UNIVERSITÀ DI PISA

MSc in Computer Engineering

Computer Architecture Project

PARALLELIZED MERGE-SORT ALGORITHM

TEAM MEMBERS:

GIOVANNI BARBIERI

FEDERICO CAVEDONI

ALESSIO DI RICCO

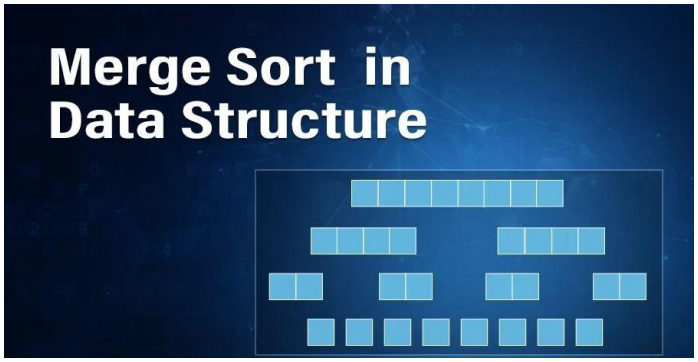
Anno Accademico 2022-2023

Summary

INTRODUCTION

PROBLEM DESCRIPTION

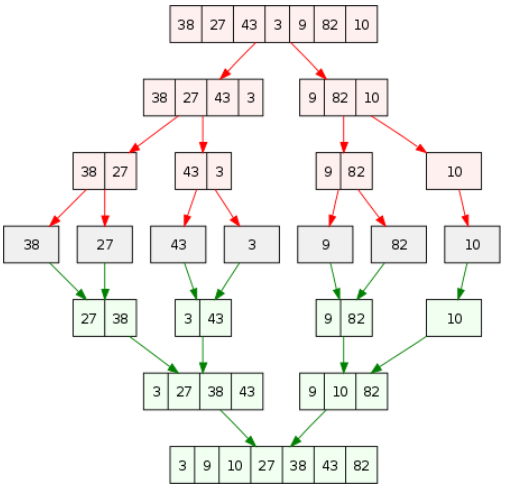
Merge Sort is a popular sorting algorithm that aims to efficiently sort an array or list of elements in ascending or descending order. The algorithm follows a divide-and-conquer approach, recursively dividing the input into smaller subproblems until they can be easily solved, and then merging the results to obtain the final sorted output.



ALGORITHM

The Merge Sort algorithm can be described in the following steps:

1. **Divide**: The input array is recursively divided into two halves until each subarray contains only one element or is empty. This process continues until we reach the base case.
2. **Conquer**: Once the subarrays reach the base case, they are considered sorted by definition. If a subarray contains more than one element, the algorithm recursively sorts each half.
3. **Merge**: After all subarrays are sorted, the merge operation combines them to produce a single sorted output array. It compares the elements from each subarray and selects the smaller (or larger) element to place in the output array. The process continues until all elements are merged.



The merge operation is performed by maintaining two pointers that traverse the subarrays. These pointers compare the elements and add them to the output array in the appropriate order. After one subarray is exhausted, the remaining elements from the other subarray are appended to the output array.

The Merge Sort algorithm has a time complexity of O(n log n) in all cases, where "n" represents the number of elements in the input array. This makes it one of the most efficient comparison-based sorting algorithms, especially for large data sets.

Overall, Merge Sort is a reliable and efficient algorithm for sorting arrays, providing a stable and consistent sorting solution.

The goal of our project is therefore to write parallel software for both GPU and CPU that is efficient.

As the main evaluation parameter for performance we have chosen the execution time.

CPU IMPLEMENTATION

HARDWARE SPECIFICS

For this project, we utilized the Ryzen 7 5700U processor. Its specifications are shown in the image below.

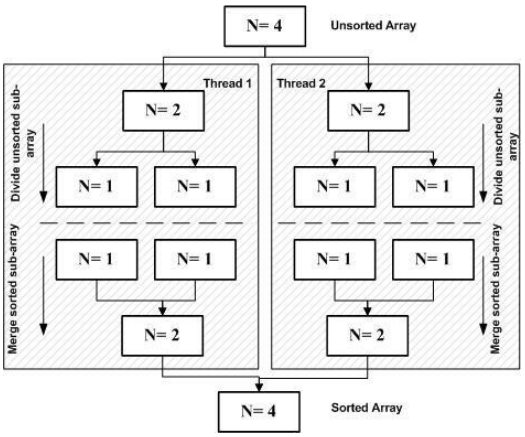
AMD Ryzen™ 7 5700U

General Specifications			
	Platform: Laptop	**Product Family:** AMD Ryzen™ Processors	**Product Line:** AMD Ryzen™ 7 Mobile Processors with Radeon™ Graphics
	# of CPU Cores: 8	**# of Threads:** 16	**Max. Boost Clock:** Up to 4.3GHz
	Base Clock: 1.8GHz	**L2 Cache:** 4MB	**L3 Cache:** 8MB
	Default TDP: 15W	**AMD Configurable TDP (cTDP):** 10-25W	**Processor Technology for CPU Cores:** TSMC 7nm FinFET
	CPU Socket: FP6	**Max. Operating Temperature (Tjmax):** 105°C	**Launch Date:** 1/12/2021

CODE IMPLEMENTATION

We used the C++ language to develop the code.

We split the initial array into several sub-arrays and assigned each one to a thread. Each thread did the merge-sort on the sub-array assigned to it and finally we did the final merges on the sorted sub-arrays.



We have paid particular attention to load-distribution within threads. In case of arrays that are not multiples of the number of threads we made sure that the sub-arrays vary no more than one element between them.

```
void create_division(int& low, int& high, int thread_part){
    int resto = MAX % num_threads;
    low = thread_part * (MAX / num_threads);
    if (thread_part >= 1 && thread_part <= resto)
        low += thread_part;
    if (thread_part > resto)
        low += resto;

    int c = (thread_part + 1) * (MAX / num_threads);
    if (thread_part < resto)
        c += thread_part;
    else
        c += resto-1;

    high = c < MAX? c : MAX;
}
```

The execution times used for our analyzes are taken as in the figure. The reading of the input data has been omitted from the total.

START →

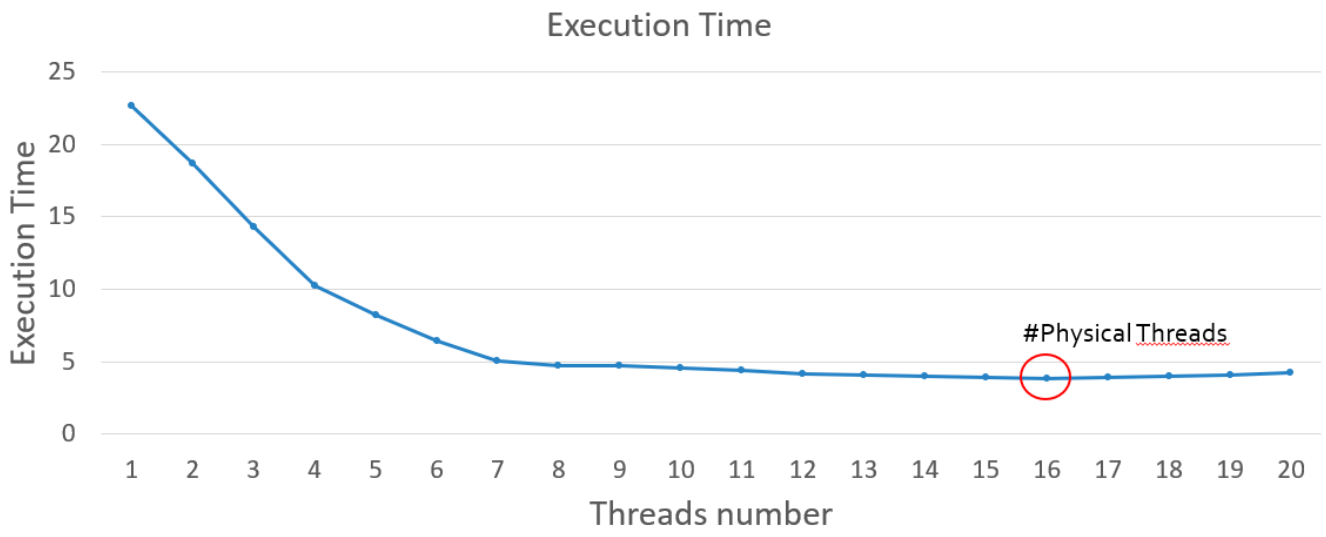
END →

```
clock_t t1, t2, end1;

t1 = clock();

pthread_t threads[num_threads];

for (int i = 0; i < num_threads; i++)
    pthread_create(&threads[i], NULL, merge_sort_thread, NULL);

for (int i = 0; i < num_threads; i++)
    pthread_join(threads[i], NULL);

for(int j = 2; j < num_threads; j = j*2){
    for(int i = 0; i < num_threads/j; i++){
        int c = num_threads/j;
        int start = (MAX/c)*i;
        int e = ((MAX)/c)*(i+1)-1;
        int mid = (start+e)/2;
        merge(start, mid, e);
        //printf("%d %d %d\n", start, mid, e);
    }
}

merge(0, (MAX - 1)/2, MAX - 1);

t2 = clock();
```

ANALYSIS

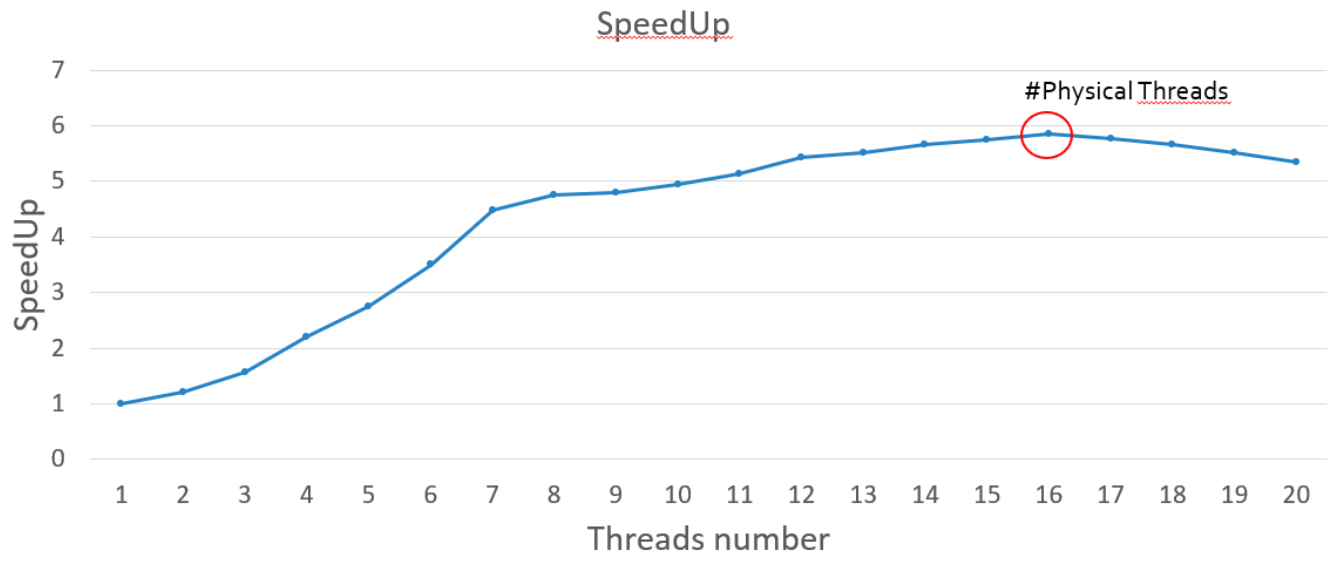
Each experiment was performed in 36 repetitions, in order to exploit the central limit theorem. Therefore, by calculating the **'confidence intervals'** using the specific formulas for the normal distribution, we have calculated the 95% confidence intervals.
The results are not shown in the graphs because the interval was too small, less than 0.05 amplitude.

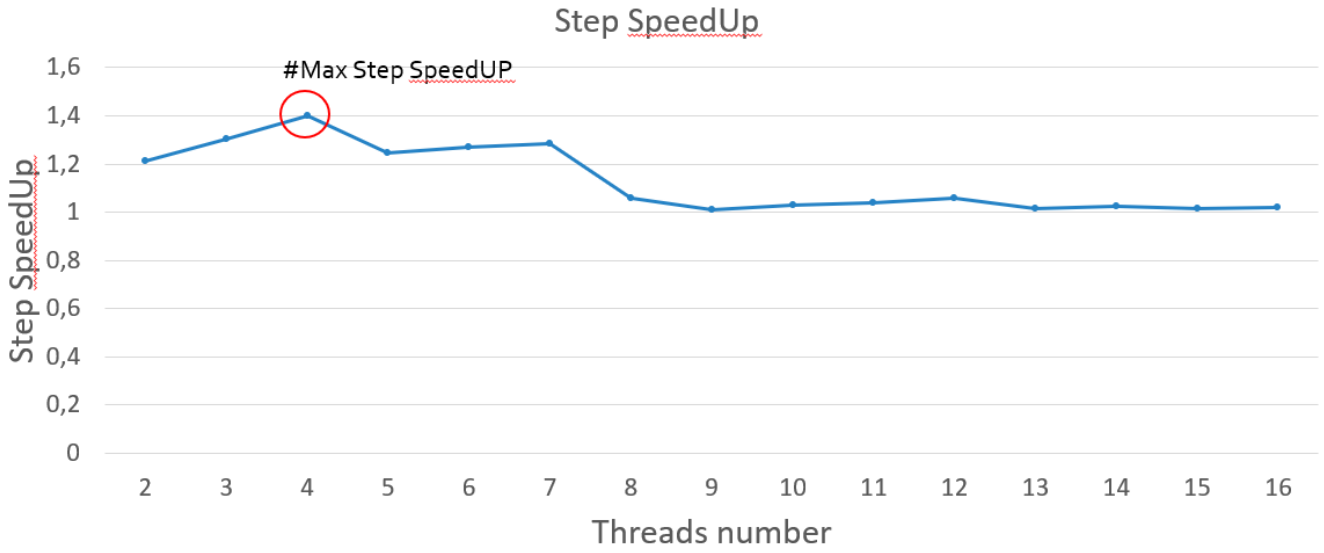
Initially we chose an array of fixed size (10 million) and we studied the variation of the execution time as the number of threads varied. We chose 10 million as the size as we felt it was large enough so that the overhead of multithreading wasn't the predominant part. We got the result shown in the figure below.



The figure clearly shows that the best execution time is obtained with 16 threads, which is effectively the number of hardware threads the processor has. The result is therefore consistent with the architecture used.

Then we performed the study of the speedup and step-speedup as the threads vary, with the array of fixed length (10 million). The speedup is calculated between parallel and sequential execution.

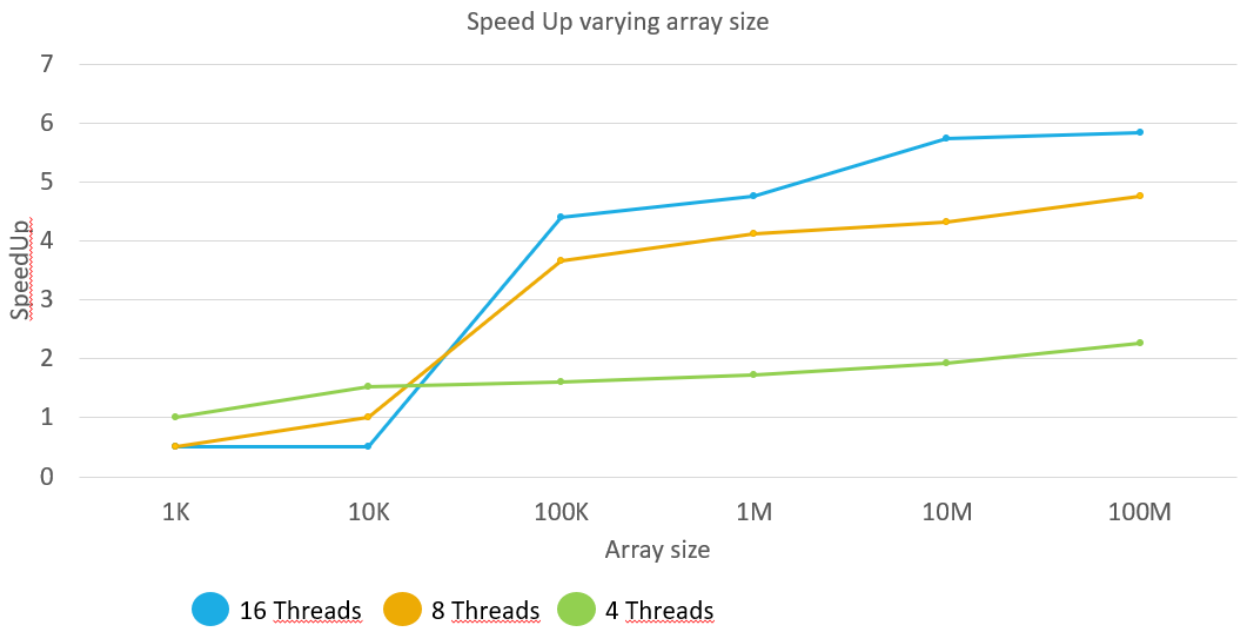




From the speedup analysis we obtained consistent results with the previous graph, as we obtain the maximum speedup at the value of 16 threads, which correspond to the hardware threaded ofthe used processor.

As for the step-speedup we get a higher gain with a lower number of threads. This is due to the fact that as more threads increase there will be more sub-arrays and therefore the overall array will be messier due to the greater number of segments. Therefore the final merge is overall more expensive as the threads increase and consequently we obtain a lower gain, even if overall it is still advantageous.

Finally, we performed an analysis of the speedup both as the threads and the size of the array varied and we obtained the graph below.



We have obtained that for small arrays parallelization is not convenient and in fact the speedup is less than one.
As the size of the array increases, parallelization becomes more and more convenient. In particular, it can be seen that initially a lower number of threads is convenient, to then have an optimal result with 16 threads over a certain threshold.
We therefore see that as the size of the array increases, parallelization is increasingly convenient, because the overhead due to multithreading affects less and less.

Through the profiler "Perf" we have analyzed the algorithm to find possible bottlenecks, and we have obtained the following results.

```
cupar   /proc/kcore                  0x7fffb72ca2f4   k [k] update_curr
cupar   /home/cavedoni/cupar         0xe7b            B [.] merge_sort(int, int)
cupar   /home/cavedoni/cupar         0xd94            B [.] merge(int, int, int)
```

```
 46.08 |      add    %rdx,%rax
  0.09 |      mov    %rax,somma_ricorsione
  0.14 |    → callq  clock@plt
       |      mov    %rax,-0x10(%rbp)
  1.14 |      mov    -0x38(%rbp),%edx
  0.09 |      mov    -0x24(%rbp),%ecx
       |      mov    -0x34(%rbp),%eax
       |      mov    %ecx,%esi
       |      mov    %eax,%edi
       |    → callq  merge(int, int, int)
  0.05 |    → callq  clock@plt
       |      mov    %rax,-0x8(%rbp)
  0.46 |      mov    -0x8(%rbp),%rax
       |      sub    -0x10(%rbp),%rax
       |      mov    %rax,%rdx
       |      mov    somma,%rax
 48.72 |      add    %rdx,%rax
  0.23 |      mov    %rax,somma
  0.36 | c2:  nop
```

From the following results we can see that the recursion represents a bottleneck for the analyzed algorithm.

The recursive call, having to save the memory necessary for the function at each call, causes a greater overhead and later, since the cache will fill up with memory for new calls, it will cause the increase in capacity misses.

RESULTS

From the analyzes carried out we concluded that the most optimal number of threads to use is 16, which correspond to the hardware threads present on the processor used.

However, parallelization is optimal only for arrays with a size of at least 10 thousand elements, otherwise the overhead due to parallelization is greater than the gain obtained with it and therefore it is advisable to use a sequential program.

Then analyzing the algorithm through the profiler 'Perf' we came to the conclusion that the recursion is a bottleneck of the algorithm, but the algorithm itself cannot ignore this as it is implemented. Recursion is a bottleneck because each recursive call allocates memory, and therefore the memory usage is large and thus increases the cache miss rate.

Furthermore, another considerable bottleneck is the final merge of the various subarrays, since it must necessarily be performed sequentially.

GPU IMPLEMENTATION

HARDWARE SPECIFICS

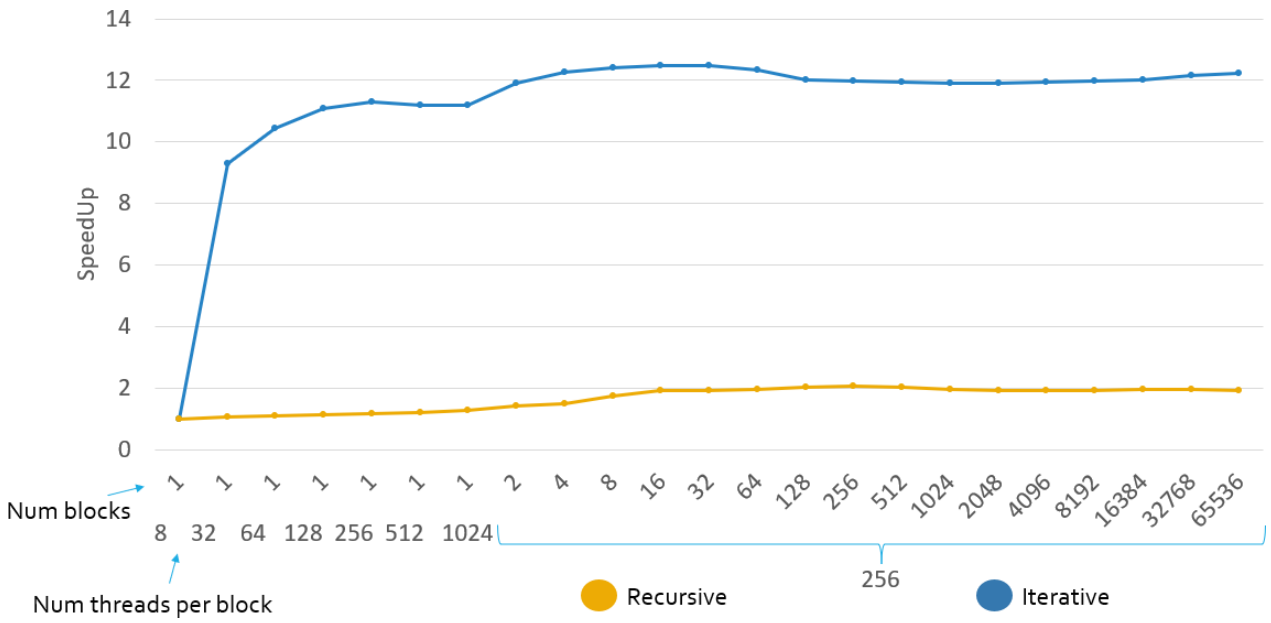
For this project, we utilized the Nvidia Tesla T4 GPU. Its specifications are shown in the image below.

SPECIFICATIONS	
GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/sec**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

CODE IMPLEMENTATION

We used the CUDA C++ language to develop the code.

We used an iterative implementation of the merge-sort algorithm on the GPU. This is because recursion on the GPU is poorly performing, as we can see from the graph in the figure where we compare a first recursive version of the algorithm with the iterative one actually used for subsequent analyses.



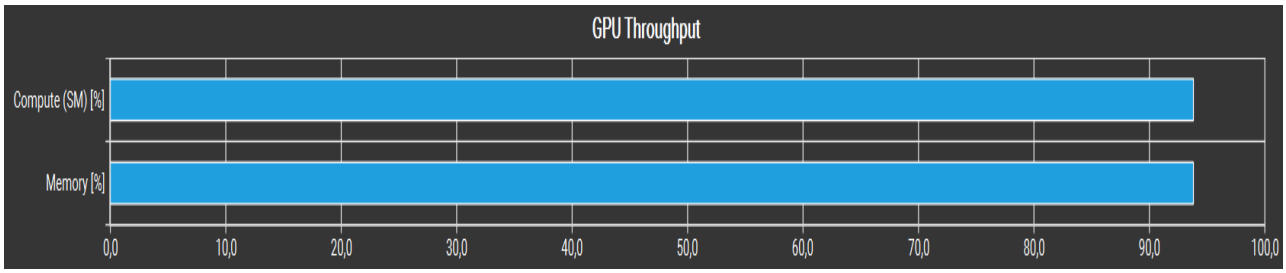
To calculate the execution times to carry out our analyses, we used CudaEvents, which allowed us to evaluate only the actual execution time of the GPU without counting the work on the CPU.

```
cudaEvent_t start, stop;
float elapsedTime;

cudaEventCreate(&start);


cudaEventCreate(&stop);



long *data;
readList(&data);


cudaEventRecord(start);


mergesort(data);

cudaEventRecord(stop);

cudaEventSynchronize(stop);


cudaEventElapsedTime(&elapsedTime, start, stop);
```

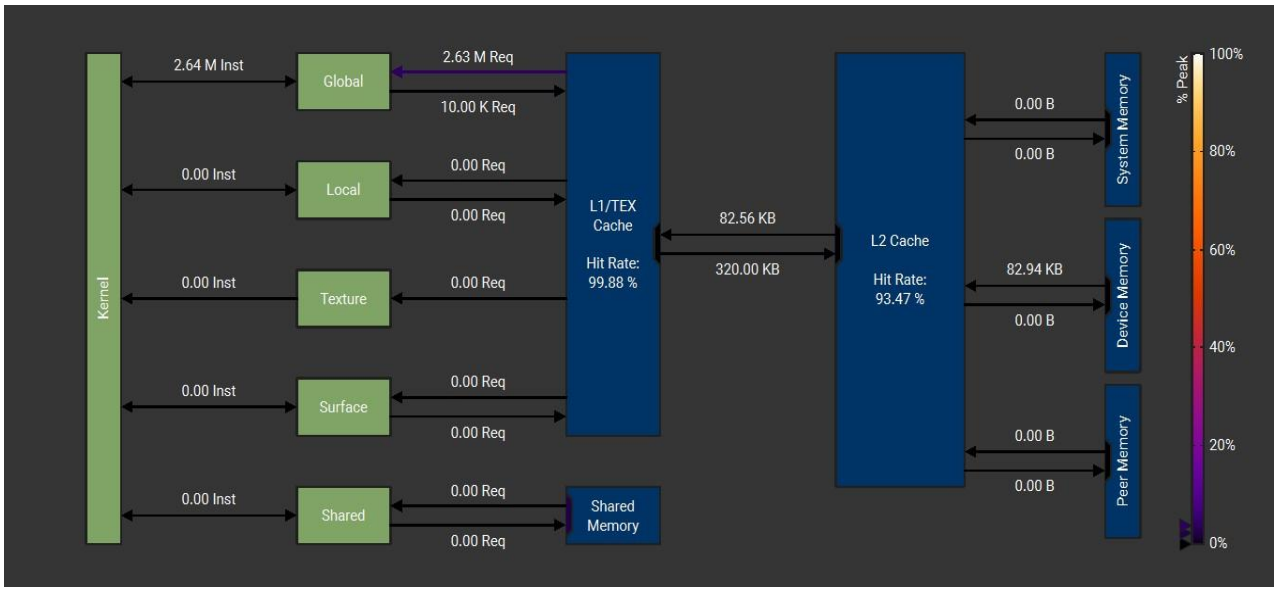
ANALYSIS

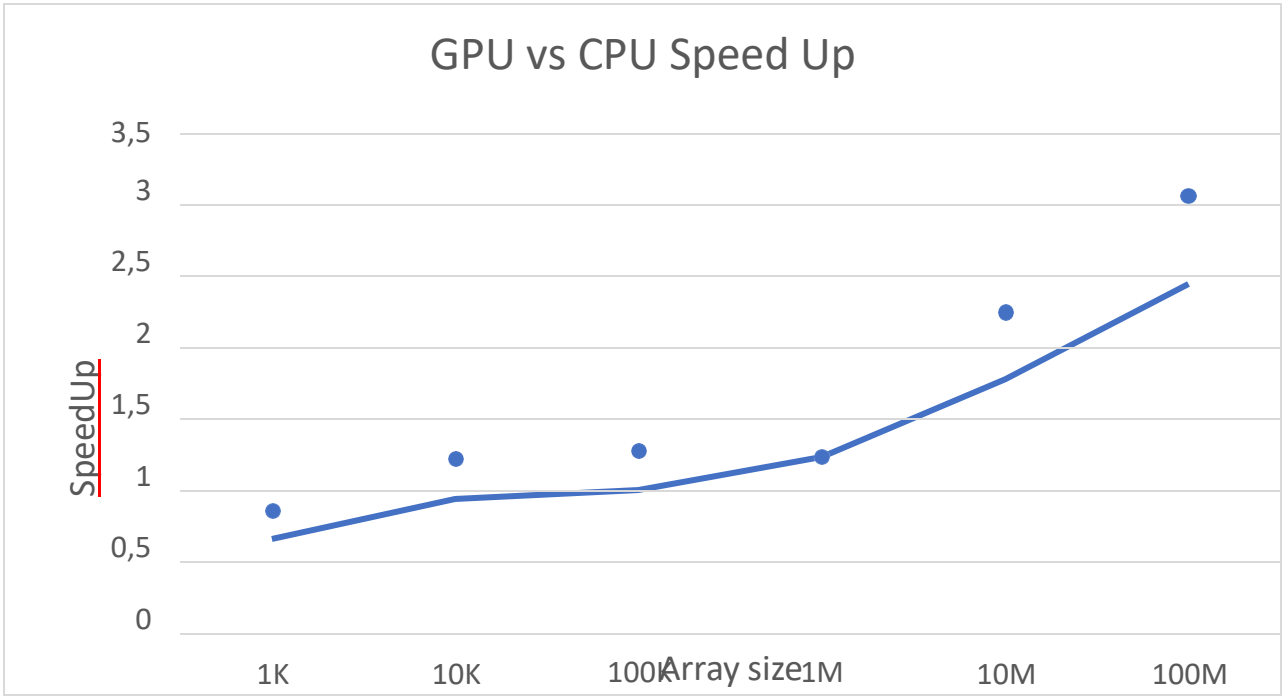
We analyzed the algorithm using two software, **Nvidia Nsight Compute** and **Nvidia Nsight Systems**.

Through these tools we have analyzed the algorithm and we have obtained the results shown in the graphs below.



Theoretical Occupancy [%]	100
Theoretical Active Warps per SM [warp]	32
Achieved Occupancy [%]	68,04
Achieved Active Warps Per SM [warp]	21,77





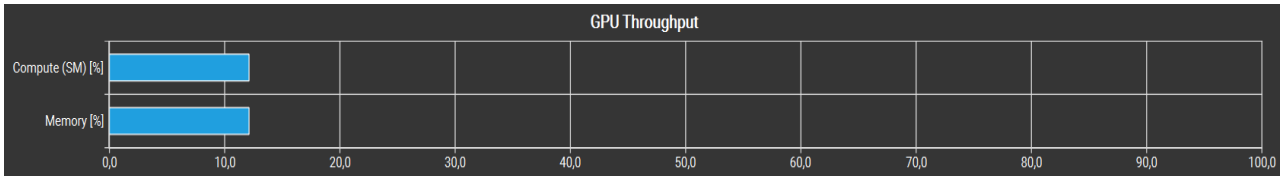
These first graphs show the strong points of our algorithm.

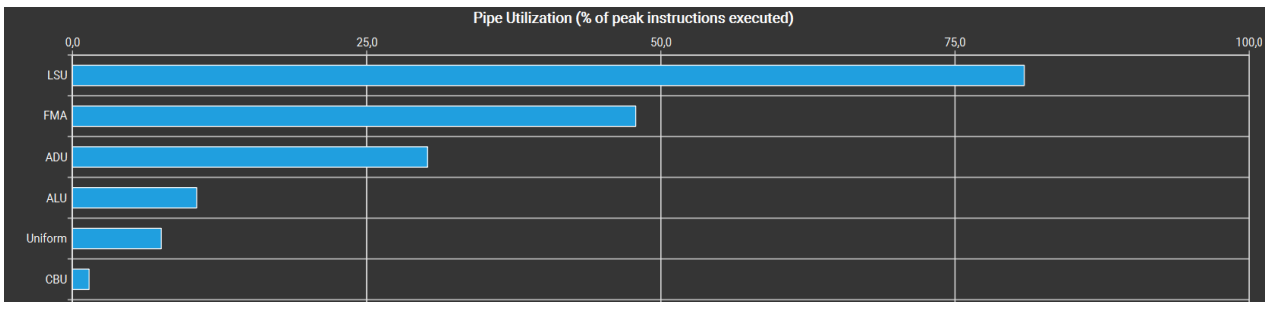
First, we have over 90% throughput for most kernel calls made.

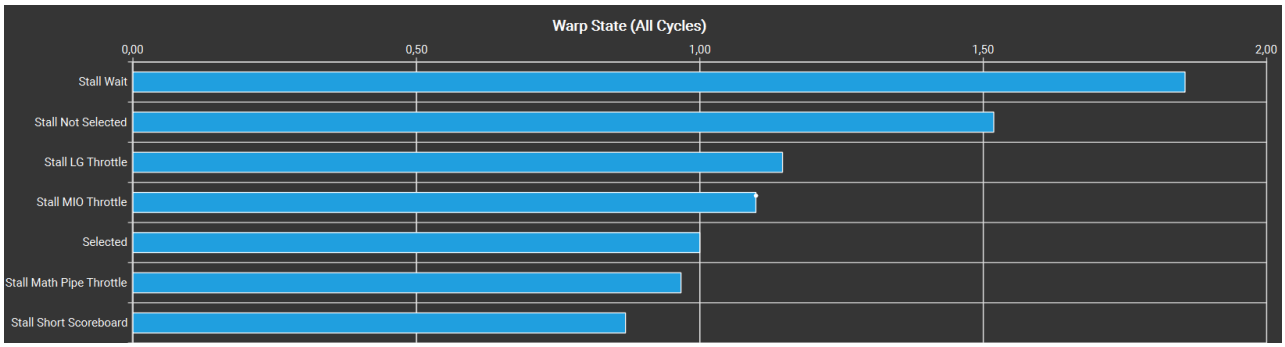
We also have a very high warp occupancy of around 70% and an equally high cache hit of almost 100% for both L1 and L2 caches.

We finally compared the speedup obtained on the GPU versus that obtained on the CPU, and we have that the CPU performs better for a smaller array size but, as the array size increases, we have a higher and higher gain.

This result is due to the fact that for a vector with few elements the GPU multithreading management overhead exceeds the gain obtained from it and therefore it is not suitable as a solution.







In these last graphs we show some negative sides of the algorithm.

First of all, as we have already seen for the CPU, the final merges are a bottleneck since they must be executed sequentially and therefore we have a low throughput in the last few calls, even if for most of the total calls to the kernel the throughput is very high.

In this case we have a different implementation compared to the CPU, in which the degree of parallelism decreases as the calls progress, until we have a sequential last call and therefore not very performing.

Furthermore, our algorithm is memory-bound, i.e. it spends most of its time waiting for data from memory and then performs very fast computations on it, and these types of algorithms do not fully exploit the computational power of the GPU.

This property of the algorithm can be seen from the fact that most of the pipeline is used in the "Load and Store" operation and the warps spend most of the time in the "Stall Wait" state and therefore waiting for the operands to be able to proceed with the calculations.

IMPROVEMENTS

After analyzing the algorithm through the profiler and identifying the bottlenecks we have made improvements to the code in order to increase performance.

```
for (int width = 2; width < (size << 1); width <<= 1) {
    long slices = size / ((nThreads) * width) + 1;

    gpu_mergesort<<<blocksPerGrid, threadsPerBlock>>>(A, B, width, slices, D_threads, D_blocks);

    A = A == D_data ? D_swp : D_data;
    B = B == D_data ? D_swp : D_data;
}
```

This first image shows a first improvement in which we replaced the multiplication by two with a shift inside the for. This improvement reduces the computational costs as the shift is a much lighter operation than the multiplication, as we learned during the labs.

```
__device__ void gpu_bottomUpMerge(long* source, long* dest, long start, long middle, long end) {
    long __align__ (8) i = start;
    long __align__ (8) j = middle;
    for (long k = start; k < end; k++) {
        if (i < middle && (j >= end || source[i] < source[j])) {
            dest[k] = source[i];
            i++;
        } else {
            dest[k] = source[j];
            j++;
        }
    }
}
```

We have inserted the keyword 'align' in the variables of the device code because aligning data in memory can improve performance in several ways:

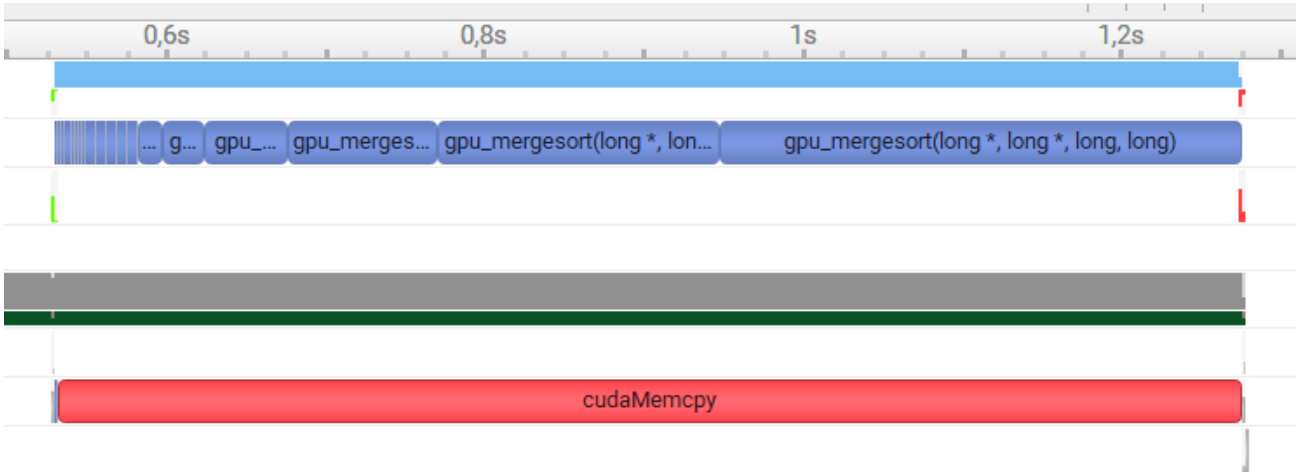
1. Efficient memory access: Modern processors often access memory in fixed-size blocks called cache lines. If data is not aligned properly, it may require multiple memory accesses to retrieve all the required data. In contrast, if data is aligned, more data can be fetched in a single memory access, reducing the overall number of accesses and improving efficiency.

2. Avoiding memory access penalties: Some hardware architectures may impose performance penalties when accessing unaligned data. This can result in additional delays in accessing the data and reduce overall performance.

3. Compiler optimizations: Data alignment can enable the compiler to generate more efficient code. The compiler can make assumptions about data alignment and optimize memory access instructions to make the best use of underlying hardware features.

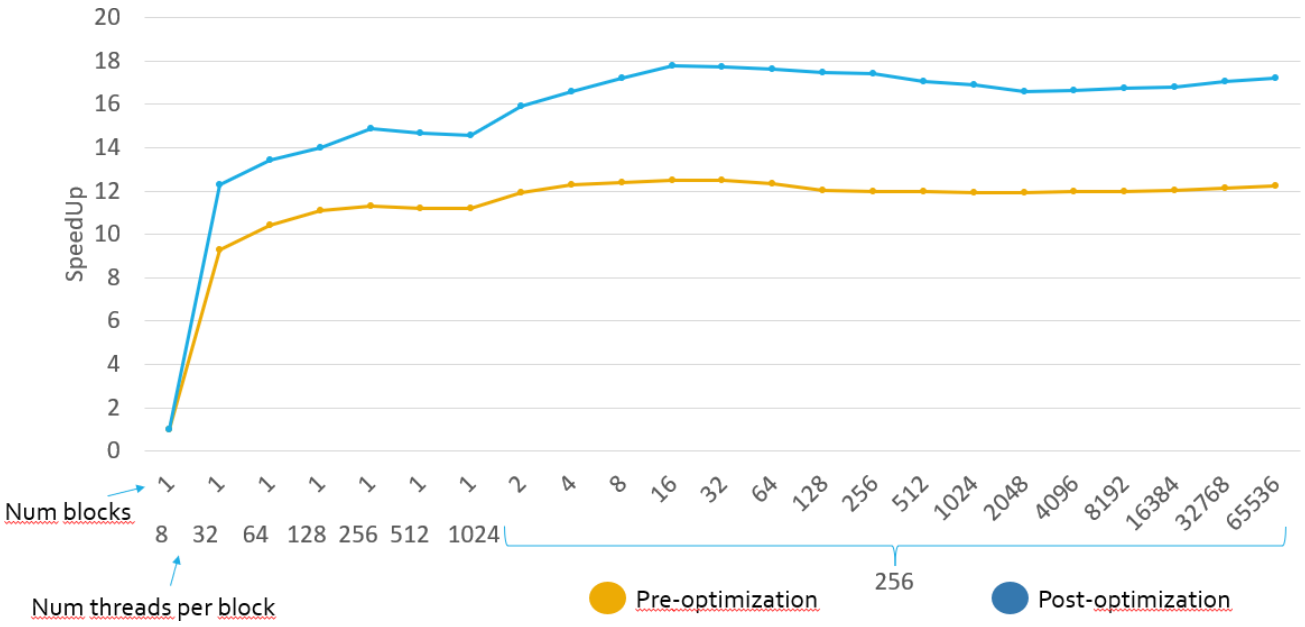
```
#define min(a, b) (a < b ? a : b)
#define size 1000000
#define NumThreads 1024
#define NumBlocks 65535
```

We replaced the constant variables with 'define' ones, so we don't have memory access.



Through the 'Nvidia Nsight Systems' profiler we were able to ascertain that the CudaMemcopy function is performed overlapped, therefore CPU and GPU work simultaneously and maximum utilization is obtained. This mechanism allows us to hide memory latency issues.

RESULTS



The graph in the figure shows the improvements obtained by applying the previously exposed improvements. We ultimately get a peak speedup, with respect to the pre-optimization version, of 18 using 256 threads per block and 16 blocks per grid.

One observation we came to after doing some analysis is that the merge-sort algorithm doesn't benefit too much in the GPU implementation.

This is due to several factors:

1. First of all, it is an algorithm that performs operations on one-dimensional arrays and therefore does not take full advantage of GPU parallelism.

2. Recursion is not properly implemented in cuda and therefore we had to implement an iterative version.

3. The degree of parallelization decreases as the calls progress, up to a recursive call and therefore not effective with the GPU implementation.