PARALLEL SORTING

Giorgio Locicero

Abstract

Every person that studies computer science should dedicate some thoughts into the topic of data parallelism and methods to exploit data parallelism where there is no evident independent data.
Sorting is a good example to show the difficulty and the insight behind algorithms that rely heavy on sequential computation but can be decomposed into small problems, it is also a good exercise to the mind to start doing things concurrently when they are not supposed to be done in parallel.

Synopsis

This short study is about sorting using parallel methods, taking advantage of some aspect of sorting numbers or objects(if transitive property is respected).

With this research I am trying to establish a framework for building parallel models for sorting or similar problems(especially problems where we can construct solutions for partitions of the problem and “merge” these solutions into one solution of a larger partition of the problem, trying to maximize the usage of the parallel nature of these problems).

During the implementation there will be many problems regarding the architecture, so I will try to find a solution that takes into account these implementation-specific task. I will also try to implement it using a hybrid approach in later chapters, using both GPU and CPU, cutting the task of sorting into lesser problems, alternating various algorithm to maximize the usage of more devices (I will only visit this argument because I only briefly visited the concept).
In the end of the paper I will present my most powerful algorithm (it is NOT the most powerful algorithm, it is MY most powerful).

The code and the story of the development is available at
https://github.com/josura/university-sad/tree/master/prog_GPU/miecoseprogmoderna/progettoGPU
everything is open source.
I will use OPENCL because it is not device specific and it is a flexible language.

While explaining the algorithms and strategy behind sorting, I will try to optimize the algorithm for a particular GPU architecture, but the study is nonetheless general for every architecture that could compute in parallel (multiple-thread, FPGA,etc…).
This paper will not consider things like transfer between devices and host, that is common knowledge to every person that studies computer science and GPGPU, and the topic is almost insignificant to the time required to sort an array of millions of elements.
The next section presents the intuition behind the parallel aspects of sorting and other algorithm that present a locally parallel behavior.

This is an University project, I will probably make some conceptual or logic mistakes because I lack the experience, I am open to constructive criticism, new ideas and optimizations of the algorithms that I will present.

Introduction

A sorting algorithm is an algorithm that puts elements of an array-list-data_structure in a certain order (increasing or decreasing, non-decreasing or non-increasing, other types of order that require a binary operator that are reflexive, anti-symmetric and transitive, also known as partial ordering over a set P with a binary operator).

The ordering problem is not an easy problem to parallelize: there is no obvious data stream that could proceed in parallel, there is no way of considering a subset of the data to find the global solution, and in some kind of way, every element is dependent to all other elements (every element needs to be compared in some kind of way to all other elements, this supposition can be avoided-diminished using the properties of the ordering operator, like the transitive property, in the chapter where I analyze some type of algorithms I will also talk about some algorithms that depend heavily on the transitive property).

With this type of problems, We need to find a solution that tries to convert the sequential heavy burden of computation, and transform it in a more parallel pattern of execution.

During the implementation we also depend heavily on the hardware that we use, and the hardware has limits that decrease our assumption on the possible parallelism of parallel sorting algorithms (not only sorting but every other algorithm that is not embarrassingly-parallel is affected), and during the implementation we will find our assumptions often stuck in a burden with the hardware, so that we need to accept a trade-off of parallelism-sequential computation that will affect our usage.

I will try different methods to resolve my problems, from the more simple and straightforward methods to augment the workload of a workitem (sliding window and such), to methods that rely on the instruments that are available from the device (local memory, vectorization in loads,store and operations).

Through this presentation we need not to forget that sequentialism in instruction does not mean only instructions that can only be executed one after another, but it also means that there will be a lot of conditional branches, and conditional branches are not good at all for parallel programming, processors in the same grid need to execute in lock-step, the code will execute in either case, only masked PE will not execute the instructions inside the branch, and this means that there are no real improvements in using conditional into the code (other than obvious and necessary logic).

If there are some missing parts or some parts of the code does not work properly (especially the firsts implementations of sorting), It is entirely my fault because such implementations are two months old and they are the result of my theories and hypothesis that have resulted to be erroneous (see the full parallel merge, or the mergesort from the low level), I know that I could have tried a little more to correct the old code (and I tried for a very long time), but I was so eager to implement new algorithms and experiment my ideas that I totally did not want to return to some of my old theories that are now deprecated, there would be hardly any point to correct the old code (at the end I will present my “final” algorithm, final for this presentation).

For the most part I will not present host code (I will show It only when it is important for the algorithm)

1. CONSTRUCTION OF AN ALGORITHM

To construct an algorithm to compute an optimal solution to a subset of a problem, we need to take into account a lot of variables that can tremendously affect the computation and our mind during the process of development.

As I have said, if the algorithm can be partitioned into sub-solutions of a greater problem, we can construct a type of parallelism that depends heavily on this assumption.
These type of problems are very important in computer science and are also referred to *Dynamic,* in fact Dynamic programming is a method for optimizing bigger problems that can be decomposed into sub-problems.
These type of algorithms are conceptually parallel, but at the same time share the sequential behavior that is seen in some naive implementation.

For this type of problem we will focus on a Bottom-up-approach, that is locally parallel and that does not depend too much on sequential computation, but the more we climb the bottom-up-tree, the more sequential and computation heavy it becomes.

We can think of different approaches to overcome the sequential burden that awaits at the head of the bottom-up approach, we could transfer the computation to hardware that can handle sequential code, but we also need to consider the memory transfer (if there is a memory transfer between two or more banks, some devices and ad-hoc machines use an hybrid approach that integrates a single memory space for all the devices).
We could try to evade the sequential burden, using some tricks to transform the sequential process of finding the optimal sub-solution, into a kind-of-parallel process that will slow down the sequential approach, but will parallelize the sub-problem. This type of solution depends greatly on the nature of the problem that we need to resolve.

As an example (It is the main focus of one chapter of this paper), the sorting algorithm depends heavily on sequential computation when we have two sorted sub-arrays that need to be merged, we can transfer the data in the CPU and merge the rest of the sub-problems using the CPU, but we have to waste some time to transfer the data between devices (GPU→CPU and also if we need the data in the GPU we need to re-transfer from CPU back to GPU), and also there is no guarantee that it would be more efficient in this way.

We will see that we can decompose the computation of the place of every element, making every item to search for itself its place.

Other times, there will be some strategies of resolving problems that need pre-processing to find a network of places to compare that does not need many conditional branches and instructions, this type of strategy will produce high usage during the true computation, but the pre-processing will be time-expensive(see bitonic search for a possible implementation, or ,more in general, sorting networks or other concepts that need this kind of pre-processing).

2.SORTING FOR SMALL PARTITIONS

Like I have said so far, sorting is not an easy task to do in parallel, there are a lot of technical difficulties during experimentation, and a lot of these are coming from hardware.

In this chapter I will present the main ideas behind some of my algorithms (those that worked properly) and find a way to implement these algorithm exploiting the parallel nature along with the software implementation.

I will start by presenting a naive approach (naive merge) that is an implementation of merge-sort where I will explore some parallel aspects (very few aspects in fact).

This part of the paper will focus over the sorting of small partition but the concepts presented are general, for the full algorithm of sorting see Merging(Ch.3)

After the discussion of the possible algorithm I will discuss about whether to do everything in one pass (with only a sorting algorithm that will sort the entire array) or divide the problem with a different method (the intuition behind this approach lays in speed, some algorithms will be very fast for small arrays but will lack speed when the length becomes too great, other algorithms will be slow with small arrays but will make little changes in speed when the amount of elements to sort will become high).

I will also discuss about where to do computation for larger partition, because if the array to sort will become too large, we will need to do some computation in CPU because the code will probably become a little too much sequential heavy (every PE will work on too many elements and the code will not be parallel, lots of conditions will arise, and the speed will decrease drastically).

During the discussion and explanation of the ideas of the algorithms, I will only present one algorithm and integrate all the optimization directly in the shown code; if anyone wants to see the code for older, buggier and not optimized version, go to the web page where I keep all my code (see Synopsis).
However, the discussion will involve the comparison in usage, bandwidth and speed between various versions (bandwidth and usage are not a reliable measure if done wrong, I do not have a profiler for OpenCL, so the bandwidth is calculated as (Byte_read + Byte_wrote) / execution_time; for a more reliable measure search for a profiler or confront the speed of different algorithms by brute_forcing plugging numbers and see how it goes).

2.1 Naive Merge

In this first algorithm I will give a first impression of parallel thinking, this chapter is the first plate, the more basic(other than the direct sequential approach) and intuitive.

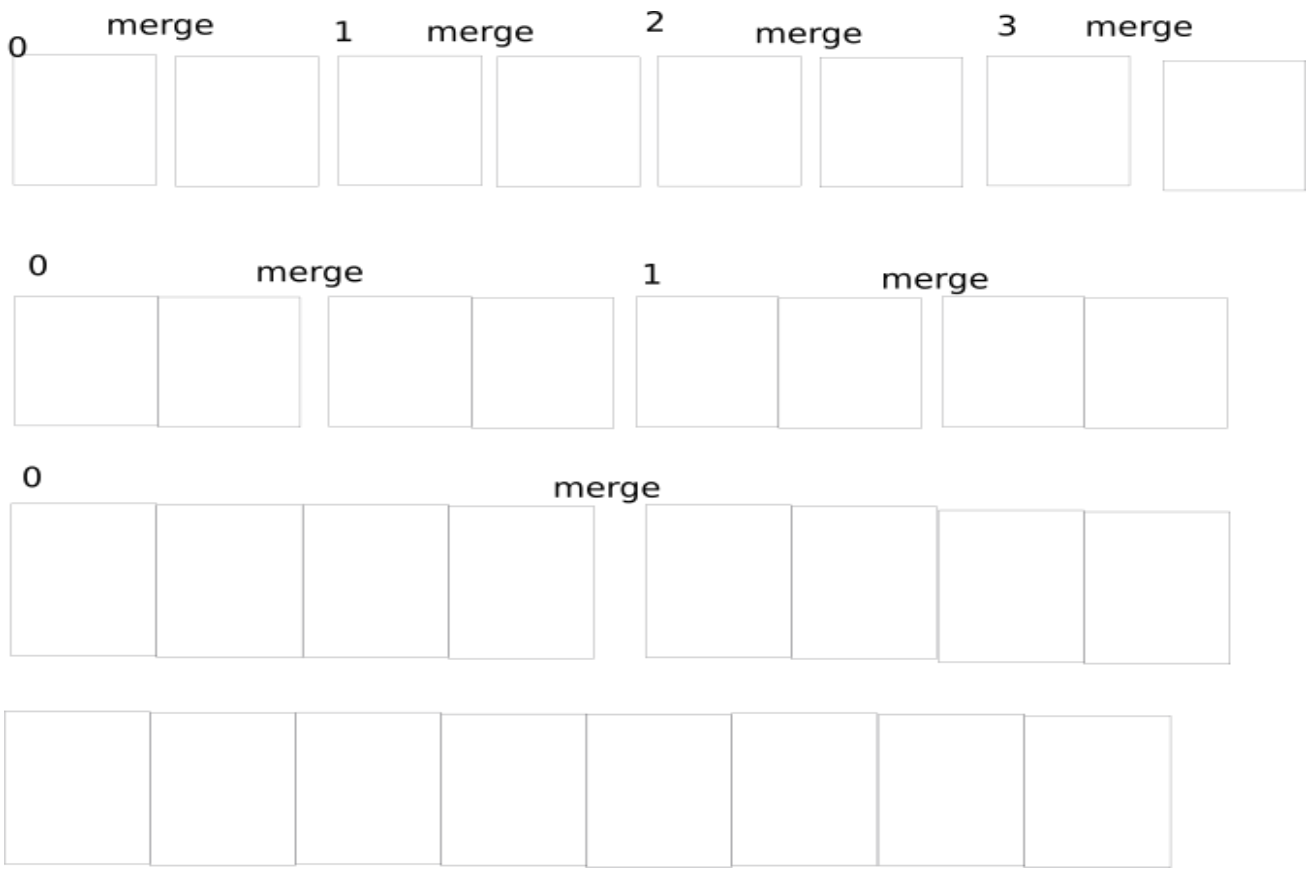


Figure 1: Every PE (a total of half the size of the number of elements) works in the first “iteration” and merges the two sorted sub-arrays, the mask of the working elements will grow, the number of working elements will be halved.

Substantially, every processing element (we will start with number_of_elements/2 processing elements, one more one less) works first on two elements and confronts the two (like two sorted list of length 1, the usual merge), merging the two elements. In the next step the PEs that will work to merge the subsequences will be halved, and every PE will merge its part, doubling the size of the sorted sub-arrays.

My implementation will only work with a power of 2 number of elements, this is because when I first implemented it the performance were not so great(it was really slow) so I have abandoned the possibility of expanding the concept, but the theory and intuition behind are there (also it is one of the worst algorithms just because of the heavy dependence of a single PE to many elements that are waiting to be ordered, and also because of not contiguous access to memory, there is no presence of memory coalescing).

The code of the kernel is shown below:

```
kernel void mergesortnaive(global int * restrict arr,int nels){
        unsigned actives = nels >> 1;
        //in the first iteration the active workitem are half the number of elements to be sorted
        uint gi=get_global_id(0);
        if(gi>=nels)return;
        uint initial_el = gi << 1;
        uint offset = 1;
        for(int i=1;i<nels;i<<=1){
                uint active_mask = i-1;
                offset <<= 1;
                if((gi & active_mask))return;    //the workitem is not active
                merge(arr + initial_el,arr + initial_el + (offset >> 1) , (offset >> 1), (offset >> 1) );
        }

}
```

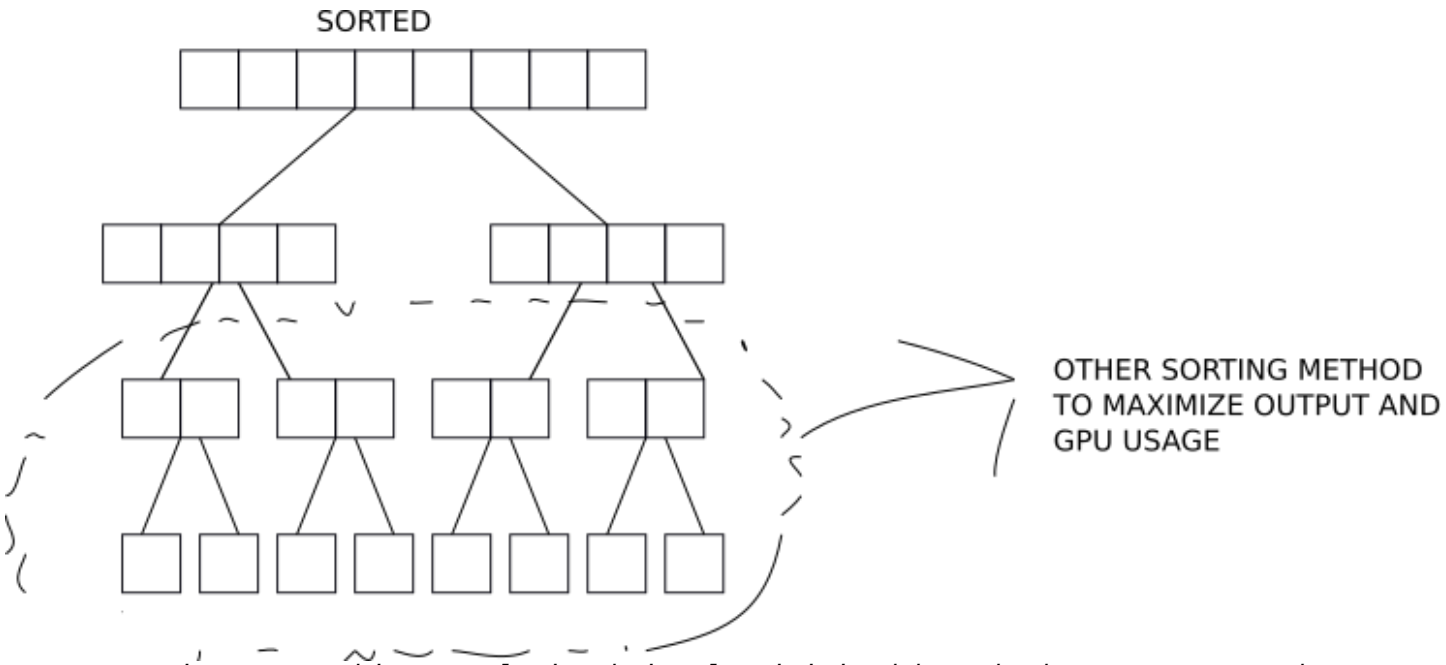
The bandwidth is so low that even after minutes of processing(every processing element that is active at the time will read `current_size * log(current_size)` and will write `(current_size)` more or less) it does not seem to end, there is no meaning in doing data analysis of this algorithm and its performance.

This algorithm is important to understand the problem behind sorting and the intuition for the solutions that I will present through this paper.
The **sorting tree(**or **merging tree)** is a tree where **every node is a sorted list-array**, and the **father** of two nodes **is the merged siblings** (the root is the complete sorted list-array).

This algorithm is constructed with a **bottom-up** manner with a merge-sorting algorithm(the logic is exactly that of the merge-naive presented before).
During the research for an algorithm I have also tried making a **top-down** type, but the results were not the one that I wished for(too slow, not working for specific cases, difficulty in optimization and intuition)

The bottom of the chain is the quickest part to sort, but yet it serves as a pillar to construct the full sorting algorithm.

Strategies differ from one another but I will take an hybrid approach, by first considering the sorting of small sub-sequences (with optimized algorithms), and then thinking about merging them.



:Merging tree, this example is obviously minimized but the bottom part may be composed of more levels.

2.2 Sliding window

During the research for an algorithm that could take advantage of parallel qualities of sorting, I stumbled upon various problems regarding the workload of a single work-item and the limited amount of processing elements that could execute in lock-step (also there were other problems on the maximum amount of local-memory and the limited number of work-items in a work-group).
A solution to these problems could come from a type of parallel programming that consists in making the processing elements work on elements that are distant from each other by a stride that is in fact the size of the PE that are in the same Compute Unit (probably not all executed in parallel but they are “awake” at the same time in the same CU).

This type of parallel programming (increasing the *workload* of single PE) can cover the ***latencies*** of memory access because while there is a grid of PE that can execute in parallel that will in fact execute, the other grid that is waiting for the memory access can wait and leave the control of the CU to the other grid.
We can also do everything with lockstep execution in mind, so that we don’t need barriers and synchronization points in the code, but in this way the workload will be low as well, independently of the approach that we use (sliding etc…).
We need an algorithm that is conscious of the sliding window approach: in the construction or adaptation of an already seen algorithm we and the code need to be aware of the stride, and the code of the kernel will grow along the logic behind it.
Augmenting the workload of a single PE could and should be beneficial, but at the same time there are situations were more simple and straight-forward parallel code with no assumption on the structure of our logic will be more quick (for example, in the next section I will cover an algorithm that is linked with the concept of bubble-sort; I will also do a sliding window approach, trying to resolve the problems that arise from limited memory and work-size).

By increasing the workload of every work-item we can evade memory latencies, like threads that switch between different process/threads, we can ease the burden of memory access (read or write).

This method can be used with almost every algorithm, I will use it only to demonstrate that in sorting It is not so convenient as an approach.

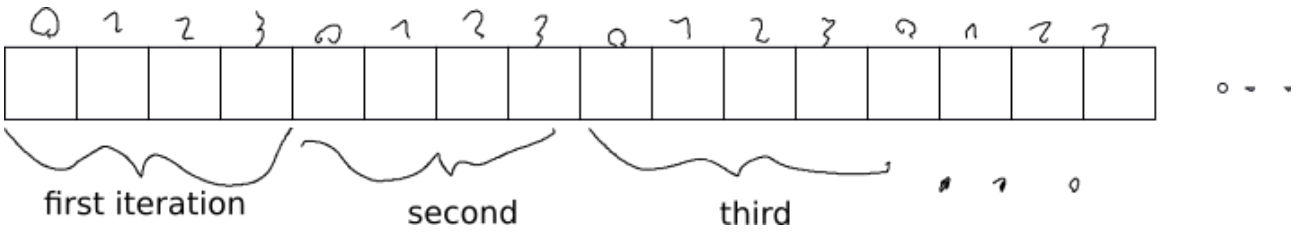


Figure 2: As an example of a sliding window, the stride is 4, the PE will work on aligned data, and close PE will work on close data (memory coalescence, the warp, in CUDA, will read contiguous location, accessing memory at the same time). It can not be seen in this example, but if there are reads from memory or writes in memory, there will probably be another work-group that will execute while the first is waiting for memory transfer.

2.3 Bubble-sort / Assembly-lane

This algorithm was my first functioning parallel algorithm that was effective and was functioning properly.

The fundamental idea is that of **bubble_sort**, where larger numbers will go at the end of the partition of the array (non decreasing order) and small number will go down. How it is done differs almost completely from the sequential bubble_sort.

I also thought of another name for this algorithm that summarizes the operations that it does during computation: I thought of it as an **assembly lane**, because every element will work on elements that fall in a partition of the array and will perform the same kind of computation for a limited number of iterations (in an assembly line, mechanical arms will work on objects that pass under a small distance under them, performing the same work over and over).

The algorithm is divided in two main components, the comparison between elements(and which element are compared) and how every PE will contribute by making greater elements go up in the array and small elements go down in the array.

Every PE will first work on two consecutive elements and will sort them, after all the PE have done this work, they will work again on two consecutive elements, but the first element will be the last of the first step.
This process is iterated *(Number_of_elements/2)* times to shift bigger numbers to the end (worst case is when the array is sorted in the opposite order that we are trying to obtain).

Also with this function we encounter one of the worst problems that will always be present in our implementation (especially for “old” hardware like mine), the last work-group should have less work-items because the last part of the array is not the perfect size most of the times, some work-items need to go “out” or not execute the code, if we use the return instruction It does not really return, and if we encounter a barrier the last work-group will get stuck because not all work-items in the work-group will call barrier, and the last part of the array won’t be sorted. There are many ways to solve this problem (I will show an implementation in later chapters of a more suitable algorithm with better performance that will need this solution), but for this problem I didn’t want to implement it because it was slow, not too slow, but enough to make me regret ever creating this algorithm (it is slow probably because of the apparent unconditional iteration, it is slow because the access is not vectorized and operations are plain simple, no binary operator), and yet there were times where this algorithm was surprisingly fast.

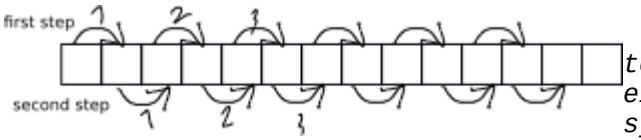


Figure 3: In the first step all PE work to compare and switch in case of two elements; after some kind of synchronization, the next step will take place.

The kernel code is the following:

```
kernel void local_miosort_lmemV3(global int2 * restrict arr,int nels, local int* lmem){
	const int gi=get_global_id(0);
	const uint lws=get_local_size(0);
	if(gi>=(nels+(lws>>1))/2)return;
	const uint groupid=get_group_id(0);
	const uint start=get_group_id(0)*(lws<<1);
	const uint end=(start+(lws<<1)>nels ? nels : start+(lws<<1));
	const uint el_lws = end-start ;
	const int li=get_local_id(0);
	const uint arr_local_index = li << 1;
	int2 tmp= arr[gi];
	//we are supposing that the number of elements is divisible for 2
	int indexfin1=arr_local_index | 1;
	int indexin2=(indexfin1<el_lws-1 ? indexfin1 : 0);
	int indexfin2=(indexin2 != 0 ? indexin2 + 1 : el_lws - 1);
	lmem[arr_local_index]=tmp.x;
	lmem[indexfin1]=tmp.y;
	barrier(CLK_LOCAL_MEM_FENCE);
	#pragma unroll
	for(int i=0;i< lws;i++){
		int confront1 = lmem[arr_local_index],confront2 = lmem[indexfin1];
		char comp=(confront1>confront2);
		lmem[indexfin1]=comp ? confront1 : confront2;
		lmem[arr_local_index]=comp ? confront2 : confront1;
		barrier(CLK_LOCAL_MEM_FENCE);
		confront1 = lmem[indexin2],confront2 = lmem[indexfin2];
		comp=(confront1>confront2);
		lmem[indexfin2]=comp ? confront1 : confront2;
		lmem[indexin2]=comp ? confront2 : confront1;
		barrier(CLK_LOCAL_MEM_FENCE);

	}
	tmp = (int2)(lmem[arr_local_index],lmem[indexfin1]);
	arr[gi]=tmp;
}
```

This version will sort only a partition of the array.
In this version, I have **optimized memory access with vector reads** from global memory and cache (local memory) where the elements can be sorted quickly.
Also this version is **limited** by the amount of local memory and number of work-items in a work-group that we can call to execute the kernel, we could use a **sliding window** to cover the full array (this type of algorithm involves only one work-group that will do everything, also local memory will be useless because we change it at every iteration, from start to end of the array), the work_size is half the number of elements (this is a local version that will only sort partitions that can be kept in local memory and then copied).
As we can see, the code is pretty straight-forward, loading into local memory the values (2 values per work-item), than do the compare and switch iteration for `lws` times (for a partition of *lws*2* elements).

This version needs a number of elements that are divisible by 2 (for the vector access to memory, we can even overcome this problem but this will be a discussion for the future).

There is also the sliding window version, not optimized with local memory because, as I have already said, it will be useless with the logic behind, no other optimization (no vectorization for coalesced reads or writes). It is the simple “assembly lane” with a sliding window that will slide through the array until the end.
Only one workgroup is permitted.

```
kernel void miosort_sliding(global int * restrict arr,const int nels){
    int gi=get_global_id(0);
    int lws = get_local_size(0);
    int gws = (nels+1)>>1;
    if(gi>=gws)return;
    int shift = lws << 1;
    for(int i=0;i<nels >> 1;i++){
                //every workitem will go through elements that are lws<<1 away from each other
        for(int sliding_off=0;sliding_off<nels;sliding_off += shift){
            int iniziamo = (((gi << 1) + 1 + sliding_off) >= nels);
            int indexin = ( iniziamo ? 0 : ((gi << 1) + sliding_off)) ;
            int indexfin= ( iniziamo  >= nels ? nels-1 : indexin+1);
            if(arr[indexin]>arr[indexfin]){
                int tmp = arr[indexfin];
                arr[indexfin]=arr[indexin];
                arr[indexin]=tmp;
            }
        }
        for(int sliding_off=0;sliding_off<nels;sliding_off += shift){
            int iniziamo = (((gi << 1) + 1 + sliding_off) >= nels);
            int indexin = ( iniziamo ? 0 : ((gi << 1) + sliding_off)) ;
            int indexfin= ( iniziamo  >= nels ? nels-1 : indexin+1);
            int indexpast=indexin;
            indexin=(indexin+2>=nels ? 0 : indexin+1);
            indexfin=(indexpast+2 >= nels ? nels-1 : indexpast+2);
            barrier(CLK_GLOBAL_MEM_FENCE);
            if(arr[indexin]>arr[indexfin]){
                int tmp = arr[indexfin];
                arr[indexfin]=arr[indexin];
                arr[indexin]=tmp;
            }
            barrier(CLK_GLOBAL_MEM_FENCE);
        }
    }

}
```

2.4 Parallel Counting sort

This algorithm is very simple, but yet very effective when implemented well in parallel. Taking into account architecture attributes and platform specification, it is **the most useful and quick** among all other implementations that I have tried (for **sorting small partition**, at the bottom end of the sorting and merging chain, and for old hardware).

The idea behind it is straight-forward, every PE will search ,for the element that is working on, the place where it should be.
It similar to the normal counting sort in the counting of every number that is less or equal than the pivot element.

It is probably for the simplicity of the algorithm, for the intuitive optimization with both local memory and vectorization (both with memory vectorization and vector operation, I do not know about bank conflicts), or the low amount of instruction required, but this algorithm outspeeds every other algorithm in every implementation for a small number of partition's cardinality.
The code of the kernel to sort small partitions of the array of the size of the workgroup is shown below:

```
kernel void local_count_sort_vectlmemV3(global int4 * restrict arr,int nels, global int* restrict res,local int4 * lmem){
    const int gi= get_global_id(0);
    const int group_id = get_group_id(0);
    int pivot;
    const int li=get_local_id(0);
    const int lws=get_local_size(0);
    const int first_el = group_id*lws;
    // "mask" for the workitem that need to read the global memory to store in local
    const int reading_quarts=(group_id == nels/lws ? (nels-(nels/lws)*lws)>>2 : lws>>2);
    const int reading_place = (group_id * (lws>>2)) + li;
    //casting to take the pivot
    global int* restrict arr_scalar = (global int* )arr;
    //reading into local memory and taking the pivot to sort
    if(gi<nels){
        pivot = arr_scalar[gi];
        if(li<reading_quarts){
            lmem[li] = arr[reading_place];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    int counter=0,repetition=0;
    int4 counter4 = (int4)(0),repetition4 = (int4)(0);
      //compare the pivot with the element of the partition
    if(gi<nels){
        for(int i=0;i<(reading_quarts); i++){
            const int4 compar = lmem[i];
            repetition4 -= compar == pivot ;
            counter4 -= compar < pivot;
        }
    }
      //this barrier is not necessary but if the destination array is the same as the input array,
      //this fence will take care of it
    barrier(CLK_LOCAL_MEM_FENCE);
    if(gi<nels){
        repetition=repetition4.s0+repetition4.s1+repetition4.s2+repetition4.s3;
        counter=counter4.s0+counter4.s1+counter4.s2+counter4.s3;
        for(int i = first_el + counter; i < first_el + counter + repetition; i++){
            res[i]=pivot;
        }
    }
}
```

This version is **not stable**: if the order of two “equal” elements matter, don’t use this version (a stable version needs to **check the position of equal element**).
The code for a stable version is below:

```
kernel void stable_local_count_sort_vectlmemV3(global int4 * restrict arr,int nels, global int* restrict res,local int4 * lmem){
    const int gi= get_global_id(0);
    const int group_id = get_group_id(0);
    int pivot;
    const int li=get_local_id(0);
    const int lws=get_local_size(0);
    const int first_el = group_id*lws;
    const int reading_quarts=(group_id == nels/lws ? (nels&(lws-1))>>2 : lws>>2);
    const int reading_place = (group_id * (lws>>2)) + li;
    global int* restrict arr_scalar = (global int* )arr;
    if(gi<nels){
        pivot = arr_scalar[gi];
        if(li<reading_quarts){
            lmem[li] = arr[reading_place];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    int counter=0;
    int4 counter4 = (int4)(0);
    if(gi<nels){
        for(int i=0;i<(reading_quarts); i++){
            const int4 compar = lmem[i];
            const int4 positions = (int4)(0,1,2,3)+(i<<2)<li;
            counter4 -= (compar < pivot)|( positions && (compar == pivot));

        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if(gi<nels){
        counter=counter4.s0+counter4.s1+counter4.s2+counter4.s3;
        res[first_el + counter]=pivot;
    }
}
```

These versions need **the size of the array to be divisible by 4** (for the vectorization, there are other methods to solve this problem but it will be tedious and it will depend on the taste).

There was also a sliding window version (with local memory and vectorization) but it was slow (not too much but slow in comparison to the merging algorithm in the next chapter) and was not functioning as intended.

2.5 Binary Merge-sort

Confronting the merging problem in parallel is not easy, every implementation needs a **trade-off between access in memory or complicated logic** and many conditional branches.
The problem of merging is also not quick to solve for small arrays, the logic is not equal for all the work-items, and optimization to make the code less branch-full will result in more code to execute (I am considering speed for small arrays because the target for this chapter is to sort the bottom part of the sorting tree).

For the bottom implementation of parallel merge_sort, I will use a **full parallel** approach, where *every PE works to find the position for its pivot* (the HALF implementation is easy to implement, I will use it in the MERGING (ch. 3) chapter, but the key of choosing this implementation is that this is full parallel and the data used to find the elements is in local memory so the latency is low and the access can be done quickly).
As I have said, we need to **merge the siblings** and form the greater merged array. Every work-item will work on its pivot and will **binary search** the elements that are less than (or equal to depending on the siblings and avoiding collision in position of sorting) the elements of the other siblings.
The code of the two main bugged and slow binary_search is below (variations for HALF and FULL parallel are in the source, I do not want to be redundant).
The final position is exactly the element behind the current sibling (the position in the array) plus the binary_index found in the previous step.
To find the place we use a binary search on the siblings, there are two implemented versions, one that is more conditional (less access to memory and minor instruction, more waiting and masks), and one that will always access the memory for log(length) times, where length is the current size of the siblings to be merged.
The code for old bugged binary_search is below:

```
inline int binaryIndex(int pivot,global int* find1, int size){
    if(pivot>find1[size-1]){
        return size;
    }
    if(pivot<find1[0])return 0;
    int l=0 ,m, r=size-1 ;
    while (l <= r ) {
        m = l + ((r - l)>> 1);
        if ((m+1)<(size) && (pivot > find1[m]) && (find1[m+1] >= pivot))
          return m+1;
        if ((find1[m] < pivot) )
          l = m + 1;
        else
          r = m - 1;
    }
    return m;
}

inline int binaryLoc(int pivot, global int* find1, int size){
    int pos = 0;
    for (int inc=size;inc>0;inc>>=1) // binary search in the sub-sequence
    {
        int j = pos+inc-1;
        int confront2 = find1[j];
        pos += (confront2 < pivot)  ? inc:0;
        pos = min(pos,size);
    }
    return pos;
}
```

The new binary search used(optimized and corrected for full parallel merge and normal searching for HALF parallel merge) is below:

```
inline int binaryIndexFullPower(int pivot,global int* find1, int size){
	int l=0 ,m, r=size-1;
	while (l<=r ){
		m = l + ((r - l)>> 1);
		int confront1 = find1[m];
		if ((confront1 < pivot) )
		  l = m +1;
		else
		  r = m -1;
	}
	return  l;
}

inline int binaryIndexFullPower_corr(char condition,int pivot,global int* find1, int size){
	int l=0 ,m, r=size-1;
	while (l<=r ){
		m = l + ((r - l)>> 1);
		int confront1 = find1[m];
		if ((confront1 < pivot)||(condition && confront1 == pivot) )
		  l = m +1;
		else
		  r = m -1;
	}
	return  l;
}
```

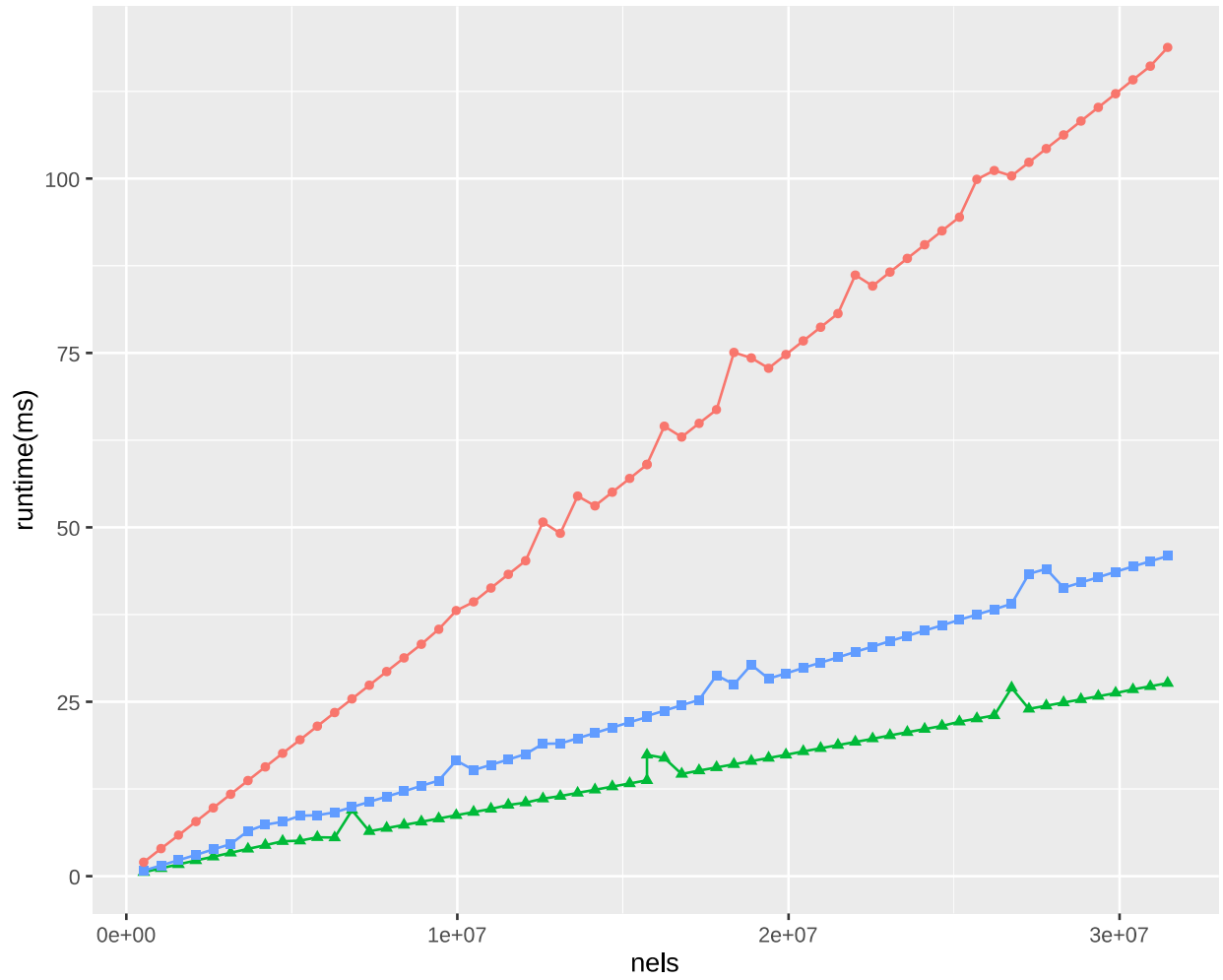
The code for the kernel of merge-sorting small sub-sequences is below:

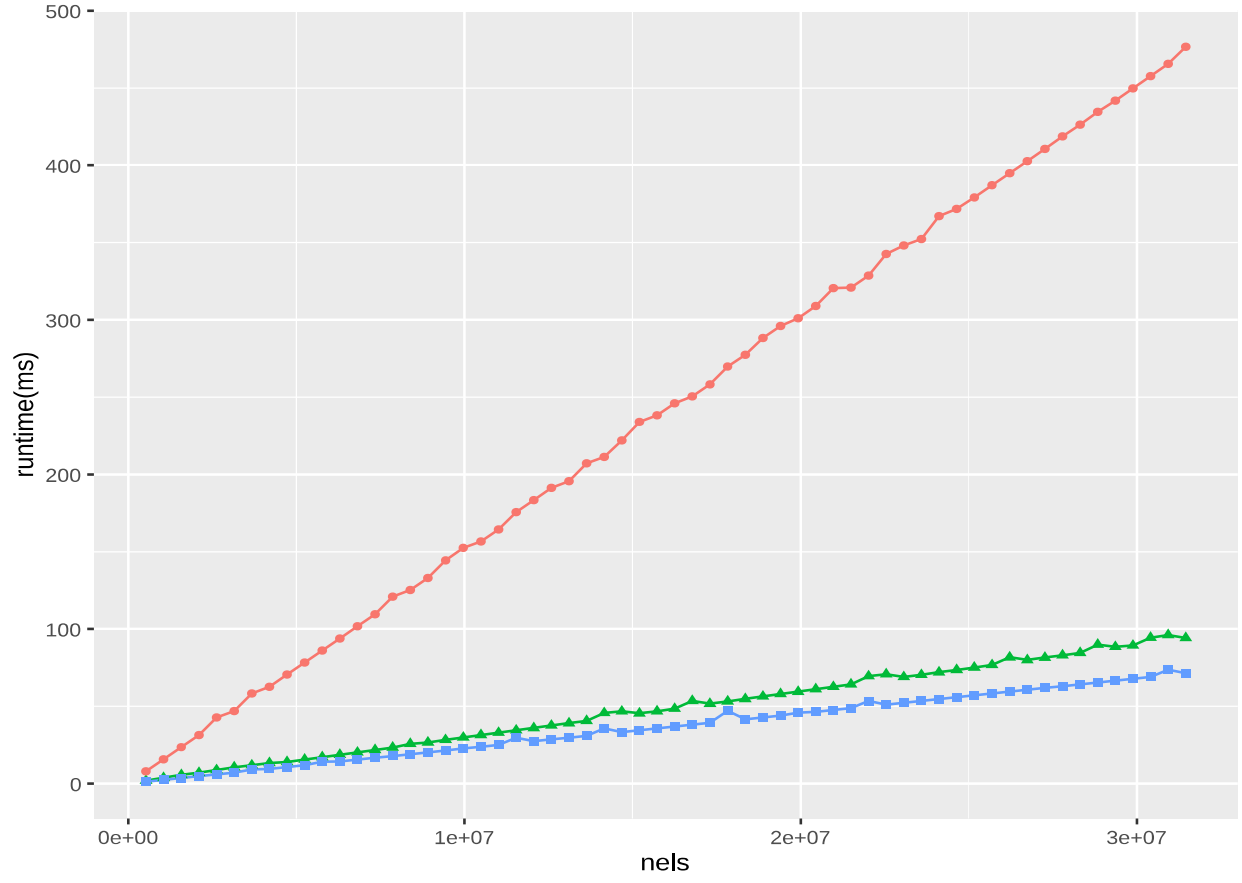
```
kernel void ParallelMerge_Local(global const int * in,int nels,global int * out,local int * lmem)
{
    const int li = get_local_id(0);
    int wg = get_local_size(0);
    const int gi = get_global_id(0);
    // Move to the start of the subarray
    int offset = get_group_id(0) * wg;
    // see if we are at the end of the array
    wg = (offset+wg)<nels ? wg : nels - offset;
    in += offset; out += offset;

    //loading in local memory
    if(li<wg)
    lmem[li] = in[li];
    barrier(CLK_LOCAL_MEM_FENCE);

     // merging sub-sequences of length 1,2,...,WG/2
    for (int length=1;length<wg;length<<=1)
    {
        int pivot = lmem[li];
        int ii = li & (length-1);  // index in our sequence in 0..length-1
        int sibling = (li - ii) ^ length; // beginning of the sibling to find the position
        int pos = 0;
        for (int inc=length;inc>0;inc>>=1) // binary search in the sub-sequence
        {
            int j = sibling+pos+inc-1;
            int confront2 = (j<wg ?  lmem[j] : INT_MAX);
            bool smaller = (confront2 < pivot) || ( confront2 == pivot && j < li );
            pos += (smaller)?inc:0;
            pos = min(pos,length);
        }
        int bits = (length<<1)-1; // mask for destination
        int dest = ((ii + pos) & bits) | (li & ~bits); // destination index in merged sequence
        barrier(CLK_LOCAL_MEM_FENCE);
        lmem[dest] = pivot;
        barrier(CLK_LOCAL_MEM_FENCE);
    }

// Write output
    if(li<wg)
    out[li] = lmem[li];
}
```

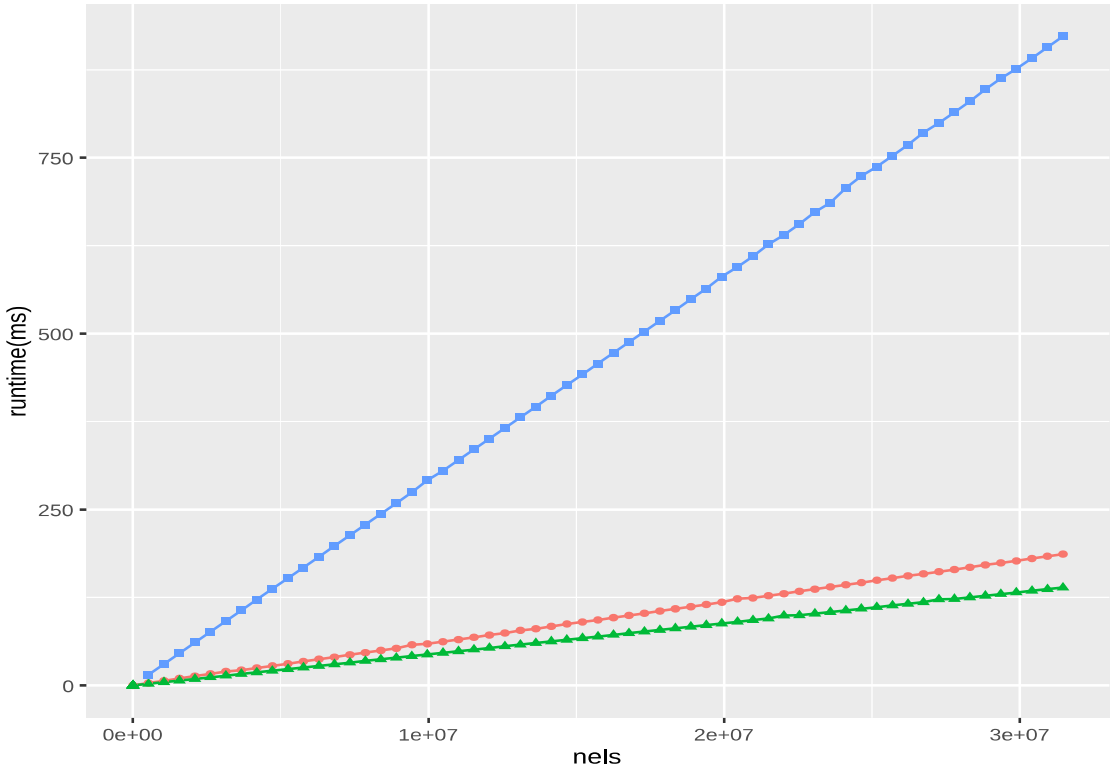
There was a problem with this algorithm, There was a bug if equal elements in the two parts of the sorted array to merge differ from one other one element, one of the elements to be sorted will be lost(the problem of this version is with floating point numbers, for an absurd reason, for integers the problem will not arise). Finally, I had found the problem after troubleshooting for two weeks, I have **fixed** it by changing the function to search for the index(also the new version will not cause segmentation faults)).

The version that will always work(hopefully),modified for floats, is below:

```
inline int local_binaryIndexFullPower_corr(char condition,float pivot,local float* find1, int size){
    int l=0 ,m, r=size-1;
    while (l<=r ){
        m = l + ((r - l)>> 1);
        float confront1 = find1[m];
        if ((confront1 < pivot)||(condition && confront1 == pivot) )
          l = m +1;
        else
          r = m -1;
    }
    return  l;
}

kernel void ParallelMerge_Local(global const float * in,int nels,global float * out,local float * lmem)
{
    const int li = get_local_id(0);
    int wg = get_local_size(0);
    const int gi = get_global_id(0);
    // Move to the start of the subarray
    int offset = get_group_id(0) * wg;
    // see if we are at the end of the array
    wg = (offset+wg)<nels ? wg : nels - offset;
    in += offset; out += offset;

    //loading in local memory
    if(li<wg)
    lmem[li] = in[li];
    barrier(CLK_LOCAL_MEM_FENCE);
     // merging sub-sequences of length 1,2,...,WG/2
    for (int length=1,shiftcond=0;length<wg;length<<=1,shiftcond++)
    {
        float pivot = lmem[li];
        int ii = li & (length-1);  // index in our sequence in 0..length-1
        int sibling = (li - ii) ^ length; // beginning of the sibling to find the position
        int pos = 0;
        pos = local_binaryIndexFullPower_corr((sibling>>shiftcond)&1,pivot,lmem+sibling,length);
        int bits = (length<<1)-1; // mask for destination
        int dest = ((ii + pos) & bits) | (li & ~bits); // destination index in merged sequence
        barrier(CLK_LOCAL_MEM_FENCE);
        lmem[dest] = pivot;
        barrier(CLK_LOCAL_MEM_FENCE);
    }

 // Write output
    if(li<wg)
    out[li] = lmem[li];
}
```


runtime(ms)
100
75
50
25
0
0e+00
1e+07
2e+07
3e+07
nels
GPU NVIDIA 850M
LOCAL WORK SIZE = 64
nome
sorting_assemblyLane
sorting_counting
sorting_merging


runtime(ms)
500
400
300
200
100
0
0e+00
1e+07
2e+07
3e+07
nels
GPU NVIDIA 850M
LOCAL WORK SIZE = 256
nome
sorting_assemblyLane
sorting_counting
sorting_merging



GPU NVIDIA 930MX

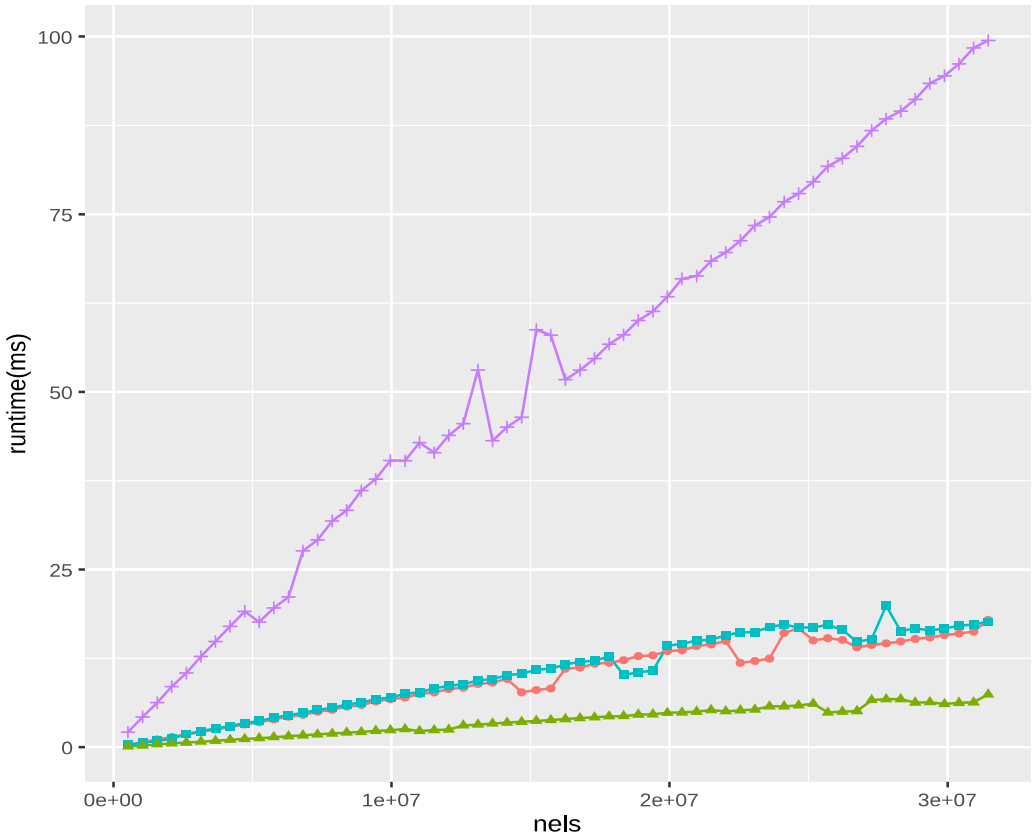
LOCAL WORK SIZE=256

nome

./sorting_counting_parallel_merge

./sorting_merging_parallel_merge

sorting_assemblyLane



GPU NVIDIA 1660 SUPER

LOCAL WORK SIZE=256

nome

./sorting_counting_parallel_merge

./sorting_merging_parallel_merge

./sorting_stablecounting_parallel_merge

sorting_assemblyLane

Augmenting the work-size will result in bigger sorted sub-partition of the bottom part of the sorting tree, but the tradeoff in execution time with the global merging algorithm could be greater.
For a secure analysis, run the code and see what is the fastest(changing the *workgroup size*).
A small optimization could be done when loading into local memory, vectorizing the loads, memory bandwidth will grow (notable speed enhancement).
The vectorized version of merge_local is below:

```
kernel void ParallelMerge_LocalVect(global int4 * restrict in,int nels,global int * out,local int4 * lmem)
{
    const int li = get_local_id(0);
    int wg = get_local_size(0);
    const int gi = get_global_id(0);
    const int group_id=get_group_id(0);
    // Move to the start of the subarray
    int offset = group_id * wg;
    // see if we are at the end of the array
    wg = ((offset)+wg)<nels ? wg : nels - (offset);
    in += offset>>2; out += offset;

    //loading in local memory
    const int reading_quarts=(group_id == nels/wg ? (nels&(wg-1))>>2 : wg>>2);
    if(li<reading_quarts){
        lmem[li] = in[li];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    local int* lmem_scalar = (local int* )lmem;
     // merging sub-sequences of length 1,2,...,WG/2
    for (int length=1,shiftcond=0;length<wg;length<<=1,shiftcond++)
    {
        int pivot = lmem_scalar[li];
        int ii = li & (length-1);  // index in our sequence in 0..length-1
        int sibling = (li - ii) ^ length; // beginning of the sibling to find the position
        int pos = 0;
        pos =
local_binaryIndexFullPower_corr((sibling>>shiftcond)&1,pivot,lmem_scalar+sibling,length);
        int bits = (length<<1)-1; // mask for destination
        int dest = ((ii + pos) & bits) | (li & ~bits); // destination index in merged sequence
        barrier(CLK_LOCAL_MEM_FENCE);
        lmem_scalar[dest] = pivot;
        barrier(CLK_LOCAL_MEM_FENCE);
    }

 // Write output
    if(li<wg)
    out[li] = lmem_scalar[li];
}
```

2.6 Other famous parallel algorithm for sorting

Bitonic sort is an algorithm that uses a type of strategy that is called **“sorting network”**, the idea behind is that the sequence of comparison between elements can be calculated early.
The name comes from the kind of sequences that will result during the algorithms:

$$x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$$ Bitonic sequence

I will not implement this kind of algorithm because most of the times the computation of the sorting network is extremely inefficient, and the amount of memory required to store the sorting network is not to be underestimated (I have tried other implementations of other people but every implementation presents the same problems).

3. MERGING TO THE TOP

Up until now I have focused on sorting small arrays on GPU to maximize the output for very big arrays and find solution to sub-problems quick enough to leave the problem of small arrays aside.
The topic of **merging large partitions** is different, as we have seen in the chapter of Binary merging sort(2.5), merging can become quite the task to do in parallel, and while constructing and designing an algorithm, we encounter many possibilities and critical points to consider.

One of the main critical points is the **repetition of elements** in the array, in parallel merging this is the most troublesome of problems (I have wrote an implementation for a parallel merge that does consider arrays with no repetition, obviously it is “quick”, however impractical and not so useful).

Two main problems with two main ideas to resolve them are the main focus of this chapter: what would we like to have, **full parallelism** and more access to memory or **half parallelism** and a good amount of access to memory?

For now we can forget about optimization with local memory (in the next paragraph we will see two implementations that will use local memory to merge).

The full parallel merging was already visited, but the HALF-parallel is new.

Basically, when we find the index of the pivot, we can directly copy the elements that are less than the pivot and greater or equal than the previous element to the pivot.

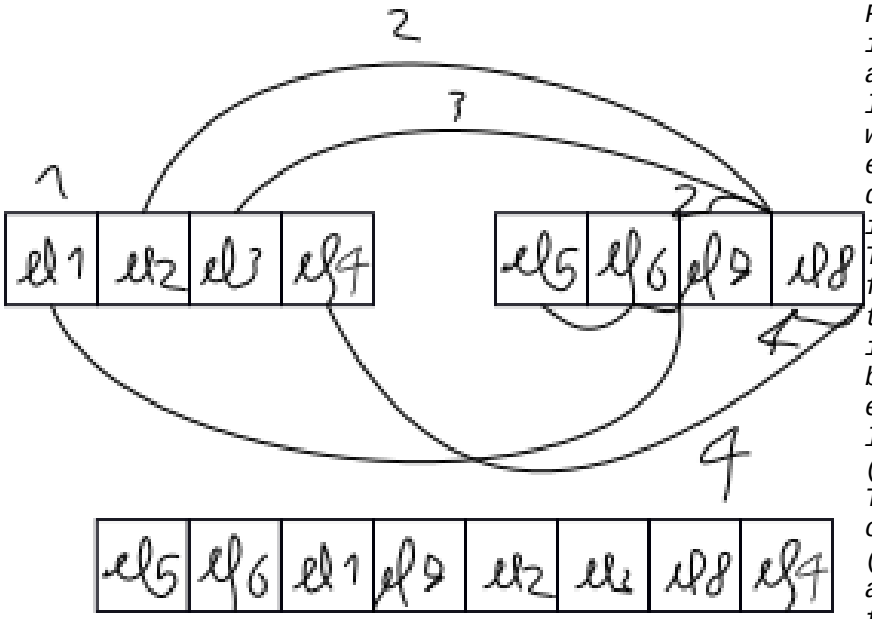


*Figure 4: The first work item will find that there are two elements that are less than its pivot, it will also copy the two elements to the destination array because it is the first element. The second element will find its place and copy the previous element where it had found its place, because the previous element to the pivot was less than this element (el7>=el1 && el2>el7). The third element will only find its place (el2>el7)
and the last element will find its place, copy the element before (el8>el3), and eventually copy the elements after (in this case there are no other elements).*

The code for the two main implementations(full parallel and HALF parallel) is below:

```
kernel void mergebinaryWithRepParallelV4(global int * out ,global int* arr,int nels, int subsize ){
    const int gi = get_global_id(0);
    if(gi>=nels)return;
    const int subsetid = gi/subsize;
    int start,end,index;
    const int pivot = arr[gi];
    char condition = subsetid & 1;
    start = condition ? (subsetid - 1) * subsize : (subsetid + 1) * subsize;
    end = (start + subsize) < nels ? start + subsize : nels;
    int locindex=0;
    if(start<nels) locindex = binaryIndexFullPower_corr(condition,pivot,arr + start , end - start);
    index=(gi-(condition ? subsize : 0))+locindex;
    out[index] = pivot;

}
```

As with the algorithm already visited in chapter 2.5 , every PE will work to find the destination for its pivot.

```
kernel void mergebinaryWithRepHalfParallelV2(global int * out ,global int* arr,int nels, int subsize ){
    const int gi = get_global_id(0);
    if(gi>=round_mul_up((nels)>>1,subsize))return;
    const int gws = get_global_size(0);
    short subindex = 0;
    for(int shiftsize=subsize;shiftsize>1;shiftsize>>=1)subindex++;
    //This algorithm requires the subsize to be a power of 2
    int subitem = gi & (subsize-1);
    //arr_start is the starting point where work-items will search for the index
    const int arr_start = (gi>>subindex)*(subsize<<1);
    if(arr_start + subitem < nels){
        int pivot = arr[arr_start + subitem];
        int start = arr_start+subsize , index, size;
        size = ((start + subsize) < nels ? subsize : nels - start);
        int locindex=0;
        if(start<nels) locindex = binaryIndexFullPower(pivot , arr + start , size);
        int element_index = arr_start + subitem;
        index = element_index + locindex;
        out[index] = pivot;
        // index of the place in between the numbers(or at the end or start) where the pivot was "found"
        int place_index = start + locindex - 1;
        int current_index = index;
        //copy all elements that are less than or equal to the previous element
        if(element_index > arr_start){
            int comp_previous = arr[element_index-1],comp_other;
            while(place_index>=start && arr[element_index-1]<=(comp_other=arr[place_index]) ){
                out[--current_index]=comp_other;
                place_index--;
            }
        }
        //if It is the first element, It copies the first part of the second array
        if(!(subitem & (subsize - 1))){
            while(place_index>= start)out[--current_index]=arr[place_index--];
        }
        //If It is the last element, It copies the last part of the second array
        if(subitem == subsize-1){
            int repetition = start+locindex;
            while(repetition< start + size){
                out[++index]=arr[repetition++];
            }
        }
    }
}
```

Obviously, other versions and tests are available at my gitHub page.

The two algorithms are different in their nature, like I had said already when I was talking about the local_merging algorithm, favoring a full parallel algorithm will result in more access to global memory and less instructions to

compute, but in global memory more access means hundreds of instruction cycles wasted.
The HALF-parallel algorithm, on the other hand, will require more instructions to execute and more conditionals, but the access to global memory will be a lot less.
The main problem with the HALF-parallel implementation is that at the head of the merging chain it becomes a little too much sequential and some elements will do a lot of access to memory, while others in the same work-group(blocks or grids) will wait until these elements have finished.

The merge algorithm will merge two sorted array and will create a bigger partition, to sort until the end we need to iterate for every remaining level of the merging tree, until we reach the root.

The host code to merge until there is only one sorted array is below:

```
int turn=0,pass=1;
double total_time_merge=0;
int current_merge_size = lws1;
merge_evt1=sort_evt;
(current_merge_size<nels){
    if(turn ==0){
        turn = 1;
        merge_evt2 = sortparallelmerge(sort_merge_k, lws2, que, d_Sort1,
                                        d_Sort2, nels, merge_evt1, current_merge_size);
        clWaitForEvents(1, &merge_evt2);

        const double runtime_merge_ms = runtime_ms(merge_evt2);
        total_time_merge += runtime_merge_ms;
        const double merge_bw_gbs = memsize*log2(nels)/1.0e6/runtime_merge_ms;
        printf("merge_parziale_lws%i destinazione Sort2: %d int in %gms: %g GB/s %g GE/s\n",
          current_merge_size,nels, runtime_merge_ms, merge_bw_gbs, (nels)/1.0e6/runtime_merge_ms);

          }
    else{
      turn = 0;
        merge_evt1 = sortparallelmerge(sort_merge_k, lws2, que,
                                        d_Sort2,d_Sort1, nels, merge_evt2,current_merge_size);
        clWaitForEvents(1, &merge_evt1);
        const double runtime_merge_ms = runtime_ms(merge_evt1);
        total_time_merge += runtime_merge_ms;
        const double merge_bw_gbs = memsize*log2(nels)/1.0e6/runtime_merge_ms;
        printf("merge_parziale_lws%i destinazione Sort1: %d int in %gms: %g GB/s %g GE/s\n",
                current_merge_size,nels, runtime_merge_ms, merge_bw_gbs,
(nels)/1.0e6/runtime_merge_ms);

        }
        current_merge_size<<=1;
        pass++;
}
```

the function that will call the kernel from host and enqueue the execution to the command queue is below, small variation to the HALF parallel version, where the number of workitem will be halved (and rounded to the preferred size).

```
cl_event sortparallelmerge(cl_kernel sortinit_k,cl_int _lws, cl_command_queue que,
    cl_mem d_v1,cl_mem d_vout, cl_int nels, cl_event init_event,cl_int current_merge_size)
{
    const size_t workitem=nels;
    const size_t gws[] = { round_mul_up(workitem, gws_align_init) };
    const size_t lws[] = { _lws };
    printf("init gws e workitem : %d | %zu = %zu  %li\n", nels, gws_align_init,
gws[0],workitem);
    cl_event sortinit_evt;
    cl_int err;

    cl_uint i = 0;
    err = clSetKernelArg(sortinit_k, i++, sizeof(d_vout), &d_vout);
    ocl_check(err, "set mergeinit arg1", i-1);
    err = clSetKernelArg(sortinit_k, i++, sizeof(d_v1), &d_v1);
    ocl_check(err, "set mergeinit arg3", i-1);
    err = clSetKernelArg(sortinit_k, i++, sizeof(nels), &nels);
    ocl_check(err, "set mergeeinit arg2", i-1);
    err = clSetKernelArg(sortinit_k, i++, sizeof(current_merge_size),&current_merge_size);
    ocl_check(err, "set mergeinit arg4",i-1);

    err = clEnqueueNDRangeKernel(que, sortinit_k, 1,
        NULL, gws, lws,
        1, &init_event, &sortinit_evt);

    ocl_check(err, "enqueue sortinitparallelmerge");

    return sortinit_evt;
}
```

The major and most expensive computation in these algorithms are obviously the part of the code where there is access to memory and where there are conditional branches.

In the next chapter I will implement two modifications of the algorithm presented in this chapter.

3.1 Merging large partitions with local memory cache

A strategy to **evade**(in theory) the memory transfer burden involves the local memory.

I thought of two types of use of local memory, one is pretty simple, the other involve relative complex logic and takes inspiration from the standard cache **memory associativity** to place elements.

The first algorithm involves a modification of the function to find the index in the second array, now a part of global memory is stored in local memory and PE of the same work-group can address directly to local memory to search for the index(if local memory is not enough, the access is to global memory).

To store a part of the global array in local memory, we need the start and the end of the partition to copy, so I first search for the first index(the first PE in a work-group will search for this) and the last index(last element in a work-group).

When I have the two index, Every PE will load in local memory trying to keep the memory coalescing(consecutive PE will load consecutive elements from global memory).

I present the full parallel version of the algorithm and the binary_index variation below:

```
inline int binaryIndex_local_blocks(int pivot,global int* find1, int size,local int* lmem, int localsize){
    int l=0 ,m, r=size-1;
    while (l<=r ){
        m = l + ((r - l)>> 1);
        int confront1 =(m<localsize ? lmem[m] : find1[m]);
        if ((confront1 < pivot) )
          l = m +1;
        else
          r = m -1;
    }
    return  l;
}
```

```
kernel void mergebinaryWithRepParallelV3local(global int * out ,global int* arr,int nels, int subsize,
local int* lmem,int local_subsize ){
	const int gi = get_global_id(0);
	if(gi>=nels)return;
	const int lws = get_local_size(0);
	const int subsetid = gi/subsize;
	const int li = get_local_id(0);
	int start,end,index;
	local int first,last;
	char condition= subsetid &1;
	const int pivot = arr[gi];
	if((condition)){
		start = (subsetid - 1) * subsize;
	} else{
		start = (subsetid + 1) * subsize;
	}
	end = (start + subsize) < nels ? start + subsize : nels;
	int size = ((start + subsize) < nels ? subsize : nels - start);
	int locindex=0;
	//finding first element index and last element index
	if(li==0){
		first= binaryIndexFullPower_corr(condition,pivot, arr+start, size);

	}
	barrier(CLK_LOCAL_MEM_FENCE);
	if(li==get_local_size(0)-1){
		last =binaryIndexFullPower_corr(condition,pivot, arr+start+first, size - first);

	}
	barrier(CLK_LOCAL_MEM_FENCE);
	//loading elements from global memory to local memory
	for(unsigned stride=0;stride <local_subsize && stride<=last;stride+=lws){
		int locplace=li+stride;
		if(locplace<last)lmem[locplace] = arr[start+first+li+stride];
	}
	barrier(CLK_LOCAL_MEM_FENCE);
	if(start<nels) locindex = binaryIndex_local_blocks_corr(condition,pivot,arr + start + first ,
last,lmem,local_subsize);
	index=(gi-(condition ? subsize : 0))+locindex + first;
	out[index] = pivot;

}
```

The second alteration of the merge algorithm involves **HASHING** into local memory using the index from global memory, the new index in local memory is the index from global memory multiplied by the size of the local memory and divided by the size of the partition.
This algorithm associate groups of elements into a single place in local memory, the elements are evenly distributed.
If a place is free, the element in global memory is loaded in local memory at the place and mapped.
If a place is occupied and the address is the same as the one searched, the element is read from local memory.
If the element is not the one searched, the PE will load from global memory and write to local memory.

Code below:

```
inline int binaryIndex_local_HASH_corr(char condition,int pivot,global int* find1, int size,local int2* lmem, int localsize){
    if(size <= 0)return 0;
    if(size ==1 ){
        const int conf=find1[0];
        return (pivot>conf || (condition && pivot==conf));
    }
    int l=0 ,m, r=size-1;
    float normalizer = (localsize-1) / (float)(size-1);
    while (l<=r){
        m = l + ((r - l)>> 1);
        int confront1 ;
        const int localplace =  normalizer * m;
        int2 comp1 = lmem[localplace];
        if(comp1.y != m){
            comp1  = (int2)( find1[m],m);
        }
        confront1=comp1.x;
        lmem[localplace]=comp1;
        if ((confront1 < pivot) || (condition && confront1==pivot) )
          l = m +1;
        else
          r = m -1;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    return  l;
}
```

```
kernel void mergebinaryWithRepParallelV4local(global int * out ,global int* arr,int nels, int subsize,
local int2* lmem,int local_subsize ){
    const int gi = get_global_id(0);
    const int lws = get_local_size(0);
    const int subsetid = gi/subsize;
    const int li = get_local_id(0);
    int start,end,index;
    local int first,last;
    char condition= subsetid &1;
    int pivot=INT_MAX;
    if(gi<nels)
    pivot = arr[gi];
    if((condition)){
        start = (subsetid - 1) * subsize;
    } else{
        start = (subsetid + 1) * subsize;
    }
    end = (start + subsize) < nels ? start + subsize : nels;
    int size = ((start + subsize) < nels ? subsize : nels - start);
    int locindex=0;
    //search for first and last index
    if(li==0){
        first= binaryIndexFullPower(pivot, arr+start, size);

    }

    barrier(CLK_LOCAL_MEM_FENCE);
    if(li==get_local_size(0)-1){
        last =binaryIndexFullPower_corr(condition,pivot, arr+start+first, size - first);

    }
    barrier(CLK_LOCAL_MEM_FENCE);
      //in the first element, the mapped index should not be 0
    if(li==0)lmem[0] = (int2)(0,-1);
    barrier(CLK_LOCAL_MEM_FENCE);
    if(start<nels) locindex = binaryIndex_local_HASH_corr(condition,pivot,arr + start + first ,
last,lmem,local_subsize);
    index=(gi-(condition ? subsize : 0))+locindex + first;
    if(gi<nels)
    out[index] = pivot;

}
```

These two versions were constructed to amend the global memory transfer burden, but neither of these versions seem to improve the performance, rather they aggravates the access to global memory(even when there is memory coalescence), because the time to load elements in local memory is more than accessing directly in global memory.
Even with the HASH algorithm to use the local memory as a normal cache, the logic becomes complicated, the speed drops drastically (I thought that this loss in speed was caused by the search of the end and start index, but I was wrong, the runtime is worse without them, and it fluctuates).

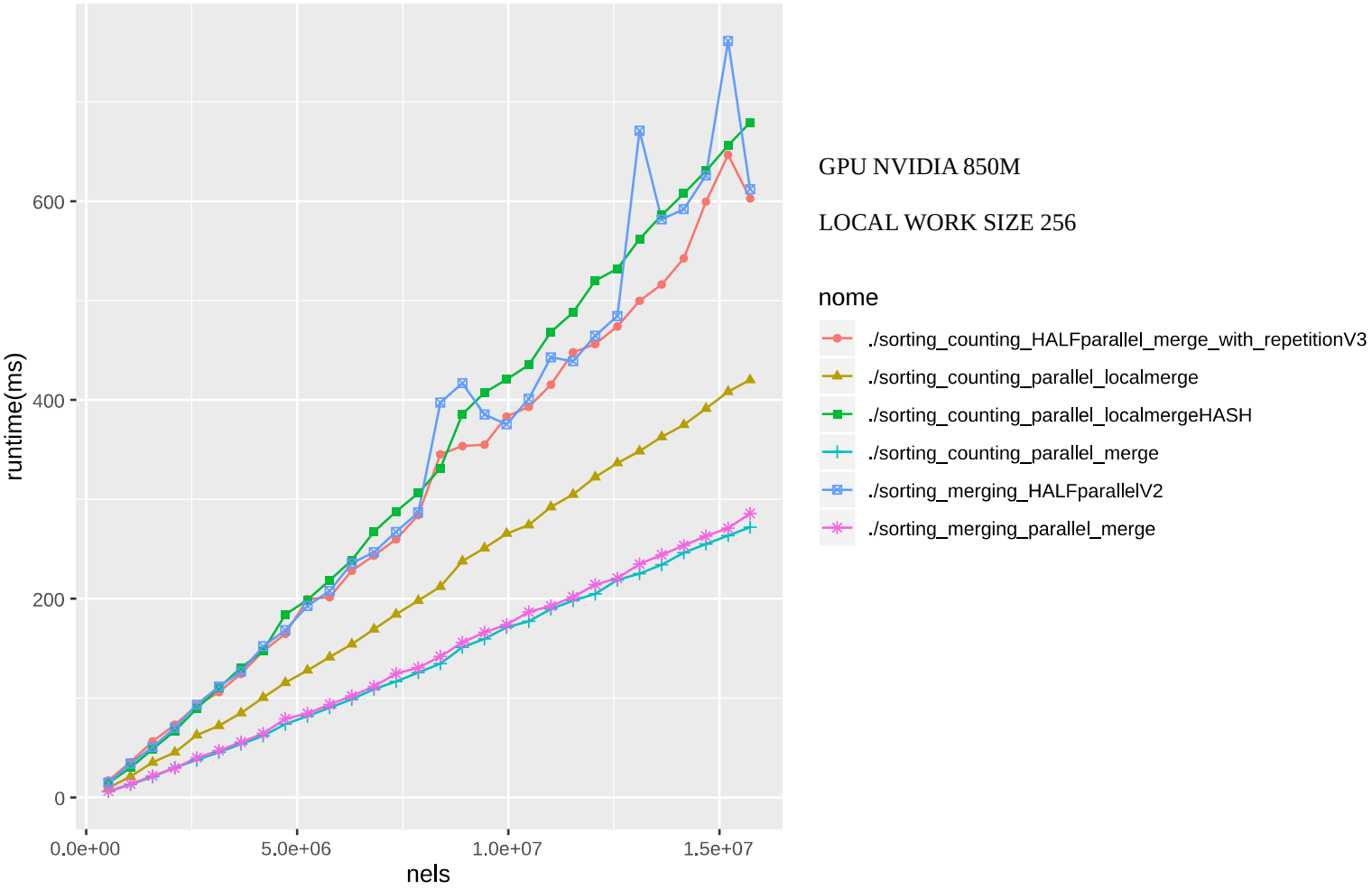
CONCLUSION

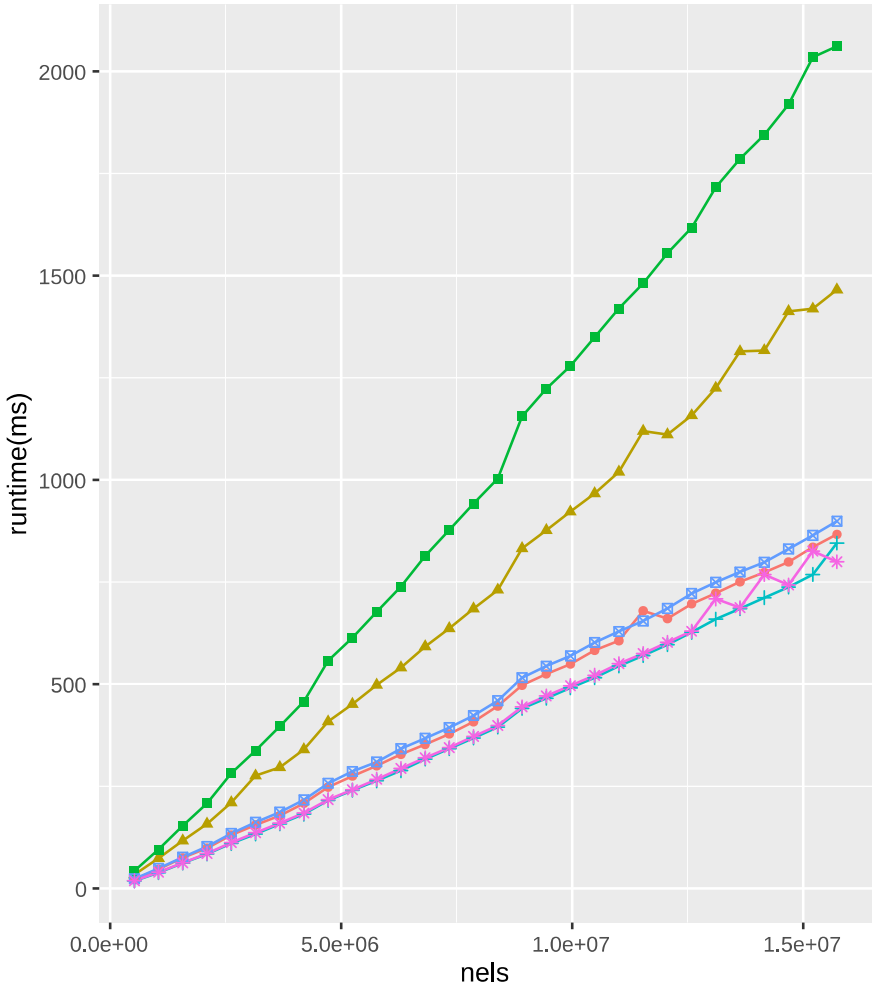
In the end, there are other optimizations, other algorithms and other implementations, but for most situations,it will resolve to waiting for two large array to merge, so we can shift one more time the blame to the CPU, when the time comes.
Also, in my github there are sources where I adopted an Hybrid CPU-GPU approach where part of the computation (merging primarily) was devoted to the CPU.

However, for large enough sizes, the GPU would always outrun the CPU.

I have tried to implement other types of algorithms, but,for most cases, the results were disappointing(other algorithms were slow, sliding versions of the local algorithms presented in the 2nd chapter were really slow, for some algorithms, vectorizing or using local memory were a good choice, but the gain in performance were very little, and most of the times the performance declined drastically).
To see all the results run the script sorting.sh.

The benchmark of the various versions(the last version for the most outstanding algorithms ,the assembly line is not considered) on discrete GPU(850M,930XM , 1660 SUPER) and integrated GPU(5500U) and a box plot of bandwidth(not reliable) are below.



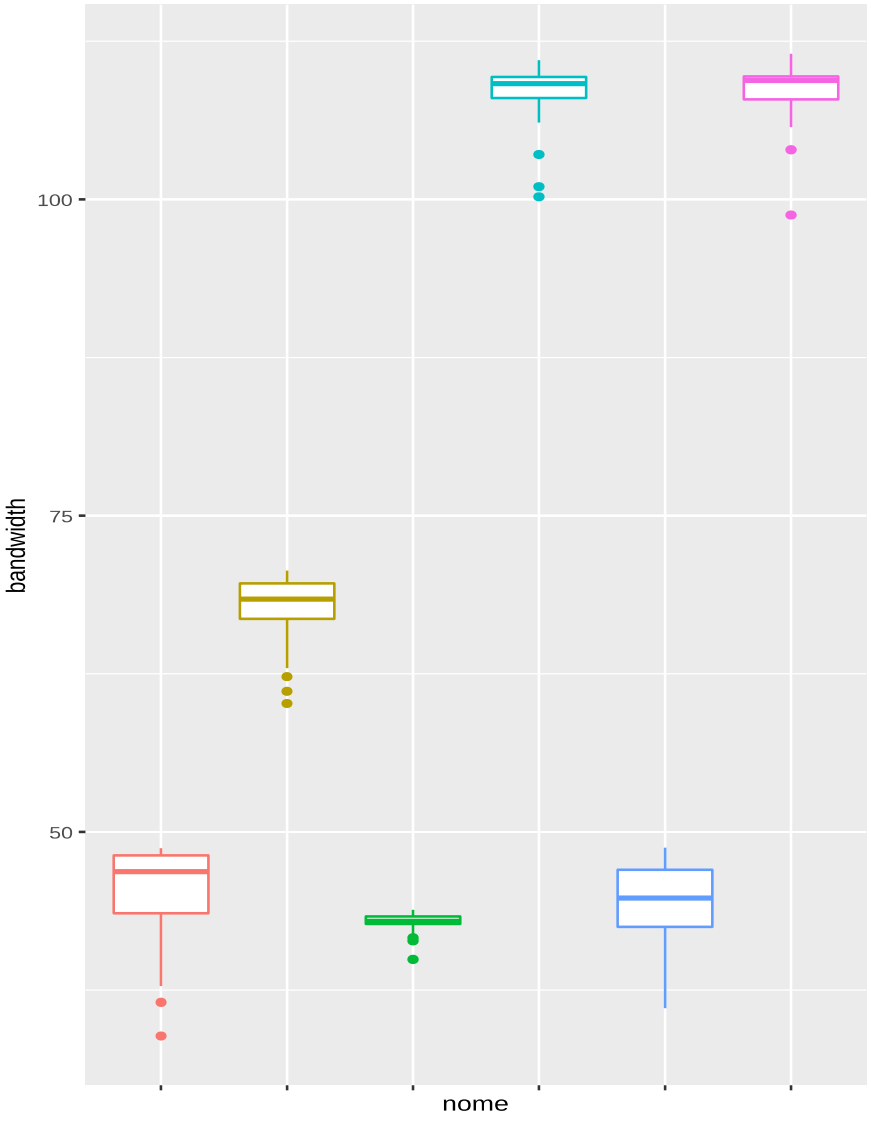


GPU INTEGRATA 5500U

LOCAL WORK SIZE 256

nome

- ./sorting_counting_HALFparallel_merge_with_repetitionV3
- ./sorting_counting_parallel_localmerge
- ./sorting_counting_parallel_localmergeHASH
- ./sorting_counting_parallel_merge
- ./sorting_merging_HALFparallelV2
- ./sorting_merging_parallel_merge

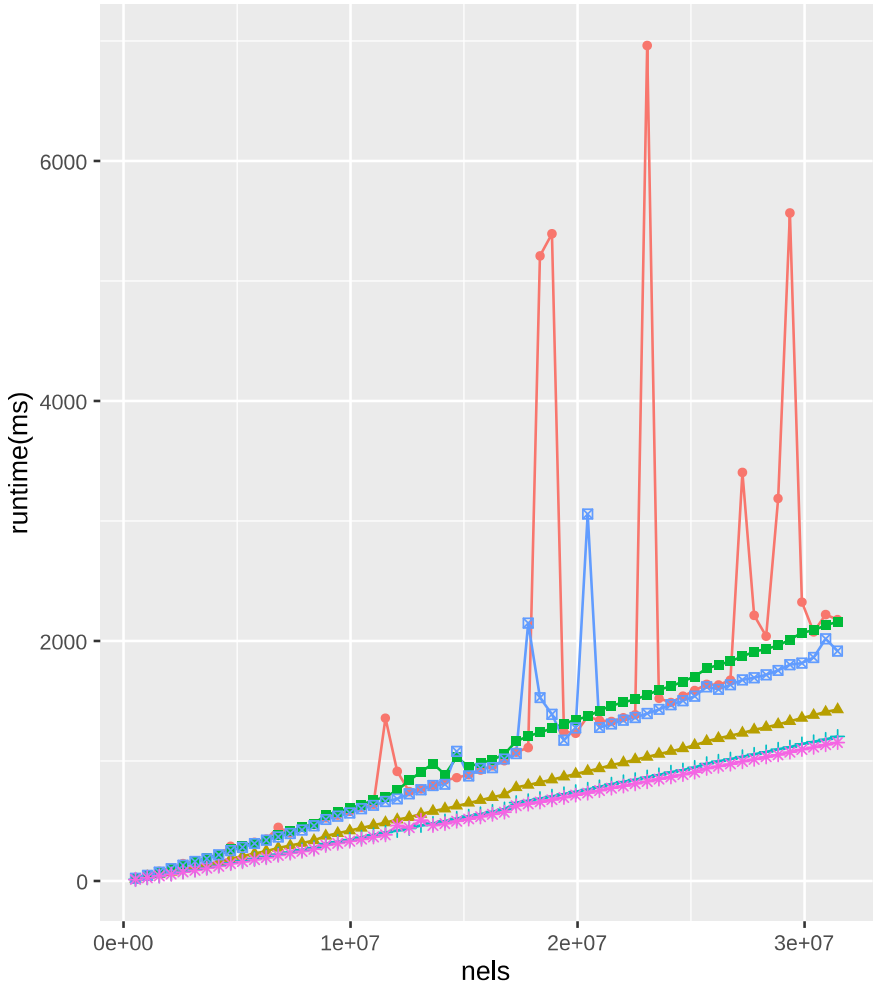


GPU NVIDIA 850M

LOCAL WORK SIZE 256

nome

- ./sorting_counting_HALFparallel_merge_with_repetitionV3
- ./sorting_counting_parallel_localmerge
- ./sorting_counting_parallel_localmergeHASH
- ./sorting_counting_parallel_merge
- ./sorting_merging_HALFparallelV2
- ./sorting_merging_parallel_merge

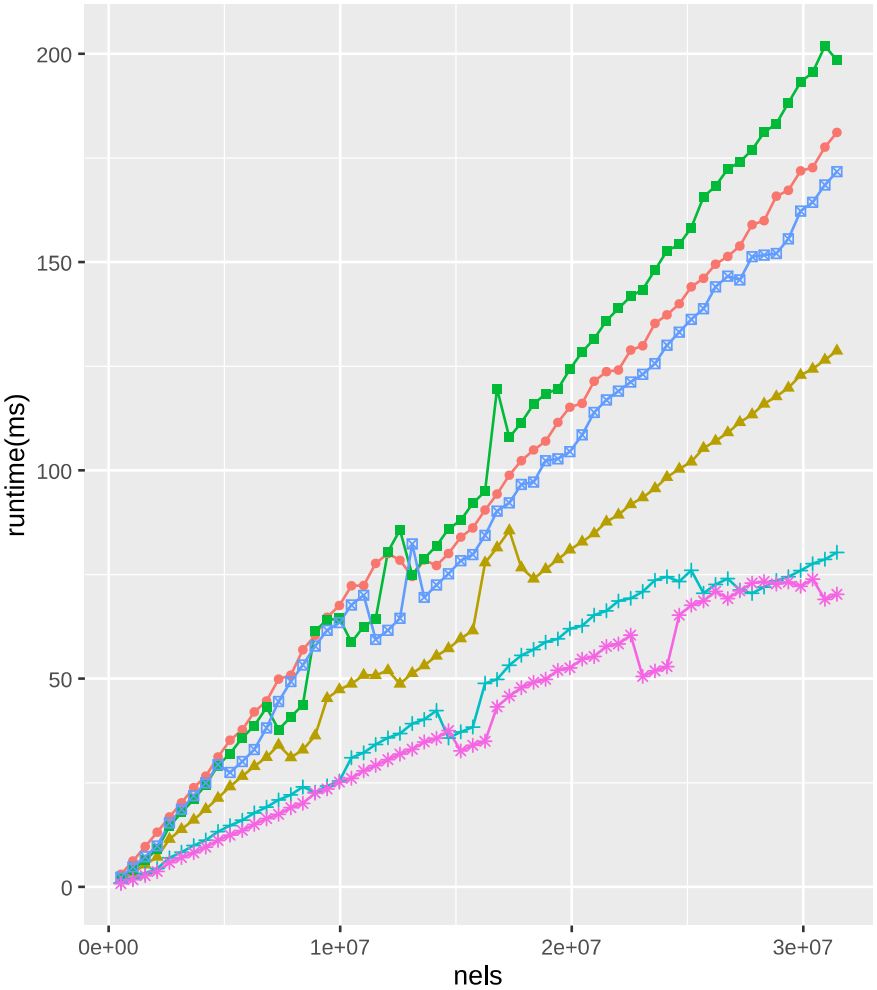


GPU NVIDIA 930MX

LOCAL WORK SIZE 256

nome

- ./sorting_counting_HALFparallel_merge_with_repetitionV3
- ./sorting_counting_parallel_localmerge
- ./sorting_counting_parallel_localmergeHASH
- ./sorting_counting_parallel_merge
- ./sorting_merging_HALFparallelV2
- ./sorting_merging_parallel_merge



GPU NVIDIA 1660 SUPER

LOCAL WORK SIZE=256

nome

- ./sorting_counting_HALFparallel_merge_with_repetitionV3
- ./sorting_counting_parallel_localmerge
- ./sorting_counting_parallel_localmergeHASH
- ./sorting_counting_parallel_merge
- ./sorting_merging_HALFparallelV2
- ./sorting_merging_parallel_merge