# Parallel Block-InsertionSort[⋆]

José Vásquez

Universidad Austral de Chile, Chile

**Abstract.** In this work, we present parallel implementations of Block-InsertionSort using OpenMP and CUDA for CPU and GPU architectures, respectively. We evaluate the performance of our implementations on different input sizes and data distributions. We also compared our implementations against various parallel versions of classical sorting algorithms and state-of-the-art parallel sorting algorithms with available implementations. Our OpenMP parallelized version[1] achieved a significant speedup compared to the sequential version, but our best results were obtained when combining BiS with multiway merge routines, with close to 17x average speedup on 32 cores. On the GPU side, we explore the foundaments of manycore architectures and the GPU CUDA programming model, and develop a *naive* implementation of CUDA-BiS that could serve as baseline for future work. Our evaluation includes performance analysis and benchmarking, highlighting the strengths and weaknesses of each implementation. Moreover, we discuss the potential and limitations of our approach and provide insights for future research in this area.

**Keywords:** Sorting algorithm, InsertionSort, Block-InsertionSort, OpenMP, CUDA

## 1 Introduction

Sorting is frequently described as one of the most fundamental problems in the study of algorithms [19,33]. Sometimes, an application inherently needs to sort information. For example, the contact list in our phones must be sorted alphabetically, and multimedia files may be sorted by date, name, type, or another criterion. Furthermore, sorting is a crucial step in many other big problems; such as searching, information retrieval, particle simulation, data compression, data extraction and so on [19, 20, 33, 38]. This makes sorting an important building block for many other algorithms. Given the importance of this problem, a huge amount of research has been done, and several algorithms and implementations have arisen, giving us many options to choose from. However, despite all this effort, it is hard to say that the problem is fully solved. Technologies are constantly evolving, and so are the requirements in terms of the data that needs to be processed, reason why algorithms are constantly being challenged.

In today's Big Data era, data sources are ubiquitous, and the data itself is a valuable resource with many usages. Two clear instances that may deal with a lot of data are database management and machine learning. Sorting in databases is becoming increasingly important for efficient data management. Sorting can be used to organize and query data, making it easier to retrieve information and perform analytics. In the case of machine learning, current models are being trained with vast amounts of data, where sorting can be an essential pre-processing step for. For example, sorting can be used to organize and clean datasets, remove duplicates, and prepare data for feature extraction. With the increasing availability of large datasets, sorting has become critical to ensure the quality and accuracy of machine learning models.

Efficient use of available resources and reducing the execution time of any computer routine are goals worth striving for, regardless of the scale of the task or the availability of supercomputers. One approach to improve the execution time of programs is through **paralellism**, which is a run-time property where two or more tasks can run simultaneously [33]. Parallel computers, equipped with multiple processing units, have become increasingly common across a wide range of prices and performance levels. Even relatively inexpensive desktop and laptop chip multiprocessors house multiple processing ”cores”, each of which can access a common memory [19]. In addition, highly parallel graphics processing units (GPUs) are now being adopted for a wider range of computations beyond graphics-related tasks [34]. As a result, the design, analysis, and implementation of parallel algorithms, including sorting, have become increasingly relevant.

[⋆] Partially supported by ANID, Fondecyt Iniciación grant N°11221029, Chile.

[1] Implementations available in our GitHub repository

Although sorting is a conceptually simple problem—generate an ordered permutation of the input—it is challenging to find an algorithm that performs optimally in all practical scenarios. The choice of sorting algorithm can depend on various factors, such as available hardware, data type, and input distribution [19]. When dealing with large amounts of data, sorting becomes a critical operation and the execution time could vary to a great extent depending on the algorithm chosen, but it is not always possible to know all the constraints beforehand and make the optimal choice.

Sorting algorithms can be classified based on various characteristics. Some algorithms are *in-place*, meaning they only need constant additional memory to operate, while others require additional memory proportional to the input size. Some are *stable*, which means items with the same value retain their relative order, whereas others are not. There are also algorithms that sort by *directly comparing input elements* and those that use alternative approaches. When considering the factors previously mentioned, such as hardware and input distribution, some algorithms may perform better on certain types of inputs but poorly on others, or they may be limited to specific input types. For instance, there are some integer sorting algorithms which are not based on comparison, and these can provide the best performance due to their lower asymptotic bound and parallelization capabilities, but may require additional preprocessing of the data or be inapplicable in some cases [12, 27].

Given the factors that influence the final performance of sorting algorithms, the search for an efficient general-purpose sorting algorithm is of great interest. Standard libraries in popular programming languages such as Python, C++, and Rust use hybrid algorithms like Introsort and Timsort [4, 7, 8], which provide good general performance. These hybrids combine aspects from well-known sorting algorithms such as InsertionSort, MergeSort, and QuickSort. However, QuickSort on its own, despite having non-optimal worst-case performance, is a strong candidate for a general-purpose sorting algorithm, as it has many good qualities [12, 19]. With a robust pivot-selection strategy, it requires only $O(n \log n)$ expected work independent of the input, can be parallelized, and works in place [12].

Recently, a comparison-based sorting algorithm called **Block-InsertionSort** has been proposed and shown to outperform traditional methods such as QuickSort, MergeSort, and HeapSort, as well as, for certain cases, the efficient hybrid Introsort algorithm (std::sort() method provided by the GNU C++ Standard Library) [23]. This research aims to explore the parallel potential and achievable speedup of the Block-InsertionSort algorithm in multicore and manycore architectures. Additionally, parallel versions of the algorithm will be designed and implemented for both CPU and GPU using OpenMP and CUDA, respectively. These efforts will complement the existing algorithm and make Block-InsertionSort an attractive choice for a high-performance, portable general-purpose sorting algorithm.

## 1.1 Research question

Given the outstanding performance of Block-InsertionSort in a sequential setup, even when compared to the efficient implementations of the C++ STL, the following research questions arise:

1. **Is it possible to develop a parallel implementation of Block-InsertionSort that achieves significant speedup compared to the original sequential implementation?**
2. **Can our parallel implementation be competitive against other parallel implementations, including specialized libraries and state-of-the-art approaches?**

## 1.2 Roadmap

The outline of this work is the following: In section 2, we present relevant concepts and definitions related to the sorting problem and parallelism, including necessary aspects of the OpenMP API and a brief description of the CUDA programming model. In section 3 we review recent work on the field of parallel sorting algorithms. In section 4 we describe the original sequential Block-InsertionSort algorithm, analyze its execution time and potential parallel paths. In section 5 we present Parallel Block-InsertionSort. In section 6 we describe our experimental framework, evaluate different aspects of our parallel implementations and compare them against state of the art sorting implementations. We also show our *initial* results regarding our GPU work. Finally, we comment about our findings and potential improvements in the Conclusions sections.

## 2 Preliminaries

**Notation.** We use $A[1 \ldots n]$ to denote an input array with $n$ elements. We use $t$ to denote the number of threads, and we may use this term interchangeably with $p$ processors to refer to a independent processing unit.

We define the **sorting task** as the rearrangement of an input sequence, usually an n-element array, composed by elements from an ordered set.
**Input:** A sequence of $n$ elements $\langle a_1, a_2, ..., a_n \rangle$
**Output** A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

We consider both main classes of sorting algorithms: **comparison-based** and **non comparison-based.** For the first class, the sorted order they determine is based only on comparisons between the input elements. Any comparison sort must make $O(n \log n)$ comparisons in the worst case to sort $n$ elements [19]. In contrast, rather than relying on comparisons to sort elements, non comparison-based sorting algorithms employ extra information about keys and procedures apart from comparisons to determine the sorted order. [19, 33], consequently, $O(n \log n)$ lower bound does not apply to them. Comparison-based sorting algorithms, while losing in terms of asymptotic bound, have the advantage in terms of versatility and may be the only choice when a custom comparison function is needed [27] or if all we know about the keys is that they are from an ordered set [33].

**Parallell Compute Models.** Many theoretical models have been devised in attempt to capture the key features of a large class of parallel architectures, but it is hard to say that there is one model that is universal enough [33]. One that is widely used is the *parallel random access machine* (PRAM). A PRAM computer consists of processors that have uniform access to a large shared memory. These processors share a common clock but may execute different instructions in each cycle. Therefore, a PRAM computer can be classified as a synchronous, Multiple Instruction Multiple Data(MIMD), Uniform Memory Access (UMA) computer [33]. There are four versions of the PRAM model that describe how concurrent memory accesses are handled [17, 33]. Additionally, there exist other models and variations of the PRAM model that aim to consider factors such as memory hierarchy, I/O operations, and communication overhead.

The **Work-Span** model is another model we can use in order to analyze algorithms. In this model, the work is the number of operations used by the algorithm and the span is the length of the longest sequential dependence in the computation [19]. We say a parallel algorithm is work-optimal if the total of operations performed considering all processing units is equal to the number of operations of an optimal sequential algorithm.

We denote the running time of a parrallel algorithm as $T_P$, where $P$ is the number of processors. We use $T_1$ for the running time of the sequential algorithm ($P = 1$). We define the **speedup** of a computation on $P$ processors by the ratio $T_1/T_P$ which says how many times faster the computation is on $P$ processors than the sequential computation. When $T_1/T_p = \Theta(P)$, we say that the computation exhibits linear speedup, and when $T_1/T_p = P$, we have perfect linear speedup [19]. From a practical point of view, parallel algorithms that achieve some limited speedup at the cost of a great increase of total work should be avoided [40].

**Static and Dynamic Parallelism.** One important class of concurrency platforms is dynamic parallelism/multithreading, which is complemented by static multithreading as its counterpart [19]. In Static parallelism, the number of iterations assigned to each thread is determined at compile time and remains the same throughout the execution of the program. This approach may work well when only a single level of parallelism is needed or when the number of iterations for each thread is relatively equal [19, 34]. In contrast, dynamic multithreading enables the specification of parallelism in applications without worrying about communication protocols, low-level load balancing, and other complexities of static-thread programming [19]. Another good use case for dynamic parallelism is nested parallelism, which allows a subroutine to be spawned, enabling the caller to proceed while the spawned subroutine is computing its result [19, 34].

**CPU vs GPU.** The central processing unit (CPU) is optimized for executing a sequence of operations or sub-processes known as **threads**. CPUs exhibit high performance per thread but typically support only a few dozen threads simultaneously. On the other hand, the graphics processing unit (GPU) is specifically designed to handle thousands of threads in parallel, thus amortizing lower performance at the individual thread level [34]. With massive parallel capabilites in mind, GPUs are designed with more transistors in charge of processing data/computation than tasks related to caching and flow control (Fig. 1) [31,34,39].

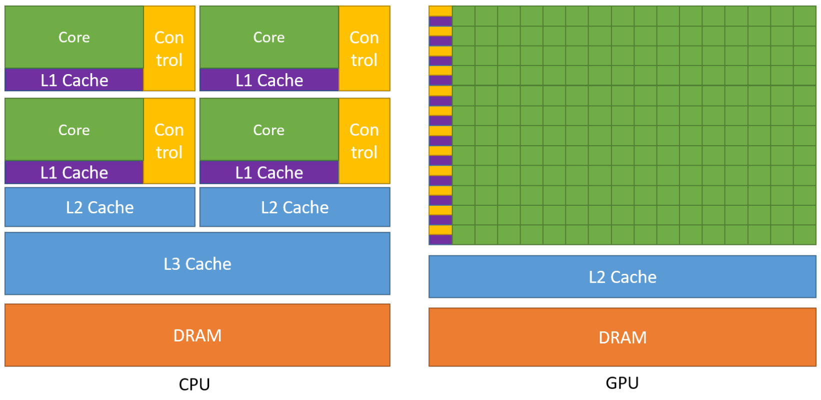


**Fig. 1.** The GPU Devotes More Transistors to Data Processing [34]

**OpenMP** is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs [3]. OpenMP is a relatively small and simple specification, supports several compilers from various vendors, and supports incremental parallelism [3]. An OpenMP program can be described as a sequential program added with compiler directives to set out parallelism [10].

An OpenMP executable directive and the associated statement, loop or structured block, if any (not including the code in any called routines) is known as a *construct* [28]. The set of directives provided by OpenMP extend supported programming languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs [28]. These directives also provide support for sharing, mapping and privatizing data [28]. OpenMP fully supports loop-level parallelism, nested parallelism with parallel constructs inside parallel regions, and task parallelism [3].

OpenMP directives for C/C++ are specified with *#pragma* directives. Each directive starts with *#pragma omp* followed by optional clauses. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives [28]. For instance, the construct *#pragma omp parallel num_threads(4)* specifies a parallel region with four threads. To utilize the OpenMP constructs in a C/C++ program, it is necessary to include the *omp.h* header file, which provides necessary function prototypes and types. These constructs generally apply to a structured block, which refers to a block of statements containing an entry point at the top and an exit point at the bottom. This structured block enables the proper execution and coordination of parallel tasks [28].

**CUDA** is an application programming interface (API) that allows parallel programs to run cooperatively between the CPU (host) and GPU (device), via what is known as a **kernel**, or a section of code compiled for the express purpose of running on a coprocessor like the GPU [11]. At the core of CUDA are three fundamental abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions are exposed to the programmer through a minimal set of language extensions [34]. Compared to alternative APIs such as OpenCL, CUDA is better for getting the most performance with NVIDIA GPU cards and a homogeneous device environment [11].

The CUDA execution model is based on primitives of **threads**, **thread blocks**, and **grids**, with kernel functions defining the program executed by individual threads within a thread block and grid [34]. Each thread has local private memory. Each block has shared memory visible to its own threads. All threads have access to the same global memory [34]. GPUs are built around an array of Streaming Multiprocessors (SMs), which create, manage, schedule, and execute threads in groups of 32 parallel threads called warps [34]. A multithreaded program is divided into independent blocks of threads, enabling GPUs with more multiprocessors to execute the program faster than GPUs with fewer multiprocessors [34].

In the CUDA execution model, performance optimization revolves around **maximizing parallel execution**, **maximizing memory and instruction throughput**, and **minimizing memory thrashing** [34]. Since GPUs have high latency global memory and very limited caches, memory access patterns become an important issue [31]. When accessing global memory, the hardware can fuse memory accesses if the threads access consecutive memory locations, resulting in **coalesced memory accesses** [34]. Similarly, in the case of block-wide shared memory, it is divided into equally-sized memory modules known as banks, enabling simultaneous access [34]. Hence, any memory read or write request consisting of $n$ addresses that span $n$ distinct memory banks can be serviced simultaneously. However, when two addresses of a memory request fall within the same memory bank, a **bank conflict** occurs, and the access must be serialized [34]. It is widely recognized that uncoalesced access to global memory and shared memory bank conflicts can significantly impact the overall performance of GPU implementations, sorting included [27,31,39].

**CUDA Dynamic Parallelism** is an extension to the CUDA programming model that enables a CUDA kernel to create and synchronize new work directly on the GPU. This extension provides improved support for recursion and irregular loop structures [34]. Previously, it was possible to use *cudaDeviceSynchronize* –a function that blocks until the device has completed all preceding requested tasks– as a synchronization mechanism in both host and device code, but the use of cudaDeviceSynchronize in device code was deprecated in CUDA 11.6 and removed for compute_90+ compilation [34]. As an alternative, the CUDA programming guide suggests utilizing a *tail launch stream* to achieve the same functionality. A **stream** in CUDA represents a sequence of commands, which can be issued by different host threads and execute in order. The tail launch stream, named *cudaStreamTailLaunch*, enables a grid to schedule a new grid for launch after its completion [34]. Each grid in CUDA has its own tail launch stream, and any non-tail launch work initiated by a grid is implicitly synchronized before the tail stream is triggered [34].

## 3 Related work

Many parallel sorting algorithms for multicore and manycore have been proposed in the literature. The existing comparison-based multicore algorithms take two main approaches [18]. The first one is the **merge** principle, a process that converts two ordered sequences of data into a single, larger ordered sequence [17]. In order to sort an array, **MergeSort** [17,21,24,25,27,37] divides it into two halves, sorts the two halves (recursively), and then merges the results [38]. The second approach is **sampling** [12,18,29,30]. This approach splits the input into subsets, sorts the subsets, samples the sorted subsets, sorts the sample, partitions about a subsample, and recursively sorts the resulting sets [18].

**Partitioning**, as a means to obtain disjoint segments of a sequence that can be processed in parallel by different threads, is an important operation for sorting algorithms [40]. In the case of sampling-based approaches that need to split an unsorted sequence, elements are distributed to other regions based on a pivot value [19,31] or a sample from the input [12,18]. Partitioning schemes also play a crucial role in parallel merge operations, allowing each thread to work on specific regions of the sorted subsequences that need to be merged. However, since these sequences are already sorted, strategies like binary search [19] and prefix sum/global rank are employed [18,31,40] to obtain the splitter indexes or output locations where each thread will work on.

Merge sort, due to its inherent divide-and-conquer paradigm and numerous independent subproblems at certain execution points, presents itself as a promising candidate for parallelization [19]. For a pair-wise merge sorter, the smaller subproblems can be done in parallel, resulting in sorted segments that increase in size by a factor of two at each level, with a total $\log_2 n$ merge rounds or levels [27]. For the last few levels, where the number of processors can be greater than the number of subproblems, the merge itself can be done in parallel as it was described earlier.Merging typically involves using additional memory to construct the sorted sequence in an auxiliary array. In-place merge, where the merging operation is performed directly on the input array, is also possible [12]. However, as discussed in [12], in-place implementations of mergesort tend to exhibit slower performance compared to non-in-place implementations.

K-way merging or **multiway merge** (MWM) routines, as the name suggests, merge more than two sequences into one larger sorted sequence at each step and can be performed in parallel. In contrast to the pairwise algorithm that requires $O(\log_2 n)$ merge rounds, a k-way mergesort only takes $O(\log_k n)$ merge rounds. Achieving multiway merge, or more specifically, to be able to keep track of the next element in each sequence, has been accomplished using strategies such as utilizing a highly tuned tournament tree data structure [40] or a variation of a minHeap structure [27]. According to [12], Parallel Multiway Mergesort algorithm (MCSTLmwm), a non-in-place, stable sorting algorithm from the MCSTL library [40] was identified as the best practical multi-core mergesort algorithm at the time. How MCSTLmwm works is that each thread sequentially sorts $\leq \lceil n/p \rceil$ elements, and the resulting sorted sequences are then merged using a parallel multiway merge algorithm.

**Quicksort** [16, 31, 43, 44] is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively [33]. Quicksort, however, works by selecting a pivot element and partitioning the array such that all elements smaller or equal than the pivot are in the left part and all elements larger than the pivot are in the right part [12]. Quicksort has a worst case of $T(n) = O(n^2)$ that happens when partitions are unbalanced [19], but this worst case can be less frequent with more robust pivot selection strategies such as random sampling or median of $k$ elements [19, 33]. Analog to parallel mergesort, both the partitioning and the recursion can be parallelized [12, 40]. The oneTBB library [36] provides a parallel Quicksort that uses a pseudo median of nine pivot selection strategy. The MCSTL also features parallel implementation of Quicksort, including Load-Balanced Quisccksort that relies on the library's parallel partitioning algorithm [40]. **Sample Sort** could be considered a generalization of the technique used by Quicksort that works with more pivots and splits the input into several parts.

Currently, the best performing general-purpose parallel sorting algorithm seems to be **In-place Parallel Super Scalar SampleSort (IP$S^4$o)** [12]. IP$S^4$o is a recursive algorithm where each recursion level has a partition step that divides the input into $k$ buckets. These buckets are sorted directly with a base case algorithm when they reach small enough size. The partitioning is done in-place, and the multithreading for this step is controlled by their own scheduling algorithm. They combine these techniques with branchless decision trees to avoid branch mispredictions. In an extensive experimental work they show that IP$S^4$o is the fastest algorithm for roughly three fourths of their test cases, even beating integer sorting algorithms in many of them. In [42], the authors propose **vqsort**, a vectorized implementation of quicksort that supports seven instruction sets with close-to native performance. The authors of vqsort integrate it with IP$S^4$o, using their work as a base case for the efficient sample-sorter, achieving a geometric mean speedup of 1.59.

**Radix Sort** [12, 35, 37, 41] is one of the fastest algorithms for sorting integers, it is a non comparison-based algorithm that repeatedly sorts on a constant number of bits of the keys. Most significant digit (MSD) radix sorting starts from the higher order bits, and recursively sorts each set of keys that share the same higher-order bits. Least significant digit (LSD) radix sorting starts from the lower-order bits instead. LSD requires a stable sort to be used as the intermediate sorting algorithm, while MSD does not [19, 35]. In [35] the authors propose **Regions Sort**, a highly parallel MSD, non-stable radix sort. It is considered an in-place algorithm because for sorting $n$ integers from a range $r$, and a parameter $K$, it requires only $O(K \log r \log n)$ auxiliary memory[2] In the same work where IP$S^4$o was proposed, the authors also presented

[2] Some works differentiate between strictly in-place (with constant additional space) and refer to it as in-place when the additional space is logarithmic in the input size.

**In-place Parallel Super Scalar Radix Sort** (IP$S^2$Ra) [12], another MSD radix sort that also uses the in-place partitioning framework they developed for IP$S^4$o. IP$S^2$Ra performs the best when dealing with near-uniform input distributions, small keys, or a sequential setting, covering cases where IP$S^4$o does not offer the top performance.

## 3.1 GPU Sorting

GPU programming is a highly dynamic field that constantly improves in various aspects and presents new challenges. The hardware of GPUs evolves rapidly, with significant differences in architecture between older and more recent generations [39]. In the NVIDIA environment alone, several new GPU architectures have been released, accompanied by regular updates to the CUDA Development Kit, each time supporting greater compute capabilities and providing improvements to existing functionalities [34].

The two approaches mentioned earlier, merging and sampling, have been extensively studied and improved in recent publications related to sorting on manycore architectures. For example, in [37], the authors developed an algorithm for **pairwise parallel merging** of sorted sequences that exposes fine-grained parallelism on their merge sort for GPUs. In [21], the authors propose a GPU merge sort that reduces warp divergence, avoids over-segmenting blocks of data, and increases register utilization instead of shared memory for communication, addressing challenges related to communication overheads in GPU sorting. In a more recent work [27], the authors study the relationship between different factors that affect the performance of GPU sorting and propose **GPU-MMS**, a GPU-efficient multiway merge sort algorithm that uses a improved heap structure they call minBlockHeap.

Both Quicksort and Sample sort have also been studied on the GPU side. In [16], the authors propose **GPU-Quicksort**, an algorithm designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and interthread synchronization needed. Later, in [31], the authors propose **CUDA-quicksort** an iterative GPU-based implementation of the sorting algorithm started from GPU-quicksort, with improvements in the form of optimized memory accesses, use of atomic primitives for inter-block communication and using a different Bitonic sort as a base case. They also present a recursive version using CUDA Dynamic Parallelism, but its results proved inferior to their iterative version.

In the case of Sample sort, [29] proposes **GPU sample sort**, while another deterministic sample sort is later proposed in [22]. The deterministic sample sort is advantageous because its running time does not depend on the input data, eliminating the fluctuations that may occur with randomized sample sort. These works demonstrated improved results compared to existing implementations of GPU sorting algorithms at the time. However, according to [27], one reason why these sample sorts may not be the fastest in practice is the presence of a large number of bank conflicts in shared memory.

Finally, in addition to the above-mentioned techniques, radix sort has also been explored for efficient GPU sorting. In [37], the authors present a radix sort based on efficient parallel scan operations, which can achieve high performance on GPUs. One of the most recent works on radix sort on GPUs is an hybrid MSD radix sort **(HRS)** [41], which can efficiently sort on eight bits with each pass. This radix sort uses low-overhead scheduling mechanisms of the GPU to avoid any load imbalance, by subdividing every bucket into tiny, fixed-size blocks that can be evenly distributed amongst the GPU's Streaming Multiprocessors (SMs). More recently, Onesweep was presented [9], an LSD radix for large GPU sorting problems residing in global memory that requires less global read/write operations. Due to improving other contemporary GPU radix sorting implementations, with even more consistent performance compared to HRS, Onesweep was integrated into the CUB library [5,9].

**Challenges in GPU sorting.** A significant challenge that arises when utilizing the immense parallelism of GPUs is the fact that a key's position within the output sequence depends on all other keys [41]. Additionally, sorting algorithms usually do not have high computation to global memory access ratio [32]. Recent work attempt to deal with these challenges and general performance optimizations through coalesced memory accesses [16,27,31], fine grained parallelism [27,37,41], smart use of shared memory [16,31], and hybrid

CPU-GPU approaches [25,41]. The hybrid CPU-GPU approach is particularly intriguing. Many studies on GPU sorting argue that the CPU-GPU data transfer time can be omitted, as it often forms only a step within a larger process that involves additional modifications to the input performed on the GPU. However, in scenarios where sorting is the sole necessary step or when input datasets are too large and exceed the global memory capacity of the GPU, resulting in communication overhead between the CPU and GPU that may degrade overall performance [25], hybrid CPU-GPU approaches could offer advantages over GPU-only solutions.

### 3.2 Parallel Algorithms in the C++ STL

Starting from the standard specification version C++17 of the C++ programming language, the C++ Standard Library introduced official support for parallel execution in several algorithms. The *algorithm* header now provides additional overloads for routines such as *sort* and *merge*, which received an *execution policy tag* as an argument. Examples of execution policies include *std::execution::par* and *std::execution::par_unseq*. These execution policies are used to specify the type of parallelism allowed in parallel algorithms [8]. The specific parallel algorithm used may vary depending on the implementing library or the compiler used, such as *libstdc++* (GNU Standard C++ Library) or *libc++* (LLVM project's C++ Standard Library). As of now, libstdc++ seems to have better support for parallel algorithms compared to libc++ [1, 2]. At least for libstdc++, compiled programs need to be linked to the TBB [36] library (-ltbb) in order to use parallel execution policies [15].

### 3.3 Sorting in HPC projects

There are active open-source projects that offer parallel primitives and reusable software components for high performance computing in multicore and manycore architectures. NVIDIA mantains two important libraries: CUB and Thrust. **CUB** [5] provides state-of-the-art, reusable software components for every layer of the CUDA programming mode, including block-wide, warp-wide and device-wide highly optimized radix and merge sorting algorithms. **Thrust** [6] is a higher-level library that provides a set of algorithms and containers similar to the standard template library (STL) in C++ while enabling performance portability between GPUs and multicore CPUs. **ParlayLib** [14] is a C++ library for programming parallel algorithms on shared-memory multicore machines that provides additional tools and primitives compared to the C++ STL. All of these projects have released new versions subsequent to March 2023, showcasing the activity in the subject.

## 4 Block-InsertionSort (BiS)

Block InsertionSort is a comparison based sorting algorithm that improves the running time of the well known **InsertionSort** algorithm, which is an efficient algorithm for sorting a small number of elements [19].

InsertionSort applies a very intuitive method of ordering, iteratively it inserts the elements one by one in their correct positions. To do so, it compares the current item -the element to be inserted- with the elements already inserted, displacing each item by one index until finding its correct position. The invariant is that, for the sequence $S[1 \ldots n]$, always there is a subsequence $S[1 \ldots k], k \leq n$, that is already sorted, and it is still pending to insert $S[k+1 \ldots n]$. This simple iterative approach makes InsertionSort a great algorithm for small or nearly sorted sequences [19], but its average and worst case running time of $O(n^2)$ means, as the input size grows, it quickly loses it advantage over asymptotically optimal sorting algorithms.

**Block-InsertionSort** improves upon the classic InsertionSort algorithm by inserting multiple elements in each iteration. While it sacrifices sorting stability and requires extra memory, it offers optimal worst and average case performance. In practice, it has been shown to perform on par with IntroSort. BiS introduces an additional parameter, $k$, which determines the number of sorted blocks to be inserted in each step. This constant value determines the number of block-insertion levels ($t = \log_k n$) required, as the sorted blocks grow in size as powers of $k$. This $k$ parameter is a small power of 2, and according to the author's publication [23], experimental results indicate that a value of $k = 16$ yields the best performance in a sequential setup.

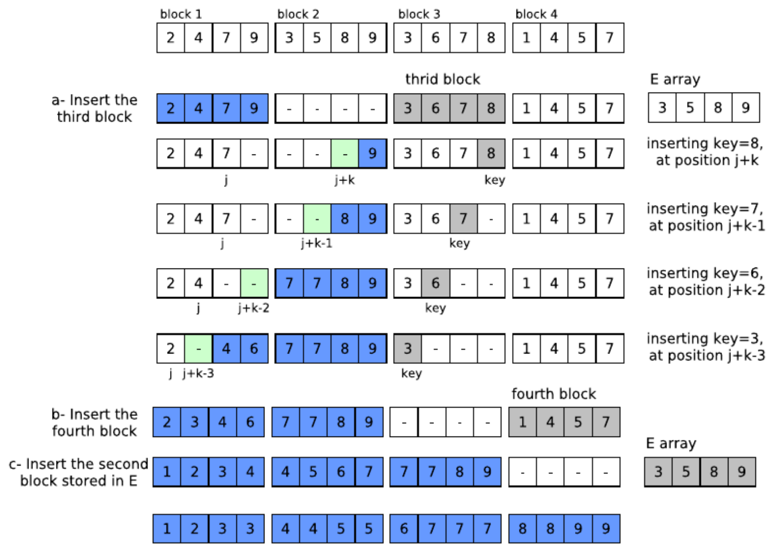

block 1
block 2
block 3
block 4
thrid block
E array
a- Insert the third block
inserting key=8, at position j+k
j
j+k
key
inserting key=7, at position j+k-1
j+k-1
inserting key=6, at position j+k-2
j+k-2
inserting key=3, at position j+k-3
j+k-3
fourth block
b- Insert the fourth block
c- Insert the second block stored in E

**Fig. 2.** To make room for the first block-insertion, the block 2 is copied into auxiliary memory. After that, BiS uses the room left by the previous block to insert the current block with just one traversal [23].

## 4.1 The algorithm

Lets assume that $n$, the input size, is a power of $k$. Block-InsertionSort starts by sorting segments of size $k$ with the classic Insertionsort (base case) obtaining $n/k$ sorted blocks. We can now take $k$ elements already sorted to insert them by performing only one iteration to shift the largest elements to the right, inserting the elements in their correct position. To do this, an additional space to store $k$ elements is required. Therefore, we can insert a fully sorted block with just one traversal of the sorted subarray, instead of inserting its elements one by one independently by running a different traversal for each item as Insertionsort does [23].

Figure 2 display the process of sorting an array $A$ of size $n = 16$ with $k = 4$. After the base case sorting, we want to insert $k$ elements into the sorted segment colored in blue (initially block 1). For this, we start by storing the block 2 into auxiliary memory $E$ and proceed to insert the third block into the sorted segment by comparing the largest elements, storing the result in $A[1 \dots 2k]$. After that, since the third block is already inserted into the sorted segment, we can use its original place in $A[2k+1 \dots 3k]$ to repeat the process and insert the fourth block, obtaining a sorted segment stored that covers $A[1 \dots 3k]$. All that is left is inserting the second block previously stored in $E$ using the same approach to complete the process, effectively sorting the whole $k \cdot k = n$ elements.

## 4.2 Worst case complexity of Block-insertion

In the previous example, the whole input was sorted after just one block-insertion level. For larger inputs, multiple levels of block-insertion may be needed, each level increasing the size of the sorted blocks by a factor of $k$ while also reducing the total number of blocks by the same rate, until completion in the form of a single sorted block of size $n$ is achieved. Table 1 displays the number of blocks, their size and the worst case cost, in terms of the number of comparisons, for each level of the algorithm. The final time complexity of the algorithm is obtained by adding the cost of the $t$ levels:

$$t \cdot O(n) = \log_k n \cdot O(n) = O(n \log n) \tag{1}$$

which is optimal for a comparison-based sorting algorithm. For the space complexity, the max amount of additional memory required by the algorithm comes from storing the second block in the last level of execution, that is, for an input of size power of $k$, storing $n/k$ elements in auxiliary memory $O(n/k) = O(n)$

for any constant $k$. In general, the required amount of cells of memory is equal to the closest power of $k$ that is strictly less than $n$:

$$\min(k^{\lfloor \log_k n \rfloor}, k^{\lceil \log_k n \rceil - 1}) = O(n) \tag{2}$$

| Level | Prev. Length | Cost by Block | # Blocks | Level Cost |
|---|---|---|---|---|
| 1 | 1 | $1 + 2 + \ldots + (k-1) \leq k^2$ | $n/k^1$ | $kn = O(n)$ |
| 2 | $k$ | $k + 2k + \ldots + (k-1)k \leq k^3$ | $n/k^2$ | $kn = O(n)$ |
| ... | ... | ... | ... | ... |
| $t$ | $k^{t-1}$ | $k^{t-1} + 2k^{t-1} + \ldots + (k-1)k^{t-1} \leq k^{t+1}$ | $n/k^t$ | $kn = O(n)$ |

**Table 1.** Breakdown complexity for the Block-insertion algorithm [23]

So far we have assumed that the input size is a power of $k$. For arbitrary input sizes, more often than not we will need to process between 1 and $k-1$ exact blocks (blocks of size power of $k$) and a remaining segment that is smaller than those exact blocks. BiS handles this by finding the biggest $t$ such as $k^t \leq n$ and applying its sorting method for $S[1 \ldots k^t]$. After that, $S[k^t + 1 \ldots n]$ is considered as an independent array that is sorted with the same process to finally insert it into $S[1 \ldots k^t]$.

### 4.3 Block-insertion vs Merge

The insertion strategy used by BiS is very similar in nature to the merge algorithm: we combine two or more sorted sequences into a single, larger sorted sequence. BiS, instead of merging pairs of sequences, combines $k$ sorted sequences of size $n/k^i$ into a sequence of size $n/k^{i-1}$ inserting one at a time. Compared to the classic, non-in-place mergesort algorithm, the additional memory requirements of BiS are slighlty lower since it only needs the size of a block of additional memory ($n/k$ for the last level) to perform the block-insertion procedure and combine $k$ sorted blocks into a single sorted block. The BiS described here is clearly not an in-place algorithm, but by taking advantage of the fact that most of the work is done inside the input array itself, the author also proposed an in-place version of the algorithm that only needs a constant amount of memory. This was achieved by limiting the maximum block size, but this memory usage improvement comes with a greater asymptotic bound and slightly inferior practical performance as trade-off [23].

## 5 Parallel Block-InsertionSort (PBiS)

Block-InsertionSort can be easily expressed as a recursive, top-down algorithm that employs a divide and conquer strategy to sort an input array $A[1..n]$. It divides the array into $k$ segments of size $n' = n/k$ that can be sorted recursively by splitting and block-inserting smaller segments, until reaching a base case where the classic insertionsort handles inputs small enough ($n' \leq k$). By leveraging the dynamic parallelism model, we can describe a parallel version of this algorithm (1), which demonstrates how the problem can be split into independent tasks and identifies where synchronization is needed.

Again, assume that $n$ is a power of $k$, and that the $E$ array has been allocated in the same fashion as in the original algorithm. Similar to other recursive algorithms that operate within a range, we return if $l \geq r$. As for the base case, we utilize the classic Insertionsort to handle segments of size $n = r - l \leq k$ and then return. Otherwise, we spawn $k$ recursive tasks within the for loop of line 10 to sort each block of size $n/k$. Since these blocks are disjoint with respect to the input array $A$, the recursive calls can be executed in parallel. We wait for the completion of the $k$ parallel tasks at the synchronization point of line 16 and then proceed with the same block-insertion process described in section 4. This process involves copying the second block into $E$, inserting consecutive blocks $3 \ldots k$ of size *current_block_size* (insertAllButSecondBlock), and finally inserting the second block that was previously stored in $E$ (insertSecondBlock). Similar to how each parallel task operates on its respective section of the input array $A$ delimited by $l$ and $r$, the *e_start* parameter is utilized to define the region of the auxiliary array $E$ that each task will work on.

**Algorithm 1** Parallel Block-InsertionSort

```
Input: Array $A$, Array $E$, $l, r, e_{start}$. Constant parameter $k$
Output: Sorted array $A$ within indices $l$ and $r$
procedure ParallelRecBiS( $A, E, l, r, e_{start}$ )
    if $l \geq r$ then
        return
    end if
    $n \leftarrow r - l$                                          ▷ for now, suppose that $n = k^t$
    if $n \leq k$ then
        return InsertionSort($A, l, r$)                           ▷ Small segments are sorted with the classic InsertionSort
    end if
    $current\_block\_size \leftarrow n/k$
    for $i \leftarrow 0$ to $k - 1$ step 1 do
        $block\_l \leftarrow l + (i \times current\_block\_size)$
        $block\_r \leftarrow block\_l + current\_block\_size$
        $e\_idx \leftarrow e_{start} + (i \times current\_block\_size/k)$       ▷ Each task works on its own region of the E array
        spawn ParallelRecBiS($A, E, block\_l, block\_r, e_{idx}$)            ▷ Spawn parallel task
    end for
    sync                                                          ▷ Wait until all blocks are sorted
    $second\_block\_start \leftarrow l + current\_block\_size$
    copy $current\_block\_size$ elements starting from $A[second\_block\_start]$ into $E[e_{start}, \ldots]$
    InsertAllButSecondBlock($l, r, current\_block\_size, A$)      ▷ Insert blocks 3..k into the sorted subarray
    InsertSecondBlock($l, r, current\_block\_size, A, E + e_{start}$)   ▷ Insert block 2 previously stored in E
end procedure
```

## 5.1 Auxiliary array E and additional memory usage

We know that BiS requires, for any constant $k$, $O(n/k) = O(n)$ additional memory, which is the maximum size of the block that needs to be stored in the last level of the algorithm. For the same last level of BiS (or the first in our top-down recursive algorithm), we also know that each of the $k$ blocks of size $n/k$ was sorted by inserting $k$ blocks of size $n/k^2$ (Table 1) with each requiring to store their respective second block outside the input array, amounting a total of $k \cdot n/k^2 = n/k$ elements that had to be stored in additional memory. In the same way, the previous level of $k^2$ blocks of size $n/k^3$ had to do the same for a total of $k^2 \cdot n/k^3 = n/k$. So for any level $i > 1$, a total of $k^i \cdot n/k^{i+1} = n/k$ elements were stored in the auxiliary $E$ array. Therefore, by carefully assigning an independent segment of $E$ to each parallel task, we can use the same $E$ array to sort multiple blocks in parallel. Aditionally, since we wait until all tasks are completed in the synchronization point before copying the second block of the current level, no elements yet to be inserted from $E$ are overwritten in line 18.

The segmentation of $E$ for parallel processing is achieved by introducing the parameter $e\_start$. In the recursive algorithm, the initial invocation of ParallelRecBiS uses $e\_start = 1$ since the final level requires the entire $E$ array. To handle parallel tasks at each level, within the for loop that iterates over the number of blocks, we calculate the offset for each recursive call. This offset is determined by adding the offset for the current task, $e\_start$, to the product of the block index, denoted by $i$, and the next block size, which is given by $current\_block\_size/k$.

## 5.2 Sequentiality of the block-insertion operation.

The key operation of Block-insertionSort involves the successive insertion of the $k$ sorted blocks, each of size $k'$. At each level, the sorted segment size increases iteratively until a single sorted segment of size $k \cdot k'$ is achieved. This process offers good general performance in a sequential environment, but, as similar to the insertion of elements in the classic Insertionsort, it is inherently sequential and hard to parallelize without deeply changing how it works or increasing the additional memory requirements. There is a strong data dependency between sorted blocks since the insertion of a block requires the space left by the previously inserted block, and the resulting position of its elements also depends on the previously inserted block.

We can enhance the achievable speedup by transitioning from sequential block-insertion to a more parallel algorithm for combining the last sorted blocks. This is where a parallel multiway merge routine would prove beneficial, as it allows us to achieve the same outcome of block-inserting the last $k$ sorted blocks of size $n/k$ while increasing parallelism. Taking it a step further, we can switch to MWM when the number of independent subproblems becomes lower than the available processing units, rather than solely replacing the last level. However, determining the optimal switching point or finding the ideal value of $k$ can be challenging as the number of independent subproblems in the last levels depends on the chosen $k$ parameter and the input size.

Another even more direct approach would be to simply divide the input array into $p$ segments, allowing each of the $p$ threads or processing units to sort a segment of size $n/p$ using sequential BiS. After sorting all the segments, we simple merge them using the MWM routine. Going back to the memory complexity of sequential BiS ($O(n)$ from Equation 2), note that in the best case we need to store exactly $n/k$ elements, that is when the input size is an exact power of $k$. In the general case, the required memory is given by the power of $k$ that is the closest to $n$, with the worst scenario being when we only have two full blocks and a small remainder, making it so we need to store close to $n/2 < n$ elements. Considering the general case, let $dt = k^{\lfloor \log_k n \rfloor} < n$. If $n' = n/p$, for this splitting approach we would need a maximum of $dt' = k^{\lfloor \log_k n' \rfloor} < n'$ additional memory for each of the $p$ segments, so in total $p \cdot dt' < p \cdot n/p$, which is still $O(n)$ for any constant $p$. By doing this, our general sorting approach would be similar to MCSTLmwm, but with the practical BiS serving as a large base case. Therefore, we can expect to achieve performance similar to that of the multiway mergesort algorithm used in the libstdc++ parallel mode.

## 5.3 Work and span of PBiS

The analysis of the simpler PBiS is quite straightforward. Similar to the work-span analysis of mergesort with sequential merge [19], since we are just spawning the parallel tasks and block-inserting serially, the work of the algorithm remains unchanged with respect to the sequential algorithm, while the span $O(n)$ is given by the serial block-insertion (just like serial merge). For the case of our hybrid PBiS, we need to refer to the MCSTL publication for the total execution time of their multiway merge algorithm. To merge $k$ sorted sequences $S_1 \ldots S_k$, The total execution time of their MWM algorithm is $O(\frac{m}{p} \log k + k \log k \cdot \log \max_j |S_j|)$, where $m$ is used for multi-sequence partition with global ranks $\{m/p, 2m/p, \ldots, (p-1)m/p, m\}$ [40]. In our case, assuming a constant $p$, the cost of merging $k = p$ sequences of size $n/p$ is $n \log k + k \log k \cdot \log n = O(n)$, meaning no considerable increase in total work.

## 5.4 Implementation detals

We use C++17 for all our CPU implementations. For MWM, we use *__gnu_parallel::multiway_merge* algorithm included in the *parallel/algorithm* header, a file that is a GNU extension to the Standard C++ Library. Unlike algorithms executed with the parallel execution policy, which require linking to ltbb, GNU parallel algorithms only require the -fopenmp flag during compilation.

Our initial approach involved exploring various combinations of sequential pair-wise merges, sequential merges inside parallel for-loops, and *std::inplace_merge* with parallel execution policies. Since MWM is not part of the STL, it was challenging to find its availability. However, it is well-documented in the The GNU C++ Library Manual and within the code itself in order to configure its runtime behavior and to support its usage outside of the library's internal operations. After conducting experiments and observing the results, we found that, with enough parallelism available, multiway merge consistently yielded superior outcomes, making it our chosen parallel merging strategy. An important consideration is that multiway merge does requires an explicit output array of the same size as the input. As a result, we had to adjust our memory requirements by allocating the extra array E to size $n$ instead of $n/k$, which still keeps us within the bounds of O(n) space complexity.

Previously, we used a top-down, recursive version of the algorithm based on parallel tasks to describe PBiS. However, for our final implementations presented in the following results, we switched to a bottom-up implementation using parallel for-loops. Additionally, in our initial hybrid version, we incorporated a check to evaluate the number of independent segments available, allowing us to switch to MWM if the number of subproblems was less than the number of available processing units. Our final hybrid implementation, as described earlier, simply splits the input into p segments of size $n/k$, sorts each segment with BiS and finally merges them all with MWM.

## 6 Experimental Results

### 6.1 Setup

The experiments were conducted on a machine equipped with an AMD Ryzen™ Threadripper™ PRO 5975WX 32-Core processor and 64GB of DDR4 RAM running at 3200MHz. The machine used Arch Linux with kernel version 6.2.10-arch1-1. The C++ implementations were compiled and linked with g++ (GCC) version 12.2.1, utilizing the following flags: -O3 -ltbb -fopenmp -latomic -lpthread -std=c++17.

Each data point in our final results represents the average time calculated from 10 executions. We included only those data points that produced valid outputs, meaning they matched the correct results generated by the sequential std::sort.

### 6.2 Data Types and Input Distributions

We conducted our experiments using *int*, *float*, *double* and *size_t* (unsigned long) datatypes. To generate random numbers, we utilized the *std::mersenne_twister_engine* random number engine along with *uniform* and *normal* random number distributions, which are defined in the *random* header of the C++ standard library. For both normal and uniform distributions, we also generated sorted, almost sorted (randomly swapping $p = 5\%$ of the elements) and reverse sorted inputs. Regarding the values generated, for normally distributed data we set the mean to $n/2$ and the variance to $0.1n$. For uniformly distributed data, we generated positive values up to $2^{27}$, excluding the case of unsigned long where we used $2^{54}$ as the upper limit. We used a fixed seed value of 0 for all random-related operations throughout the experiments. Again, we use $p$ to denote the number of cores/threads used.

### 6.3 Implementations used

**Parallel Block-InsertionSort (not in-place)**

- **PBiS:** bottom-up iterative parallel BiS using sequential Block-insertion only.
- **Hybrid PBiS:** implementation that switches to MWM when the number of remaining independent subproblems is lower than certain threshold proportional to the number of available processors ($p$).
- **Hybrid PBiS PSplit:** sort, in parallel, $p$ segments of size $n/p$ with sequential BiS and finally merge these p segments using MWM from the libstdc++ parallel mode.

**External Implementations:**

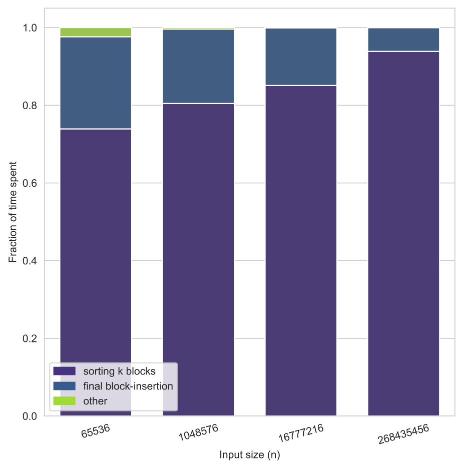
- **tbb::parallel::sort()** Pseudo median of nine Quicksort from the oneTBB library (in-place) [36]
- **ipso::parallel::sort()** Inplace Super Scalar Sample sort [12]
- **gnu_parallel::qs** Quicksort from GNU parallel algorithms, invoked by using the tag *gnu_parallel::quicksort_tag()* (in-place) [40]
- **gnu_parallel::qsb** Load-Balanced Quicksort from GNU parallel algorithms, invoked by using the tag *gnu_parallel::balanced_quicksort_tag()* (in-place) [40]
- **gnu_parallel::mwms** Multiway Mergesort from GNU parallel algorithms, invoked by using the tag *gnu_parallel::multiway_mergesort_tag()* (not in-place) [40]
- **omp_task::mergesort** Custom implementation of Mergesort with OpenMP Tasks. It uses std::inplace_merge() to merge and std::sort() as base case (non in-place[3]) [26]
- **ipra::parallel::sort()** Inplace Super Scalar RadixSort, only for unsigned long inputs [12]

[3] inplace_merge needs to be able to allocate enough additional memory in order to guarantee O(n) time complexity

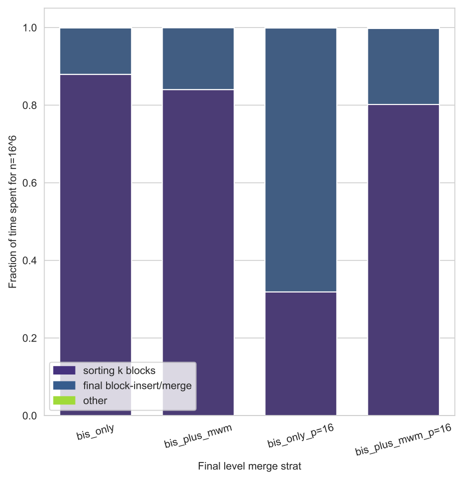
## 6.4 Block-Insertion vs Multiway Merge

Figure 3 illustrates the distribution of execution time for the algorithm across different stages, depending on the input size. The stage *sortking $k$ blocks* represents the time taken to sort $k$ blocks of size $n/k$ with sequential BiS. The *final block-Insertion* (a) and *final merge* (b) stages refers to the Block-insertion (a) and either MWM or Block-insertion (b) operations performed on those $k$ sorted blocks in the last level of the algorithm (Table 1). The *other* stage refers to tasks such as memory allocation and index computation, which are only is significant for small inputs.

Figure 3 (a) shows that for inputs of size around $10^6$, the *final block-Insertion* step represents approximately 20% of the total execution time. As the input size increases, this fraction gradually decreases, reaching about 15% for $n = 16^6$. In Figure 3 (b), we compare the same bar from (a) for $n = 16^6$ with a version that incorporates MWM instead of Block-Insertion in the last level of the algorithm. When we use multiway merge to combine the final $k$ segments in a sequential setup ($p = 1$), the overall time actually increases compared to the pure BiS approach. However, when parallelism is available ($p = 16$), the sequential block insertion in the last level becomes the primary bottleneck. Switching to MWM at this stage reduces the total execution time to roughly half of that achieved with Block-insertion alone, and the fraction of time spent between sorting and inserting/merging returns to looking similar to the sequential ones.



(a) Fraction of time spent on each stage by input size, sequential ($p = 1$), $k = 16$.

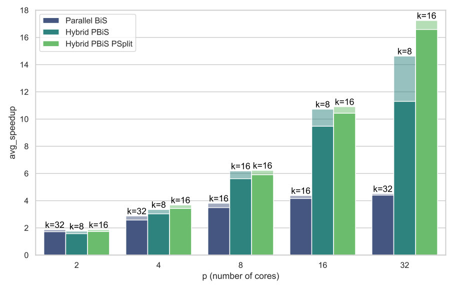


(b) Fraction of time spent by the two main approaches for $n = 16^6$, $p = k = 16$.

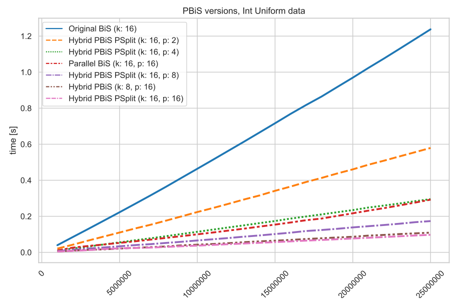
**Fig. 3.** For large enough inputs $n > 10^7$, less than 20% of the execution time is spent on the final block-insertion of $k$ blocks (a). This may not seem like a lot, but when we can sort these $k$ blocks in parallel, the sequential insertion step becomes the primary limiting factor for achieving speedup (b).

In Figure 4 (a), the average speedup is shown for best and worst choices of $k$, $k \in [8, 16, 32]$ with varying numbers of cores, denoted as $p$. Pure parallel PBiS demonstrates a slight advantage over the Hybrid versions when using 2 cores, but its speedup plateaus at around 4x. The difference between Hybrid PBiS and Hybrid PBiS PSplit is not significant for $p \leq 16$, but it becomes more apparent at $p = 32$, where the difference reaches approximately 2 points. Notably, for $p = 32$, Hybrid PBiS is more sensitive to the choice of $k$, with a difference in speedup of over 3 points between the best ($k = 8$) and worst ($k = 32$, number not shown) selections. Additionally, it is interesting to observe that Hybrid PBiS PSplit shares its optimal $k$ value with the sequential BiS of the original work, as one would initially expect.

Figure 4 (b) ilustrates the execution time for different versions of Parallel BiS with focus on Hybrid PBiS PSplit. As it could be observed by comparing the average speedups in Figure 4 (a), the performance of PBiS with $p = 16$ is comparable to our hybrid implementations using only $p = 4$. In any case, comparing the different parallel versions with the original Block-InsertionSort, the benefits in terms of execution time are evident. However, the impact is even greater for $p > 2$ with our hybrid implementations that take advantage of a parallel algorithm to insert/merge the final $k$ blocks that have been sorted using the efficient sequential BiS approach.



(a) Impact of $k$ on average speedup. The lighter colored sections highlight the difference between the best and the worst $k$ on each setup.



(b) Execution times for the original BiS and different Parallel-BiS setups.

**Fig. 4.** Comparing Parallel BiS versions on Int uniform inputs with sizes between $10^6$ and $25 \times 10^6$

## 6.5 General comparisons

From Figure 5 to Figure 7, we present a comparison of the execution times for the different implementations using various input types and distributions. All experiments were conducted with $p = 32$ and input size increments of $10^6$. In this comparison, we exclude the initial *Hybrid PBiS* based on the results shown in Figure 4 (a). We also exclude *gnu_parallel::bqs* in favor of *gnu_parallel::qs*, which has provided better results in our experiments. The results for *PBiS* are obtained using the optimal $k = 32$ value that was also in the same Figure 4 (a).

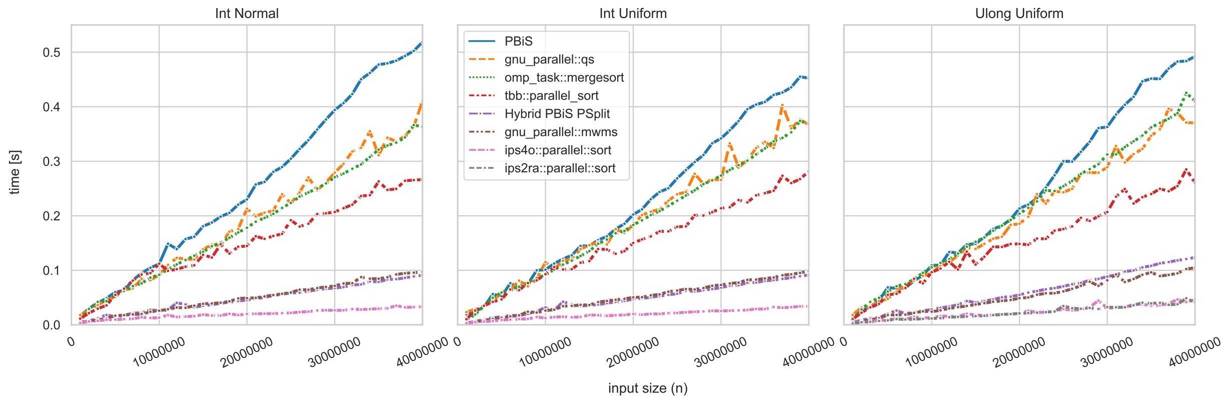


**Fig. 5.** Time vs input size (N) using $p = 32$ cores sorting random Integer data.

The first observation from Figure 5 is the presence of three distinct performance groups when sorting random data from the two main distributions. Ips4o demonstrates a clear dominance, consistent with previous findings. In the second group, we find the MWM-based implementations, including mwms from gnu_parallel and our hybrid parallel BiS. The remaining implementations form the third group, where PBiS ranks lower compared to other quicksort and merge-based approaches. Interestingly, Figure 5 reveals only minimal variations related to key size (int and ulong uniform data) and the main distributions (int normal and int uniform data), with no significant impact on the relative rankings among the implementations.

Figure 6 presents a different scenario with inputs that already exhibit some degree of sortedness. In the case of sorted data, Ips4o is surpassed by TBB's quicksort. This can be attributed to the periodic checks performed by TBB parallel sort to determine if the input is already sorted in order to stop earlier. When considering reverse sorted data, we observe that the mwms implementation achieves the best execution times, outperforming other approaches. In all three cases depicted in Figure 6, our Hybrid PBiS PSplit consistently maintains its position relative to other implementations and demostrates reliable execution times.

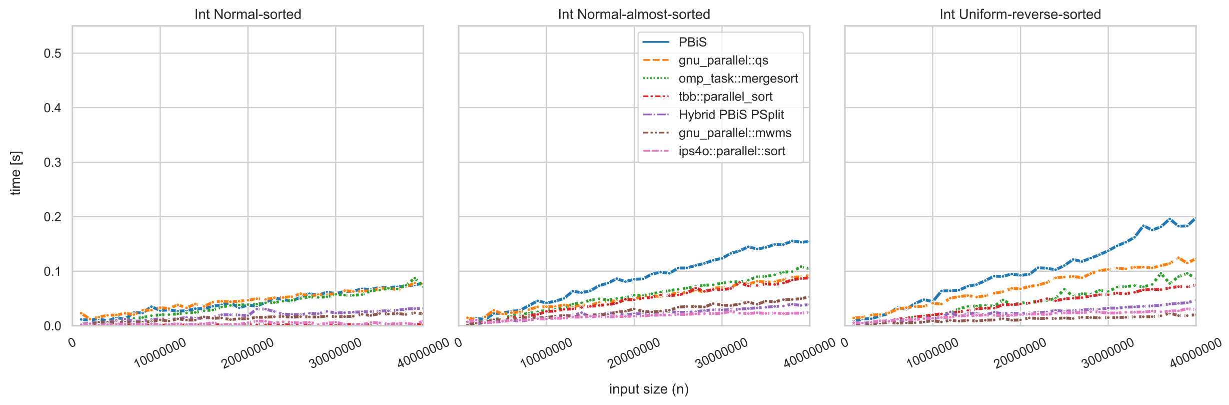


**Fig. 6.** Time (log scale) vs input size (N) using $p = 32$ cores. Input with different grades of sortedness.

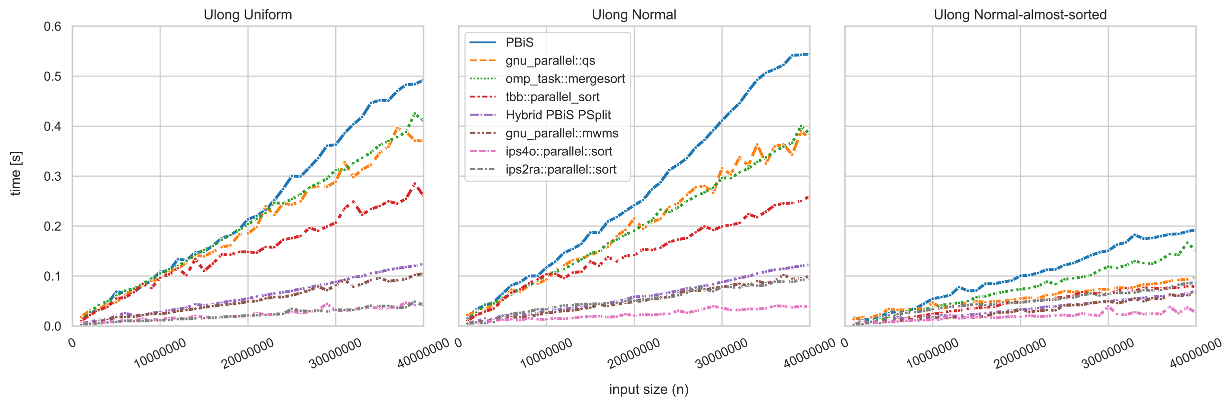


**Fig. 7.** Time vs input size (N) using $p = 32$ cores sorting random ulong data.

Figure 7 focuses on unsigned long data and includes the radix sorter Ip2ra. In the ulong uniform plot (left), we observe that Ip2ra performs similarly to Ips4o, with no significant advantage over the comparison-based algorithm. This lack of noticeable improvement may be attributed to the wide range of values generated for ulong data, resulting in large keys that impact the performance of Ip2ra. Figure 7 also highlights the sensitivity of certain algorithms to different input distributions. Both PBiS and Ips2ra exhibit noticeable increased execution times when sorting normally distributed data compared to uniformly distributed data. However, the difference in performance for Ips2ra is more pronounced in relative terms rather than absolute values. In any case, these results align with the observation that Ip2ra excels with small keys and uniform data. Regarding unsigned long, nearly sorted normal data (right), it is clear that the included implementations handle this particular input well, with lower execution times all across the board.

| | **Hybrid PBiS PSplit** | **PBiS** | **gnu parallel mwms** | **ips4o parallel sort** | **tbb parallel sort** |
|---|---|---|---|---|---|
| **double_normal** | 1.0 | 4.4045 | 0.9049 | 0.3477 | 2.4580 |
| **double_uniform** | 1.0 | 4.8648 | 0.9023 | 0.3531 | 2.4886 |
| **float_normal** | 1.0 | 5.2249 | **1.0210** | 0.3604 | 2.9633 |
| **float_uniform** | 1.0 | 5.7139 | **1.0142** | 0.3603 | 3.0173 |
| **int_normal** | 1.0 | 5.3327 | **1.0431** | 0.4165 | 3.1505 |
| **int_normal-almost-sorted** | 1.0 | 3.9052 | **1.2314** | 0.7866 | 2.1248 |
| **int_normal-sorted** | 1.0 | 2.2508 | 0.7235 | 0.2070 | 0.0804 |
| **int_uniform** | 1.0 | 4.6463 | **1.0489** | 0.4010 | 3.1321 |
| **int_uniform-reverse-sorted** | 1.0 | 4.1561 | 0.4904 | 0.8128 | 1.6614 |
| **ulong_normal** | 1.0 | 4.4675 | 0.8587 | 0.3905 | 2.4075 |
| **ulong_normal-almost-sorted** | 1.0 | 2.9843 | 0.9219 | 0.5580 | 1.3376 |
| **ulong_normal-sorted** | 1.0 | 1.8107 | 0.5928 | 0.1278 | 0.0639 |
| **ulong_uniform** | 1.0 | 3.9517 | 0.8687 | 0.3884 | 2.4896 |
| **ulong_uniform-reverse-sorted** | 1.0 | 3.1773 | 0.4312 | 0.5332 | 0.9258 |

**Table 2.** Average execution times in relation to Hybrid PBiS PSplit for a selection of implementations (using $p = 32$ and sizes between $10^6$ and $25 \times 10^6$). Values lower than 1 mean better execution time than our hybrid algorithm.

Finally, Table 2 summarizes the results of our Hybrid PBiS PSplit implementation compared to a selection of parallel implementations used in our experiments. It is evident that Ips40 dominates the performance, with our implementation being two to three times slower on random (non-sorted) inputs. As expected, our implementation achieves similar performance compared to the MWMS from libstdc++ parallel mode. Except for sorted and reverse sorted data, MWMS is approximately 15% faster on 64-bit keys and nearly equal on 32-bit keys. Notably, int normal almost sorted data proves to be a particularly favorable case for Hybrid PBiS PSplit, as our implementation surpasses MWMS significantly in this scenario. For the case of TBB parallel sort, even if we exclude sorted inputs it is still a pretty good implementation, considering it works in-place unlike MWM-based implementations.

## 6.6 CUDA BiS

Additional to the CPU parallel BiS implementations described earlier, we also developed a naive GPU implementation of the algorithm and compared it against both simple and state-of-the-art, highly optimized GPU sorting implementations. We used CUDA Dynamic Parallelism with CUDA Streams to implement a recursive PBiS with a base case that sorts with iterative BiS blocks of up to 4096 int32 elements directly in shared memory (naive_gpu_bis).

We included another experimental implementation (third_gpu_cbis) that increases shared memory utilization but applies the same sequential block-insertion process. This small change of copying smaller segments of the sorted block into shared memory slightly improves the execution times of our first naive approach, even if it involves additional work of copying data and bound checking, highlighting the importance of memory hierarchy on the CUDA model. Since the shared memory size and the number of elements that it can store depends on the hardware and the data type, this implementation only supports sorting keys of type $int32$ for the time beign.

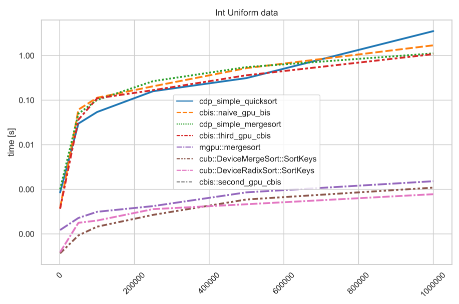
## GPU implementations

Implementations compiled with nvcc from CUDA compilation tools, release 12.1, V12.1.105 using the flags –expt-relaxed-constexpr -dc -arch=sm_60 –expt-extended-lambda. Initial execution times were obtaining from running on a NVIDIA GeForce GTX 1050 Ti, 4 GB GDDR5 memory.

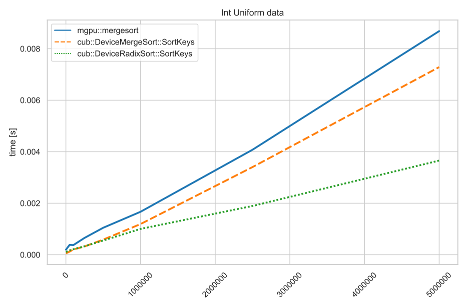
- Custom implementation of Mergesort using CUDP2 and thrust's device merge
- CUDP2 Quicksort from NVIDIA CUDA Samples
- mgpu::mergesort (Pairwise Mergesort from the ModernGPU library [13])
- cub::DeviceMergeSort::SortKeys() (Mergesort from the NVIDA CUB library [5])
- cub::DeviceRadixSort::SortKeys() (Onesweep, LSD Radixsort [5,9])

GPU-MMS and CUDA-Quicksort have their code publicy available but we were not able to include their implementations in our GPU experimental setup.

It is clear that the difference between naive and highly optimized implementations is abysmal when working in manycores architectures and specifically in the CUDA model. Figure 8 (a) displays the order of magnitude difference that exist between those two type of implementations. Figure 8 (b) shows the high-performance implementations only. In this case, the integer sorter from the cub library **cub DeviceRadixSort** show considerably better results than the highly optimized comparison-based implementations from cub and moderngpu starting from input sizes $n = 10^6$



(a) All GPU implementations



(b) High-performance implementations only

**Fig. 8.** Time vs input size (N) for GPU.

## 7 Conclusions and Future Work

In this work, we have explored the strategies used in the implementation of parallel sorting algorithms and examined two important parallel API or compute models, such as CUDA and OpenMP. With these insights, we designed and implemented parallel versions of the efficient comparison-based algorithm, BlockInsertionSort.

Our OpenMP implementations for the CPU have resulted in significant speedup without major increases in memory requirements. Additionally, our hybrid implementation, which relies on algorithms from the libstdc++ Parallel Mode of the GNU project, achieves performance very close to their multiway mergesort. On average, our hybrid implementation achieves a speedup of around 17x when sorting int uniform data on 32 cores. For larger inputs, we have observed up to a 20x speedup compared to the original sequential BiS in the same setup.

The block insertion part of the algorithm demonstrates exceptional efficiency in a sequential setup, outperforming both the classic 2-way and multiway merge procedures when generating a sorted sequence from multiple smaller sorted sequences. However, in scenarios where parallelism is available, parallel merges prove to be a better choice in comparison to block-insertions. Even if the strictly sequential step does not represent a large fraction of the total execution time, it can really limit the speedup when introducing enough parallelism.

To further enhance our parallel implementations, we can focus on improving the performance in cases involving sorted inputs and explore potential micro-optimizations to enhance the sequential algorithm. One possible improvement is to incorporate validations to directly copy or move blocks that are already sorted relative to one another, particularly benefiting sorted and nearly sorted inputs. These efforts have the potential to yield even greater performance gains and ensure more consistent execution times.

Regarding our GPU-BiS results, while they are comparable to simple implementations of algorithms like Mergesort and Quicksort, they fall short of state-of-the-art, highly optimized work. There is ample room for improvement in this area, and further optimizations can be explored to achieve better performance. To maximize the potential of GPUs for high-performance computing, it is crucial to design algorithms that effectively utilize the massive parallelism they offer. Merely porting an algorithm to the GPU does not guarantee optimal performance. To achieve the best results, memory access must follow optimal patterns, there should be a well-balanced mix of fine and coarse-grained parallelism, synchronization should be minimized, to name a few relevant considerations that should be addressed.

With the results presented and the discussions held, we can now address the research questions. It is evident that, particularly on the CPU side, we have successfully developed a parallel implementation that achieves significant speedup for a small number of processing units. By leveraging existing implementations of interesting algorithms such as multiway merge, we have improved our initial parallel implementation, resulting in enhanced execution times and an implementation that can compete with important sorting algorithms. On the GPU side, our work has been mainly exploratory, and although we have developed implementations comparable to other naive approaches using classical algorithms, there is still significant room for improvement in order to narrow the performance gap and approach the level of state-of-the-art implementations.

Parallel computing has become increasingly accessible, thanks to available libraries and active open-source projects that offer high-performance building blocks for many computer science problems and general programming tasks. The exceptional efforts put into developing these high-performance libraries, including notable implementations like Ips4o, play a vital role in advancing the field of computer science. For cases where high-performance libraries cannot help, tools such as OpenMP and CUDA provide options for both "plug and play" parallelism and fine-grained control over available resources. Understanding the programming models and underlying architectures of computing devices is crucial for maximizing performance in both cases.

## References

1. Compiler support for C++17 - cppreference.com — en.cppreference.com. https://en.cppreference.com/w/cpp/compiler_support/17#C.2B.2B17_library_features, [Accessed 13-Jul-2023]

2. libc++ Parallel STL Status; libc++ documentation — libcxx.llvm.org. https://libcxx.llvm.org/Status/PSTL.html, [Accessed 13-Jul-2023]
3. OpenMP FAQ — openmp.org. https://www.openmp.org/about/openmp-faq/, [Accessed 21-Apr-2023]
4. Cpython/sorting.rst at main - python/cpython. https://github.com/python/cpython/blob/main/Doc/howto/sorting.rst#sort-stability-and-complex-sorts (2022), [Accessed 21-Apr-2023]
5. Nvidia - cub. https://github.com/NVIDIA/cub (2023), [Accessed 21-Apr-2023]
6. Nvidia - thrust. https://github.com/NVIDIA/thrust (2023), [Accessed 21-Apr-2023]
7. slice#method.sort - rust. https://doc.rust-lang.org/std/primitive.slice.html#method.sort (2023), [Accessed 21-Apr-2023]
8. Std::sort. https://en.cppreference.com/w/cpp/algorithm/sort (2023), [Accessed 21-Apr-2023]
9. Adinets, A., Merrill, D.: Onesweep: A faster least significant digit radix sort for gpus (2022)
10. Ali, M., Nazim, Z., Ali, W., Hussain, A., Kanwal, N., Paracha, M.: Experimental analysis of on(log n) class parallel sorting algorithms. INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND NETWORK SECURITY **20**, 139–148 (01 2020)
11. Asaduzzaman, A., Trent, A., Osborne, S., Aldershof, C., Sibai, F.N.: Impact of cuda and opencl on parallel and distributed computing. In: 2021 8th International Conference on Electrical and Electronics Engineering (ICEEE). pp. 238–242 (2021). https://doi.org/10.1109/ICEEE52452.2021.9415927
12. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. ACM Trans. Parallel Comput. **9**(1) (jan 2022). https://doi.org/10.1145/3505286, https://doi.org/10.1145/3505286
13. Baxter, S.: moderngpu 2.0 (2016), https://github.com/moderngpu/moderngpu/wiki
14. Blelloch, G.E., Anderson, D., Dhulipala, L.: ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. ACM (Jul 2020). https://doi.org/10.1145/3350755.3400254, https://doi.org/10.1145/3350755.3400254
15. Carlini, P., Edwards, P., Gregor, D.: Gcc 5. 2 Standard C++ Library Manual. Samurai Media Limited (2015), https://books.google.cl/books?id=7mEVjwEACAAJ
16. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. ACM J. Exp. Algorithmics **14** (jan 2010). https://doi.org/10.1145/1498698.1564500, https://doi.org/10.1145/1498698.1564500
17. Cole, R.: Parallel merge sort. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986). pp. 511–516 (1986). https://doi.org/10.1109/SFCS.1986.41
18. Cole, R., Ramachandran, V.: Resource oblivious sorting on multicores. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) Automata, Languages and Programming. pp. 226–237. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
19. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
20. Dagum, L.: Data parallel sorting for particle simulation. Concurrency: Practice and Experience **4**(3), 241–255 (1992). https://doi.org/https://doi.org/10.1002/cpe.4330040304, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330040304
21. Davidson, A., Tarjan, D., Garland, M., Owens, J.D.: Efficient parallel merge sort for fixed and variable length keys. In: 2012 Innovative Parallel Computing (InPar). pp. 1–9 (2012). https://doi.org/10.1109/InPar.2012.6339592
22. DEHNE, F., ZABOLI, H.: DETERMINISTIC SAMPLE SORT FOR GPUS. Parallel Processing Letters **22**(03), 1250008 (Jul 2012). https://doi.org/10.1142/s0129626412500089, https://doi.org/10.1142/s0129626412500089
23. Ferrada, H.: A sorting algorithm based on ordered block insertions. Journal of Computational Science **64**, 101866 (2022). https://doi.org/https://doi.org/10.1016/j.jocs.2022.101866, https://www.sciencedirect.com/science/article/pii/S1877750322002253
24. Gao, Q., Liu, Z.: Sloping-and-shaking. Science in China Series E: Technological Sciences **40**(3), 225–234 (Jun 1997). https://doi.org/10.1007/bf02916597, https://doi.org/10.1007/bf02916597
25. Gowanlock, M., Karsin, B.: A hybrid cpu/gpu approach for optimizing sorting throughput. Parallel Computing **85**, 45–55 (2019). https://doi.org/https://doi.org/10.1016/j.parco.2019.01.004, https://www.sciencedirect.com/science/article/pii/S0167819118302515
26. a Jan Dvořák, L.B.: Advanced programming with openmp, [Accessed 21-Apr-2023]
27. Karsin, B., Weichert, V., Casanova, H., Iacono, J., Sitchinava, N.: Analysis-driven engineering of comparison-based sorting algorithms on gpus. In: Proceedings of the 2018 International Conference on Supercomputing. p. 86–95. ICS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3205289.3205298
28. Klemm, M., Board, O.A.R., de Supinski, B.: OpenMP Application Programming Interface Specification Version 5. 0. Independently Published (2019), https://books.google.cl/books?id=sgGpyAEACAAJ
29. Leischner, N., Osipov, V., Sanders, P.: Gpu sample sort. In: 2010 IEEE International Symposium on Parallel Distributed. pp. 1–10 (2010). https://doi.org/10.1109/IPDPS.2010.5470444

30. Li, X., Lu, P., Schaeffer, J., Shillington, J., Pok Sze Wong, Shi, H.: On the versatility of parallel sorting by regular sampling. Parallel Computing **19**(10), 1079–1103 (1993). https://doi.org/https://doi.org/10.1016/0167-8191(93)90019-H, https://www.sciencedirect.com/science/article/pii/016781919390019H
31. Manca, E., Manconi, A., Orro, A., Armano, G., Milanesi, L.: Cuda-quicksort: an improved gpu-based implementation of quicksort. Concurrency and Computation: Practice and Experience **28**(1), 21–43 (2016). https://doi.org/https://doi.org/10.1002/cpe.3611, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3611
32. Mišić, M.J., Tomašević, M.V.: Data sorting using graphics processing units. In: 2011 19thTelecommunications Forum (TELFOR) Proceedings of Papers. pp. 1446–1449 (2011). https://doi.org/10.1109/TELFOR.2011.6143828
33. Neapolitan, R., Naimipour, K.: Foundations of Algorithms. Jones & Bartlett Learning, LLC (2009), https://books.google.cl/books?id=u7LztCWIb6QC
34. NVIDIA Corporation: NVIDIA CUDA C++ programming guide (2023), version 12.1
35. Obeya, O., Kahssay, E., Fan, E., Shun, J.: Theoretically-efficient and practical parallel in-place radix sorting. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures. p. 213–224. SPAA '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3323165.3323198, https://doi.org/10.1145/3323165.3323198
36. Oneapi-Src: Oneapi-src/onetbb: Oneapi threading building blocks (onetbb), https://github.com/oneapi-src/oneTBB
37. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: 2009 IEEE International Symposium on Parallel Distributed. IEEE (May 2009). https://doi.org/10.1109/ipdps.2009.5161005, https://doi.org/10.1109/ipdps.2009.5161005
38. Sedgewick, R., Wayne, K.: Algorithms, 4th Edition. Addison-Wesley (2011)
39. Singh, D.P., Joshi, I., Choudhary, J.: Survey of GPU based sorting algorithms. International Journal of Parallel Programming **46**(6), 1017–1034 (Apr 2017). https://doi.org/10.1007/s10766-017-0502-5, https://doi.org/10.1007/s10766-017-0502-5
40. Singler, J., Sanders, P., Putze, F.: Mcstl: The multi-core standard template library. In: Kermarrec, A.M., Bougé, L., Priol, T. (eds.) Euro-Par 2007 Parallel Processing. pp. 682–694. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
41. Stehle, E., Jacobsen, H.A.: A memory bandwidth-efficient hybrid radix sort on gpus. Proceedings of the 2017 ACM International Conference on Management of Data (2016)
42. Wassenberg, J., Blacher, M., Giesen, J., Sanders, P.: Vectorized and performance-portable quicksort. Software: Practice and Experience **52**(12), 2684–2699 (Aug 2022). https://doi.org/10.1002/spe.3142, https://doi.org/10.1002/spe.3142
43. Wheat, M., Evans, D.: An efficient parallel sorting algorithm for shared memory multiprocessors. Parallel Computing **18**(1), 91–102 (1992). https://doi.org/https://doi.org/10.1016/0167-8191(92)90114-M, https://www.sciencedirect.com/science/article/pii/016781919290114M
44. Zhang, W., Rao, N.S.V.: Optimal parallel quicksort on EREW PRAM. BIT **31**(1), 69–74 (Mar 1991). https://doi.org/10.1007/bf01952784, https://doi.org/10.1007/bf01952784