Optimizing Sorting for Chiplet-Based CPUs

Alessandro Fogli
Imperial College London
a.fogli18@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

Jana Giceva
Technical University of Munich
jana.giceva@cit.tum.de

ABSTRACT

This paper explores the challenges posed by modern chiplet-based CPU architectures in the context of sorting algorithms, a fundamental component of many computer science applications. We highlight how the heterogeneity introduced by chiplet-based processors—including varying access times to partitioned L3 caches, inter-core latencies, and bandwidths—can lead to suboptimal performance when using traditional sorting algorithms that assume uniform memory access and consistent processor performance.

To address these issues, we propose a set of chiplet-aware optimizations designed to enhance the efficiency of memory-intensive sorting algorithms on these modern architectures. Our approach includes four key strategies: (1) partitioning input data at a chiplet-level granularity to minimize inter-chiplet communication and balance the computational load, (2) extending the memory hierarchy phase to account for distinct L3 cache partitions, (3) scheduling tasks based on data size relative to local and combined L3 cache capacities, and (4) avoiding expensive data shuffling. We provide a comprehensive analysis of chiplet architectures and detail chiplet-aware implementations of LSB Radix-Sort and Comparison Sort. Our evaluation demonstrates that chiplet-conscious sorting algorithms can enhance performance by up to 4.5× compared to NUMA-aware approaches.

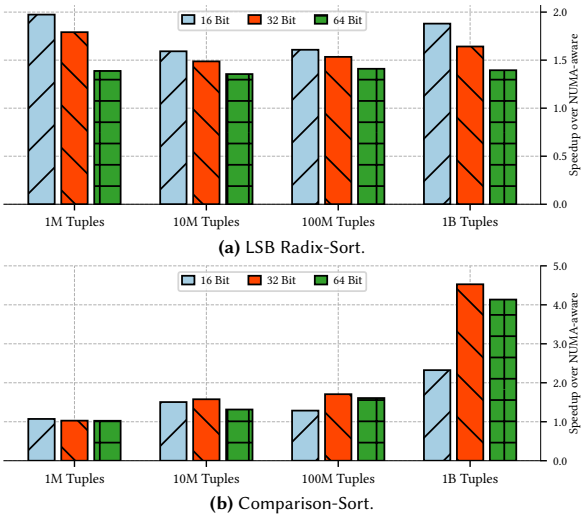
VLDB Workshop Reference Format:
Alessandro Fogli, Peter Pietzuch, and Jana Giceva. Optimizing Sorting for Chiplet-Based CPUs. VLDB 2024 Workshop: Fifteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures(ADMS 2024).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Alessandro727/Chiplet-aware-sorting-algorithms.

1 INTRODUCTION

Sorting algorithms are essential in computer science, supporting critical operations in many applications. Their efficiency greatly affects the performance of systems that handle large volumes of data. Traditionally, these algorithms have been designed with the assumption of uniform memory access and consistent processor performance within a single NUMA domain. However, this assumption is becoming outdated with the widespread deployment of modern chiplet-based CPUs.

Proceedings of the VLDB Endowment. ISSN 2150-8097.



(a) LSB Radix-Sort.

(b) Comparison-Sort.

Fig. 1: Speedup of chiplet-aware sorting algorithm compared to NUMA-aware sorting algorithms for varying input array sizes and integer widths (16-bit, 32-bit, and 64-bit integers).

Chiplet-based architectures present a recent technological advancement embraced by leading processor manufacturers. They consist of several smaller chips, known as *chiplets*, which are interconnected via a high-bandwidth fabric to function cohesively as a unified multi-core CPU. This modular approach not only allows for easier scaling to more cores in a single package but also facilitates the integration of different technologies and improves manufacturing yields.

From a software perspective, one downside is that chiplet-based processors introduce new heterogeneities. For example, chiplet-based processors exhibit different (1) access times to partitioned L3 caches across chiplets, (2) inter-core latencies, and (3) inter-core bandwidths. Unfortunately, this heterogeneity has not yet been properly accounted for in the software stack and hardware-conscious algorithm design. For instance, current parallel sorting algorithms are generally chiplet-agnostic, which can lead to suboptimal CPU utilization, particularly when task distribution and data partitioning do not align with the chiplet topology.

Heterogeneity in data access latencies and interconnect bandwidth was first addressed in the context of NUMA systems. Prior work demonstrated the benefits of allocating tasks to the core where the data is located, thereby minimizing the overhead of remote memory accesses (e.g., shorter latencies, higher local NUMA bandwidth, less congestion on the interconnect, etc.). In contrast to NUMA, which offers uniform efficiency within each socket, chiplet-based CPUs partition the L3 cache at a NUMA level, where each chiplet (and all associated cores) shares a partition. In this scenario, restricting tasks to a single chiplet's cores, as with NUMA, can improve

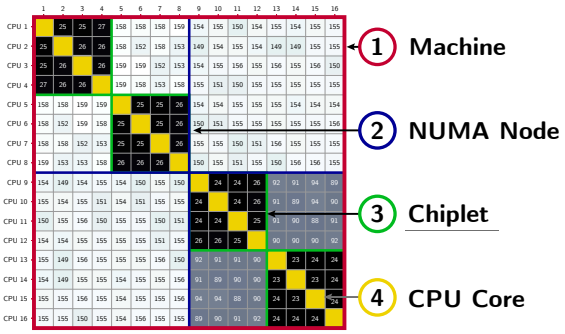


Fig. 2: Core-to-core latency and architectural components of an AMD Ryzen CPU.

cache efficiency but also limits access to other chiplets' L3 caches, reducing total cache size and increasing the time to access data from main memory when it exceeds local cache capacity. This trade-off becomes more significant with multiple chiplets in a single NUMA node, i.e., with a reduced core count relative to a full NUMA domain.

In this paper, we propose chiplet-aware optimizations that can significantly enhance the efficiency of memory-intensive sorting algorithms:

(1) Partition input data at a chiplet-level granularity: Distributing data partitions with chiplet awareness can improve sorting performance for widely used algorithms. This alignment minimizes inter-chiplet communication and balances the computational load.

(2) Extend memory hierarchy phase: We recommend extending the memory hierarchy to account for the partitions of the L3 cache within chiplet-based CPUs. Specifically, extending the in-cache sorting phase can enhance cache efficiency, promote a more balanced workload distribution, and minimize cache coherence traffic.

(3) Schedule tasks based on data size: Task-to-core assignment should be based on the relative data size compared to the local and combined L3 cache sizes. When the input data size is smaller than the capacity of the local L3 partition of a single chiplet, the algorithm should assign the task to the cores within a single chiplet. When the data size is larger than the "chiplet-local" L3 cache size but smaller than the combined size of all chiplets' L3 caches, the algorithm should distribute the tasks across a selected group of cores from all the chiplets.

(4) Avoid expensive data shuffling:. We recommend avoiding data shuffling between NUMA nodes, as it can increase inter-chiplet communication and reduce performance.

Our chiplet-aware approach demonstrates notable improvements over traditional NUMA-aware algorithms, achieving up to a 2× increase in Radix-Sort performance and up to a 4.5× enhancement in Comparison-Sort (see Fig. 1).

The rest of the paper is structured as follows:

- We provide background on chiplet architectures, analyze their challenges, and offer insights into their performance traits (§2).
- We propose an expansion of the memory hierarchy to account for the effects of a partitioned L3 cache (§3).
- We outline a series of chiplet-aware optimizations, focusing on hardware-conscious implementations of LSB Radix-Sort and Comparison-Sort (§4).
- We evaluate the impact of our optimizations across different data input sizes and distributions (§5).

2 BACKGROUND

2.1 Hardware-aware sorting

Rapid advancements in modern hardware have always provided fertile ground for academics and researchers to explore how to adjust algorithms and data structures to best leverage the underlying hardware capabilities [3, 10, 28, 39]. Sorting, as one of the most relevant and computationally expensive operations, has particularly attracted attention. For instance, prior work has explored the use of SIMD data parallelism, cache/memory-aware strategies, and NUMA optimizations [15, 22, 23, 27, 28, 41]. Here, we provide a brief overview of each category of optimizations.

SIMD data parallelism. Inoue et al.'s AA-Sort combines SIMD and thread-level parallelism to optimize data alignment on PowerPC and Cell processors[22]. Gedik et al. developed an efficient SIMD-based sorting algorithm for the Cell processor using bitonic sorting[20]. Chhugani et al. extended these techniques to x86 processors with a multi-core SIMD implementation, though it was initially limited by the hardware [15]. Satish et al. found that SIMD merge sort excels with larger keys and will benefit from future hardware improvements[41].

Cache-aware strategies. Cache-conscious approaches also make a significant difference. LaMarca and Ladner analyzed sorting algorithms in terms of cache misses and instructions and proposed cache-aware variants of Mergesort that better utilize L1 and L2 caches [27]. Their memory-tuned algorithms are often used as a reference for comparing sorting algorithms. Since then, several cache-tuned implementations for well-known algorithms have been developed, such as CC-Radix by Jiménez-González et al.[23]. Bender et al. studied cache-oblivious algorithms, which are efficient across all levels of the memory hierarchy without knowing the cache parameters, and presented a cache-oblivious sorting algorithm called Funnelsort[14].

NUMA optimizations. Several research efforts have explored NUMA-aware algorithms for operations such as sorting, joins, and data shuffling. Albutiu et al. presented a NUMA-aware design for sort-merge join algorithms on multi-core NUMA systems, which avoids cross-traffic between NUMA nodes during the sorting and merging phases, thereby improving performance [7]. They also advocate for sequential accesses to remote memory, as hardware prefetching can hide latency. Li et al. studied the data shuffling problem on NUMA architectures [28], demonstrating that a naïve shuffling implementation can be up to three times slower than a NUMA-aware approach that exploits thread binding, NUMA-aware memory allocation, and thread coordination. To prevent imbalanced use of the NUMA layer, where all transfers are directed to a subset of CPUs, they propose pre-scheduling the transfers and supervising them through synchronization to ensure load balancing.

2.2 Chiplets

Chiplet-based processors differ from monolithic integrated circuits by using a modular approach, integrating multiple smaller semiconductor dies, known as *chiplets*, onto a single package or substrate to form a functional unit. This innovative design has been adopted by all major hardware vendors, including AMD, Intel, and ARM [1, 30, 32, 34].

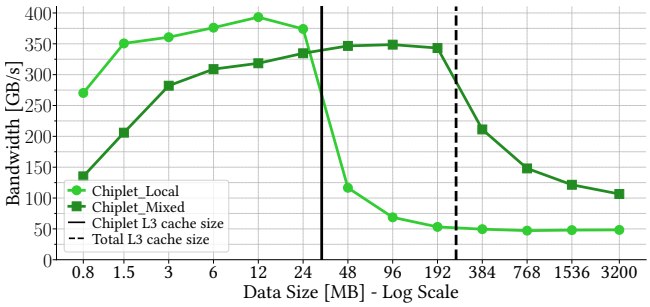


Fig. 3: STREAM Benchmark with a single process running 8 threads on an 8-chiplet AMD EPYC Milan 7713.

A key architectural feature of these chiplet processors, particularly from a database perspective, is the shift from a unified L3 cache to a partitioned L3 cache, where each chiplet has its own dedicated L3 cache segment. We argue that this partitioning of the L3 cache can have significant implications for the performance and efficiency of memory-intensive workloads such as sorting, which is the core focus of this work.

Chiplet-based CPUs introduce new types of heterogeneity: (1) they exhibit varying access times to partitioned L3 caches across chiplets, (2) they have diverse inter-core latencies, and (3) they possess different inter-core bandwidths. For instance, Fig. 2 shows the core-to-core latency of a dual-socket processor (with each socket enclosed in a blue box). In this case, inter-core latency can vary by up to 6× within the same CPU socket. This heterogeneity affects the performance of parallel processing tasks, extending beyond the challenges posed by traditional NUMA architectures [5, 19, 24, 39].

2.3 Challenges: NUMA vs. Chiplets

Workloads running on NUMA architectures can suffer from performance issues when non-local memory is frequently accessed. NUMA-agnostic programs can experience higher latency and incur bottlenecks on a particular NUMA node or the shared interconnect. These problems are often mitigated through advanced software and operating system support that optimizes memory allocation and access patterns [3, 11, 13, 26, 35], or through techniques like memory replication and migration. Note that for many of these techniques, the emphasis is on memory allocation (or migration), which can either be triggered and controlled by the user (i.e., via a user space library like libnuma [6]) or done transparently by the operating system.

Unfortunately, standard NUMA optimizations cannot address the heterogeneity introduced by chiplet architectures. This limitation is particularly evident when dealing with the partitioned L3 cache. While Intel has introduced Cache Allocation Technology (CAT) for cache partitioning and isolation, this functionality has specific constraints [4]. Intel's CAT allows the partitioning of the last-level cache into different regions, which can then be allocated to specific cores or groups of cores, known as Classes of Service (COS). This partitioning helps isolate cache usage between different applications or processes running on the same processor, thereby reducing cache contention and improving performance determinism. However, CAT does not provide a mechanism to allocate data directly to a specific chiplet's cache. CAT can be beneficial if the working set size is known and the appropriate number of cache ways needed for a chiplet can be determined in advance. If this is not the case, access to multiple chiplet caches may be required, but data allocation to these caches is managed by the hardware's cache coherence protocols rather than by CAT, potentially leading to imbalance and high inter-chiplet communication. Therefore, while CAT can partition the cache to improve isolation and performance within a single chiplet, it cannot be used to direct data to a specific chiplet's L3 cache.

This means that the options are either to adapt the structure of the algorithm and its read/write access patterns or to assign tasks specifically to a core within the chiplet with the desired cache. If we resort to the latter option, we must be careful of the trade-offs when working with the partitioned L3 cache. Assigning tasks that share a chiplet-local portion of the L3 may benefit from higher bandwidth and lower latency, but the data needs to fit into a smaller cache area. Alternatively, leveraging the full L3 capacity requires more careful access management, as it incurs inter-chiplet communication and potential overhead. Furthermore, if tasks are allowed to span across multiple chiplets, data can be fetched from one chiplet to another; however, if tasks are restricted to a single chiplet, they cannot access data from other remote chiplets.

Fig. 3 shows the bandwidth achieved during the STREAM benchmark for various data sizes. It compares the performance of using 8 cores from a single chiplet (Chiplet_Local) with using 8 cores from 8 different chiplets (Chiplet_Mixed). We observe that Chiplet_Local achieves higher bandwidth than Chiplet_Mixed until the array size reaches 32 MB, which is the single chiplet's L3 capacity. This is because Chiplet_Local avoids inter-chiplet communication. However, for larger array sizes, Chiplet_Local's bandwidth suddenly drops, as it needs to fetch data from the main memory. In contrast, Chiplet_Mixed shows more stable bandwidth due to its ability to leverage the total capacity of the L3 caches from all chiplets. Beyond this point, Chiplet_Mixed's bandwidth also declines. Based on these results, directing tasks to specific core subsets presents a trade-off in chiplet-based architectures. On one hand, it reduces the available L3 cache; on the other hand, this approach can increase the aggregate bandwidth.

3 CACHE CONSCIOUS SORT

3.1 Sorting and the memory hierarchy

Modern hardware memory hierarchies necessitate dividing the sorting algorithm into distinct phases to enhance cache efficiency:

(i) **in-register** handles runs that fit within the (SIMD) CPU registers;
(ii) **in-cache** includes runs that are confined to the CPU's L3 cache;
(iii) **out-of-cache** is for runs that exceed cache capacities.

With the widespread adoption of chiplet-based processors, where the L3 cache is partitioned, we argue that phase (ii) should be divided into two new phases: ***in-local-chiplet-cache*** and ***in-remote-chiplet-cache***.

The *in-local-chiplet-cache* phase includes sorting runs that fit within the cache of a single chiplet, thereby leveraging both the

low access latency and high bandwidth of local caches to provide superior performance.

The *in-remote-chiplet-cache* phase handles sorting when the data size exceeds a single chiplet's cache capacity but remains smaller than the total combined L3 cache across *all* chiplets. This phase must account for the higher latencies and potential slowdown associated with inter-chiplet data transfers. Effective strategies for this phase include optimizing data distribution and access patterns to ensure balanced workload distribution across the chiplets and to minimize inter-core traffic.

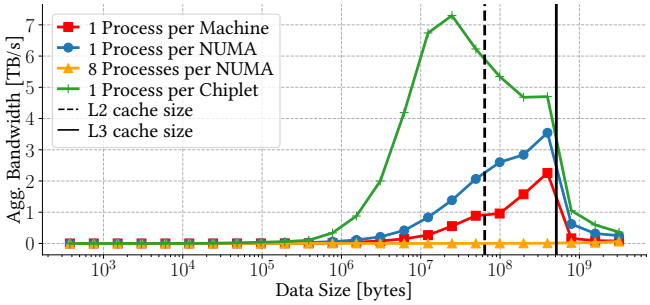


Fig. 4: STREAM Benchmark for aggregate memory bandwidth of an dual-socket 16-chiplet AMD EPYC Milan 7713.

3.2 Aggregate bandwidth on chiplet CPUs

In traditional NUMA architectures, sorting algorithms typically divide the data into equal partitions, with each partition allocated to a distinct NUMA node. Threads responsible for sorting these partitions are then assigned to cores within the same NUMA node, thereby maximizing memory locality and minimizing latency.

With chiplet CPUs, the strategy of dividing the dataset into equal partitions still holds; however, data needs to be divided into partitions corresponding to the number of chiplets to provide sufficient cache for the sorting buffers. The associated sorting threads are then assigned to specific cores within the chiplet responsible for the corresponding partition. This ensures that the sorting algorithm fully utilizes the chiplet's local cache and minimizes the need for data transfers between chiplets.

Operating sorting threads in a shared-nothing mode also results in improved aggregate bandwidth. To verify this, we measured the aggregate memory bandwidth of an AMD EPYC Milan chiplet-based processor. We use the STREAM benchmark [31] to measure sustainable memory bandwidth by performing simple operations on stored arrays of data. Specifically, we focus on the `COPY` function, which copies the contents of one array to another, using a direct memory-to-memory data transfer implemented via a loop-based copy operation. We vary the number of processes used, ranging from a single process that uses all available resources to multiple processes bound to specific chiplets or NUMA domains. Each process runs the STREAM benchmark and spawns multiple threads, which are restricted to the resources allocated to that process. For each configuration, all processes are initiated simultaneously, ensuring that the total data array size is consistently and evenly distributed among the processes.

The measured aggregate memory bandwidth for a dual-socket AMD EPYC Milan processor system, featuring 16 chiplets with 8 cores and 32 MB of L3 cache each, is illustrated in Fig. 4. It shows how aggregate bandwidth varies with increasing array sizes across different process configurations. The system achieves peak performance of approximately 7 TB/s when the data for each process fits within the L2 cache, using a configuration of one shared-nothing process per chiplet. This observed peak closely aligns with the theoretical maximum L2 bandwidth of around 8 TB/s, calculated from the measured per-core L2 cache bandwidth of 63.7 GB/s multiplied by 128 cores [44]. As the data size increases beyond L2 cache capacity, there is a transition to L3 cache utilization. Here, the one-process-per-chiplet configuration maintains superior performance, reaching 4.8 TB/s for data sizes approaching the total L3 cache size. This outperforms the NUMA-aware configuration (one process per NUMA node), which achieves 3.6 TB/s in the same range. The theoretical maximum L3 bandwidth is approximately 5.8 TB/s, based on a single-core L3 read speed of 46 GB/s [44]. Notably, the shared-nothing approach with one process per chiplet consistently outperforms the NUMA-aware configuration across various data sizes. Additionally, the eight-process per NUMA configuration uses the same number of processes as the one-process per chiplet configuration (16 processes). However, unlike the chiplet-based approach, each process in the NUMA configuration can access all resources (chiplets, caches, and main memory) within its assigned NUMA node. This shared access leads to significant resource contention, substantially reducing the achievable bandwidth despite having the same number of processes. This performance difference highlights the benefits of distributing workloads among chiplets and maximizing cache locality, resulting in significantly improved aggregate bandwidth.

4 IMPLEMENTATION

In this section, we present two chiplet-aware implementations of hardware-conscious sorting algorithms. Specifically, we implemented a stable least-significant-bit (LSB) Radix-Sort and a Comparison-Sort based on range partitioning, building on the implementations from Polychroniou et al.[2, 38]. Additionally, we discuss our chiplet-aware scheduling strategy and the optimizations we applied.

Following common practices, we operate on fixed-length keys and payloads. In read-only workloads typical of data analytics, more complex data types can be encoded into compact integer types that maintain key order. Keys and payloads are stored in separate arrays, as is typical in column-store analytical databases.

In our implementations, we adopt a shared-nothing, load-balanced partitioning strategy that disregards NUMA boundaries. This eliminates the need for separate sampling and histogram generation for each NUMA node, as well as the subsequent aggregation of these histograms. Our NUMA-oblivious (chiplet-aware) partitioning method performs better than the conventional two-step process, which involves shared-nothing NUMA-aware partitioning followed by NUMA shuffling.

4.1 LSB Radix-Sort

We propose a stable, chiplet-aware least-significant-bit (LSB) radix sort that utilizes multiple threads. As in prior work, we leverage SIMD instructions during histogram creation and partitioning to

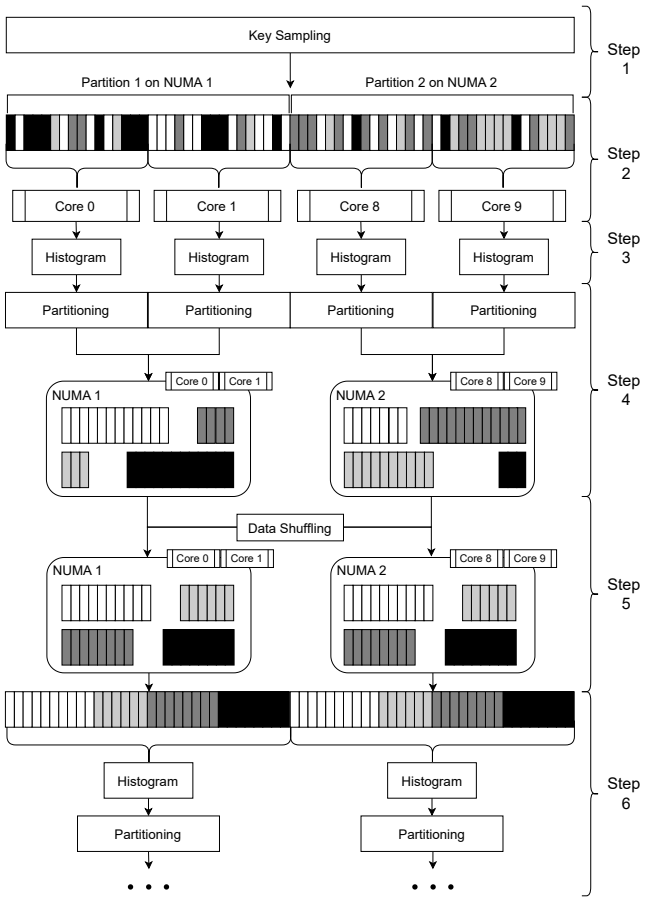


Fig. 5: NUMA-aware LSB Radix-Sort step by step execution.

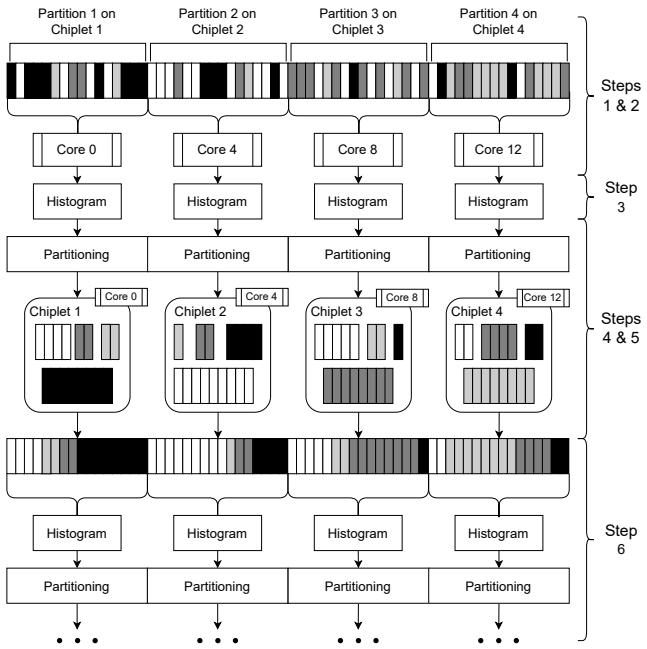


Fig. 6: Chiplet-aware LSB Radix-Sort step by step execution.

enable parallel processing of multiple data elements with a single instruction. The algorithm generally consists of six steps:

Step 1. Previous methods efficiently partition data by sampling a subset of keys to determine partition delimiters. This step is important for creating balanced partitions in radix sort and aids in data shuffling.
Our optimization: We eliminate the need for sampling by partitioning the input data based on the level of parallelism.

Step 2. Prior work splits the input data into large segments bound to NUMA regions [38].
Our optimization: We bind each thread to a specific core within a chiplet, enabling access to local chiplet caches. Our thread scheduling strategy considers input data size, chiplet's L3 share size, required parallelism, and the number of chiplets. We provide more details in §4.4.

Step 3. Each thread creates a histogram for its chunk of data. A histogram is an array where each index corresponds to a range of keys, and the value at each index is the count of keys that fall into that range.

Step 4. Based on the local histograms, each thread partitions its data into buffers allocated within the local NUMA node.
Our optimization: We keep the buffers primarily within the local chiplet cache.

Step 5. After the partitioning phase, previous methods may shuffle data to balance the load among the NUMA domains.
Our optimization: We avoid data shuffling, thereby preventing the overhead associated with frequent data transfers, including those between chiplets.

Step 6. The histogram creation and partitioning steps are repeated as necessary based on the number of sorting bits. This is done in multiple passes if the number of sorting bits requires it.
Our optimization: We dynamically adjust the number of passes based on the input array size.

Finally, by executing a prefix sum of all individual histograms, we ensure that each partition output is written to a distinct location. The final output is then a concatenation of segments of sorted data.

4.2 Comparison-Sort

The algorithm we propose is comparison-based and resembles radix-sort but, instead of using radix, it employs range partitioning [38]. As with the Radix-Sort, we leverage SIMD instructions during the histogram creation phase and the sorting phase. The algorithm determines how many range partitioning passes are needed to split the data into smaller parts that can fit in the cache. To calculate the range function, prior work uses a specialized index in the cache. Alongside building the histogram, the range partition for each tuple is then stored in this index to avoid recalculating it later. We maintain the same approach to improve the cache usage. The algorithm consists of six steps:

Step 1. Previous methods randomly sample a subset of keys from the dataset to estimate the distribution of keys. They use these samples to determine partition boundaries, which are then used to divide the dataset into smaller chunks that fit better in cache and

Tab. 1: Data shuffling times of NUMA-aware LSB Radix-Sort

Radix Bit	16	32	64	16	32	64
# Cores	Data Shuffling Time (ms)			Sorting Time Percentage (%)		
8	39	47	37	20	16	7
16	38	35	40	29	19	12
32	23	28	36	31	24	14
64	19	22	20	32	27	15

can be processed in parallel.
Our optimization: We eliminate the need for sampling by partitioning the input data based on the level of parallelism.

Step 2. Schedule threads to ensure they are located within the same NUMA node as the chunks they process.
Our optimization: As with the Radix-Sort, we initialize a thread scheduling strategy that considers chiplet boundaries to enable access to the local chiplet caches.

Step 3. Each thread generates a local histogram of the keys in its assigned partition. These histograms help determine the number of keys falling into each range.

Step 4. Calculate partition offsets using the histograms to determine where each bucket of keys starts in the final sorted array.

Step 5. After the partitioning phase, previous methods may shuffle data to balance the number of keys among the NUMA domains.
Our optimization: We avoid data shuffling. The data load is balanced during Steps 1 and 2.

Step 6. Use a SIMD-optimized comb sort algorithm to sort keys in parallel within each partition. The sorting is done in-place within each partition to maintain cache locality.

After each partition is individually sorted, merge the sorted partitions to form the final sorted array.

4.3 Data shuffling vs. core affinity

Data shuffling and thread-to-core binding are two techniques used in parallel computing to optimize performance. On the one hand, data shuffling redistributes data across different processing units to balance the workload's data and task distribution, thereby improving efficiency.

On the other hand, using core affinity binds threads to specific CPU cores. This enables tasks to take advantage of the benefits of locality and the reduced latency of the local (chiplet) cache. While data shuffling enhances flexibility and adaptability in dynamic environments, core binding offers stability and improved cache performance in scenarios where tasks have predictable and consistent workloads.

Traditional NUMA-aware algorithms often shuffle data to balance the load. Partitions are generated from the unordered input and allocated to different NUMA nodes. After creating histograms and calculating offsets, each partition is sorted locally within its respective NUMA node. Finally, data is shuffled between NUMA nodes to achieve further balancing. Some methods attempt to ensure that each data item crosses NUMA boundaries only once per pass, even if it requires reorganizing the entire array [38]. Our approach omits the shuffling and solely relies on binding tasks to CPUs and chiplets from the start. This keeps processing local,

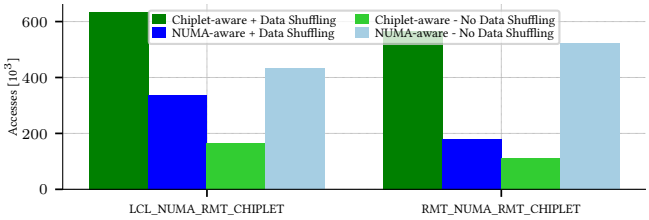


Fig. 7: Accesses to remote chiplet cache in local and remote NUMA nodes during 32-bit Radix-Sort with 1 billion tuples.

avoids the overhead of frequent data movement (including between chiplets), and enhances efficiency.

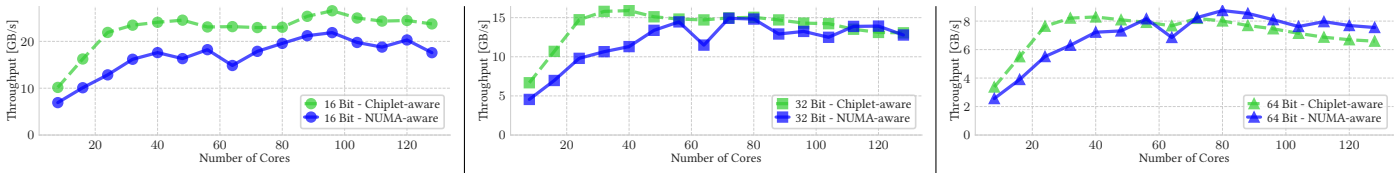
Tab. 1 shows the data shuffling times and the percentage of time the sorting algorithm spends on data shuffling, categorized by the number of cores used and different radix bit sizes. The table reveals that a significant percentage of the total sorting time is devoted to data shuffling, with this percentage increasing as the number of cores rises. For example, with 8 cores, data shuffling accounts for up to 20% of the total sorting time, while with 64 cores, this percentage increases to 32%.

4.4 Chiplet-aware scheduling

To the best of our knowledge, no sorting algorithm accounts for the partitioned L3 cache in modern chiplet processors. As explained in §2.3, each chiplet is equipped with its own local partition of the L3 cache, and fetching data from other chiplets incurs inter-chiplet communication overhead.

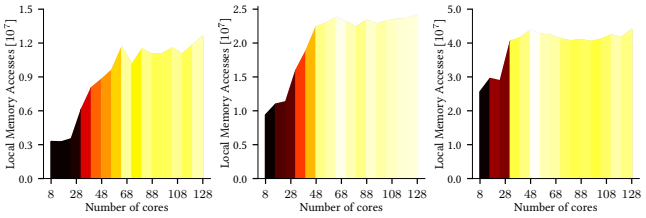
Therefore, when deciding on a placement policy, the task scheduler must consider the available cache size, the chiplet's share of the L3 cache, and the working set size. If the required degree of parallelism exceeds the number of cores in a single chiplet or if the data size surpasses a single L3 cache but fits within the combined L3 caches, tasks are distributed across different chiplets to optimize cache usage. When the data size exceeds the total L3 cache capacity, we ensure that cores use only local main memory to avoid expensive remote NUMA accesses.

Fig. 7 shows the number of accesses to remote chiplet caches in both local and remote NUMA nodes during the Radix-Sort. We measured cache and memory accesses using the *libpfm* library, which allows us to monitor specific hardware performance events. We tracked the `ANY_DATA_CACHE_FILLS_FROM_SYSTEM` event, which captures data cache fills from various levels of the memory hierarchy. We further specified the source of these fills to distinguish between fills from a remote chiplet in the same NUMA node and fills from a remote chiplet in a different NUMA node. We captured all cache fills, regardless of whether they were triggered by demand loads or prefetch requests. Notably, the chiplet-aware approach that shuffles the data shows the highest number of accesses, with 630 K in local NUMA and 562 K in remote NUMA. In contrast, the approach without data shuffling significantly reduces these accesses to 162 K and 111 K, respectively. The NUMA-aware strategies display intermediate results, with the variant that does not employ data shuffling showing an increase in remote accesses, reaching 521 K. These findings suggest that data shuffling, while potentially beneficial in NUMA-aware configurations, negatively impacts cache usage in chiplet-aware configurations due to the high number of chiplets.



(a) 16 Bit Radix-Sort, 100M Tuples **(b)** 32 Bit Radix-Sort, 100M Tuples **(c)** 64 Bit Radix-Sort, 100M Tuples

Fig. 8: LSB Radix-Sort Scalability



(a) 16 Bit Radix **(b)** 32 Bit Radix **(c)** 64 Bit Radix

Fig. 9: Radix-Sort: memory accesses when varying core counts.

The higher number of remote cache accesses for the NUMA-aware approach without shuffling, compared to with shuffling, is due to suboptimal data distribution and access patterns. Without shuffling, memory accesses concentrate in specific regions, leading to increased contention and cache conflicts. This concentration forces the system to fetch data from remote caches more frequently, resulting in higher remote cache access counts.

In contrast, shuffling in the NUMA-aware approach distributes memory accesses more evenly across NUMA nodes. This even distribution reduces hotspots and execution time, thereby minimizing remote cache accesses.

4.5 Further optimizations

We also applied the following optimizations:

We introduced prefetching during the histogram creation and partitioning phases to optimize memory access patterns. We used the following instruction:
`_mm_prefetch(void* mem, _MM_HINT_T0).`

Furthermore, we improved branch prediction during the partitioning phase by simplifying the loop structure. Specifically, we separate loops for aligned and unaligned data and minimize the depth of nested branches. We also use the `__builtin_expect` hint to inform the compiler which branches are more likely to be taken, as the loop for misaligned data is less likely to be used.

The source code for this project is accessible via our GitHub repository: https://github.com/Alessandro727/Chiplet-aware-sorting-algorithms.

5 EXPERIMENTAL EVALUATION

We conducted a range of experiments to evaluate the behavior of our approach on modern chiplet-based machines. Specifically, we answer the following questions:

- **Q1:** What is the impact of chiplet-aware optimizations on the performance of Radix-Sort and Comparison-Sort in chiplet-based processors? (§5.2, §5.3 and §5.4)
- **Q2:** How do they scale with increased data size? (§5.5)
- **Q3:** How is the performance affected by skew? (§5.6)
- **Q4:** How does the performance vary with different task placement strategies? (§5.7)
- **Q5:** What does the ablation study show about the relative performance improvements of different optimization strategies? (§5.8)

5.1 Experimental setup

The experiments are conducted on a dual-socket AMD EPYC Milan 7713 processor. Each socket features 64 CPU cores, 512 GB of RAM, and 8 chiplets, each with 32 MB of L3 cache.

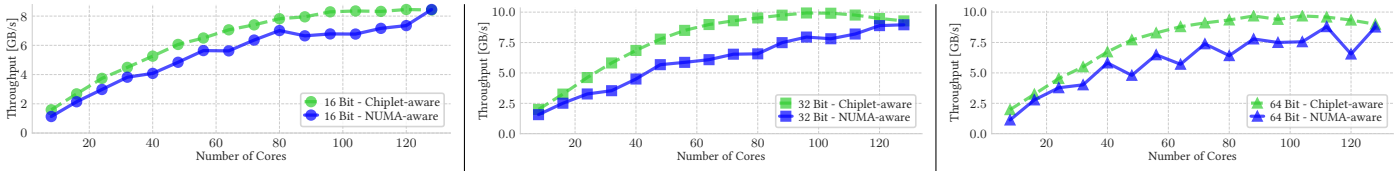
The operating system is Ubuntu 23.04, and the compiler used is GCC 12 with **-O3** optimization. We utilize SSE (128-bit SIMD) instructions on AVX registers (256-bit). Our primary focus is to analyze the impact of chiplet architectures on parallel sorting, rather than explore SIMD/AVX/AVX2 optimizations. We maintained the use of SSE instructions, consistent with the baseline NUMA-aware implementation, to isolate the effects of our chiplet-aware optimizations. When comparing chiplet-aware and NUMA-aware approaches, we ensure that each approach has access to the same hardware resources, including L3 cache and memory controllers, allowing each implementation to manage these resources independently.

We measured throughput in terms of GB/s (instead of tuples/s) to account for the varying tuple sizes and the memory bandwidth utilization of the sorting algorithm. Unless stated otherwise, the input data is uniformly random, the experiments are conducted with 1 billion tuples and 16 cores, and the results represent the average of 10 executions.

5.2 Radix-Sort

Fig. 8 shows the scalability of our Radix-Sort implementation with 16-bit, 32-bit, and 64-bit keys, respectively, comparing two policies (i.e., chiplet-aware and NUMA-aware) when varying the number of cores. For all key sizes, the chiplet-aware policy consistently outperforms the NUMA-aware policy. Notably, with just 32 cores, the chiplet-aware implementation achieves a throughput of 24 GB/s, surpassing the 16.2 GB/s achieved by the NUMA-aware approach. This trend is also observed for the 32-bit and 64-bit keys, as shown in Fig. 8b and Fig. 8c.

The limited scalability beyond 24 cores is due to increased contention for shared cache resources. As more threads are assigned per chiplet, each thread has access to a smaller portion of the local cache and thus resorts to more memory accesses, becoming memory bound. We measured the algorithm's main memory accesses as we varied the number of cores and key sizes, and the results are shown in Fig. 9. Notably, main memory access remains constant or slightly increases up to 24 cores (when the scalability slows down),



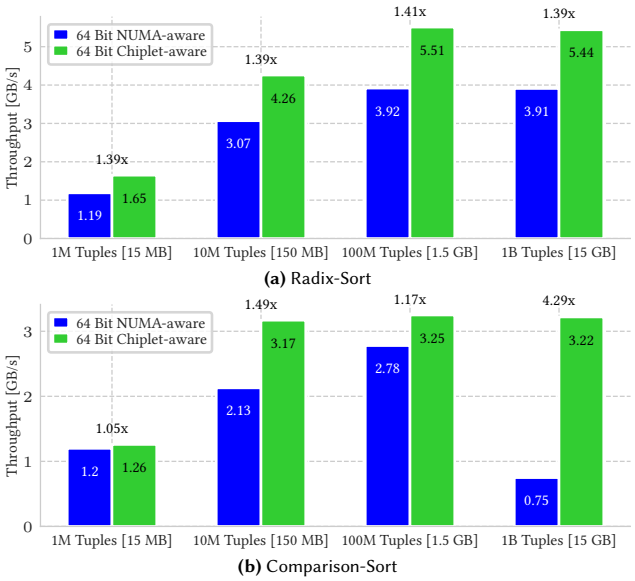
(a) 16 Bit Comparison-Sort, 100M Tuples **(b)** 32 Bit Comparison-Sort, 100M Tuples **(c)** 64 Bit Comparison-Sort, 100M Tuples

Fig. 10: Comparison-Sort Scalability

before increasing significantly across all key sizes. This, in turn, results in increased pressure on the memory subsystem.

5.3 Comparison-Sort

As with the Radix-Sort evaluation, we measure the performance of the Comparison-Sort implementation using 16-bit, 32-bit, and 64-bit keys on both the chiplet-aware and NUMA-aware policies, varying the core count. The results are shown in Fig. 10. In all cases, the chiplet-aware policy demonstrates superior performance and scalability compared to the NUMA-aware policy. For 16-bit keys, the chiplet-aware approach achieves a peak throughput of 8.3 GB/s at 96 cores, outperforming the NUMA-aware policy's 6.7 GB/s at the same core count. Overall, this translates to an average speedup of 1.23× between 8 and 96 cores, with a maximum speedup of 1.41×. This trend persists for 32-bit and 64-bit keys. With 32-bit keys, we observe an average speedup of 1.40× and a maximum speedup of 1.64×. For 64-bit keys, the chiplet-aware approach delivers an overall speedup of 1.34× between 8 and 96 cores, peaking at 1.61×. Additionally, we note that the NUMA-aware policy exhibits highly variable results, which we identified as being due to imperfect load balancing within the same NUMA node.



(a) Radix-Sort

(b) Comparison-Sort

Fig. 11: Throughput [GB/s] when varying input data size.

5.4 Radix-Sort vs. Comparison-Sort: performance comparison

The performance characteristics of Radix-Sort and Comparison-Sort algorithms vary significantly depending on the width of the integers being sorted. For 16-bit and 32-bit integers, Radix-Sort demonstrates superior overall performance, while Comparison-Sort shows better scalability. However, this dynamic shifts when dealing with 64-bit integers, where Comparison-Sort outperforms Radix-Sort. The reversal in performance for 64-bit integers is attributed to the inherent complexities of Radix-Sort when handling larger data types. Radix-Sort requires multiple passes through the data, creating histograms and partitioning the elements in each pass. As the integer width increases, so does the number of passes required, leading to increased overhead and processing time. This overhead becomes particularly pronounced with 64-bit integers, significantly impacting the algorithm's throughput. In contrast, Comparison-Sort benefits from a more streamlined approach. After a single histogram creation and partitioning phase, Comparison-Sort can efficiently sort the keys within the cache. This behavior allows it to maintain higher performance levels when sorting 64-bit integers, as it avoids the repeated histogram creation and partitioning steps that burden Radix-Sort.

The throughput of both sorting algorithms falls short of the machine's peak memory bandwidth. This shortfall is primarily attributed to the computational overhead during the partitioning phase, where managing buffers and calculating partition functions reduce the available time for pure memory operations. Furthermore, partitioning to multiple outputs induces TLB thrashing, which degrades performance. For large arrays, this issue is further exacerbated by cache misses. These factors collectively hinder the partitioning operation from fully exploiting the system's sequential memory bandwidth. While our approach mitigates some of these issues, it cannot entirely eliminate their impact on achieving maximum throughput.

5.5 Impact of data size

To assess the effectiveness of our chiplet-aware policy, we analyzed the throughput of the Radix- and Comparison-Sort implementations across various input data sizes. Fig. 11 shows the results for data sizes ranging from 1 million tuples (equivalent to 15 MB) to 1 billion tuples (equivalent to 15 GB). Specifically, we evaluated the 64-bit implementations of the NUMA- and chiplet-aware policies.

An immediate observation from Fig. 11a is that the chiplet-aware policy consistently outperforms the NUMA-aware policy across all input sizes by approximately 40%. We attribute the superior performance of the chiplet-aware policy to optimized data locality and reduced inter-core communication overhead. This optimization leads to higher bandwidth, especially as the input size increases.

Fig. 11b shows the results for the Comparison-Sort, which indicate a more mixed outcome. For smaller input sizes, such as 1

Tab. 2: Total main memory accesses (10^3).

	Size (#Tuples)	Local Memory	Remote Memory
NUMA-aware	100 Million	39683	12492
	1 Billion	3236810	652202
Chiplet-aware	100 Million	11681	20084
	1 Billion	140405	218315

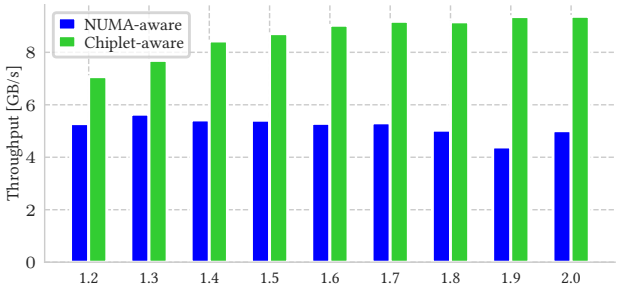


Fig. 12: Impact of skew: 32b Radix-Sort, θ in Zipf distribution.

million tuples (15 MB), the throughput for both policies is comparable. As the input size increases to 10 million tuples (150 MB), the chiplet-aware policy outperforms the NUMA-aware policy by 17-40%. However, for the largest input size of 1 billion tuples (15 GB), the NUMA-aware policy severely deteriorates. This drop in performance for the NUMA-aware policy at larger input sizes suggests inefficiencies in data distribution and memory access patterns, which are better managed by the chiplet-aware policy. For example, in Tab. 2, we display the number of main memory accesses for both policies when using 100 million and 1 billion tuples. On one hand, the chiplet-aware policy shows an increase in main memory accesses that is proportional to the increase in input data size, approximately 10 times higher. On the other hand, the NUMA-aware policy significantly increases the number of memory accesses, with remote memory accesses increasing by about 52× and local memory accesses by about 81×.

5.6 Impact of skew

We now assess the impact of data skew on performance and examine the throughput of the 32-bit Radix-Sort implementation while varying the skew parameter θ in the Zipf distribution. Fig. 12 shows the results of the two evaluated policies. Once again, we observe that the chiplet-aware policy outperforms the NUMA-aware policy. While the chiplet-aware policy can sustain throughput levels (even increasing from 6.9 GB/s to 9.2 GB/s), the throughput of the NUMA-aware policy drops from 5.1 GB/s to 4.2 GB/s. We attribute the improving trend for the chiplet-aware policy to enhanced locality in the cache-aware implementation. More specifically, profiling shows that the cache miss rate drops from 6% to 1.5% when increasing θ from 1.2 to 2. This experiment underscores the robustness of the chiplet-aware policy in managing data locality, making it more effective at handling data skew.

5.7 Impact of scheduling

We also evaluate how the performance of our chiplet-aware approach varies with different scheduling strategies. In §2.3, we discussed how bandwidth varies depending on the type of scheduling—either by assigning tasks to cores belonging to the fewest

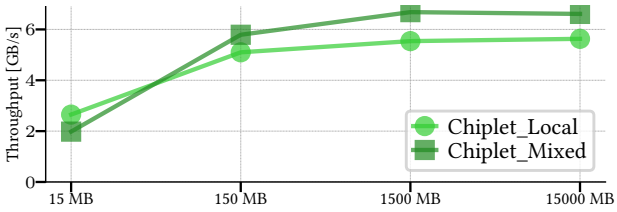


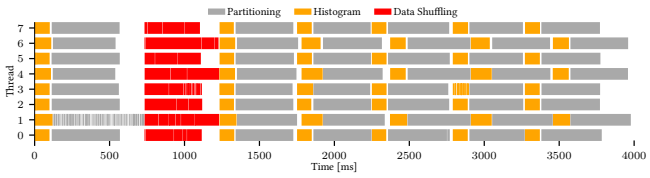
Fig. 13: Impact of scheduling on 32-bit Radix-Sort, 8 threads

chiplets possible (Chiplet_Local) or by assigning tasks to cores across as many chiplets as possible (Chiplet_Mixed). Fig. 13 shows the throughput of the Radix-Sort algorithm with 32-bit keys, evaluated using the two mentioned scheduling strategies on an 8-core setup, as in the experiment in §2.3.

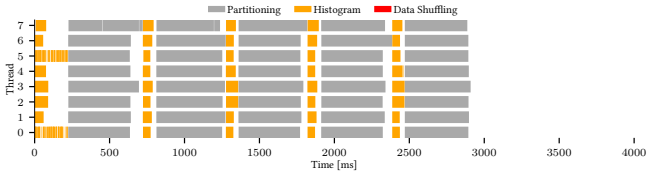
For smaller data sizes, the Chiplet_Local strategy yields higher bandwidth, leveraging the proximity of tasks to maximize throughput, as 15 MB fits in a single chiplet cache. As data sizes increase, the Chiplet_Mixed policy becomes more effective, exploiting the larger L3 cache available across multiple chiplets. These results are consistent with the analysis presented in §2.3, underscoring the importance of selecting the appropriate scheduling strategy based on the data size to optimize performance.

Fig. 14 compares the thread timeline activity for the LSB Radix-Sort algorithm using chiplet-aware and NUMA-aware scheduling approaches. With the chiplet-aware scheduling timeline, the threads exhibit a more concentrated and synchronized utilization pattern: the workload distribution is balanced across all threads, reducing idle times and enhancing efficiency.

In contrast, the NUMA-aware scheduling timeline shows a broader and more dispersed thread activity after the data shuffling operation. The spread indicates a less uniform distribution of workload, which can lead to increased latency and inefficiency. The NUMA-aware scheduling struggles with maintaining an even workload distribution due to the heterogeneity introduced by chiplets.



(a) NUMA-aware LSB Radix-Sort

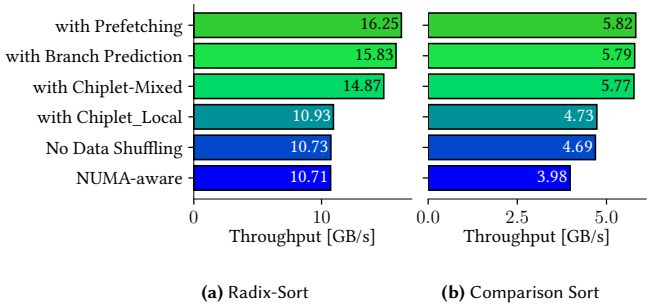


(b) Chiplet-aware LSB Radix-Sort

Fig. 14: Thread timelines of LSB Radix-Sort.

5.8 Ablation study

Finally, we performed an ablation study to evaluate the impact of each optimization strategy on the performance of our chiplet-aware implementations. We present the results in Fig. 15 for both the Radix-Sort and Comparison-Sort algorithms using 32-bit keys



(a) Radix-Sort **(b)** Comparison Sort

Fig. 15: Performance optimizations.

and 100 million tuples on a 32-core setup. Both algorithms experienced the greatest performance boost from using the Chiplet_Mixed scheduling and eliminating the data shuffling phase. Specifically, Radix-Sort's throughput increased from 10.71 GB/s to 14.87 GB/s, while Comparison-Sort's throughput increased from 3.98 GB/s to 5.77 GB/s. Additionally, prefetching and improved branch prediction increased Radix-Sort's performance by about 15%, but had no significant effect on Comparison-Sort.

6 ADDITIONAL RELATED WORK

Here we provide more information on prior work done in the context of chiplet-based CPUs, more general optimizations for hardware-conscious algorithm implementations, and on parallel sorting algorithms.

Chiplet-based CPUs. Industry documentation and product manuals from AMD and Intel provide comprehensive details on chiplet based CPU designs [30, 32, 34]. For practical insights, Velten et al. conducted thorough tests on the memory hierarchies of AMD EPYC Rome and Intel Xeon Cascade Lake SP processors [44]. Suggs et al. examined the Zen 2 architecture [43], while Schöne et al. focused on the energy efficiency aspects [42]. Naffziger et al. discussed multi-die chiplet configurations [33] while Chirkov et al. evaluated the performance of the interconnects and introduced Meduza, a write-update coherence protocol for chiplet systems [42].

Hardware-conscious algorithms and optimizations. Satish et al. [41] introduced methods for efficient out-of-cache partitioning, while Wassenberg et al. [45] highlighted the importance of software-based write-combining. Manegold et al. [29] addressed the TLB thrashing issue caused by partitioning into a large number of outputs and suggested using cache-resident hash tables for partitioning in join operations, a design later adopted by Kim et al. [25] for multi-core CPUs. Wu et al. [46] proposed hardware-accelerated partitioning to enhance both performance and power efficiency.

Sorting Algorithms. Parallel sorting algorithms have been widely studied, with advancements in both radix and comparison-based sorting approaches. Obeya et al. [36] introduced Regions Sort, a new parallel in-place radix sorting algorithm that uses a graph structure to model dependencies among elements, generating independent tasks executed in parallel. Cho et al. [16] proposed PARADIS, addressing memory overhead and load imbalance in parallel in-place radix sort. It uses speculative permutation and distribution-adaptive load balancing to permute array elements into buckets in parallel and in-place. Axtmann et al. [9] presented IPS2Ra, a sequential and parallel in-place radix sort algorithm. IPS2Ra introduces a scalar approach that processes multiple keys simultaneously to improve instruction-level parallelism.

There has also been significant prior work on parallel algorithms for comparison sorting [8, 9, 12, 17, 22, 40]. One of the most recent is IPS4o by Axtmann at al. that implements an in-place parallel samplesort [9] and compares it with IPS2Ra. Rasmussen et al. [40] developed TritonSort, a balanced large-scale sorting system designed to handle massive datasets efficiently. TritonSort focuses on balancing I/O and computation to achieve high throughput and scalability. Goodrich et al. [21] implemented parallel sorting algorithms for sorting with comparison errors in the persistent and non-persistent models. Pasetto et al. [37] conducted a comparative study of parallel sorting algorithms on multi-core hardware, evaluating unstable methods like Mapsort and Parallel Quicksort, providing insights into hardware restrictions. Dong et al. [18] introduced DovetailSort, a theoretically-efficient parallel integer sorting algorithm that effectively detects and handles duplicate keys.

7 EXTENDING CHIPLET-AWARE OPTIMIZATIONS

While our study focuses on sorting algorithms, the chiplet-aware optimizations we propose have potential applications across a wide range of High-Performance Computing (HPC) workloads. Many HPC applications share similar characteristics with sorting algorithms, such as being memory-intensive and requiring efficient data movement and processing across multiple cores. For instance, in large-scale graph processing, chiplet-aware scheduling can group closely connected nodes within the same chiplet, reducing cache misses and speeding up traversal. For real-time streaming analytics, dedicating specific chiplets to different stages of the processing pipeline enables parallel operations while maintaining low latency.

Overall, the principles underlying our chiplet-aware optimizations—such as aligning data partitions with chiplet boundaries, extending the memory hierarchy to account for partitioned L3 caches, and scheduling tasks based on data size relative to cache capacities—can be adapted to general highly parallel applications. Future work could explore specific implementations and performance gains in these diverse memory-intensive domains, potentially leading to a more general framework for chiplet-aware algorithm design in high-performance computing.

8 CONCLUSION

We evaluated the impact of chiplet-based architectures on sorting algorithms, emphasizing the necessity of finer granularity when allocating resources between sorting threads and the importance of considering the partitioned nature of the L3 cache. We propose an approach that achieves higher performance compared to widely applied NUMA-aware solutions.

Through extensive experiments, we demonstrate that careful chiplet-aware placement of tasks and in-cache resources can significantly enhance sorting performance, eliminating the need for costly data shuffling. Overall, our findings provide valuable insights for designing internal operations for databases and query engines on novel chiplet-based processors.

REFERENCES

[1] 2024. Analyzing Unconventional Logic Semiconductors – A Shift Away from Semiconductor Manufacturers. https://hacarus.com/ai-lab/03312022-graviton3/. Accessed: 2024-3-1.

[2] 2024. A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison and Radix-Sort. http://www.cs.columbia.edu/~orestis/sigmod14Isource.zip. Accessed: 2024-3-1.

[3] 2024. Configuring NUMA for SingleStore. https://support.singlestore.com/hc/en-us/articles/360058633252-Configuring-NUMA-for-SingleStore. Accessed: 2024-3-1.

[4] 2024. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html. Accessed: 2024-3-1.

[5] 2024. NUMA Balancing. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing. Accessed: 2023-6-19.

[6] 2024. numactl(8) — Linux manual page. https://man7.org/linux/man-pages/man8/numactl.8.html. Accessed: 2024-5-12.

[7] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *ArXiv* abs/1207.0145 (2012). https://api.semanticscholar.org/CorpusID:1084527

[8] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. 2014. Practical Massively Parallel Sorting. *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2014). https://api.semanticscholar.org/CorpusID:18249978

[9] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Sept. 2020. Engineering In-place (Shared-memory) Sorting Algorithms. Computing Research Repository (CoRR). arXiv:2009.13569

[10] C Balkesen, J Teubner, G Alonso, and M T Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373.

[11] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Symposium on Operating Systems Principles*.

[12] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallelism in Randomized Incremental Algorithms. *Journal of the ACM (JACM)* 67 (2018), 1 – 27. https://api.semanticscholar.org/CorpusID:12349988

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an China). ACM, New York, NY, USA.

[14] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2018. Cache-Oblivious and Data-Oblivious Sorting and Applications. In *ACM-SIAM Symposium on Discrete Algorithms*. https://api.semanticscholar.org/CorpusID:3956052

[15] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep K. Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.* 1 (2008), 1313–1324. https://api.semanticscholar.org/CorpusID:10029522

[16] Minsik Cho, Daniel Brand, Rajesh R. Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. 2015. PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort. *Proc. VLDB Endow.* 8 (2015), 1518–1529. https://api.semanticscholar.org/CorpusID:8888350

[17] Richard J. Cole and Vijaya Ramachandran. 2010. Resource Oblivious Sorting on Multicores. *ACM Transactions on Parallel Computing (TOPC)* 3 (2010), 1 – 31. https://api.semanticscholar.org/CorpusID:3182449

[18] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (2024). https://api.semanticscholar.org/CorpusID:266693888

[19] Fabien Gaud, Baptiste Lepers, Justin R. Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of memory management on modern NUMA systems. *Commun. ACM* 58 (2015), 59 – 66.

[20] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. 2007. CellSort: High Performance Sorting on the Cell Processor. In *Very Large Data Bases Conference*. https://api.semanticscholar.org/CorpusID:13349932

[21] Michael T. Goodrich and Riko Jacob. 2023. Optimal Parallel Sorting with Comparison Errors. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (, Orlando, FL, USA,) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 355–365. https://doi.org/10.1145/3558481.3591093

[22] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 2007. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)* (2007), 189–198. https://api.semanticscholar.org/CorpusID:5695020

[23] Daniel Jiménez-González, Juan J. Navarro, and Josep-Lluís Larriba-Pey. 2003. CC-Radix: a cache conscious sorting based on Radix sort. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.* (2003), 101–108. https://api.semanticscholar.org/CorpusID:16995920

[24] T Kiefer, B Schlegel, and W Lehner. 2013. *Experimental Evaluation of NUMA Effects on Database Management Systems*. BTW.

[25] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep K. Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2 (2009), 1378–1389. https://api.semanticscholar.org/CorpusID:6529485

[26] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. 2014. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *ADMS@VLDB*.

[27] Anthony LaMarca and Richard E. Ladner. 1997. The influence of caches on the performance of sorting. *J. Algorithms* 31 (1997), 66–104. https://api.semanticscholar.org/CorpusID:206567217

[28] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:854221

[29] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Very Large Data Bases Conference*. https://api.semanticscholar.org/CorpusID:40981562

[30] Michael Mattioli. 2021. Rome to Milan, AMD Continues Its Tour of Italy. *IEEE Micro* 41, 4 (2021), 78–83. https://doi.org/10.1109/MM.2021.3086541

[31] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. http://www.cs.virginia.edu/stream/.

[32] Samuel D. Naffziger, Noah Beck, Thomas D. Burd, Kevin M. Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 57–70. https://api.semanticscholar.org/CorpusID:235415451

[33] Samuel D. Naffziger, Kevin M. Lepak, Milam Paraschou, and Mahesh Subramony. 2020. 2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products. *2020 IEEE International Solid- State Circuits Conference - (ISSCC)* (2020), 44–45. https://api.semanticscholar.org/CorpusID:215800319

[34] Nevine Nassif, Ashley Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexander M. Kern, William J. Bowhill, David Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad MinHazul Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. *2022 IEEE International Solid- State Circuits Conference (ISSCC)* 65 (2022), 44–46. https://api.semanticscholar.org/CorpusID:247523158

[35] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[36] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (2019). https://api.semanticscholar.org/CorpusID:190230549

[37] Davide Pasetto and Albert Akhriev. 2011. A comparative study of parallel sort algorithms. In *OOPSLA Companion*. https://api.semanticscholar.org/CorpusID:207191141

[38] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014). https://api.semanticscholar.org/CorpusID:15344092

[39] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *Proc. VLDB Endow.* 5 (2012), 1447–1458.

[40] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. 2011. TritonSort: A Balanced Large-Scale Sorting System. In *Symposium on Networked Systems Design and Implementation*. https://api.semanticscholar.org/CorpusID:8326819

[41] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep K. Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010). https://api.semanticscholar.org/CorpusID:14972686

[42] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. 2021. Energy Efficiency Aspects of the AMD Zen 2 Architecture. *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (2021), 562–571. https://api.semanticscholar.org/CorpusID:236772121

[43] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40 (2020), 45–52. https://api.semanticscholar.org/CorpusID:214005391

[44] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (2022). https://api.semanticscholar.org/CorpusID:247681823

[45] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-core Radix Sort. In *European Conference on Parallel Processing*. https://api.semanticscholar.org/CorpusID:28042507

[46] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating big data with high-throughput, energy-efficient data partitioning. *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013). https://api.semanticscholar.org/CorpusID:16078400