UNIVERSITÀ DI PISA

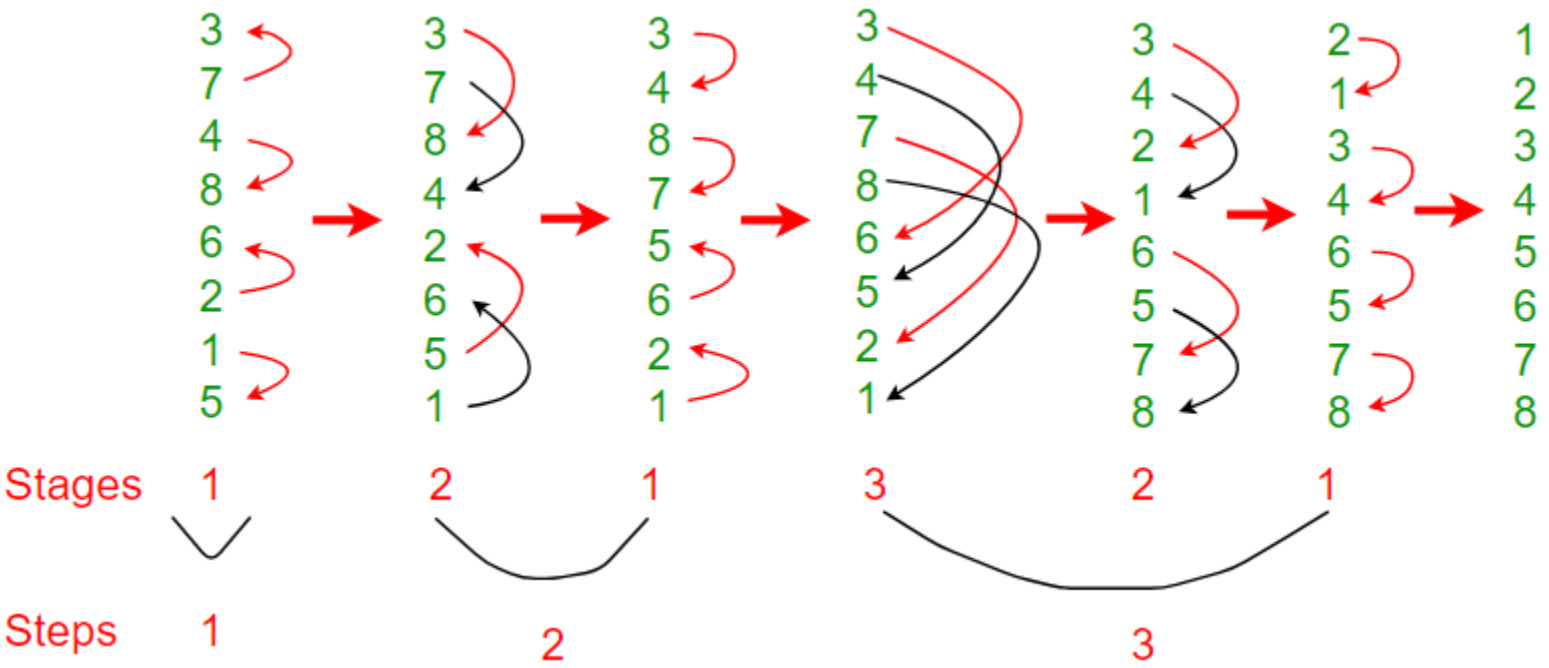
Bitonic sort

Computer architecture project

Fabio Piras
Giacomo Volpi

The algorithm and the goals

The bitonic sort is a merge sort like algorithm that exploit a bitonic sequence to sort an array

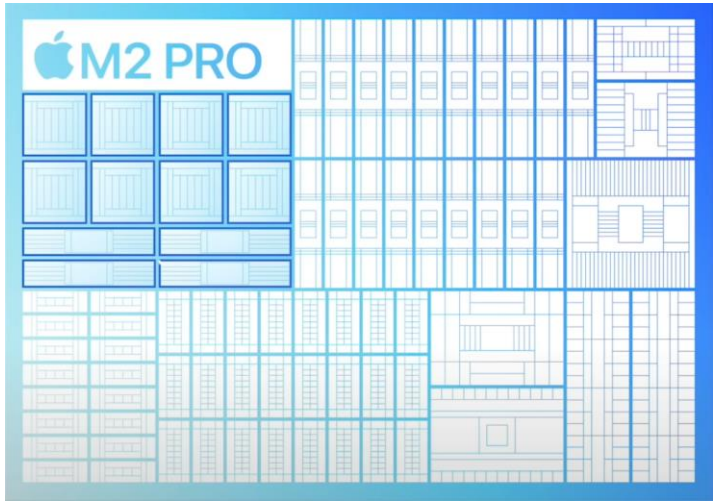


Goals - CPU:

- For the user: having an execution time less then 2 seconds up until an array with 2^24 entries
- For the developer: having a speedup higher than 50% of the number of cores with the use of multithreading

Both these goals were met

Machines and hardware



Apple M2 Pro 10-Core (ARM-based)

- 6 power-efficiency cores (up to 3.4 GHz)
- 4 high-performance cores (up to 3.7GHz)

- Cache:
  - L1i -> 192/128 Kb
  - Lid -> 128/64 Kb
  - L2 –> 32/4 Mb
  - System Level cache -> 24 Mb

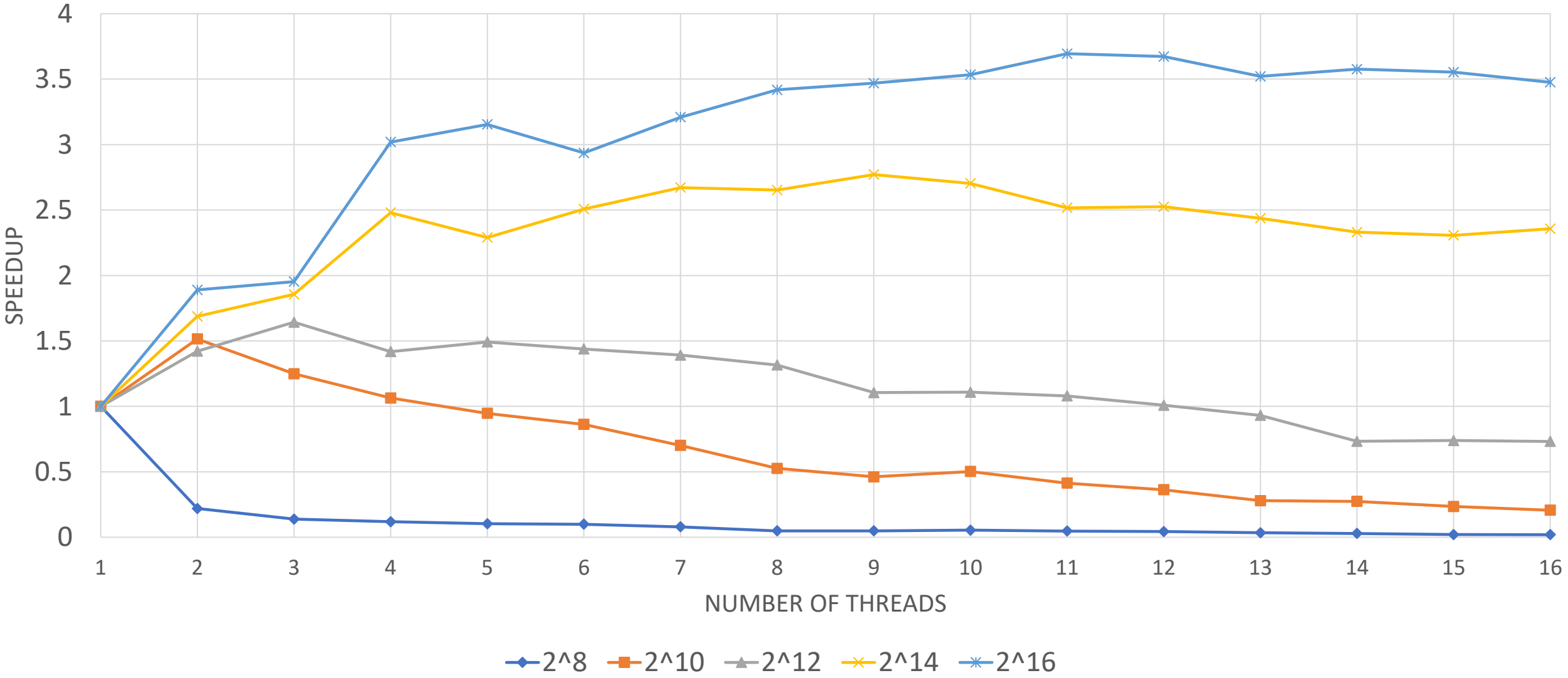
  Size of the cacheline: 128Byte

AMD Ryzen 5625U

- Speed: 2.30 GHz (base), up to 4.3 GHz
- Logical cores: 12
- Cache:

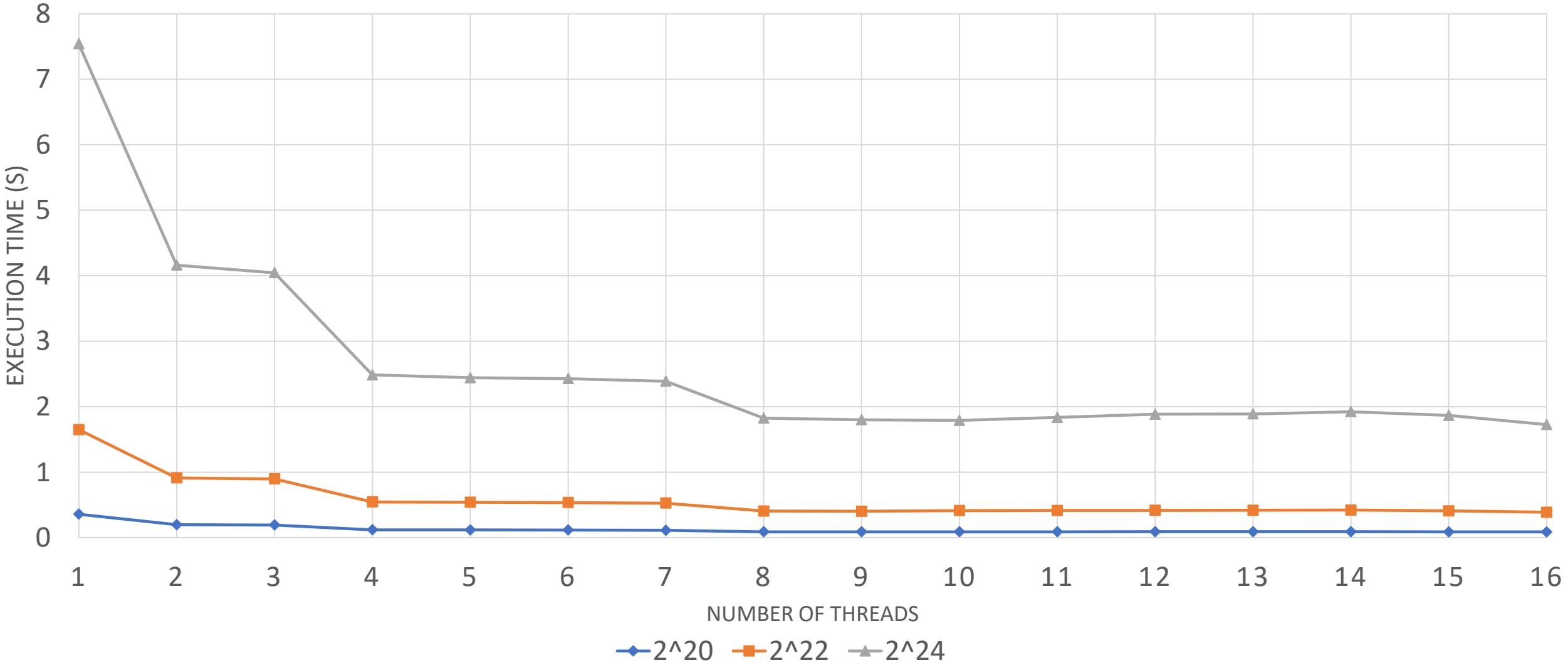
  - L1: 192 Kb IC – 192 Kb DC (8 ways)
  - L2: 3Mb (8 ways)
  - L3: 16 Mb (16 ways)

Speedup on small array – AMD case



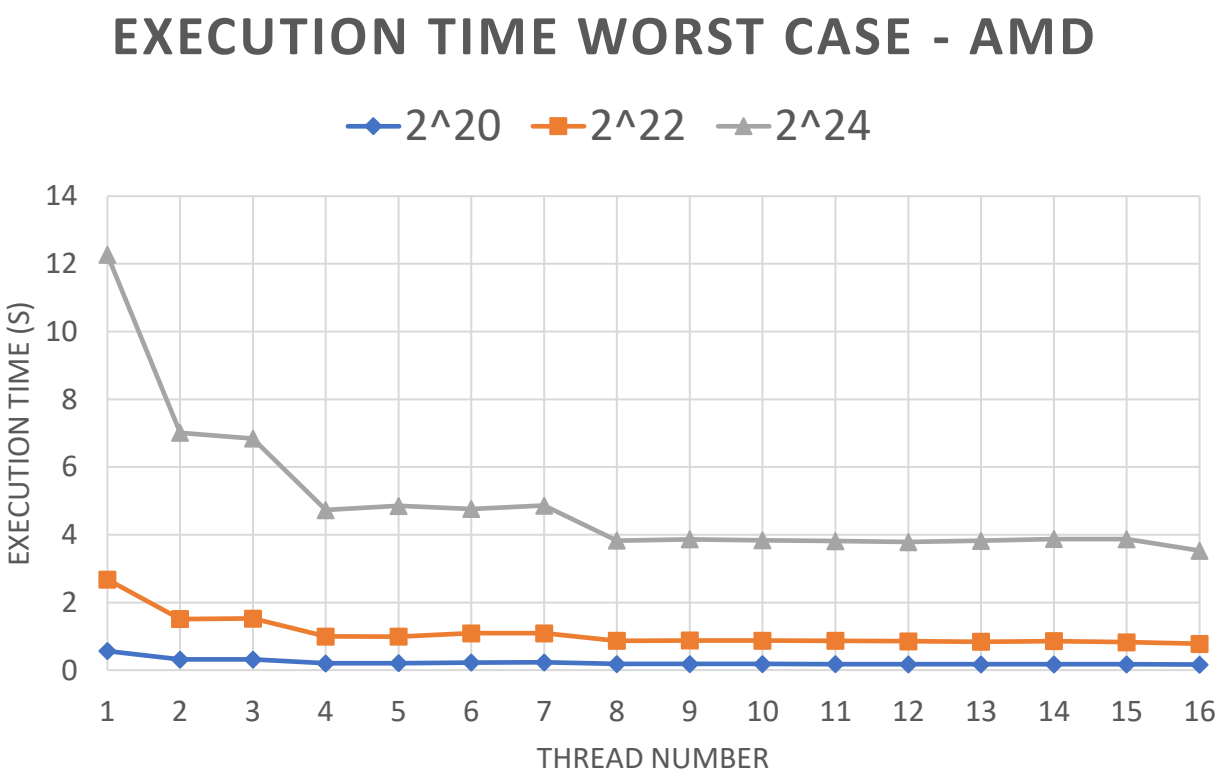
On small arrays [2^8: 2^14] the multithreading doesn't provide good enough results. Similar results were achieved with ARM architecture

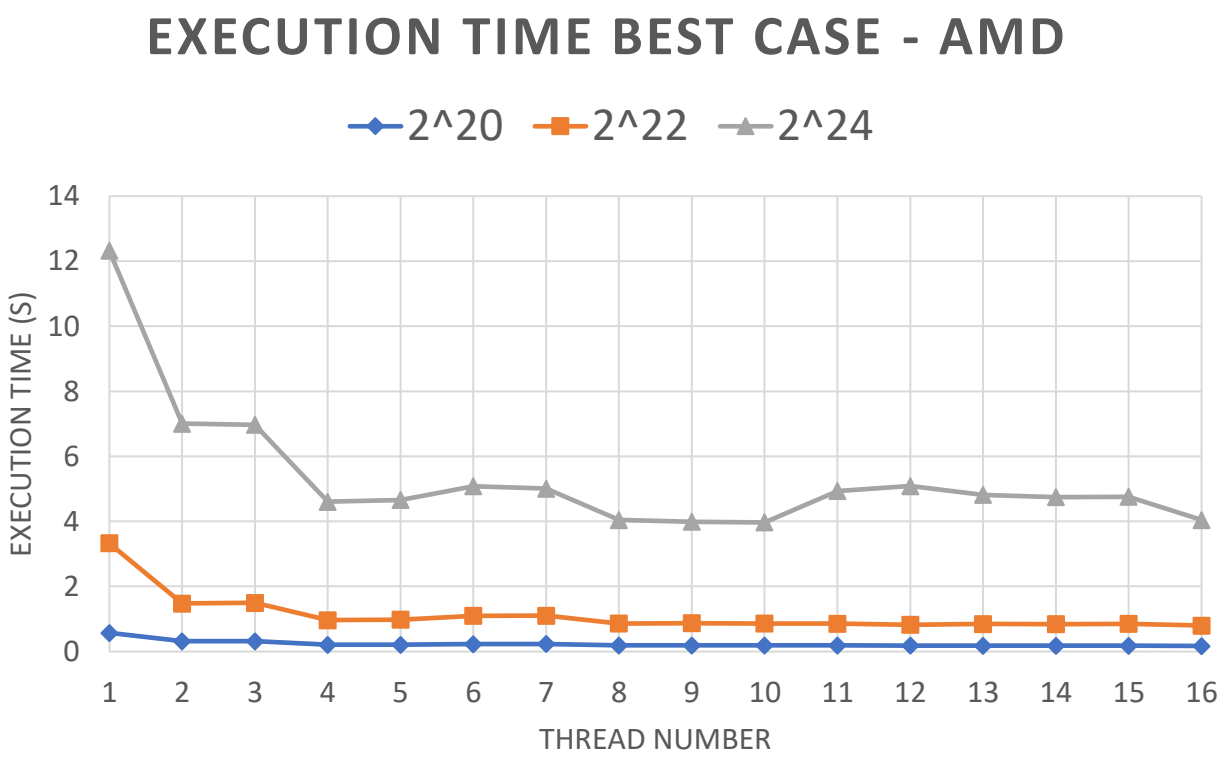
Execution time on large array – ARM case



On AMD we have similar behaviour but higher times overall

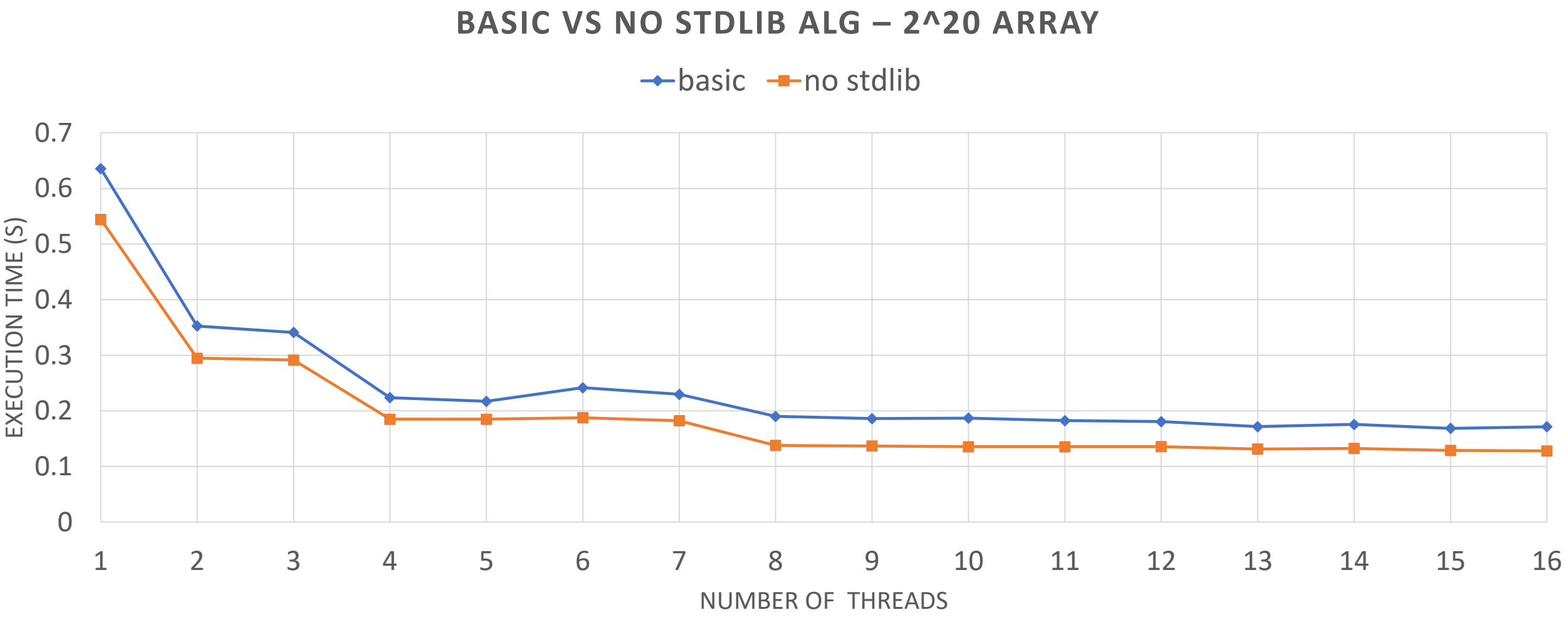
Execution time best and worst case – AMD





There are no noteworthy differeces, this is also aligns with the theory that says that bitonic sort is $O((log\ n)^2)$ in any case

Improving performace AMD – execution time

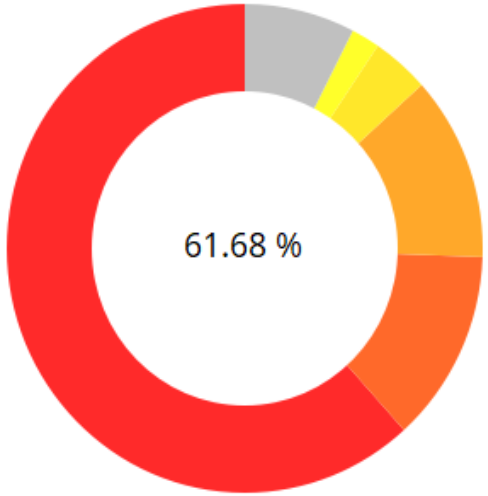


Some call to standard library function were removed in favor of using a temp variable, reducing the overall time on AMD architecture.

Improving performace AMD – cycles not in halt

Base version

Hot Functions

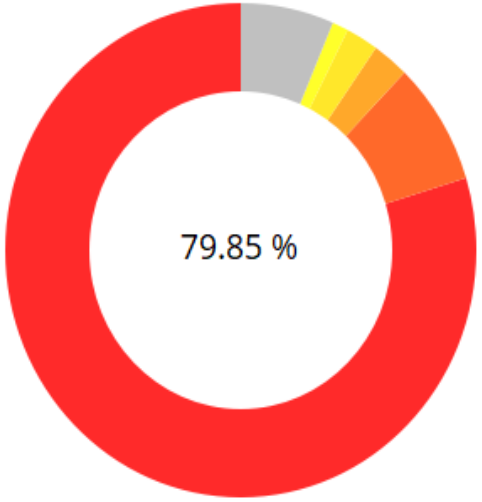


Less time spent in bitonic merge, but a higher percentage of weight in respect to other functions

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

«No stdlib» version

Hot Functions



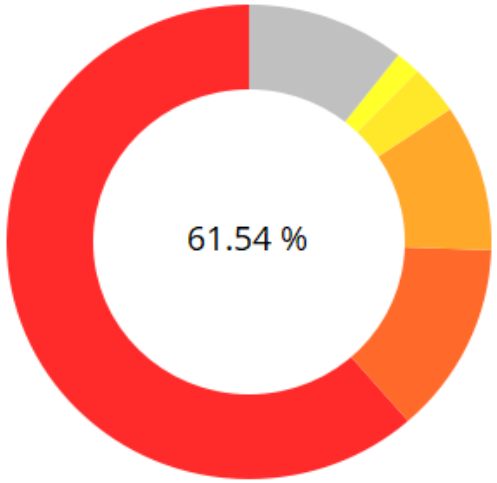
Function	CYCLES_NOT_IN_HALT [sample count]	CYCLES_NOT_IN_HALT [event count]
bitonic_merge(int * const,int,int,bool)	105354	26338500000
_CheckForDebuggerJustMyCode	10892	2723000000
ILT+1485(__CheckForDebuggerJustMyCode)	3536	884000000
ILT+415(?bitonic_merge@@YAXQEAHHH_N@Z)	3276	819000000
bitonic_sort(int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

Statics taken with uProf AMD profiler

Improving performaces AMD – L1 misses

Base version

Hot Functions

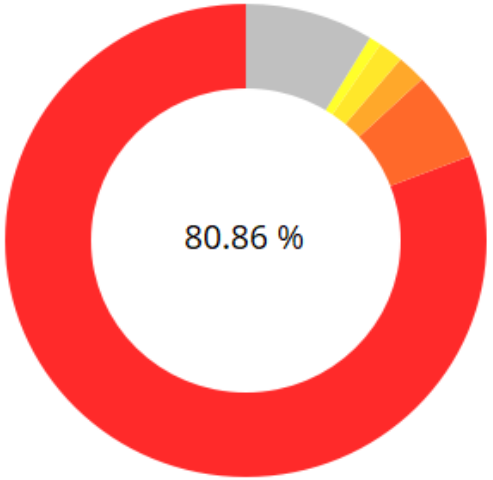


Small decrease in L2 access from L1 misses, but big increase in weight percentage

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

«No stdlib» version

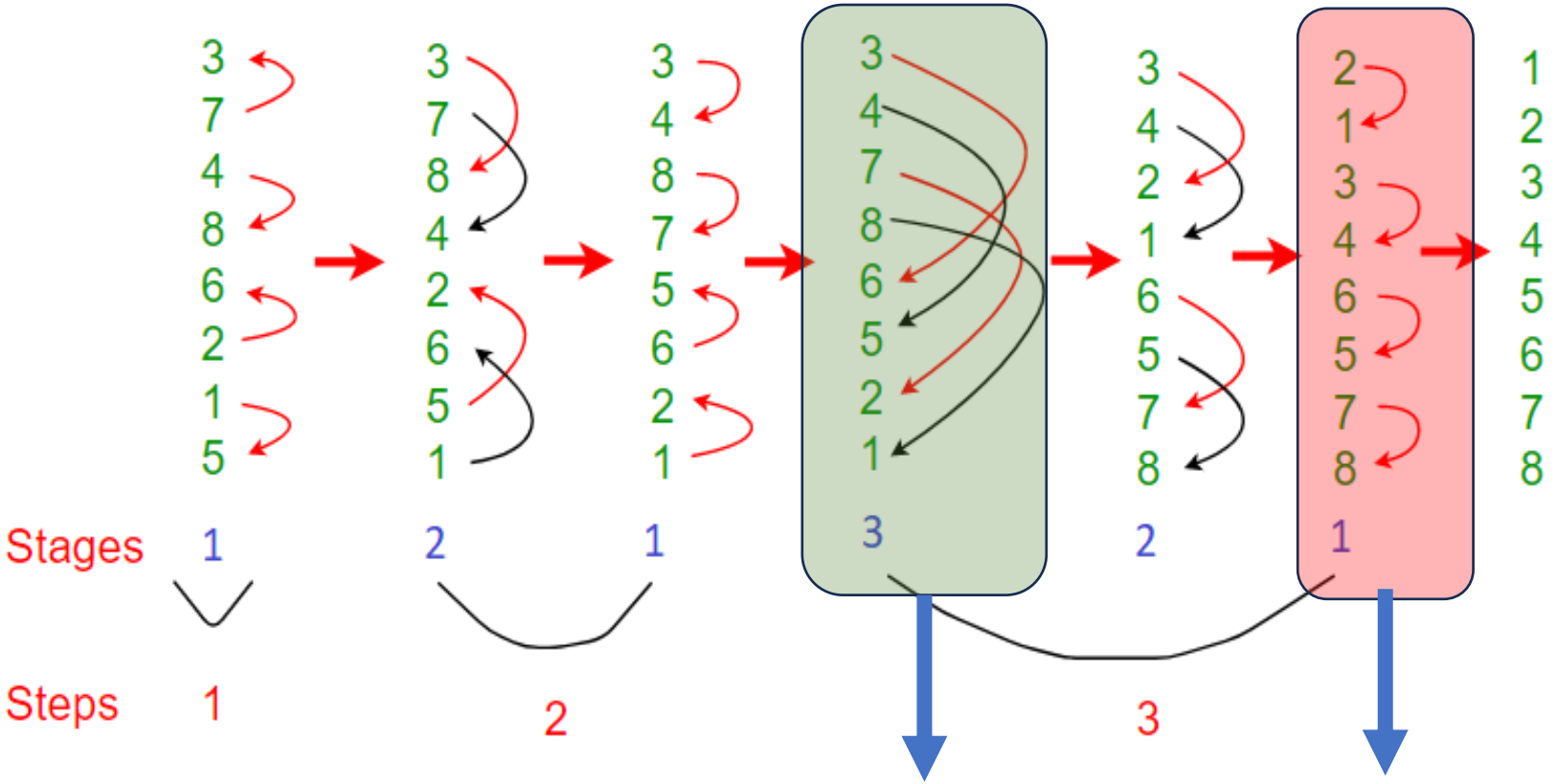
Hot Functions



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

Statics taken with uProf AMD profiler

Improving performance – parallelization of bitonic merge



Bitonic merge perform the basic swap operation and in the base version of the algorithm is executed sequentially

Improving performance ARM – L1 cache misses

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

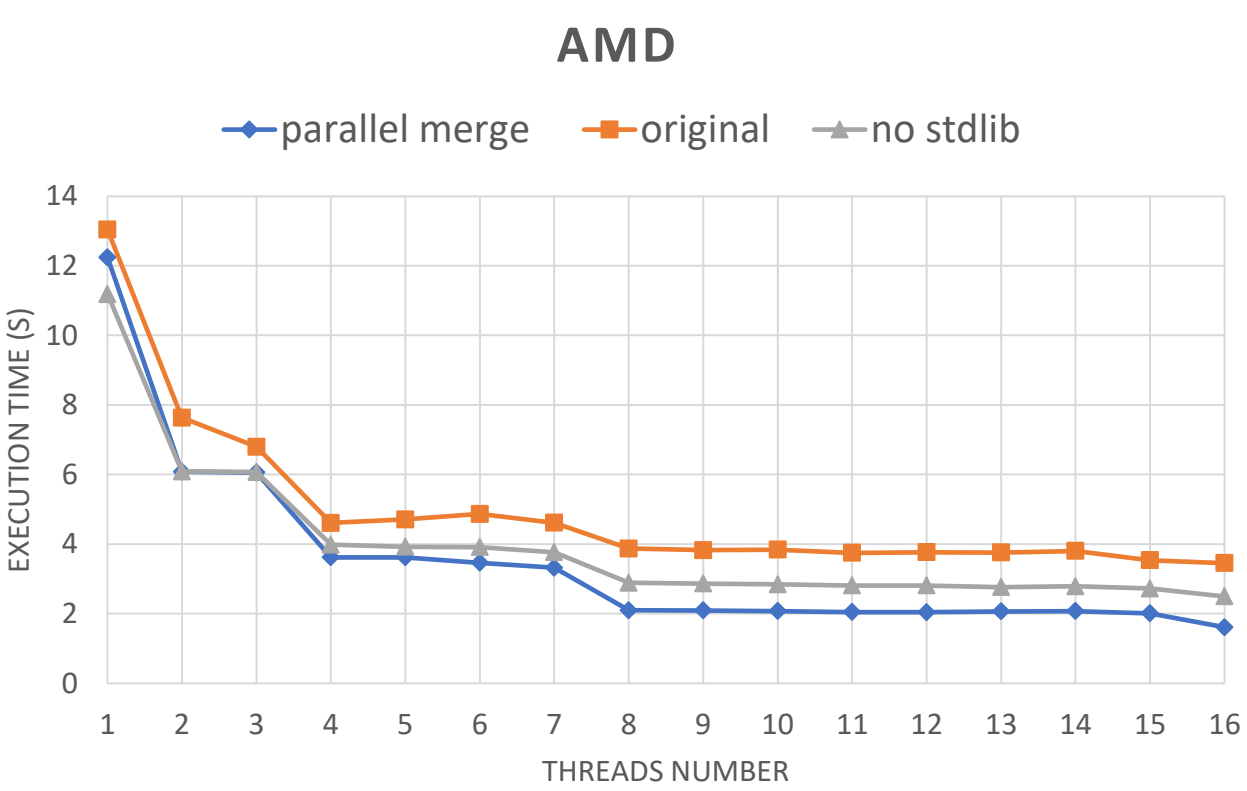
Sequential merge

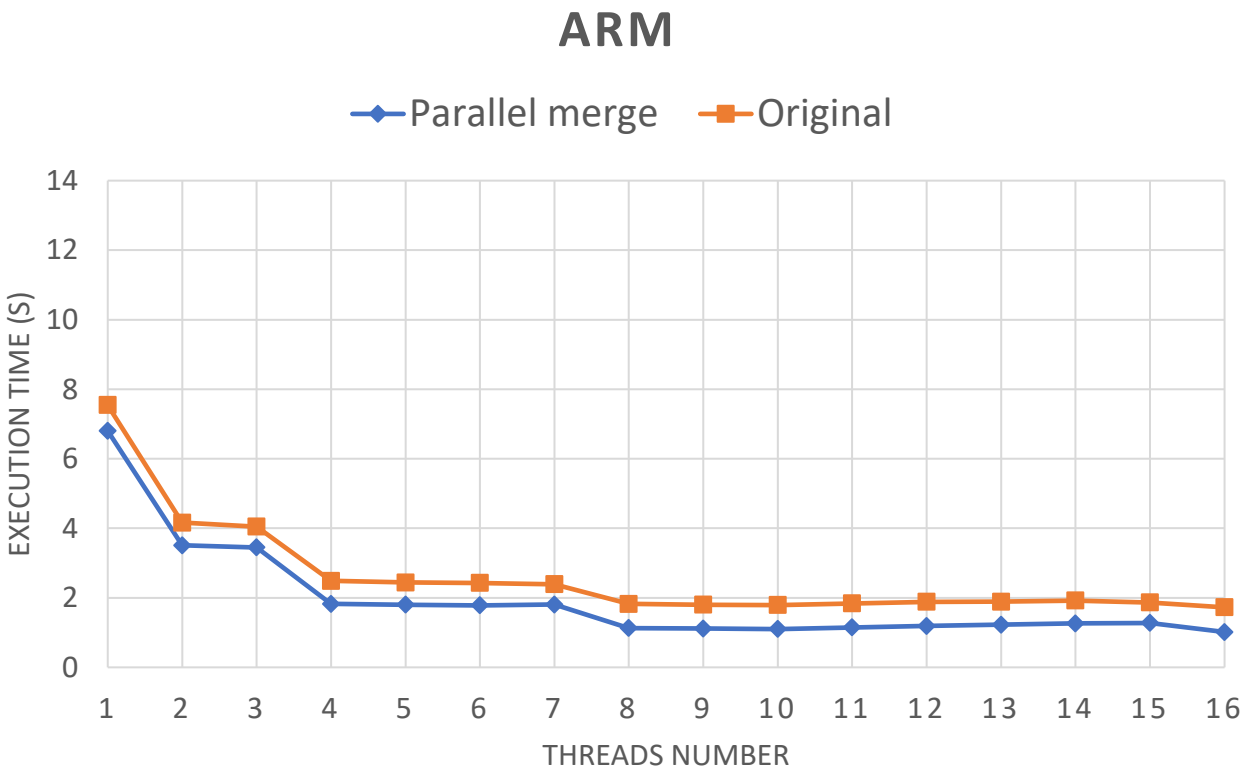
Process	# Samples	Total CPI	Total Branch miss	Total % branch mi...	Total Miss L1
* All *	855	4,5154	1.138.048	0,0006	1.473.122
BitonicOpt (1513)	855	4,5154	1.138.048	0,0006	1.473.122

Parallel merge case

Statics taken with Instruments (part of Xcode)

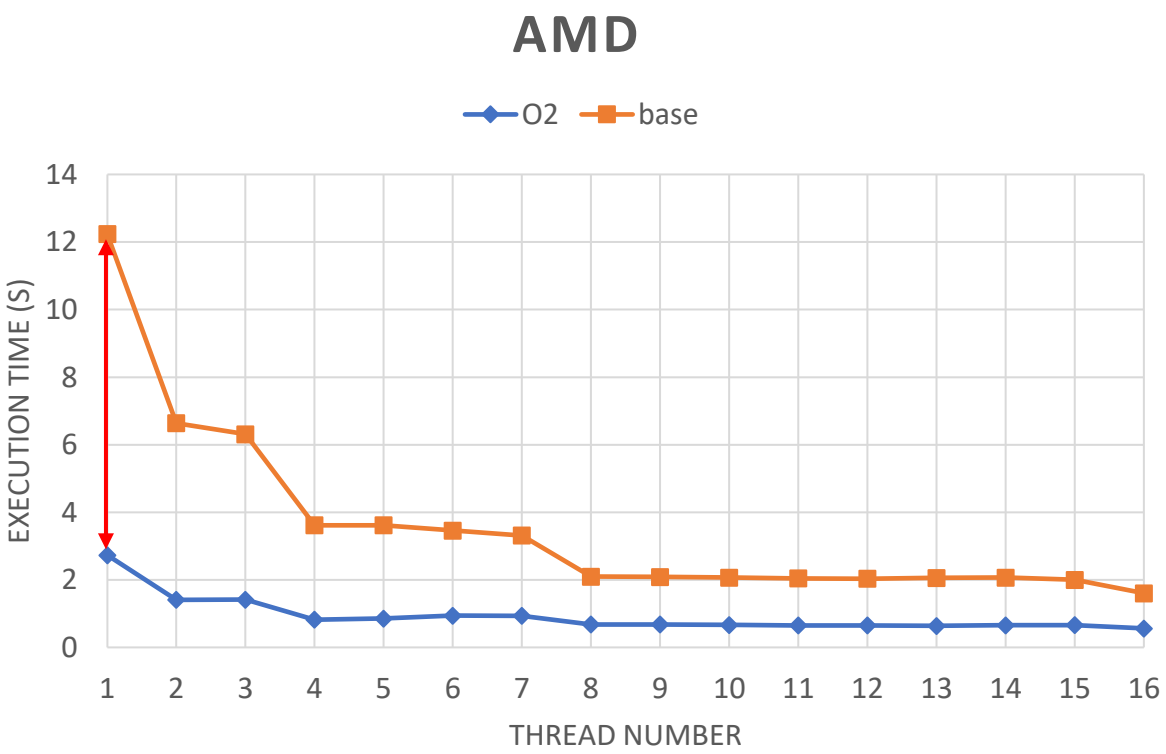
Execution time final comparison – 2^24 array

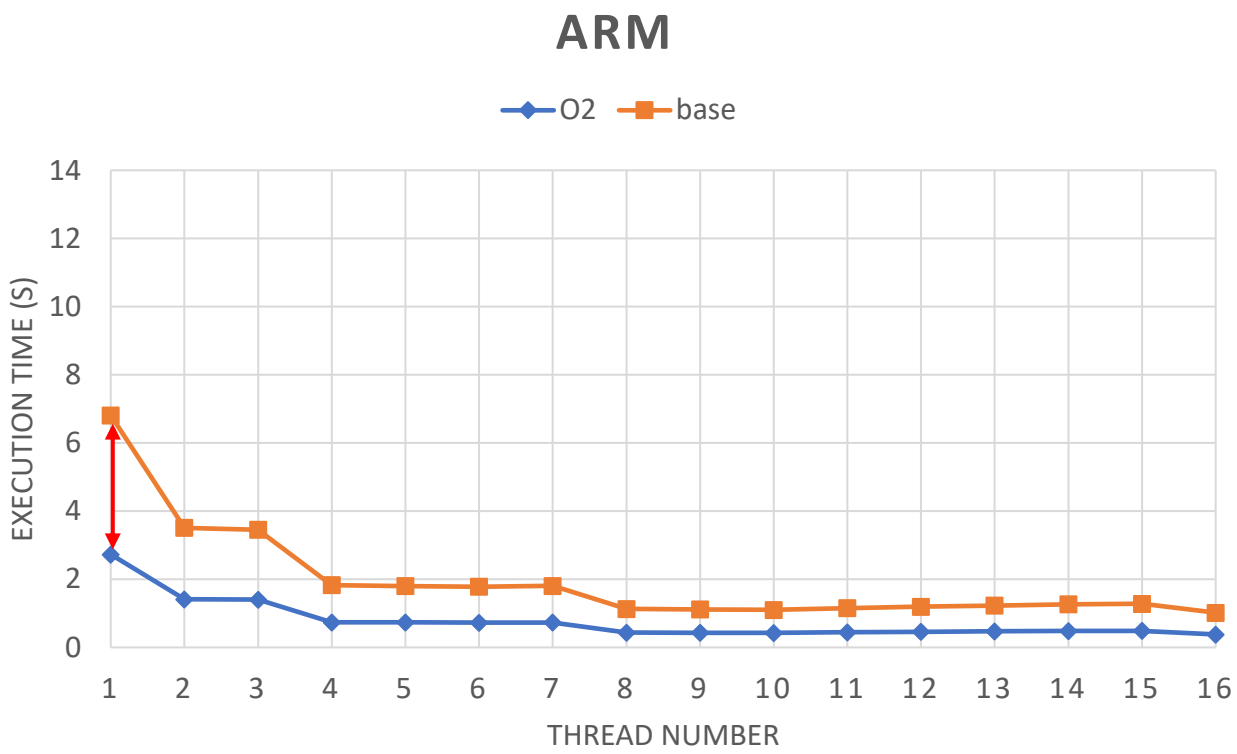




ARM chart has no «no stdlib» line since it would collapse on the «original» one

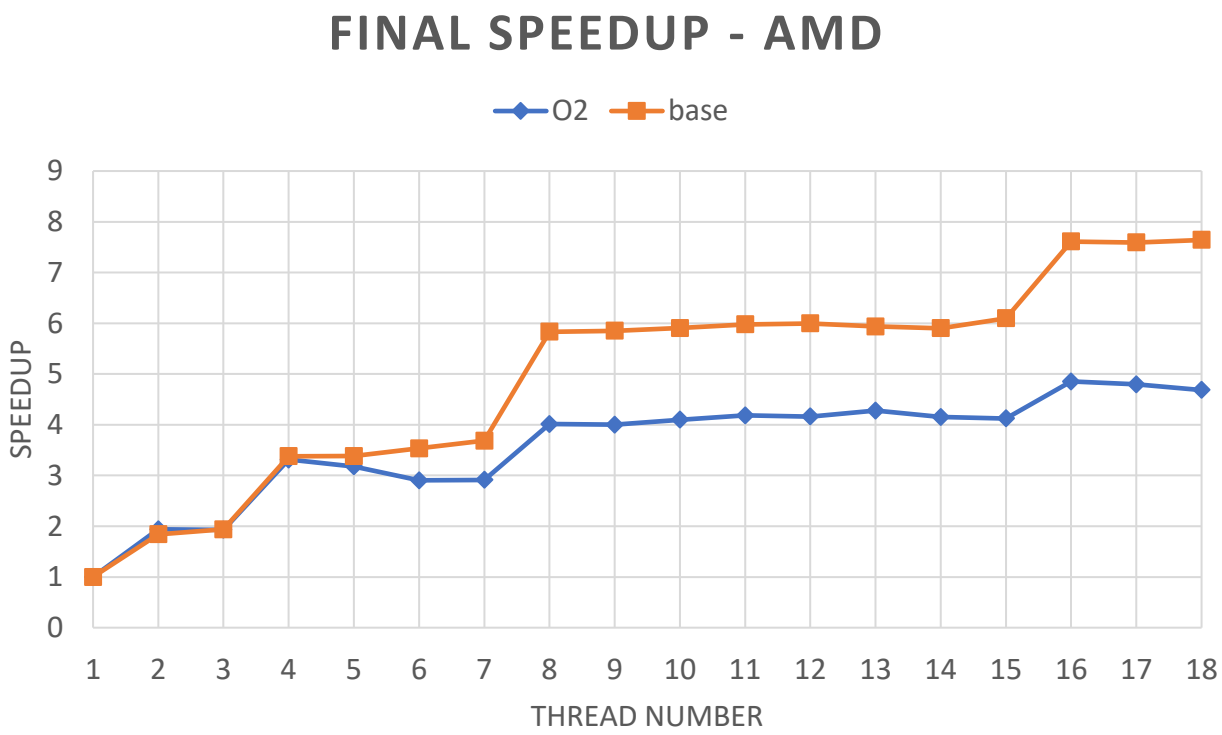
User requirements: timing - 2^24 array



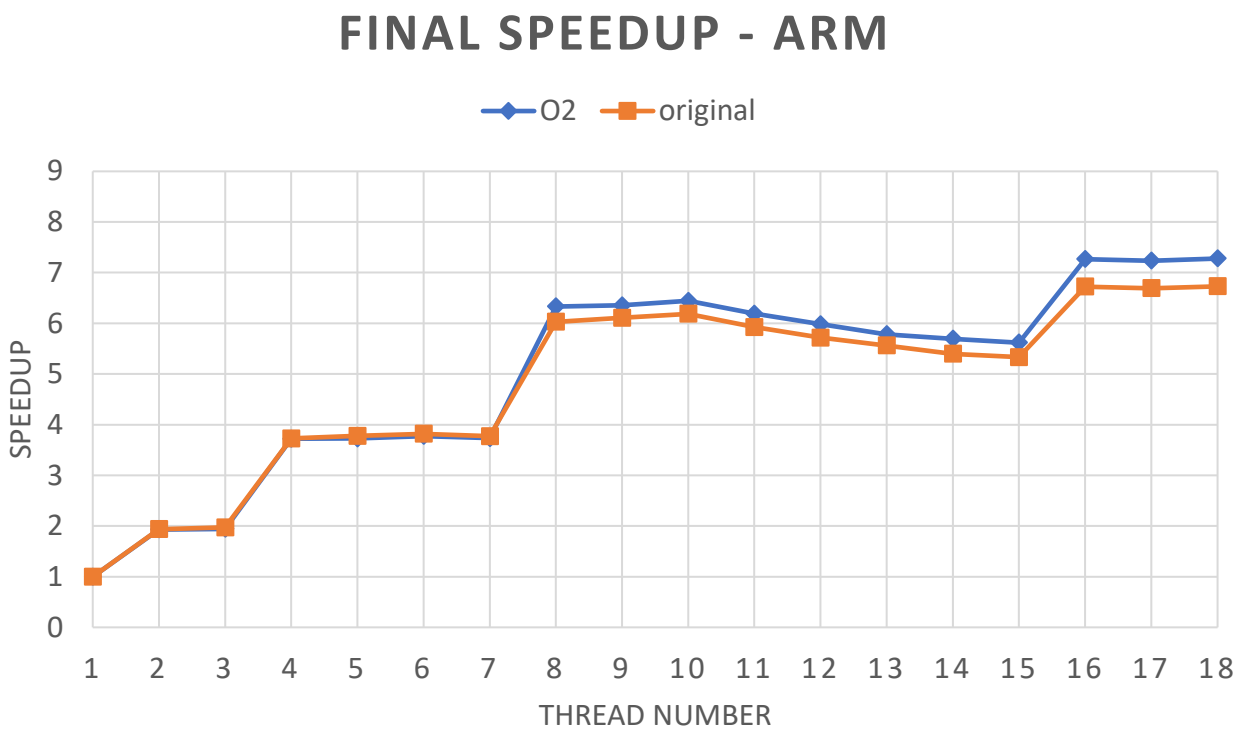


We meet our user requirements by keeping the excution time of an array of 2^24 below 2 seconds with the use of O2 compiler flag.

Developer requirements: speedup - 2^24 array



On AMD the O2 compiler flag reduces speedup, because of the monothread case being of lower value.



On ARM the O2 compiler flag brings a slight improvement

GPU hardware

NVIDIA® Tesla T4

Statistics:

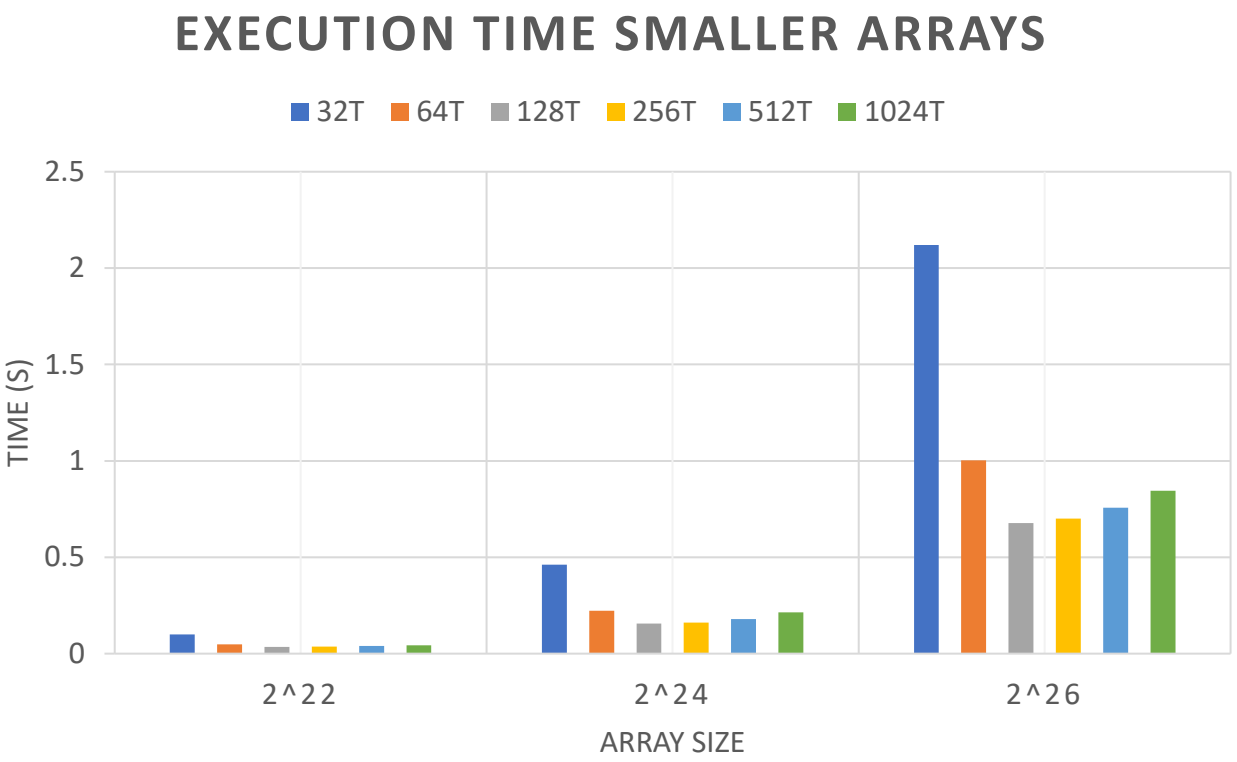
- Compute power: 7.5
- Cuda cores: 2560
- Streaming multiprocessors: 40
- Warp size: 32
- Max threads per block: 1024
- Memory: 16 GB GDDR6 bandwidth: 300 GB/s
- Interconnection bandwidth: 32 GB/s

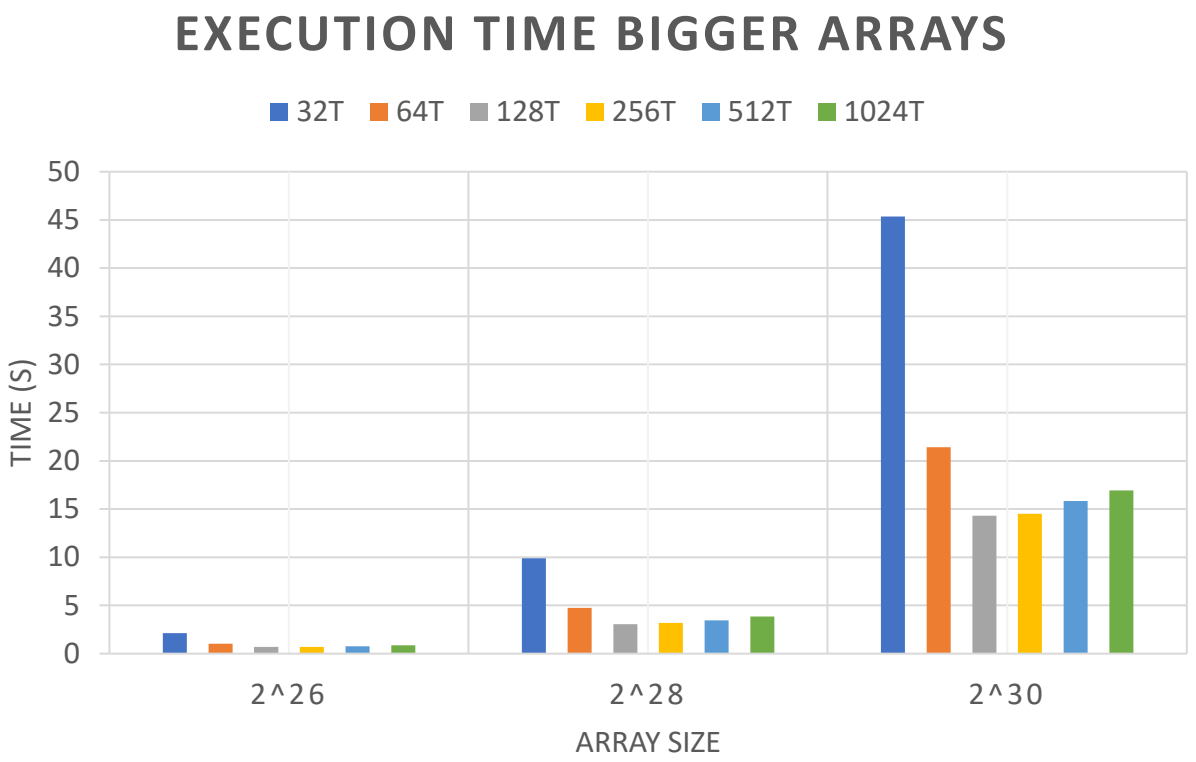
Objective:

- Minimize execution time keep arrays in the order of Mb below 2s and arrays in the order of Gb below 20s

Both were reached in our case study ✓

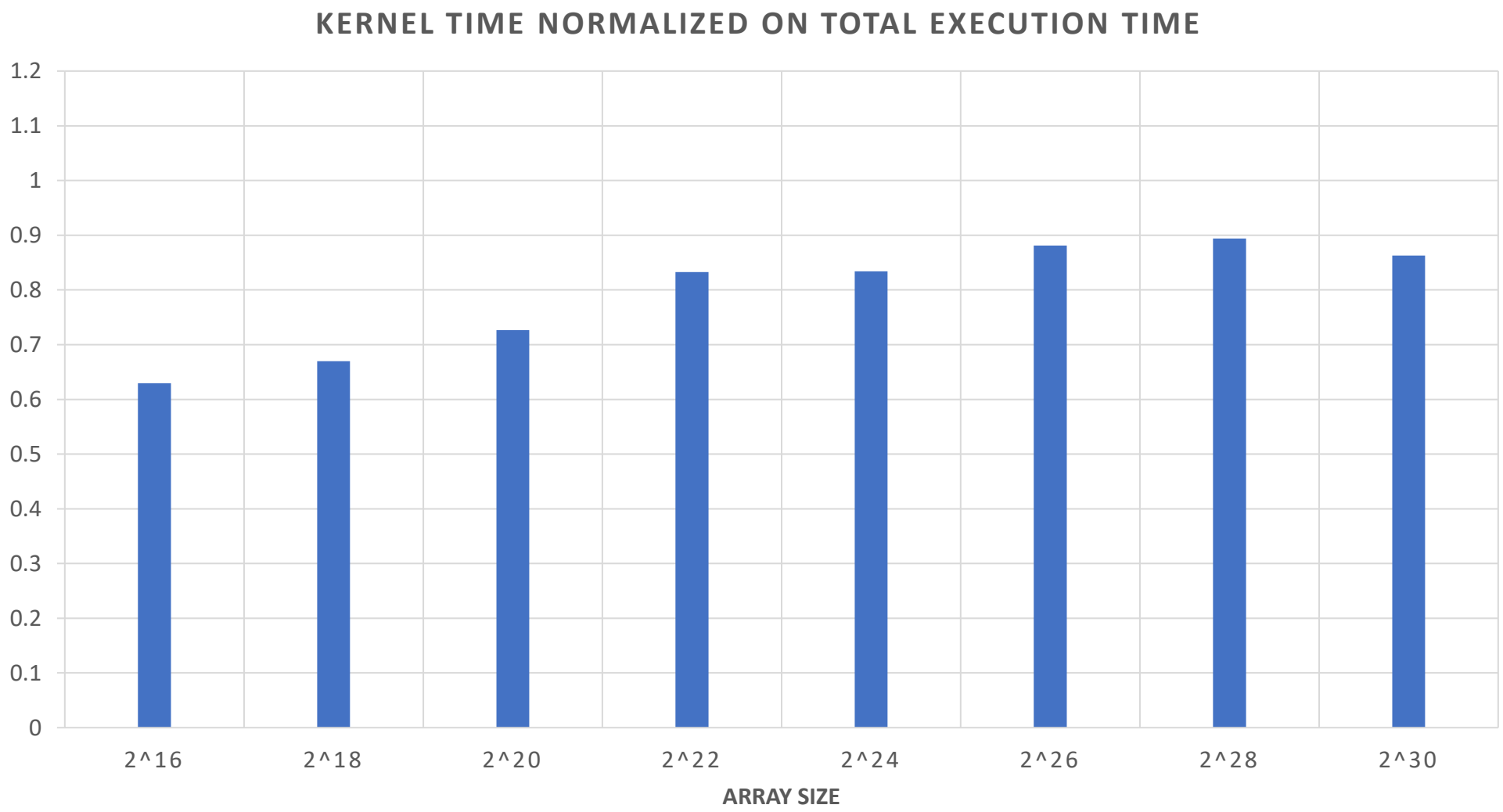
Execution time with cudaMemCpy





On smaller arrays the GPU implementation almost nullifies the sorting time, in bigger ones it is still perceptible but in both case the solution with 128 threads brings the better results

Execution time – weight of cudaMemcpy



For bigger arrays the time occupied by the CudaMemcopy (both) is not very influential

Execution time – Possible improvements

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

Memory is more heavily utilized than compute power, this can indicate memory bottleneck.
Possible solution: kernel fusion

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

Only half of the threads are active in a warp on average due to branch miss-prediction.
Possible solution: remove if blocks in kernel function

Conclusions

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

The GPU brings a lot of improvements allowing us to sort arrays up to 256 Mb in less than 1 second and one of 2 Gb in 14 seconds