Università di Pisa

Computer Architecture - Project: Bitonic Sort

Fabio Piras, Giacomo Volpi

Academic Year: 2022/2023

Contents

Chapter 1

Introduction

The algorithm we decided to implement and analyze was the bitonic sort. Sorting is a very old and well-known problem, and crucial for many applications used everyday in our life.

1 The bitonic sort algorithm

Bitonic sort is a variant of the mergesort algorithm. The algorithm was developed by Ken Batcher. The algorithm first build the bitonic sequences and the build the final sorted sequence. A bitonic sequence is a monotonically non-decreasing (or non-increasing) sequence such that $x_0 \leq ... \leq x_k \geq ... \geq x_{n-1}$ for $0 \leq k < n$

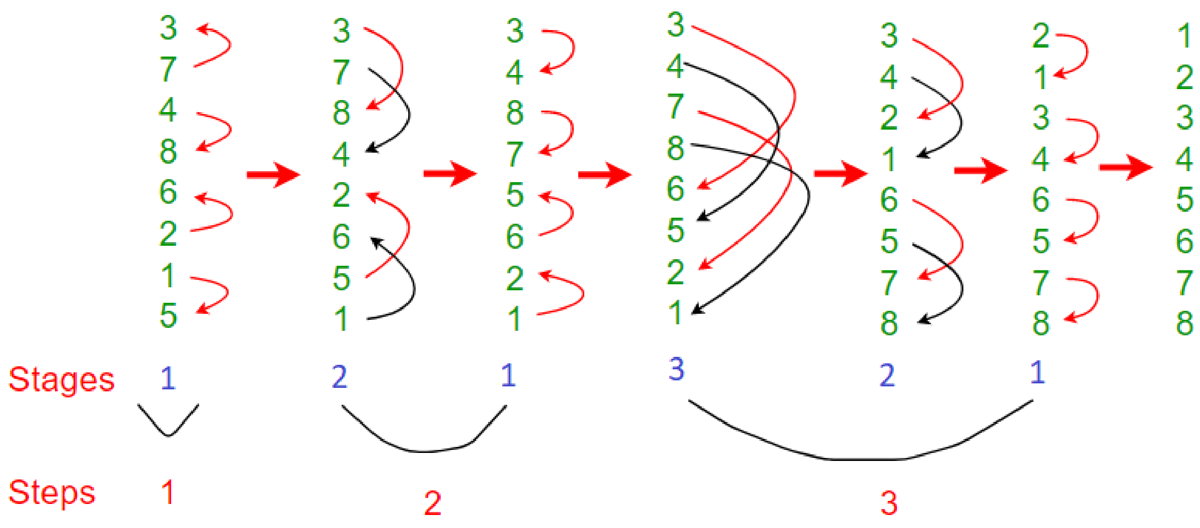


Figure 1.1: Workflow of the bitonic sort

Bitonic sort works by creating a bitonic sequence and then by exchanging values, according to the type of sort, between the ascending and descending sequence, the time complexity of the algorithm is $O(\log_2^2 n)$ for worst, average and best case.

2 Performance objective

We decide to evaluate the performance with two different points of view: the first from the user perspective, that appreciate a short execution time and the second one from developer point of view, that wants a program that can scale with the number of logical cores and that can bring a good enough improvements with respect to mono thread solutions; this parameter is measured in our case with the speedup. We set the threshold in regards to execution time on big arrays below or equal to 2 seconds, and regarding the speedup to have 50% or more in respect of the number of logical cores. Both of these objective were achieved with the use of multi-threading in CPU, however we decide to minimize the execution time.

3 Hardware - CPU

The implementation of the program was executed on two different machine, the first one presenting an AMD Ryzen 5625U as CPU, the other one an Apple M2 Pro.

3 .1 AMD Ryzen 5625U

The Ryzen 5625U is one of the latest addition to the 5000 series in regards of laptop CPUs. It presents 6 physical core and 12 logical ones. In the following are the specifications.

- CPU speed: 2.30 Ghz
- Physical/Logical cores: 6/12
- Cache:
  - L1: 192Kb IC - 192Kb DC (32Kb per core - 8 ways)
  - L2: 3Mb (8 ways)
  - L3: 16Mb (16 ways)

3 .2 Apple M2 Pro

Starting from 2020 Apple announced the shift from Intel processors to ARM-based ones. The M2 Pro is the second generation of Apple Silicon chips designed by Apple for the pro-notebook lineup, it integrates CPUs, GPUs and NPUs into a single package. It has 40 billions of transistors built on 5nm+ process and 200GB/s of memory bandwith thanks to unified memory architecture. The model we use for benchmarking the program has the following charateristics: 10 core CPU

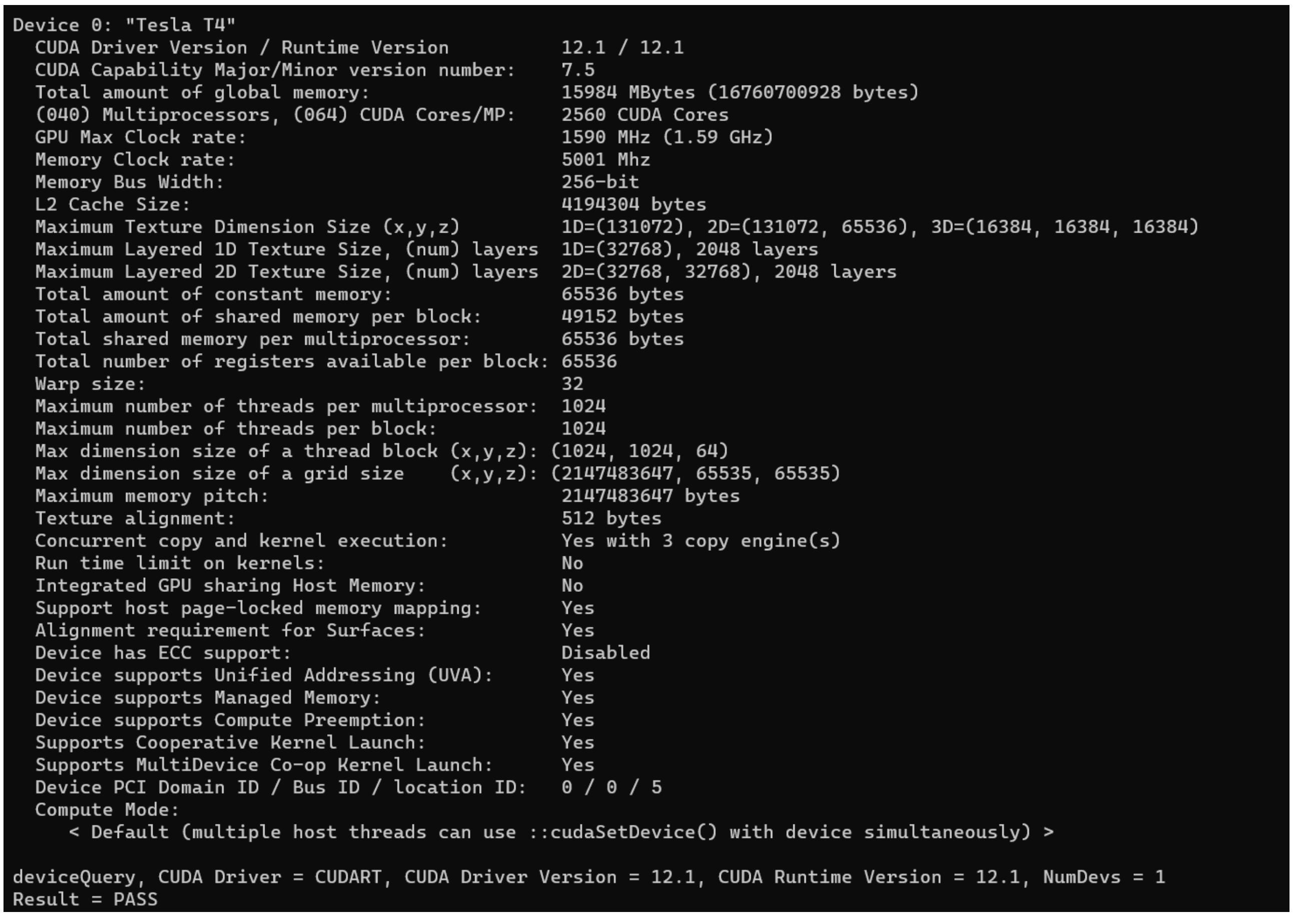
- 6 power-efficiency cores (up to 3.4 GHz). These cores have 128Kb of L1 instruction cache, 64 Kb for the L1 data cache and 4 Mb of shared L2 cache
- 4 high-performance cores (up to 3.7GHz). These cores have 192Kb of L1 instruction cache, 128 Kb for the L1 data cache and 32 Mb of shared L2 cache.

M2 Pro also has a system-level cache of 24Mb, used also by the other elements of the SoC. The size of the cacheline is 128B.

4 Hardware - GPU

4 .1 Nvidia Tesla T4

Both of us have not a Nvidia GPU so we use the one provided by the university. The model used is Nvidia T4, it is built on Turing architecture, launched in 2018. The graphic card has 2560 cuda cores, 40 streaming multiprocessors and 16 GB GDDR6 with 300 GB/sec of bandwith with ECC support (disabled in our case). The bus is 256 bit wide and the interconnect bandwith is at maximum 32 GB/sec. The maximum number of threads is 1024 per block.



```
Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          12.1 / 12.1
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15984 MBytes (16760700928 bytes)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 5
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.1, CUDA Runtime Version = 12.1, NumDevs = 1
Result = PASS
```

Figure 1.2: cudaDeviceQuery: specific of the Tesla T4

Chapter 2

CPU implementation

The very first step of the project was to implement the algorithm and to improve the performance of it by the use of multi-threading.

1 Sequential and parallel version

We started by performing some test on the sequential, mono-thread version. The code used for this can be found at the following link: `https://www.geeksforgeeks.org/bitonic-sort/`.

From this point we applied some modification to implement multi-threading, due to the divide et impera methodology used by the algorithm the main modification was to recursively creating new thread up until the desired number of working thread and dividing the data on which they work on.

1 .1 How tests were conducted

In order to evaluate the time of the program we use time checkpoints, with the implementation of high_resolution_clock provided by the C++ standard library. We decide to evaluate only the call to bitonic_sort fuction and exclude the allocation, the filling and the sorting verification of the array; this because would introduce another possible source of randomness inside the results. For statistically sound values we repeated the measurement 30 time, and try to minimize the impact of OS by closing all possible applications. Tests performed by AMD machine were done using WSL (Windows Subsystem for Linux) and the ones by ARM were done using macOS, both machines use g++ as compiler. The time displayed by the program is the minimum of the 30 iterations, in order to exclude the possible effects of other processes and the OS.

2 First result and observation

The first measurements were done on array from 2^8 elements up to 2^{16}, for reference these arrays from here on out will be called "small" arrays, meanwhile those from 2^{18} to 2^{24} will be called "big" arrays. In both case arrays size moves at steps of 2, another requirements for the array size is to be a power of 2.

After some measurements we got the following result.

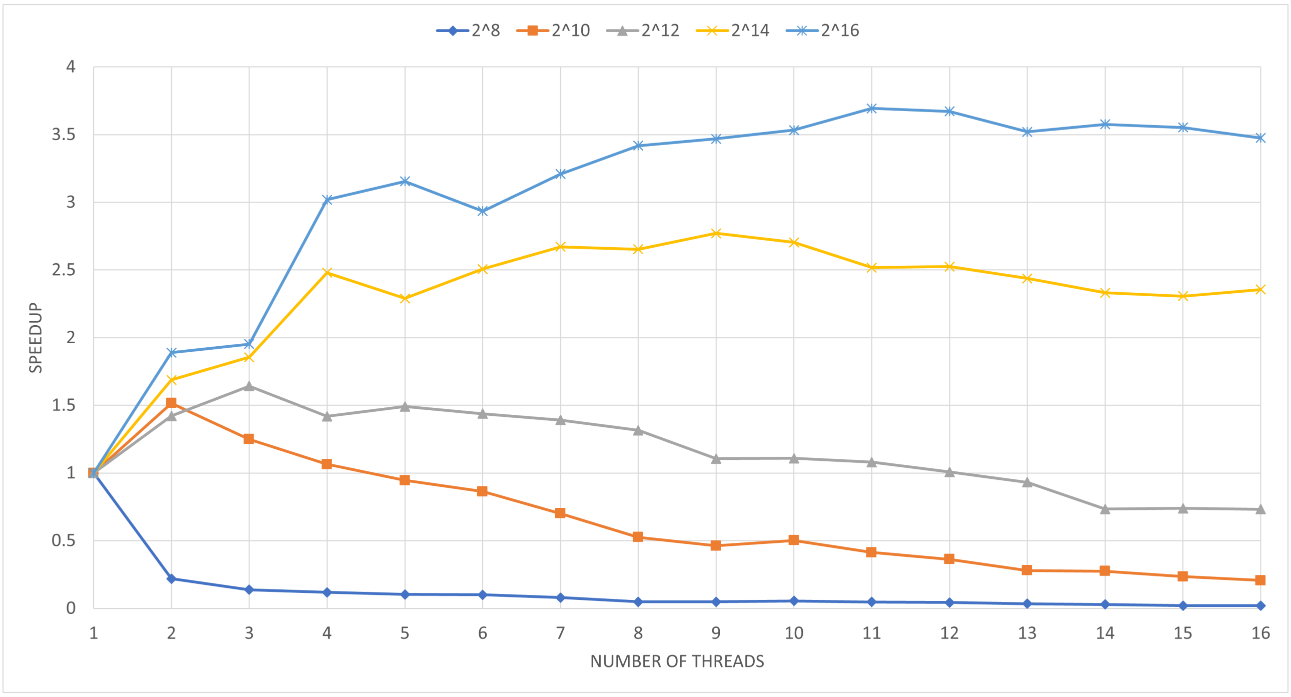


Figure 2.1: Speedup on small arrays - AMD

As can be seen on small arrays the multi-threading does not improve performance much and, in the case of 2^8, straight up worsen it. Similar results were produced by the ARM architecture.

On bigger arrays there is a difference, here multi-threading does bring an improvement, but it is still too small given the fact that we cannot reach the line of speedup equal to 5 even when we are using 10 logic cores. Similar although worse result were produced by the AMD architecture that can utilize 12 logical cores.

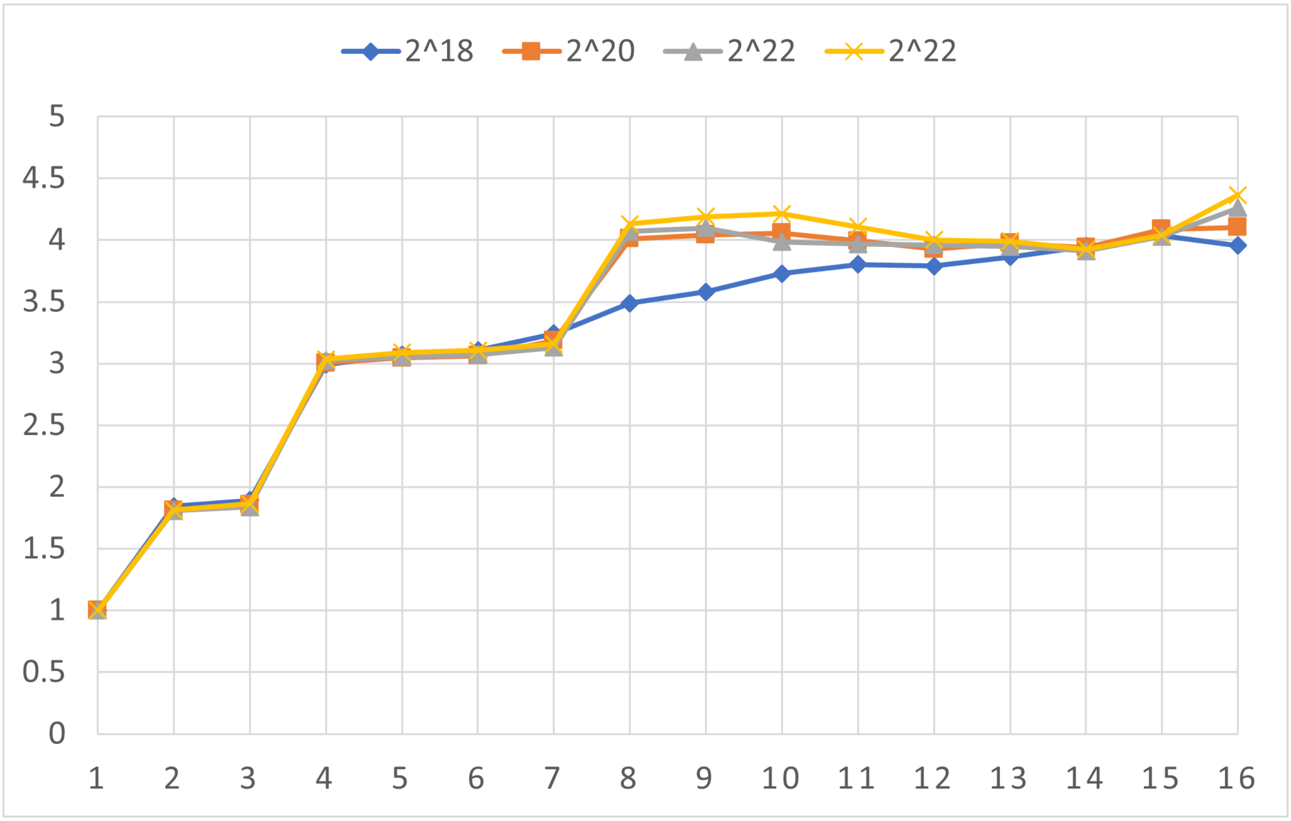


Figure 2.2: Speedup on big arrays - ARM

2 .1 Best and worst case

As previously stated the time complexity for bitonic sort is $O(\log_2^2 n)$ for worst, best and average case, this is confirmed by the results in the following figure.

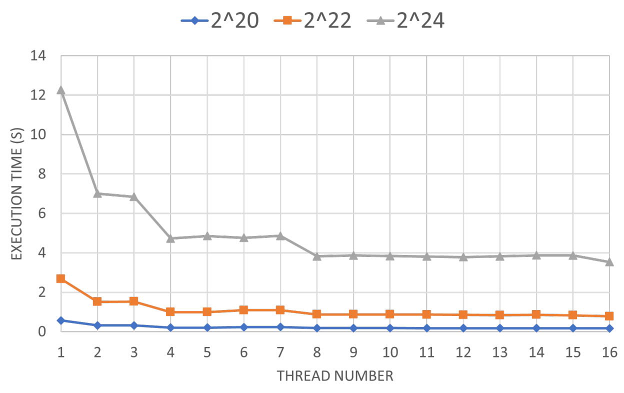


Figure 2.3: Worst case - AMD

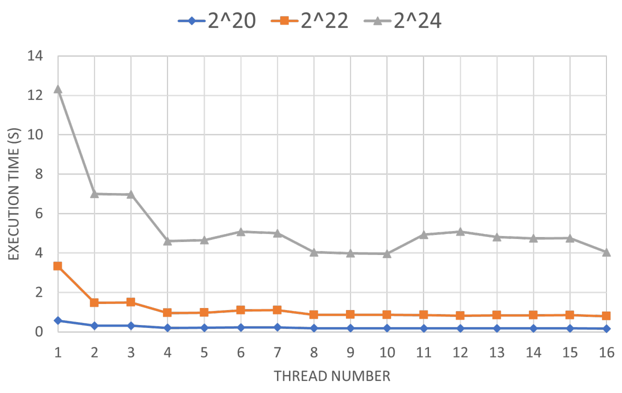


Figure 2.4: Best case - AMD

2 .2 AMD optimization: no std::swap

The following step was performed with the help of a profiler, in the case of AMD it was μProf (`https://www.amd.com/en/developer/uprof.html`), and, after a short analysis, it was noticed that in the phase of merging and exchanging of values the call to the standard library function *swap* was introducing useless overhead just to swap two values. So it was removed in favor of the use of a third temporary variable of support used to exchange integers. This produced the following results:

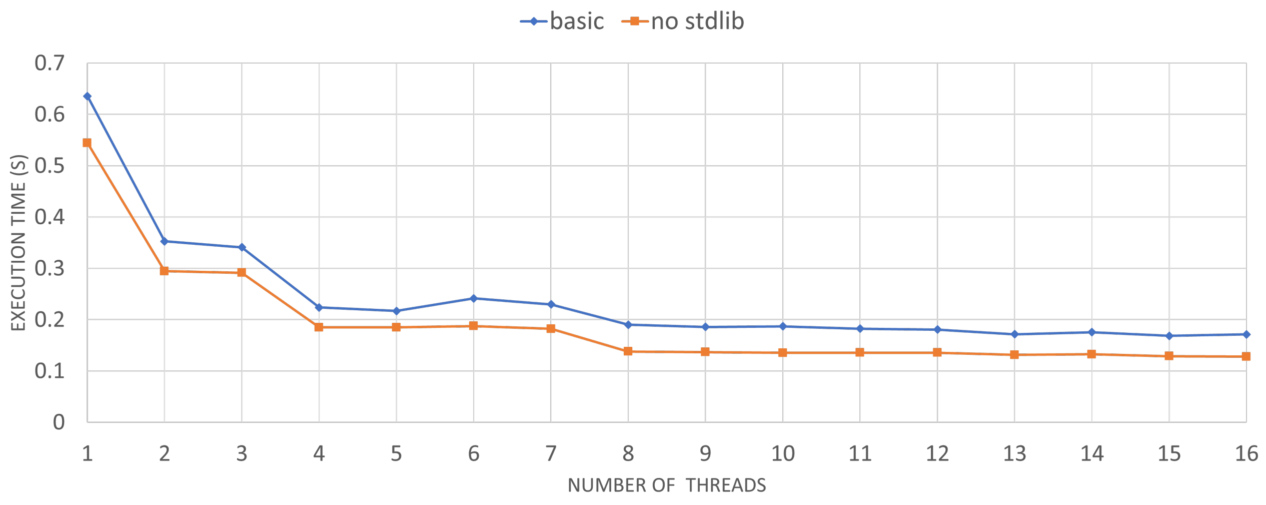


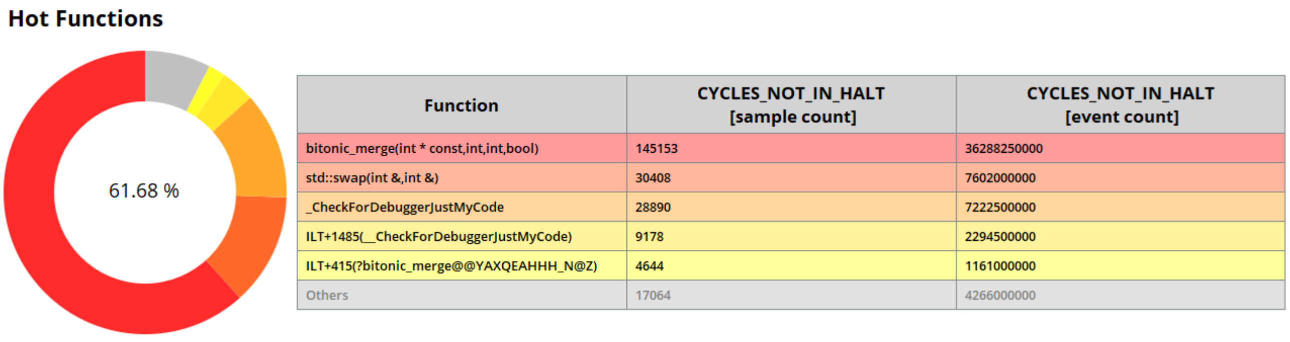
Figure 2.5: Confront between before and after removing std::swap - 2^{20} array

It is important to note that these results were not also achieved on the ARM architecture where there was no difference between the two cases. This could be due to different compiler strategies for different architectures and/or the cache size.

3 Optimization: parallel merge

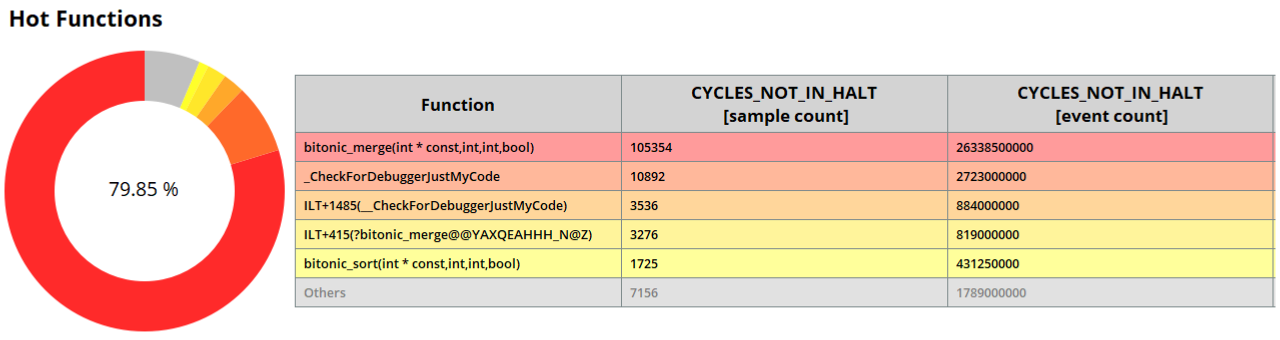
We then proceeded to analyze more in depth the program with the help of the already mentioned profiler and we found that the merging part of the software was still the most expensive and time consuming part of the algorithm. The following figures were taken from the μProf profiler on the following configuration:

- Thread: 15 (best result time-wise)
- Array size: 2^{20}



Function	CYCLES_NOT_IN_HALT [sample count]	CYCLES_NOT_IN_HALT [event count]
bitonic_merge(int * const,int,int,bool)	145153	36288250000
std::swap(int &,int &)	30408	7602000000
_CheckForDebuggerJustMyCode	28890	7222500000
ILT+1485(__CheckForDebuggerJustMyCode)	9178	2294500000
ILT+415(?bitonic_merge@@YAXQEAHHH_N@Z)	4644	1161000000
Others	17064	4266000000

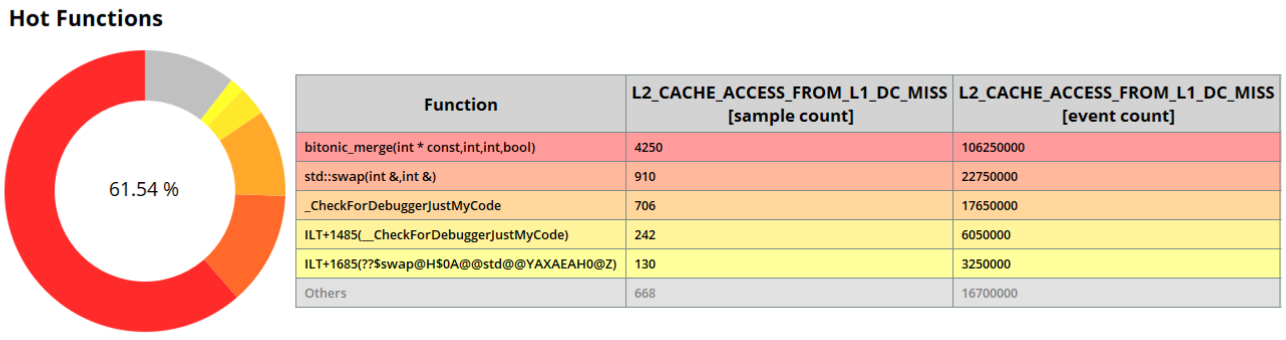
Figure 2.6: Cycles not in halt - std::swap



Function	CYCLES_NOT_IN_HALT [sample count]	CYCLES_NOT_IN_HALT [event count]
bitonic_merge(int * const,int,int,bool)	105354	26338500000
_CheckForDebuggerJustMyCode	10892	2723000000
ILT+1485(__CheckForDebuggerJustMyCode)	3536	884000000
ILT+415(?bitonic_merge@@YAXQEAHHH_N@Z)	3276	819000000
bitonic_sort(int * const,int,int,bool)	1725	431250000
Others	7156	1789000000

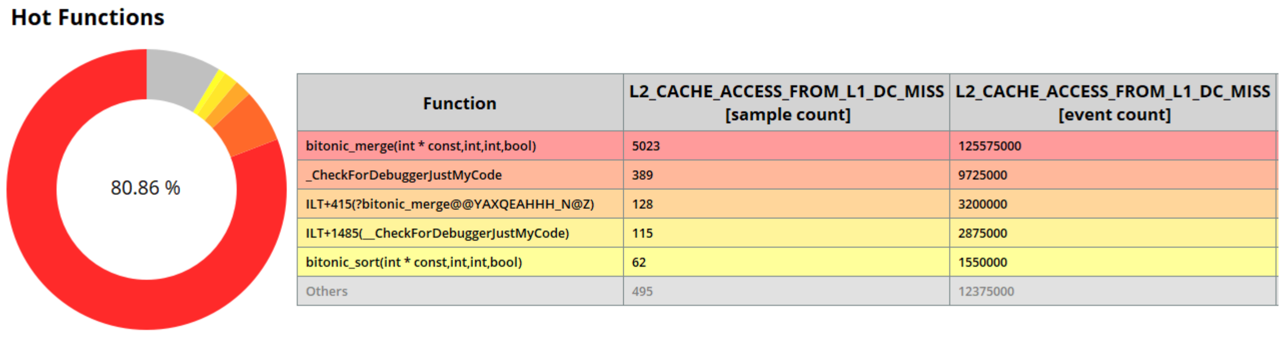
Figure 2.7: Cycles not in halt - no std::swap

As can be seen, in this case, the *"time"* spent in bitonig merge is smaller than the one spent in bitonic merge plus swap in the previous case, however the relative weight of the function is heavily increased.



Function	L2_CACHE_ACCESS_FROM_L1_DC_MISS [sample count]	L2_CACHE_ACCESS_FROM_L1_DC_MISS [event count]
bitonic_merge(int * const,int,int,bool)	4250	106250000
std::swap(int &,int &)	910	22750000
_CheckForDebuggerJustMyCode	706	17650000
ILT+1485(__CheckForDebuggerJustMyCode)	242	6050000
ILT+1685(??$swap@H$0A@@std@@YAXAEAH0@Z)	130	3250000
Others	668	16700000

Figure 2.8: L2 access from L1 misses - std::swap

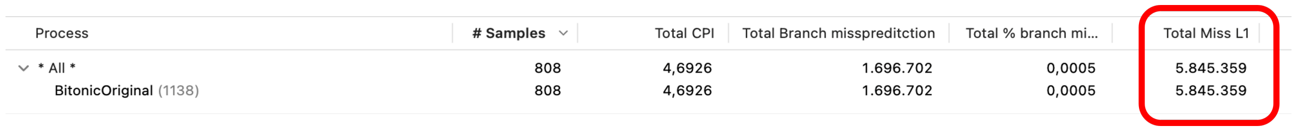


Function	L2_CACHE_ACCESS_FROM_L1_DC_MISS [sample count]	L2_CACHE_ACCESS_FROM_L1_DC_MISS [event count]
bitonic_merge(int * const,int,int,bool)	5023	125575000
_CheckForDebuggerJustMyCode	389	9725000
ILT+415(?bitonic_merge@@YAXQEAHHH_N@Z)	128	3200000
ILT+1485(__CheckForDebuggerJustMyCode)	115	2875000
bitonic_sort(int * const,int,int,bool)	62	1550000
Others	495	12375000

Figure 2.9: L2 access from L1 misses - no std::swap

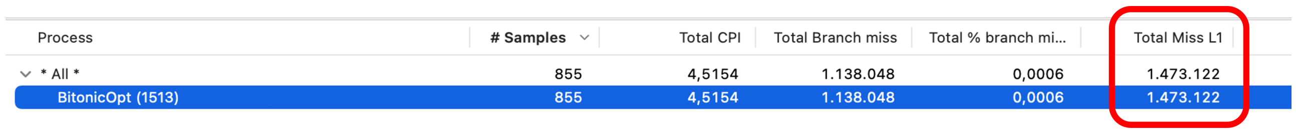
Regarding the L2 access from L1 misses, there was a very small decrease, but, as before, the relative weight of the function increased.

Given all these information, we decided to change the way in which *bitonic merge* was executed: from a sequential way to a parallel one. This small change boosted performance heavily and it is justified by the number of L1 cache misses. The following pictures were taken trough the use of *Instruments*, part of the Xcode IDE on the Apple M2 Pro.



Process	# Samples	Total CPI	Total Branch misspreditction	Total % branch mi...	Total Miss L1
* All *	808	4,6926	1.696.702	0,0005	5.845.359
BitonicOriginal (1138)	808	4,6926	1.696.702	0,0005	5.845.359

Figure 2.10: L1 misses - sequential bitonic merge



Process	# Samples	Total CPI	Total Branch miss	Total % branch mi...	Total Miss L1
* All *	855	4,5154	1.138.048	0,0006	1.473.122
BitonicOpt (1513)	855	4,5154	1.138.048	0,0006	1.473.122

Figure 2.11: L1 misses - parallel bitonic merge

As can be seen in the previous figures, the number of L1 misses decreased in the case of an array size of 2^{20}, the parallelization of the *bitonic merge* allowed to distribute the data load on more CPUs instead of just one, thus reducing the number of cache misses, due to the use of all available L1 caches.

3 .1 Final results and optimization

In this section we showed the final results achieved confronting the two architecture on an array size of 2^{24}. The first graphs show the improvements of the execution time simply by optimizing the code.

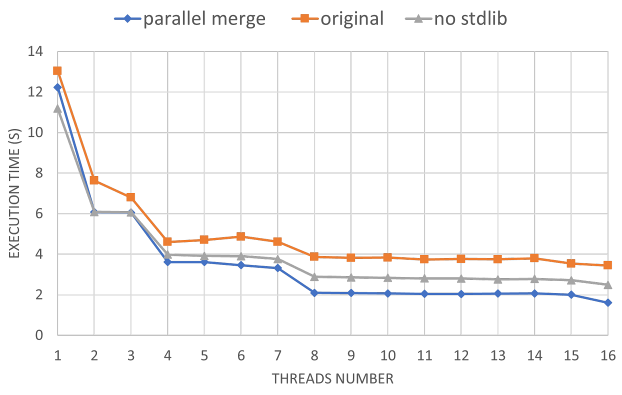


Figure 2.12: AMD final comparison - time

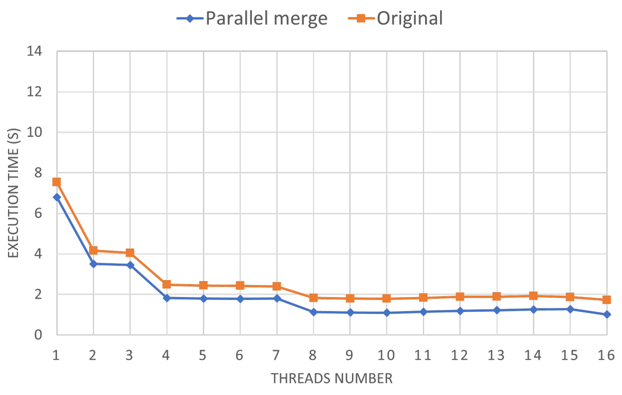


Figure 2.13: ARM final comparison - time

The parallel merge outperforms the original and no std::swap version of the code in AMD case, and the original multi-threading version ARM. However both of these solutions do not provide good enough results regarding the time constrains especially AMD. So the solution was to use the optimization at compiling through the use of the flag O2.

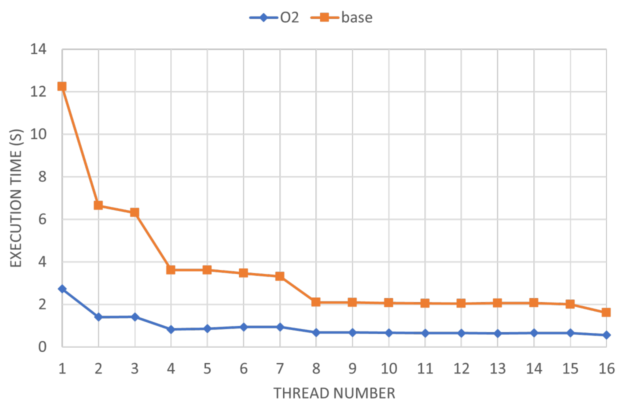


Figure 2.14: AMD time comparison with O2

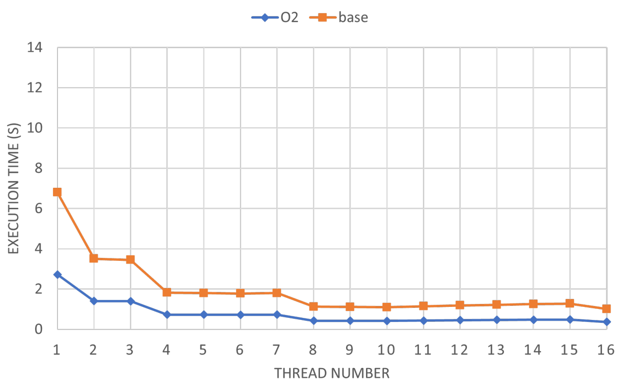


Figure 2.15: ARM time comparison with O2

This greatly reduce the execution time for the sorting operations especially regarding AMD, however this upgrade comes with a minus.

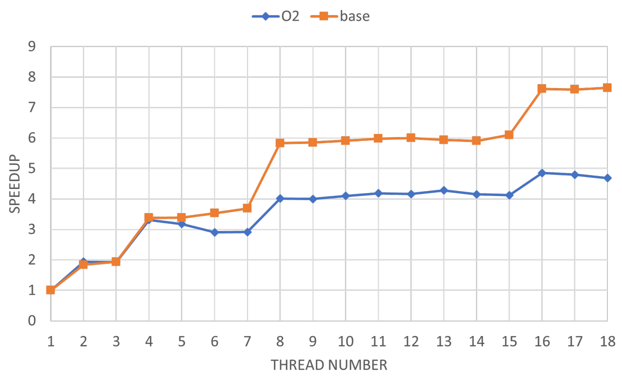


Figure 2.16: AMD speedup comparison with O2

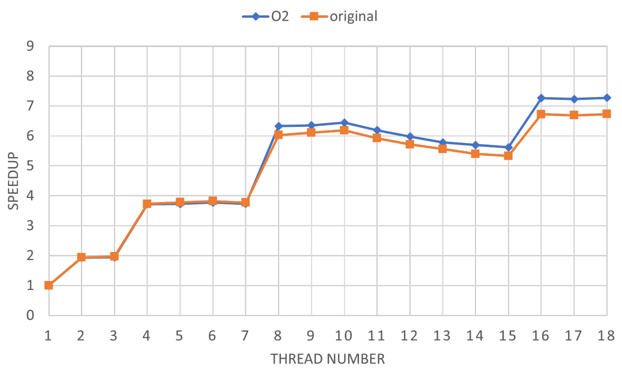


Figure 2.17: ARM speedup comparison with O2

The speedup slightly improves with the ARM architecture, but , with the AMD one, it diminish greatly. As dire as it sounds, this worsening should not be taken too much into consideration. With the O2 flag the execution time of the mono-thread case diminish and so does the speedup.

4 CPU conclusion

The use of multi-threading with a proper palatalization of the code brings satisfactory results, the execution time is below 2 second for arrays up to 2^{24} elements meaning that we can quickly sort 64 Mb of data. Similarly the speedup almost reach 8 which are impressive results for both AMD and ARM.

Chapter 3

GPU implementation

The next big step was to implement a GPU version of the algorithm. The test were performed with a Nvidia Tesla T4 offered by the university. In the introduction chapter can be seen a screen-shot of the hardware statistics, but to recapitulate here are listed the most important information:

- Compute power: 7.5
- Cuda cores: 2560
- Streaming multiprocessor: 40
- Warp size: 32
- Max threads per block: 1024
- Memory 16 Gb GDDR6 - bandwidth: 300 GB/s
- Interconnection bandwidth: 32 Gb/s

1 Timing

The algorithm was tested on arrays varying in size from 2^{16} to 2^{30} at steps of two, each run was repeated 30 times to get statistically sounds results. The number of threads for blocks was a number scaling on the power of 2 from 32 to 1024, the number of blocks was dynamically calculated as following: $(ArraySize + numberOfThreads - 1)/numberOfThreads$.

We chose as base case the configuration with 32 threads per block which corresponds to the warp size.

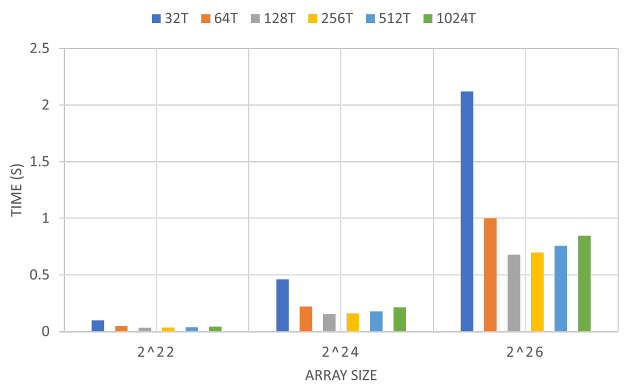


Figure 3.1: Execution time on smaller arrays

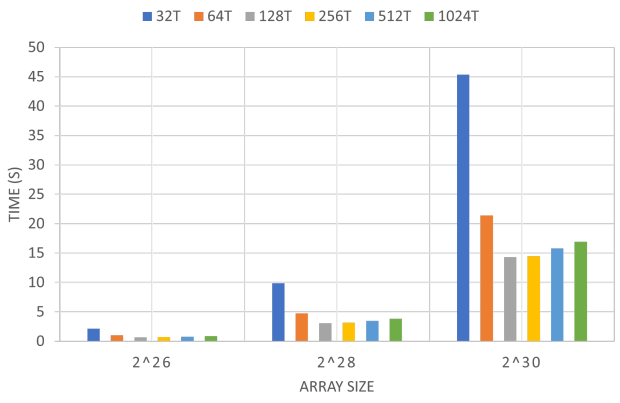


Figure 3.2: Execution time on bigger arrays

As can be seen in the graphs we have a best case with 128 threads per block. This is because the Nvidia Tesla T4, based on the Turing architecture has a SM divided into 4 partitions, so naturally the 4 warps of 32 threads map perfectly with the hardware partitions with no overhead.

2 Analysis of the results

We first profile the code with two different timers, one that consider only the kernel calls and, the second one, that includes also the cudaMemcpy in both directions (host to device and device to host). In both case we use *cudaEvent_t*.

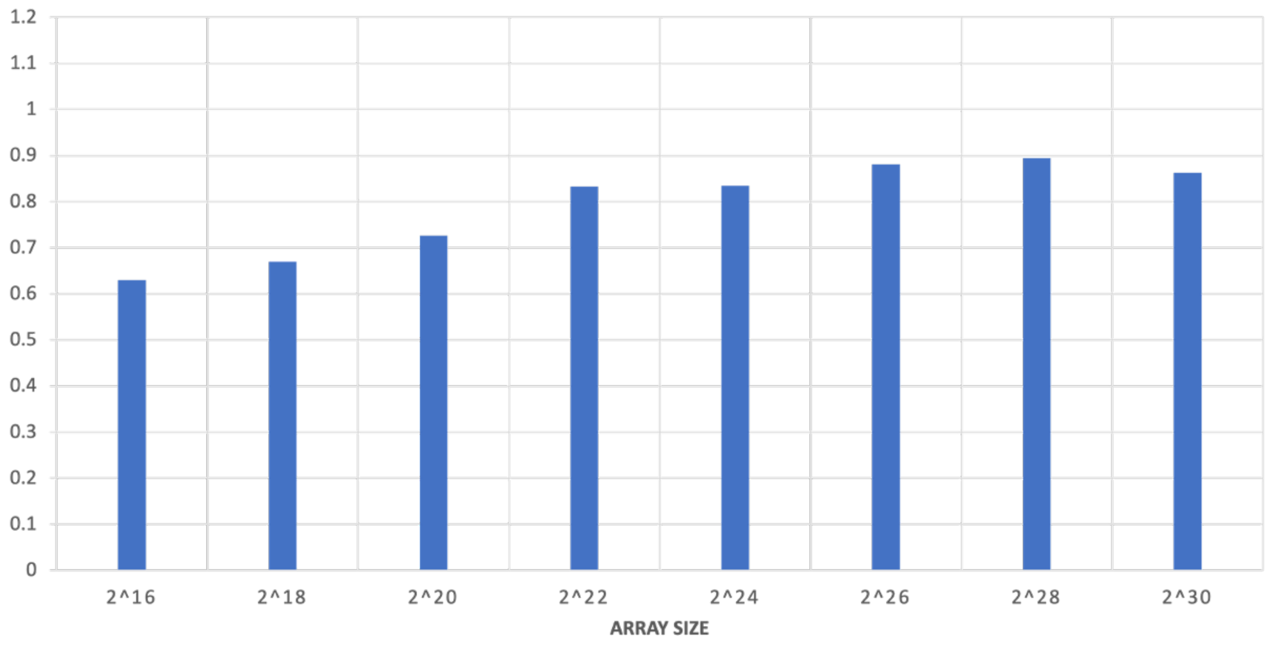


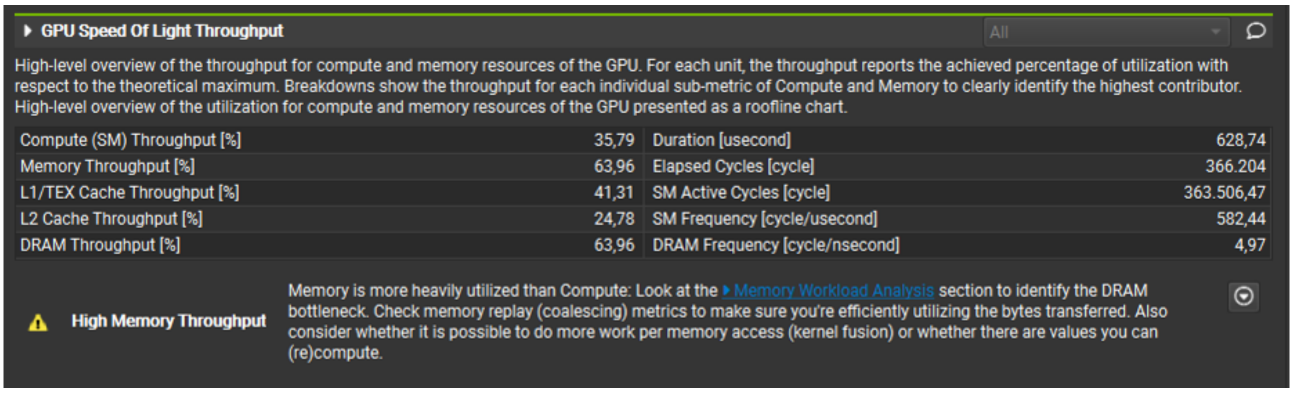
Figure 3.3: Computing time normalized on execution time

This bring us to the conclusion that on big arrays (2^{26} and over) the cudaMemcpy does not impact heavily the total execution time, so using techniques like overlapping data transfer would not bring impactful improvements. For smaller arrays there would be an improvement, however the execution time is already very low so that, improve the data transfer phase would not bring impactful improvements.

For further analysis of the code the Nvidia NSight tool was used and we noticed the following things (we used the 128 threads configuration on an array of 2^{24} elements):

- The compute throughput is approximately 36%.
- The memory throughput is approximately 64%.

These results are bounded to the program we analyze, since sorting in general, by its nature memory intensive, since the real computational part are only comparisons and swap between elements.



GPU Speed Of Light Throughput

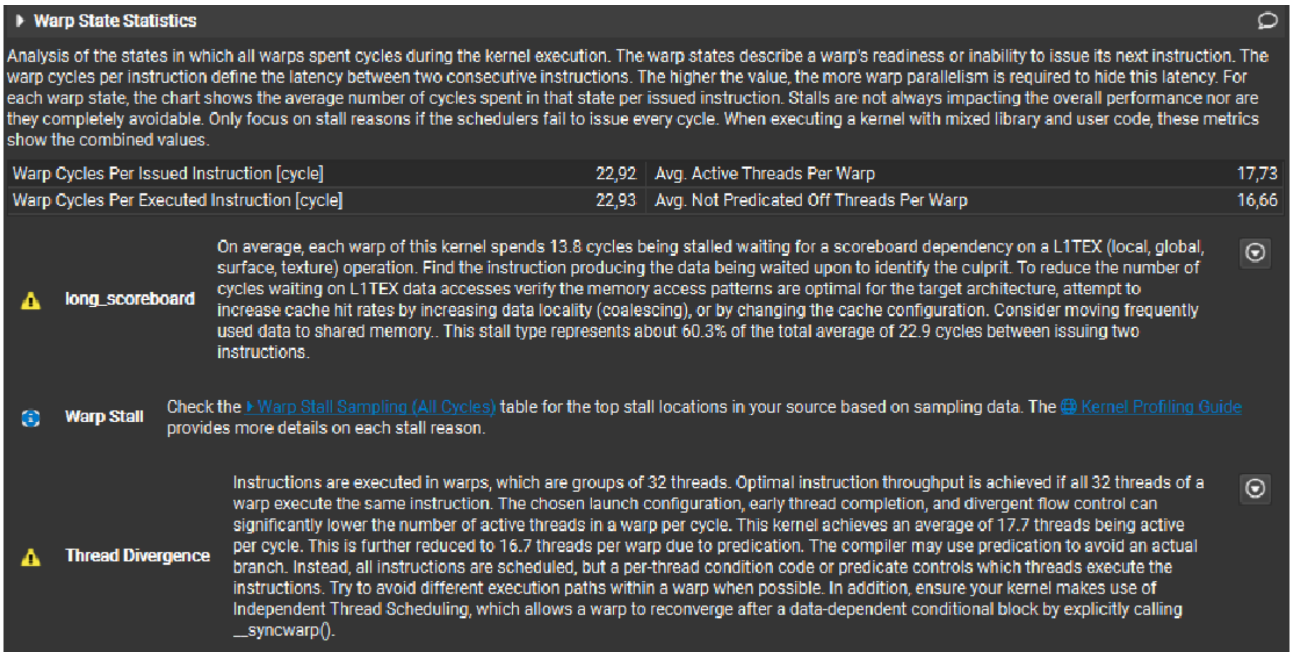
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	35,79	Duration [usecond]	628,74
Memory Throughput [%]	63,96	Elapsed Cycles [cycle]	366.204
L1/TEX Cache Throughput [%]	41,31	SM Active Cycles [cycle]	363.506,47
L2 Cache Throughput [%]	24,78	SM Frequency [cycle/usecond]	582,44
DRAM Throughput [%]	63,96	DRAM Frequency [cycle/nsecond]	4,97

High Memory Throughput — Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis section to identify the DRAM bottleneck. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.

Figure 3.4: Nvidia NSight statistics throughput

As the profiler suggests a solution could be brought by the use of kernel fusion or, more in general, by optimizing the kernel access to memory.



Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	22,92	Avg. Active Threads Per Warp	17,73
Warp Cycles Per Executed Instruction [cycle]	22,93	Avg. Not Predicated Off Threads Per Warp	16,66

long_scoreboard — On average, each warp of this kernel spends 13.8 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider moving frequently used data to shared memory.. This stall type represents about 60.3% of the total average of 22.9 cycles between issuing two instructions.

Warp Stall — Check the Warp Stall Sampling (All Cycles) table for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

Thread Divergence — Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 17.7 threads being active per cycle. This is further reduced to 16.7 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

Figure 3.5: Nvidia NSight statistics warp

Another interesting information is given by the Warp State Statistics, where, excluding the remark of the long scoreboard regarding the memory access, it is signaled that only half threads are active per warp in average. This is due to the kernel using if branches causing thread divergence; a brancheless solution was attempted which solve this issue, however it brought worse timing results.

Chapter 4

Conclusion

Although we could not improve further the implementation of the algorithm on GPU, it is important to notice that with this solution it is possible to sort up to an array of 2^{26} integer elements (256 Mb of data) in less than a second and one of 2^{30} (2 Gb of data) in 14 seconds. This is why we consider the achieved result satisfactory. Due to the differences in hardware it is not advisable to compare CPU results with those GPU, however we decided to summarize all our findings in the following table to highlight the analysis outcome.

Array size	AMD	ARM	GPU (128 th)
2^16	50,80 ms (1 th) 13,75 ms (12 th)	17,90 ms (1 th) 4,00 ms (18 th)	1,05ms
2^18	132,97 ms (1 th) 39,89 ms (13 th)	77,80 ms (1 th) 19,23 ms (14 th)	2,29ms
2^20	635,70 ms (1 th) 171,66 ms (13 th)	358.24 ms (1 th) 87,23 ms (15 th)	7,42ms
2^22	2938,17 ms (1 th) 790,80 ms (18 th)	1648,83 ms (1 th) 384,4 ms (17 th)	34,45ms
2^24	13039,70 ms (1 th) 3430,17 ms (17 th)	7542,47 ms (1 th) 1721,93 ms (17 th)	155,97ms
2^26	—	—	677,12ms
2^28	—	—	3046,14ms
2^30	—	—	14312,34ms

Table 4.1: Execution time of bitonic sort

Future analysis could focus on improving further the algorithm or analyze the same algorithm applied to float instead of integers and compare the result to find better solution for the first, especially with the great performance of GPU in floating point operations.