.
This means that there's an overhead, probably result of the Python runtime and the Numba abstraction layer, that makes the CUDA C implementation around 3 times faster.
While this seems a huge number in performances, given the fact that its constant it can be an acceptable difference in many cases when someone would prefer the faster development time that Python allows.

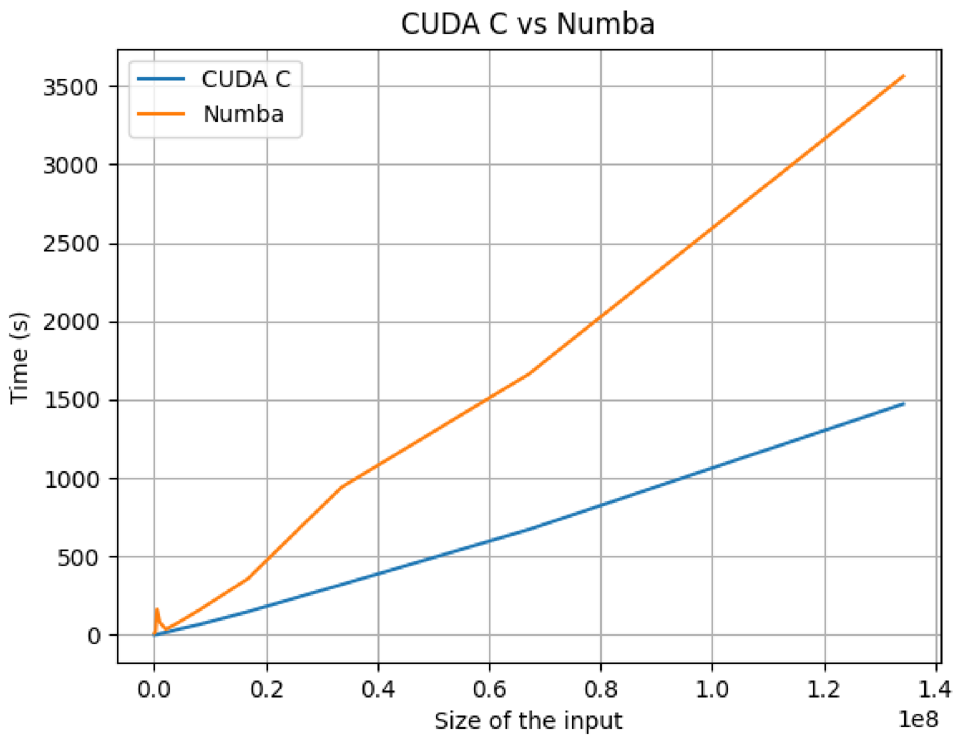

Figure 3: Time comparison between parallel implementations on different platforms

The last figure shows the same results but making the data extrapolation easier showing the values on a power of 2 scale

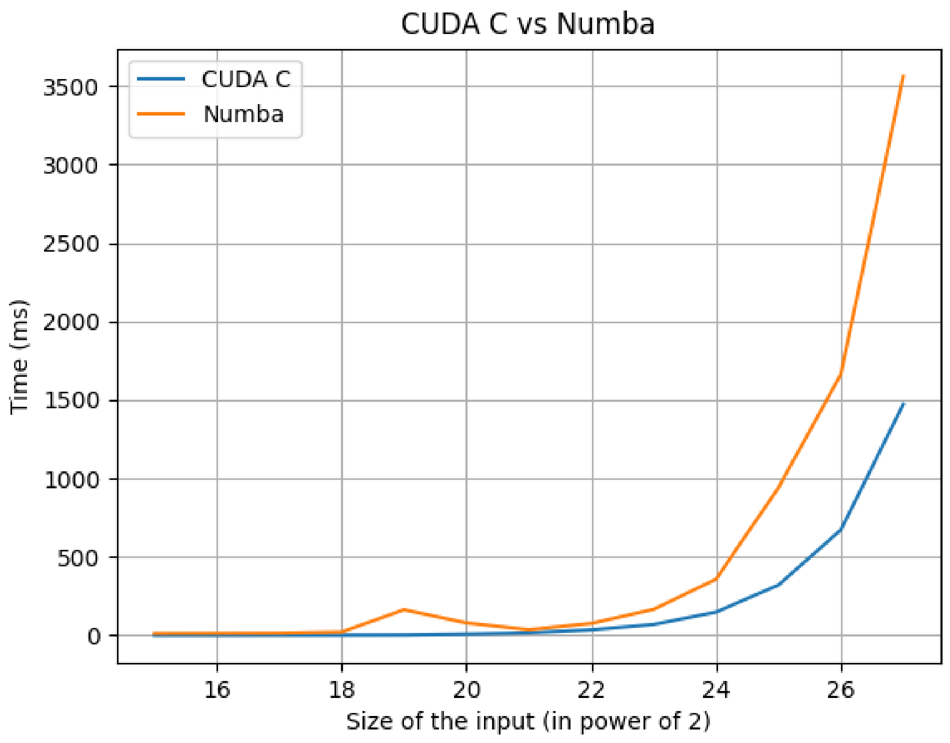

Figure 4: Time comparison between parallel implementations on different platforms with better value/time visualization

The last table of the section again shows the detailed results

Size (Power of 2)	**Cuda C++**	**Python Numba**
15	0.98 ms	4.36 ms
16	1.03 ms	5.03 ms
17	1.47 ms	4.83 ms
18	1.88 ms	22.12 ms
19	2.81 ms	25.4 ms
20	8.38 ms	36.16 ms
21	16.29 ms	76.04 ms
22	35.05 ms	79.35 ms
23	69.36 ms	165.44 ms
24	147.84 ms	357.72 ms
25	320.63 ms	940.48 ms
26	671.53 ms	1662.06 ms
27	1471.70 ms	3561.07 ms

5.5 Profiling Results

Using the profiling tool *NCU* on a kernel execution, we obtained the following results:

5.5.1 GPU Speed Of Light Throughput

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/nsecond	5,75
SM Frequency	cycle/nsecond	1,17
Elapsed Cycles	cycle	2.757.034
Memory Throughput	%	92,97
DRAM Throughput	%	92,97
Duration	msecond	2,35
L1/TEX Cache Throughput	%	19,44
L2 Cache Throughput	%	29,83
SM Active Cycles	cycle	2.746.869,40
Compute (SM) Throughput	%	16,16

5.5.2 Launch Statistics

Metric Name	Metric Unit	Metric Value
Block Size		256
Function Cache Configuration		CachePreferNone
Grid Size		131.072
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	Kbyte	8,19
Driver Shared Memory Per Block	Kbyte/block	1,02
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	33.554.432
Waves Per SM		1.092,27

5.5.3 Occupancy

Metric Name	Metric Unit	Metric Value
Block Limit SM	block	16
Block Limit Registers	block	16
Block Limit Shared Mem	block	8
Block Limit Warps	block	6
Theoretical Active Warps per SM	warp	48
Theoretical Occupancy	%	100
Achieved Occupancy	%	83,52
Achieved Active Warps Per SM	warp	40,09

Using the newer profiler **Nsys**, we obtain a binary file of a proprietary format used to reprent a complex and interactive report that can be used via the Nsys GUI.
Since this cannot be expressed in a document, we report the summary

5.5.4 CUDA API Summary (cuda_api_sum)

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
92.0	753893991	2	376946995.0	376946995.0	12413	753881578	533065998.0	cuCtxSynchronize
6.0	48795888	1	48795888.0	48795888.0	48795888	48795888	0.0	cuMemcpyHtoD_v2
0.0	4999109	325	15381.0	12594.0	8646	199198	11288.0	cuLaunchKernel
0,0	1593356	1	1593356,0	1593356,0	1593356	1593356	0,0	cuModuleLoadDataEx
0,0	1114898	1	1114898,0	1114898,0	1114898	1114898	0,0	cuMemAlloc_v2
0,0	961316	1	961316,0	961316,0	961316	961316	0,0	cuMemGetInfo_v2
0,0	145667	1	145667,0	145667,0	145667	145667	0,0	cuLinkComplete
0,0	74011	1	74011,0	74011,0	74011	74011	0,0	cuLinkCreate_v2
0,0	3236	1	3236,0	3236,0	3236	3236	0,0	cuProfilerStop
0,0	2155	1	2155,0	2155,0	2155	2155	0,0	cuLinkDestroy
0,0	411	1	411,0	411,0	411	411	0,0	cuDeviceGetUuid_v2

5.5.5 CUDA GPU Kernel Summary (cuda_gpu_kern_sum)

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
100,0	771649549	325	2374306,0	2277401,0	2201496	2944544	180394,0	cudapy::bitonic_gpu::bitonic_kernel

5.5.6 CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum)

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
100,0	40943425	1	40943425,0	0943425,0	40943425	40943425	0,0	CUDA memcpy HtoD

5.5.7 CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum)

Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
268,435	1	268,435	268,435	268,435	268,435	0,000	CUDA memcpy HtoD

6 Improvements and considerations

As we can see from the CUDA APIs summary, most of the computational time is spent on the synchronization, since the kernel is called in a loop and needs to be completed before proceding with the next iteration.
This approach is not optimal and could be improved. Also, this implementation doesn't use shared memory, since at each phase and stage the entirety of the array is needed to make the swaps, thus making impossible to use shared memory unless you want to put a huge limit on the size of the array that can be sorted. An alternative solution to use the shared memory, implicit synchronization and without warp divergence is to use the idea of the **Warp Sort** [11], that splits the sorting for each warp, merges them and finally splits them again to not lose parallelization.

7 Conclusion

Based on the results of this experiments, **Numba** seems a worthy framework that doesn't add too much overhead to the implementation, while having some advantages like a simpler use with the Python abstraction.
It's on a different order of performances compared with a serial approach in this specific problem, completing the task in less time than the overhead to load the runtime and launch the kernel (all under 100ms), while the iterative and recursive implementations took minutes, just like one would expect from a data-indipendent algorithm. The difference with the C implementation is marginal and scales by a constant, a difference that should be taken under consideration for big datasets but that is balanced by the more accessibility.

References

1. Lang, H. *Bitonic sorting network for n not a power of 2* https://hwlang.de/algorithmen/sortieren/bitonic/oddn.htm.
2. *Ken Batcher* https://en.wikipedia.org/wiki/Ken_Batcher.
3. *OpenMP* https://www.openmp.org/.
4. *CUDA C++ Programming Guide* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
5. *Numba makes Python code fast* https://numba.pydata.org/.
6. *NumPy* https://numpy.org/.
7. Phillips, M. *Array Sorting* https://web.archive.org/web/20111028201105/http://homepages.ihug.co.nz/~aurora76/Malc/Sorting_Array.htm#Exchanging_Sort_Techniques.
8. *Butterfly Network* https://en.wikipedia.org/wiki/Butterfly_network.
9. Håkan Sundell, P. T. *Lock-Free and Practical Deques using Single-Word Compare-And-Swap* http://www.non-blocking.com/download/SunT04_Deque_TR.pdf.
10. *NVIDIA CUDA Compiler Driver NVCC* https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.
11. Ye, X. *Overview of GPU Warpsort* https://www.researchgate.net/figure/Overview-of-GPU-Warpsort-It-mainly-consists-of-four-steps-1-Use-warp-based-bitonic_fig1_224140666.