IN SUPREMÆ DIGNITATIS 1343

UNIVERSITÀ DI PISA

BITONIC SORT PERFORMANCES

COMPUTER ARCHITECTURE PROJECT
A.A. 2023/2024

Giovanni Enrico Loni
Lorenzo Mancinelli

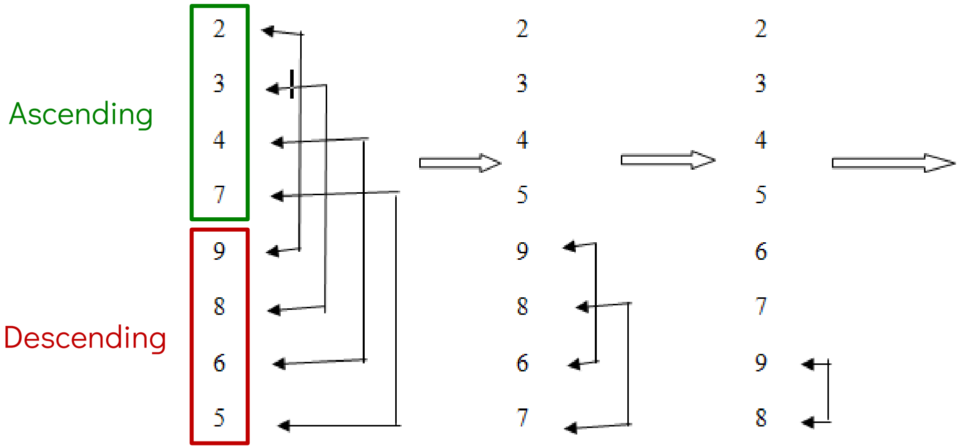
Table of contents

01
ALGORITHM & GOALS

Introduction to the algorithm and goals to be achieved

THE ALGORITHM

Bitonic sort breaks the data into "bitonic" sequences, then iteratively compare and swap elements within these sequences, progressively merging them until the entire data is sorted.



MAIN GOALS

CPU Implementation

Execution time for arrays with 2^{24} entries < 1 s.

GPU Implementation

Execution time for arrays with 2^{30} entries < 20 s.

Requirements reached! ✓

02

CPU Implementation

Study of performance on CPU as threads and load vary. Improving performance exploiting `-o2` and parallel merge.

HARDWARE & SOFTWARE

HARDWARE

AMD Ryzen 7 5800H

- 8 cores, 16 thread
- Cache:
  - L1D 256 KiB 8-way associative
  - L1I 256 KiB 8-way associative
  - L2 4 MiB 8-way associative
  - L3 16 MiB 16-way associative

OPERATIVE SYSTEM

- EndeavourOS Linux x86_64
- **Kernel**: 6.8.7-arch1-1

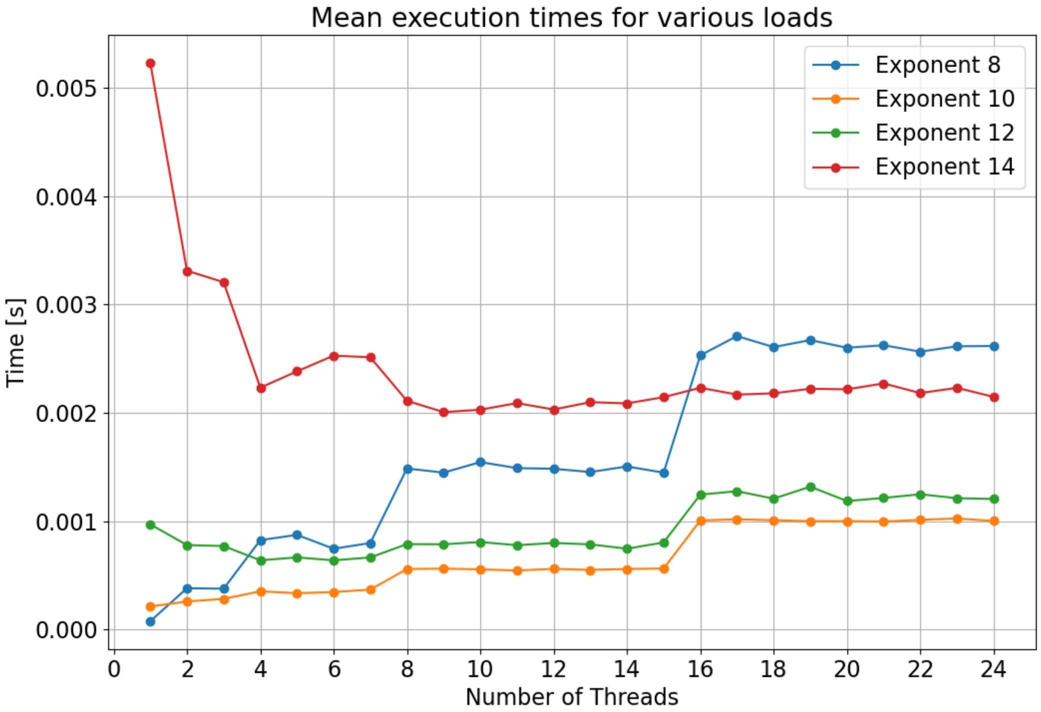
PROFILER

AMDμProf 4.2

COMPILER

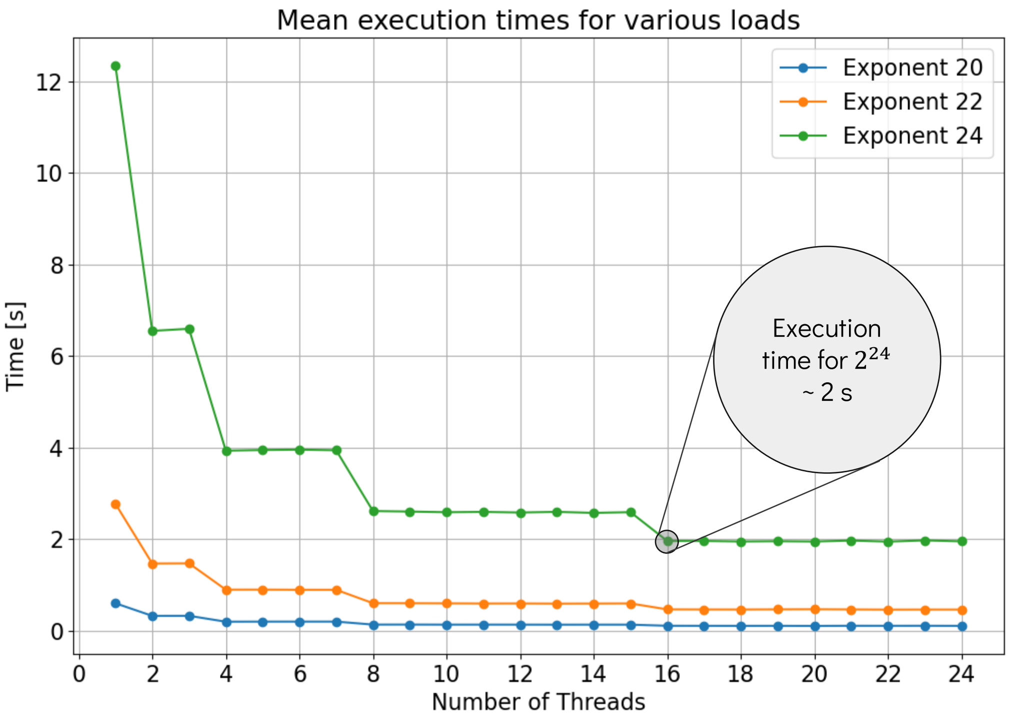
- `gcc 13.2.1`
- `-lrt -pthread`

EXECUTION TIME ON SMALL ARRAYS

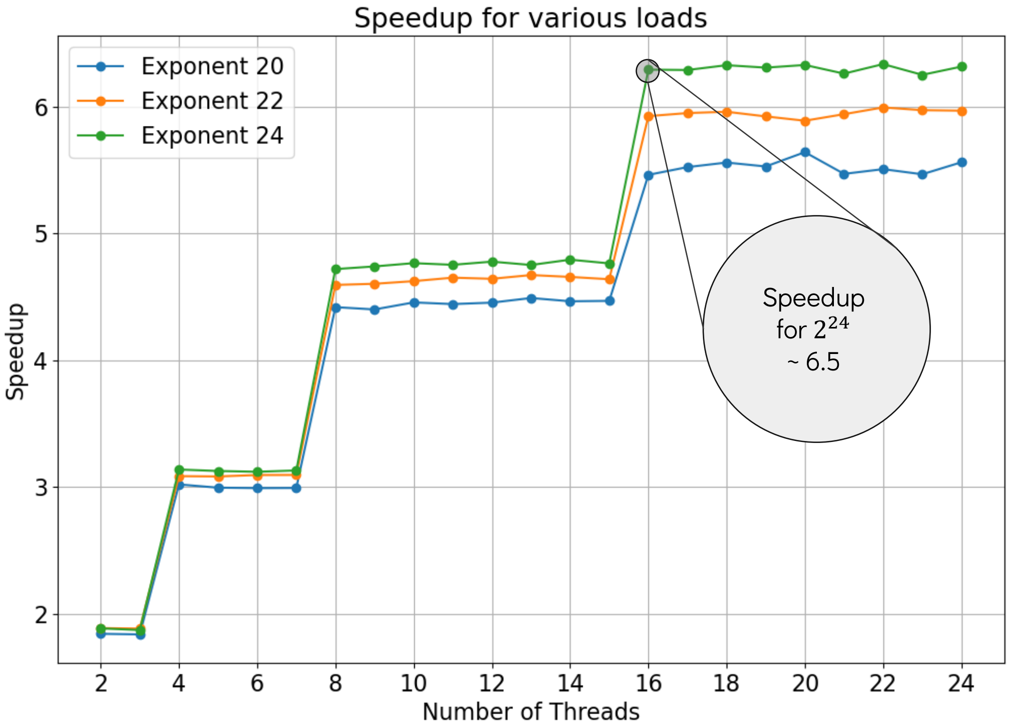


For exponents up to 12 it is better to use the sequential version of the algorithm!

EXECUTION TIME ON BIG ARRAYS

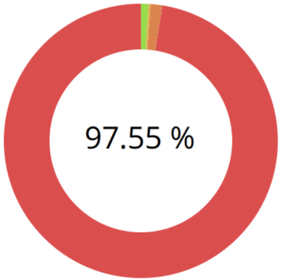


SPEEDUP ON BIG ARRAYS



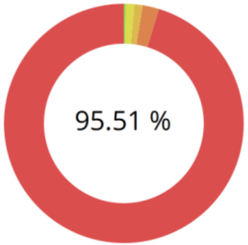
FUNCTION HOTSPOTS

- CPU TIME



Function
bitonic_merge(int*, int, int, bool)
bitonic_sort(int*, int, int, bool)
random
main
rand

- L1 MISSES



Function	L2_CACHE_ACCESS_FROM_L1_DC_MISS [sample count]
bitonic_merge(int*, int, int, bool)	85
random	2
ld-linux-x86-64.so.2!0x00008ae7	1
random_r	1
[PLT] rand	0

FUNCTION HOTSPOTS

- C++

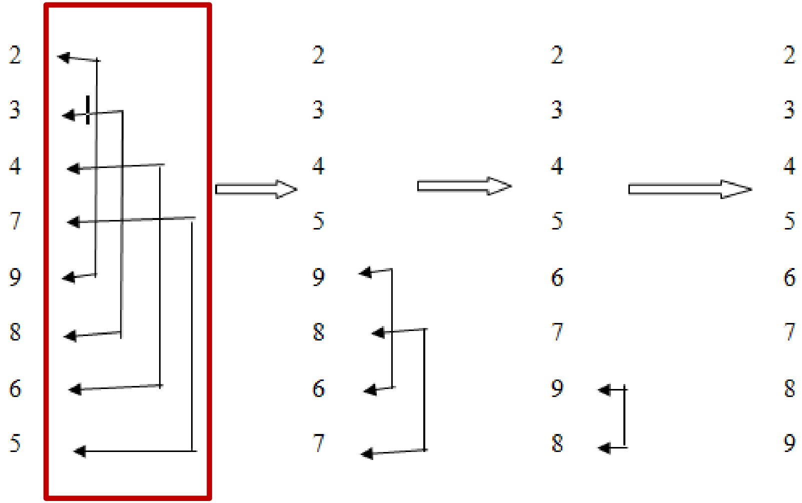
```
if ((arr[i] > arr[i + mid]) == direction) {…}
```

- Assembly

```
lea      rcx, [0+rax*4]
mov      rax, QWORD PTR [rbp-24]
add      rax, rcx
mov      eax, DWORD PTR [rax]
cmp      edx, eax
setg     al
movzx    edx, al
movzx    eax, BYTE PTR [rbp-36]
cmp      edx, eax
jne      .L22
```

Almost 25% of retired instructions of the entire program!

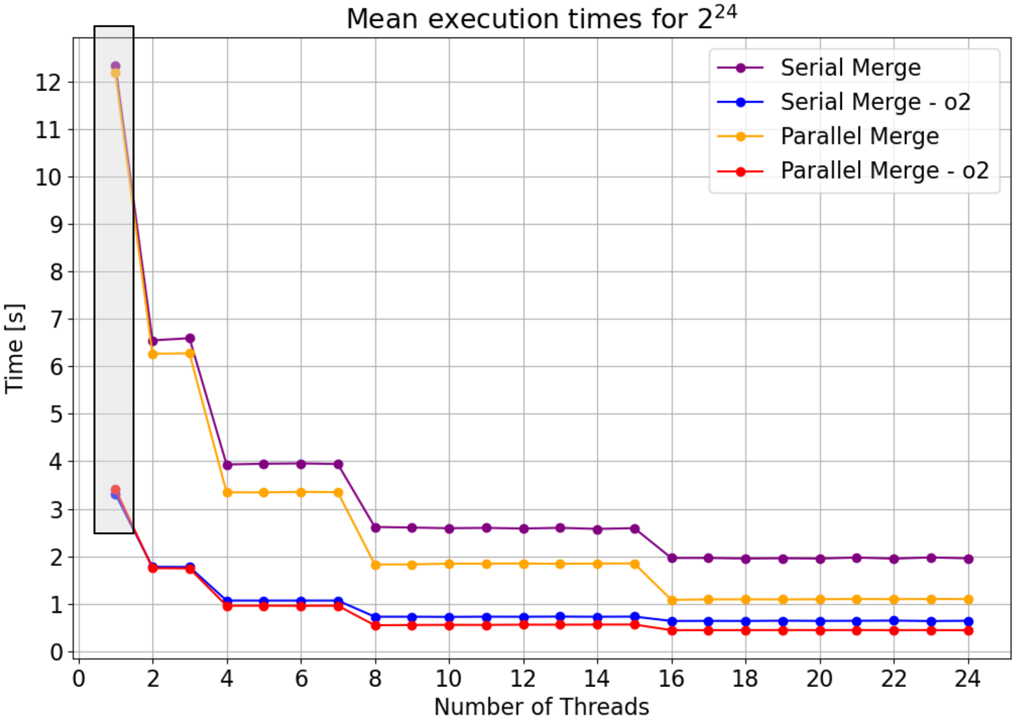
PARALLELIZATION OF BITONIC MERGE



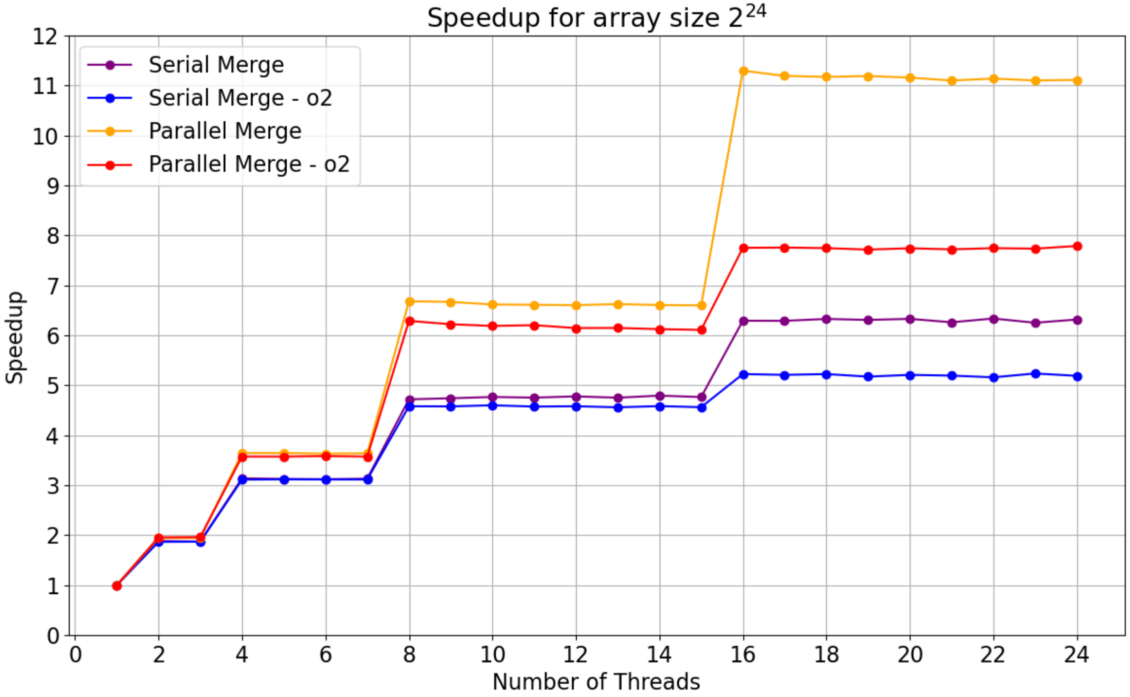
Swaps are executed in parallel, unlike the initial Bitonic Merge where they were executed sequentially.

EXECUTION TIME WITH PARALLEL MERGE

The improvement can be seen more in the single-thread case than in the multi-thread case



SPEEDUP WITH PARALLEL MERGE



Each speedup is related to its own optimization

GOALS ACHIEVED

EXECUTION TIME

- ~ Almost reached for parallel merge (1,09 s).
- ✓ Fully reached through –o2 (0,64-0,44 s).

SPEEDUP

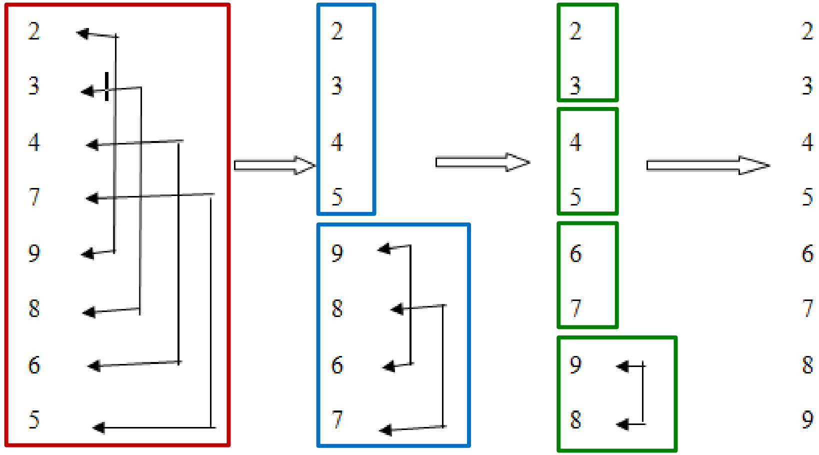
- ✓ Best value with parallel merge (x11).
- ~ Reasonable value through parallel merge –o2.

03

GPU Implementation

Study of performance on GPU as threads per block and load vary. Analysis with profiler.

NEW APPROACH FOR IMPLEMENTATION LOGIC



Instead of recursive calls for different sizes of the sequences,
swaps inside sequences with the same lenght are done in parallel.

Hardware & software

HARDWARE

NVIDIA Tesla T4

- Compute Power: **7.5**
- CUDA Cores: **2560**
- Streaming multi-processors: **40**
- Warp size: **32**
- Max threads per block: **1024**
- Memory: **16 GB GDDR6**
- Bandwidth: **300GB/s**
- Interconnection bandwidth: **32GB/s**

OPERATIVE SYSTEM

- Ubuntu 18.04.3 LTS
- **Kernel**: Linux 4.15.0-188-generic

PROFILER

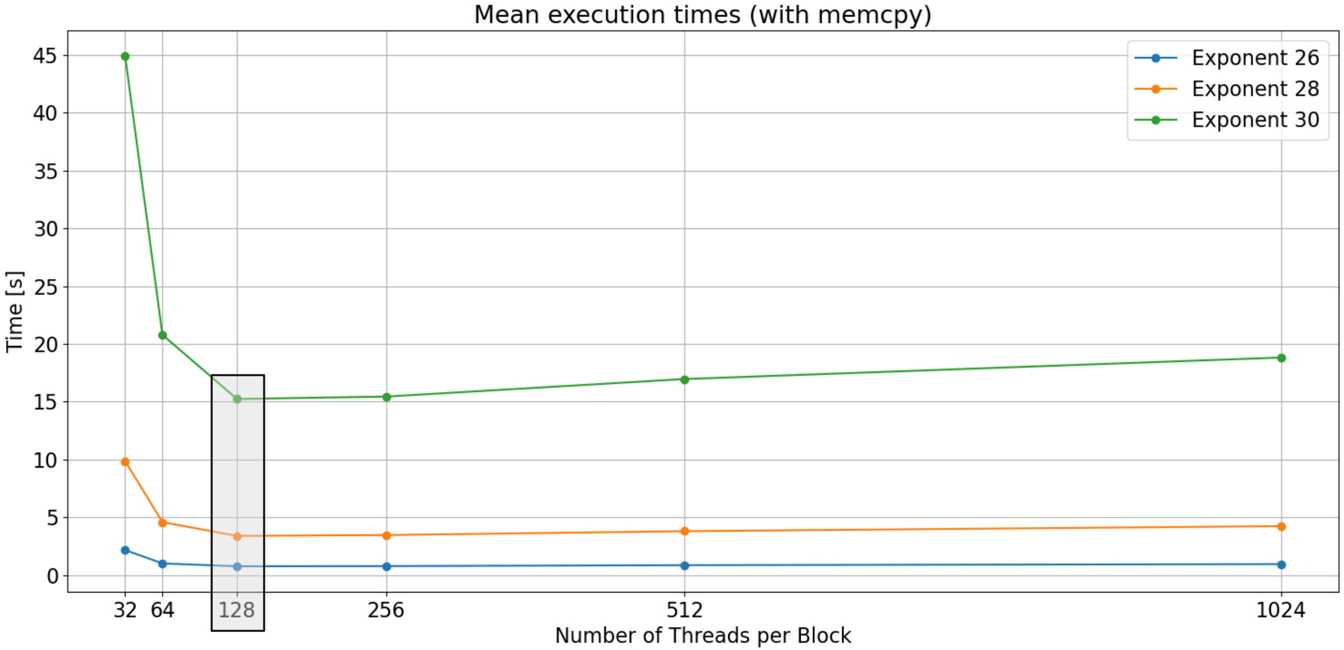
- Nsight Compute 2024.1.1

COMPILER

- `nvcc 9.1.85`

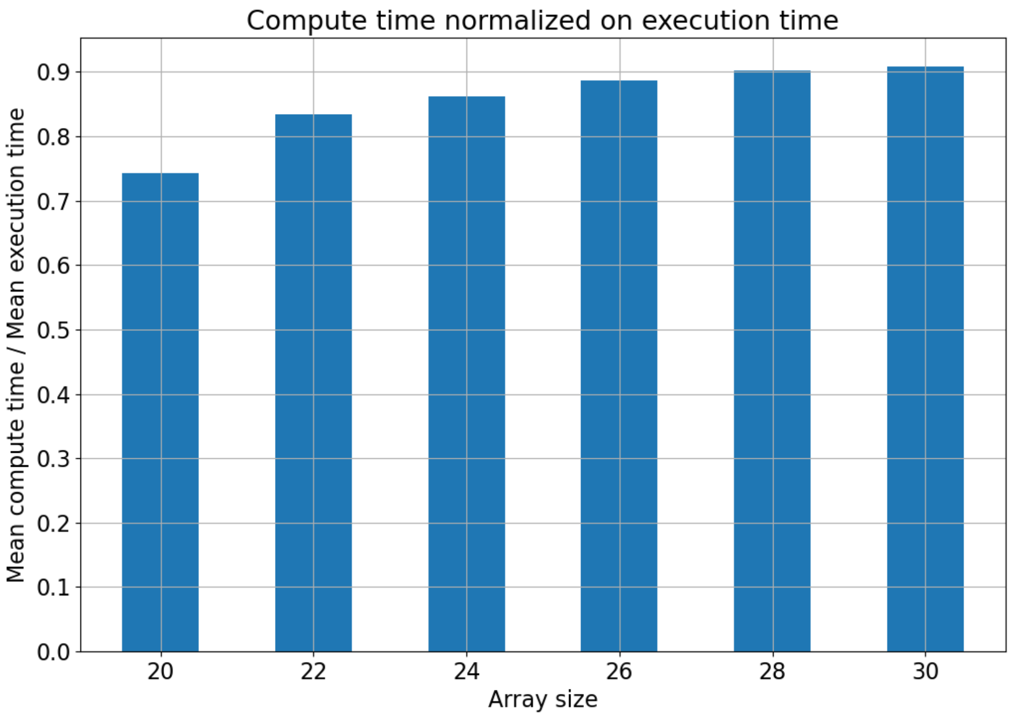
EXECUTION TIME WITH MEMCPY

```
#blocks = (arraySize + numThreads - 1) / numThreads
```



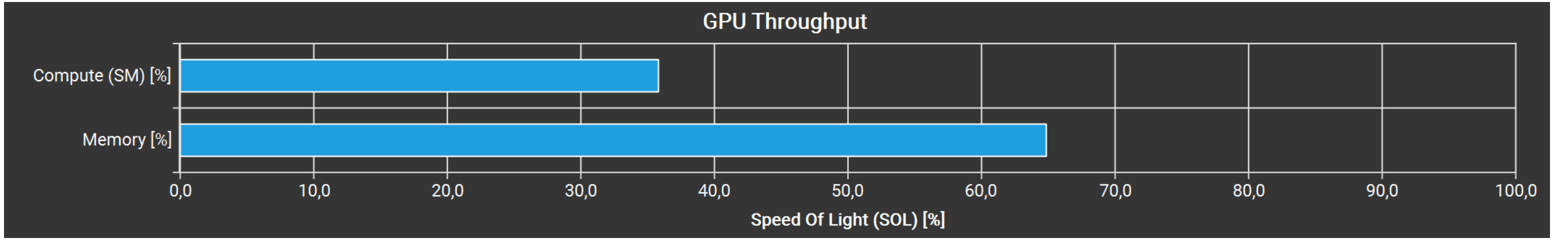
Best configuration: goal achieved!

EXECUTION TIME – IMPACT OF MEMCPY



GPU COMPUTE & MEMORY THROUGHPUT

More time spent exchanging data with memory than in the algorithm itself: possible **memory bottleneck.**

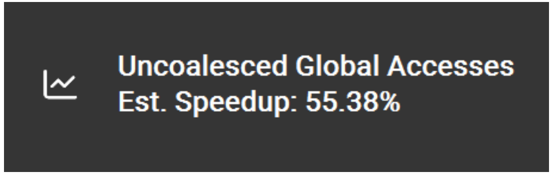


Recommended solutions

- Memory coalescence
- Kernel fusion

GPU COMPUTE & MEMORY THROUGHPUT

UNCOALESCED GLOBAL ACCESSES



Because of the nature of the algorithm, we can't ensure coalesced memory accesses to the array to sort!

KERNEL FUSION

Iterate inside the `bitonic_sort_kernel` to execute a single kernel: power of GPU not exploited, because the execution time increases a lot!

WARP STATE

Warp Cycles Per Issued Instruction [cycle]	23,02	Avg. Active Threads Per Warp	17,74
Warp Cycles Per Executed Instruction [cycle]	23,03	Avg. Not Predicated Off Threads Per Warp	16,67

Thread Divergence
Est. Speedup: 17.15%

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.

Only about half of the threads are active in a warp due to thread divergence when comparing the values to swap.

↓

Possible solution: replace the conditional blocks inside the kernel with already calculated expressions to understand which elements to exchange.

GOALS ACHIEVED

EXECUTION TIME

✓ Fully reached: sorting 4GB of `int` in ~15 s using 128 threads per block.

! *Side note:* sorting on GPU is much faster than on CPU already starting from arrays with 2^{16} elements!

04

CONCLUSIONS

Final considerations on the results obtained.

CPU

Despite having a smaller speedup, the best setup is Parallel Merge `-o2`. For this reason, this would be the final implementation of the algorithm.

64MB of `int` sorted in 0,44 s

GPU

Goal has been reached, so no other improvements are needed: modifying memory accesses and thread divergence leads to worse execution times.

4GB of `int` sorted in 15,24 s

Thanks for your attention!