Sparse Polynomial Multiplication on Chiplet-based Architectures

Miriam Okutuo and Alexander Brandt[0000−0002−1294−9710]

Faculty of Computer Science, Dalhousie University, Halifax, Canada
{mokutuo,abrandt}@dal.ca

Abstract. Chiplet-based architectures are relatively new designs for CPUs which move beyond the traditional monolithic design, allowing for many more cores to be packaged on a single CPU. For example, from 8-16 cores typical of a monolithic design up to an astonishing 288 cores. However, one hidden concern is that this new design distributes cores and their L3 cache across individual integrated circuits called chiplets, leading to non-uniform memory access. Cores on the same chiplet can communicate effectively meanwhile a core will suffer increased latency retrieving data from an L3 cache partition on another chiplet.
In this article we give a brief overview of chiplet architectures and their design, considering implications on software and developers. We present a parallel algorithm for sparse polynomial multiplication which accounts for the distributed L3 cache on chiplet-based architectures. In particular, tasks must avoid inter-chiplet communication while exploiting caching effects on a single chiplet. We discuss composition of parallel code regions, thread pinning, and thread cooperation. Our implementation achieves above linear speed-up up to 64 cores.

Keywords: sparse polynomials · parallel algorithm · hardware-aware algorithm · non-uniform memory access

1 Introduction

Sparse polynomial arithmetic is a fundamental operation in computer algebra underpinning the majority of other mid-level and high-level algorithms. As fundamental objects to manipulate in computer algebra, polynomials are a natural area for optimization in the design and implementation of a computer algebra library or system [8, 2, 26]. Sparse polynomials, those whose representation stores only terms with non-zero coefficients, are particularly important in the multivariate case where storing them densely would require huge amounts of memory. Nonetheless, due to the notorious *expression swell* faced by computer algebra [13], these sparse polynomials can still grow exceptionally large both in the number of terms and the size of their coefficients. We are particularly motivated in the case of solving systems of equations where polynomials can become exponentially large compared to the input polynomials [6, 9].

Past studies have examined efficient data structure design for sparse polynomials [12, 26] as well as their arithmetic [25, 2]. But, as polynomials grow even larger, parallel algorithms are needed to improve performance and best take advantage of the available computational power in multi-core computer architectures. Parallel sparse polynomial multiplication [4, 10, 23] and division [24, 11] algorithms have previously been studied. However, modern multi-core architectures have changed since these algorithms have been developed and new concerns must be addressed.

Now ubiquitous in even the most basic consumer hardware, multi-core architectures offer multiple cores of execution enabling parallel computing. Since the mid 2000s, computer architecture has increasingly moved toward putting more cores onto a single processor [21]. Anywhere from 2 to 16 cores was common up to the early 2020s. Recently, however, the number of available cores has increased drastically. Intel's Xeon 6+ CPUs, to be released in 2026, offer up to 288 cores in a single chip [17]. A key technology enabling this growth are *chiplets*: small integrated circuits which are combined to create a single multi-chip module that acts as a single processor.

Intel often refers to chiplets as tiles while AMD maintains the chiplet terminology. Nearly every modern CPU is based on chiplets, including laptop, desktop, and server processors [5, 15, 27].

Chiplets overcome many manufacturing limitations faced by traditional monolithic CPUs including lithographic reticle limits (the size of an integrated circuit) and yield (proportion of manufactured integrated circuits that operate nominally). They also provide many economic advantages for manufacturers. However, "chiplets are not a free lunch" [27]. A key consideration is inter-chiplet communication. In chiplet-based CPUs, the cores and shared L3 cache are distributed across multiple chiplets. Latency is higher and bandwidth is lower between chiplets compared to intra-chiplet communication. This leads to non-uniform memory access (NUMA). Cores on the same chiplet can communicate much more effectively than cores on different chiplets, posing challenges for thread synchronization and data locality in multithreaded programs.

For multithreaded programs to take advantage of these modern chiplet-based architectures, additional care is needed. Programmers can no longer rely on simplified hardware/software abstractions, they must tailor software to the executing hardware. This is a key element of software performance engineering [21].

In this work, we will explore an algorithm for parallel sparse polynomial multiplication tailored to modern chiplet-based computer architectures. We propose an algorithm which is aware of the cache partitioning between chiplets to exploit data locality within a single chiplet. Our algorithm makes use of hierarchical or nested parallelism where parallel regions are contained within other parallel regions. This nested scheme has seen success in parallelizing other algorithms in computer algebra such as in the manipulation of power series [7] or in triangular decomposition [3].

The remainder of this article is organized as follows. We discuss the memory hierarchy, chiplet-based architectures, and implications for software in Section 2. Next, Section 3 presents our parallel sparse multiplication algorithm for chiplet-based processors. Experimental data is presented in Section 4. Finally, conclusions and future work are discussed in Section 5.

2 Memory Hierarchy and Chiplet Architectures

Modern chiplet-based architectures provide a large number of cores but those cores, and their caches, are distributed across several chiplets. This disrupts the typical view of the memory hierarchy and the shared-memory programming model.

The memory hierarchy is a foundational element to the design of computer systems rooted in the *principle of locality* [16]. The memory hierarchy divides memory into levels where higher levels of cache memory are smaller and faster than the lower levels of backing memory. The need for the memory hierarchy arises primarily from the *memory wall*—the huge gap in speeds between fast processors and slow DRAM main memory. Higher levels of cache can serve data to processors with lower latency than lower levels. The memory hierarchy almost always consists of three levels of cache (L1, L2, L3) between the processor and main memory.

The memory hierarchy implies that programmers should access memory carefully to avoid paying the penalty of a cache miss, when the data requested is not available in cache and must be retrieved from slower backing memory. Programmers should employ data locality to ensure that recently accessed data is accessed again soon (temporal locality) and that data accessed should be nearby recently accessed data (spatial locality).

In multi-core processors the cache hierarchy is made even more complex. Typically, the "last-level cache" (L3 cache) is shared between all the cores meanwhile each core has its own private L1 and L2 caches [16]. *Cache coherency* thus becomes an issue where multiple copies of a piece of data must be kept in agreement across private caches. Cache coherency protocols ensure that these copies always agree. The shared last-level cache and its coherency protocol are the hardware mechanisms which allow atomic operations and, thus, synchronization in multithreaded programs. Put simply, the cores communicate with each other at the speeds of their shared L3 cache.

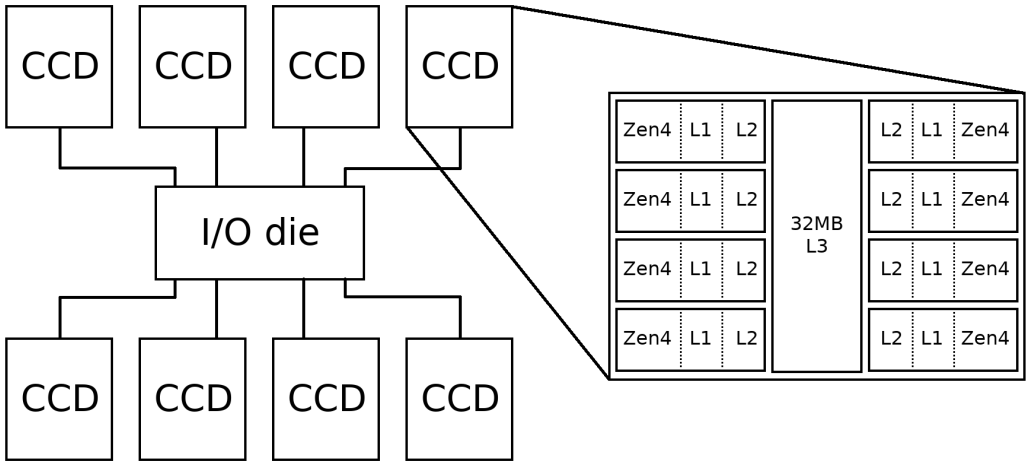

Fig. 1. The chiplet configuration of the AMD EPYC 9554 showing 8 core-complex dies (CCDs) around the central I/O die. Each CCD holds eight Zen 4 cores.

Multi-core processors have long been developed symmetrically, where each core has the same performance and can uniformly access their shared memory. This symmetry is a basic assumption of the machine model typical of parallel programming (see, e.g., [22]). Theoretical models of parallel computation, like PRAM (Parallel Random Access Machine) [14], make this symmetry explicit where processors operate in lockstep and each processor has uniform memory access. However, modern computer architectures feature many asymmetries. Nearly every recent mobile or desktop processor combines some variation of "performance" cores and low-power "efficient" cores, leading to challenges in load balancing and obtaining good parallel speedup in a multithreaded algorithm. On the other hand, chiplet-based processors distribute their cores and L3 cache across chiplets leading to non-uniform memory access (NUMA) due to inter-chiplet communication.

2.1 Chiplet-Based Architectures

Multi-chip modules and the idea of disintegrating processor components into multiple dies is not a new concept. However, AMD became the first to bring chiplets to the mass market with their Zen 2 microarchitecture released in 2019 [27]. Intel soon followed suit and released their first chiplet-based CPUs in 2023 with their 4th generation Xeon processors [5]. The basic principle behind chiplets is that it is easier to construct small chiplets and put them together than it is to construct one large monolithic processor.

Figure 1 shows the chiplet-based architecture of the 64-core AMD EPYC 9554 processor. Each compute chiplet is called a core-complex die (CCD) and contains 8 cores and a 32MB partition of L3 cache. The CCDs are connected to another chiplet called the I/O die and communicate with each other and main memory through it. The number of cores and size of cache on a CCD varies by microarchitecture generation. The number of CCD chiplets on a single processor varies across AMD's product line.

Figure 2 shows the core-to-core latency of the EPYC 9554. For each core, it shows the time to exchange data with another core using the cache coherency protocol. A clear pattern emerges: communications within a chiplet are up to 4 times faster than across chiplets. This pattern is common across all chiplet-based AMD processors to date, who all share a hub-and-spoke topology for their chiplets. This has a significant implication for L3 caches. Despite the EPYC 9554 offering a total of 256MB of L3 cache, each core only has fast access to the 32MB partition

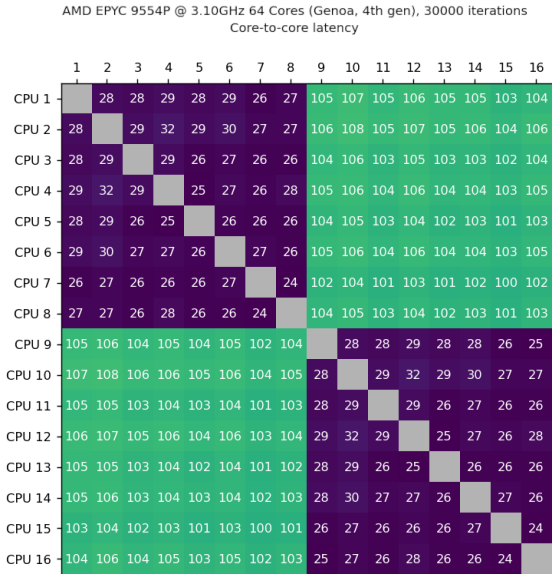

Fig. 2. The core-to-core latency (in ns) of the first 16 cores of an AMD EPYC 9554P processor. This pattern repeats where groups of 8 cores experience low latency communication and all other inter-chiplet communication is significantly slower.

of L3 on the same CCD as the core. Accessing L3 cache on a different CCD has a latency similar to accessing main memory [29].

The story is a little more complicated for Intel with different processors having different topologies. In Intel's most recent line of Intel Xeon 6 processors there may be one, two, or three compute chiplets arranged in a line bookended by two I/O chiplets [28]. Intel's compute chiplets are functionally the same as AMD's CCDs, offering computational cores and cache. In this configuration, accessing local L3 cache takes 33ns while accessing cache one or two chiplets away takes 58ns or 80ns, respectively [19]. Whether AMD or Intel, modern processors offer varied chiplet topologies, processor asymmetries, and NUMA behaviour, underscoring the need for hardware-aware algorithms.

3 Parallel Sparse Polynomial Multiplication

Let f and g be sparse multivariate polynomials sorted decreasingly with respect to some monomial order. We write $f = \sum_{i=1}^{n_f} f_i = \sum_{i=1}^{n_f} a_i X^{\alpha_i}$ and $g = \sum_{j=1}^{n_g} g_j = \sum_{j=1}^{n_g} b_j X^{\beta_j}$ where f_i and g_j are the polynomial terms with respective non-zero coefficients a_i, b_i, and respective exponent vectors α_i, β_j for the variables $X = (x_1, \ldots, x_v)$. The goal is to compute their product $h = f \cdot g = \sum_{i=1}^{n_f} \sum_{j=1}^{n_g} f_i g_j$ while combining like terms and leaving the resulting n_h terms sorted with respect to the monomial order.

A naive multiplication algorithm may produce all $n_f n_g$ terms, sort them, and then condense like terms. A more efficient approach, originally due to Johnson [18], is to perform a n_f-way merge over the partial products $f_i \cdot g$ using a heap, producing terms of the product in sorted order and combining like terms immediately. This merges sorted data sequences since each $f_i \cdot g$ maintains sorted order. This algorithm experiences good performance in practice because the heap provides good data locality and can typically fit in cache [25, 2].

This heap-based algorithm was parallelized in [23] by distributing the responsibility of computing the partial products $f_i \cdot g$ across multiple threads. We refer to this algorithm as *SDMP*. Each thread computes and merges a subset of the $f_i \cdot g$ partial products using a *local heap*, producing an intermediary polynomial. These intermediary polynomials are then similarly merged serially using a *global heap* to compute the final product. For best parallel performance, the global

merge proceeds asynchronously alongside the local heap merges. Terms from the intermediary polynomials "stream" out of the local heaps and into the global heap, using an intermediary buffer as in the classic producer-consumer problem [22]. To avoid buffer overflow and ensure global progression of the algorithm, each thread shares the responsibility of processing terms in the global heap; each thread periodically pauses processing its local heap to make some progress with the global heap before returning to its local heap.

The *SDMP* algorithm was specifically designed to take advantage of a shared last-level cache between all cores [23]. A shared last-level cache gives each core efficient access to the buffers and global heap. However, on a chiplet-based processor, we can no longer assume that the cache is shared across all cores. Communication across chiplet boundaries brings serious performance penalties. Moreover, this algorithm brings limitations to scalability since each thread must synchronize its access to the global heap.

Our *Chiplet* algorithm addresses these concerns through hierarchical parallelism. The goal is to create independent tasks which can be executed in parallel meanwhile each task is itself processed by multiple threads. In particular, each task will be processed by threads bound to the same chiplet so that they have access to a shared cache and efficient communication.

Partitioning the problem into independent tasks follows an approach by Gastineau and Laskar [10]. Consider the n_f by n_g matrix where the entry $\gamma_{i,j}$ is the exponent vector of the term $f_i g_j$. This *pp-matrix* contains all possible product term exponents. Due to the monomial ordering, every $\gamma_{i,j}$ falls in the interval $[\gamma_{1,1}, \gamma_{n_f,n_g}]$. The goal is to partition this interval so that each $\gamma_{i,j}$ belongs in exactly one sub-interval and all $\gamma_{i,j}$ with the same value are contained in the same sub-interval. These sub-intervals partition the *pp-matrix* into disjoint regions where each partition can then be processed independently, see Figure 3. For brevity, we do not detail this process here but refer the reader to [10].

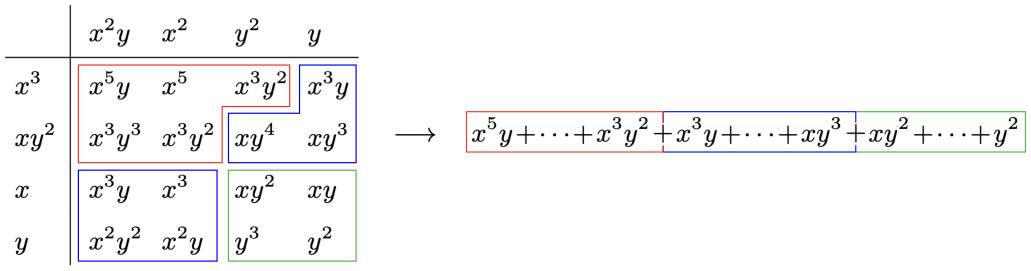

Fig. 3. A *pp-matrix* partitioned into three partitions, showing how terms from partitions can be concatenated in their final order. Partitions are not necessarily contiguous.

The *Chiplet* algorithm thus proceeds as follows. First, partition the *pp-matrix* into regions as just described, creating one task per partition. Each task is then processed using a small number of threads following an algorithm similar to *SDMP*; see Figure 4. To process a partition, each thread is responsible for merging a subset of the rows of the partition using a local heap. The outputs of these local heaps are then merged using a global heap, just as in *SDMP*. The output from this global heap is a contiguous sorted subset of terms of the product polynomial corresponding to the partition being processed. Finally, we can simply concatenate the results produced by each task to form the final product polynomial. Concatenation is valid here since each task corresponds to a continuous sub-interval of $[\gamma_{1,1}, \gamma_{n_f,n_g}]$; see again Figure 3.

An important consideration for the chiplet-based algorithm is the number of threads to use to process each task. Recall the key idea is to have each task be processed by threads executing on the same chiplet to take advantage of shared cache while avoiding inter-chiplet communication; e.g., Figure 1 shows 8 cores per chiplet. Each task should thus be executed by either 2, 4, or 8 threads to ensure maximal occupancy across all 64 cores. In practice, this number is an algorithmic parameter that should be chosen to match the chiplet topology of the executing

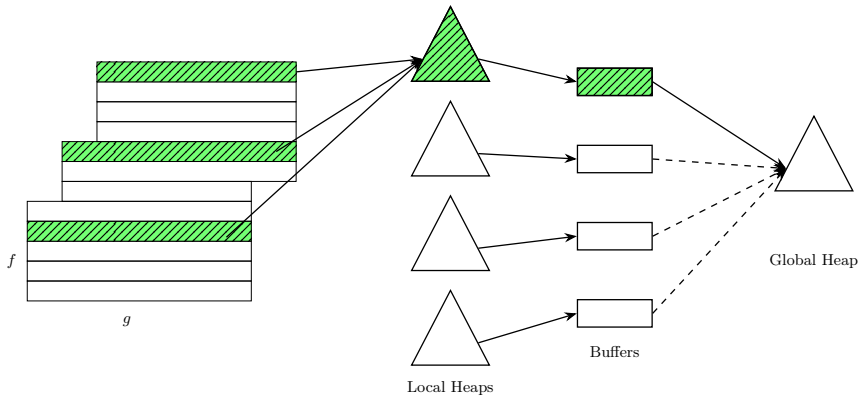

Fig. 4. A partition of the *pp*-matrix being processed by 4 threads, each with a local heap and buffer. The shaded region shows data processed by a single thread.

hardware. On the AMD EPYC 9554P, and the experimentation to follow, we use 4 threads per task.

The *Chiplet* algorithm is implemented in C++ using the BPAS Library [1]. Parallelization is provided by a thread pool interface implemented within BPAS on top of the standard C++11 concurrency support library [6]. For comparison, we also implement the *SDMP* algorithm [23] and the *TRIP* algorithm [10]. The *TRIP* algorithm begins similarly to *Chiplet* by partitioning the *pp-matrix*, but then each partition is processed serially using a standard heap-based algorithm. *Chiplet* improves upon *TRIP* through hierarchical parallelism (processing each partition using multiple threads, see Figure 4) and explicit thread pinning to chiplets to avoid any inter-chiplet communication. Thread pinning, on Linux, is implemented using pthread processor affinity.

4 Experimentation

We compare the multiplication of randomly generated polynomials each with $v = 3$ variables, $n = 80,000$ terms, and varying sparsity and coefficient sizes. Following [2], we define sparsity s as the upper bound in the difference between consecutive exponent vectors, viewing each as the digits of an integer in radix $r = \sqrt[v]{s \cdot n}$. Coefficient bound (C) is the maximum number of bits to encode the multi-precision integer coefficients. Finally, the parameter L controls the number of partitions of the *pp-matrix*; following [10] and empirical measurements we fix $L = 64$. Nonetheless, we observe that the optimal choice of L depends on the input polynomials and the underlying hardware. A thorough investigation of the optimal L value on chiplet-based architectures is left to future work. Data was collected on a workstation running 20.04.6-Ubuntu, AMD EPYC 9554P CPU at 3.76GHz, and 770GB DDR5 memory at 4.8GHz. Code was compiled with g++ v9.4 with C++17 standard. Each data point shows a median of three trials.

Figure 5 shows the parallel speedup of *Chiplet*, *TRIP*, and *SDMP* algorithms applied to the described random polynomials. SDMP behaves poorly due to inter-chiplet communication and thread synchronization between all (up to) 64 threads. *Chiplet* outperforms *TRIP* in all trials. We attribute this to explicit thread cooperation and use of shared cache in the *Chiplet* algorithm.

We note that the closed-source implementation of *TRIP* [20] by its original authors outperforms our algorithms in raw time due to further optimizations in data structures and coefficient arithmetic. However, by comparing *Chiplet* with our own implementation of *TRIP*, we are able to isolate the algorithmic benefits of aligning ones algorithm to the underlying hardware (chiplet) topology.

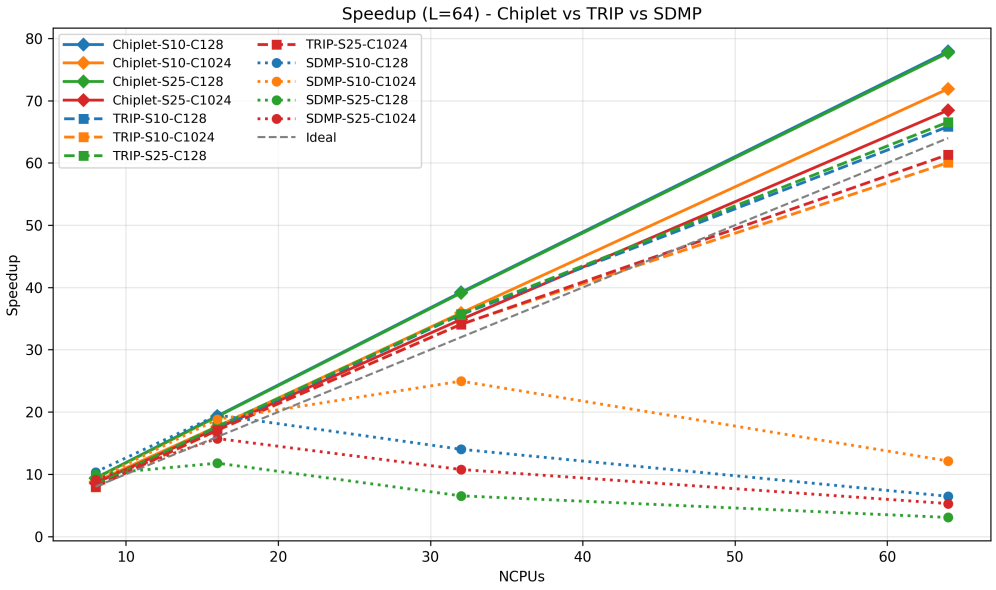

Fig. 5. Parallel speedup vs. number of threads for Chiplet, TRIP, and SDMP algorithms. Coefficient bound and sparsity vary as per the legend.

5 Conclusion and Future Work

We have presented a parallel sparse polynomial multiplication algorithm which employs nested parallelism to bring scalability and effective cache use on chiplet-based processors. The outer level of parallelism brings scalability by ensuring there are a sufficient number of independent tasks, avoiding synchronization between tasks and threads. At the same time, an inner level of parallelism is used to take advantage of the cache hierarchy and cooperatively process a task with a small number of threads pinned to a single chiplet. Thread pinning is necessary to avoid the costs of inter-chiplet communication. Our experimental results show that these chiplet-aware optimizations—cooperative multi-threading and NUMA-aware thread pinning—rather than the partitioning strategy itself are the key performance improvements over prior work like TRIP which employs the same partitioning but lacks our threading approach.

This work advocates that programmers can no longer rely on the simple shared memory model and should now consider hardware characteristics like chiplet topology when designing multithreaded algorithms. In future work we will continue exploring algorithm designs which exploit hidden hardware asymmetries. One interesting direction includes algorithms which can effectively balance work between high-performance cores and low-power efficiency cores on the same processor.

Acknowledgments. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [RGPIN-2024-05927, RTI-2024-00155]

References

1. Asadi, M., Brandt, A., Chen, C., Covanov, S., Kazemi, M., Mansouri, F., Mohajerani, D., Moir, R.H.C., Moreno Maza, M., Talaashrafi, D., Wang, L., Xie, N., Xie, Y.: Basic Polynomial Algebra Subprograms (BPAS) v. 1.791 (2022), http://www.bpaslib.org
2. Asadi, M., Brandt, A., Moir, R.H.C., Moreno Maza, M.: Algorithms and data structures for sparse polynomial arithmetic. Mathematics **7**(5), 441 (2019)
3. Asadi, M., Brandt, A., Moir, R.H.C., Moreno Maza, M., Xie, Y.: Parallelization of triangular decompositions: Techniques and implementation. Journal of Symbolic Computation **115**, 371–406 (2023)

4. Biscani, F.: Parallel sparse polynomial multiplication on modern hardware architectures. In: Proc. of ISSAC 2012. pp. 83–90. ACM (2012)
5. Biswas, A.: Sapphire rapids. In: 2021 IEEE Hot Chips 33 Symposium (HCS) (2021)
6. Brandt, A.: The design and implementation of a high-performance polynomial system solver (2022)
7. Brandt, A., Moreno Maza, M.: On the complexity and parallel implementation of hensel's lemma and weierstrass preparation. In: Proc. of CASC 2021. LNCS, vol. 12865, pp. 78–99. Springer (2021)
8. Chen, C., Covanov, S., Mansouri, F., Moreno Maza, M., Xie, N., Xie, Y.: The basic polynomial algebra subprograms. In: Proc. of ICMS 2014. pp. 669–676 (2014)
9. Cox, D.A., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms. Undergraduate texts in mathematics, Springer, 4th edn. (2015)
10. Gastineau, M., Laskar, J.: Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In: Computer Algebra in Scientific Computing, CASC 2013, Proceedings. LNCS, vol. 8136, pp. 100–115 (2013)
11. Gastineau, M., Laskar, J.: Parallel sparse multivariate polynomial division. In: Proc. of PASCO 2015. pp. 25–33. ACM (2015)
12. Gastineau, M., Laskar, J.: Development of trip: Fast sparse multivariate polynomial multiplication using burst tries. In: International Conference on Computational Science. LNCS, vol. 3992, pp. 446–453. Springer (2006)
13. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, NY, USA, 3 edn. (2013)
14. Gibbons, P.B.: A more practical PRAM model. In: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures. pp. 158–168. ACM (1989)
15. Gomes, W., Morgan, S., Phelps, B., Wilson, T., Hallnor, E.: Meteor Lake and Arrow Lake Intel next-gen 3D client architecture platform with foveros. In: 2022 IEEE Hot Chips 34 Symposium (HCS) (2022)
16. Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach. Morgan Kaufmann, 5th edn. (2011)
17. Intel Corporation: Intel Xeon 6+ processors. https://www.intel.com/content/www/us/en/content-details/866623/intel-xeon-6-processors-product-presentation.html, [Accessed 13-04-2026]
18. Johnson, S.C.: Sparse polynomial arithmetic. ACM SIGSAM Bulletin **8**(3), 63–71 (1974)
19. Lam, C.: A look into intel xeon 6's memory subsystem. https://chipsandcheese.com/p/a-look-into-intel-xeon-6s-memory (2025), [Accessed 13-04-2026]
20. Laskar, J., Gastineau, M.: Trip v1.4.120. https://www.imcce.fr/Equipes/ASD/trip/trip.php (2021), [Accessed 24-02-2026]
21. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science **368**(6495), eaam9744 (2020)
22. McCool, M., Reinders, J., Robison, A.: Structured parallel programming: patterns for efficient computation. Elsevier (2012)
23. Monagan, M., Pearce, R.: Parallel sparse polynomial multiplication using heaps. In: Proc. of ISSAC 2009. pp. 263–270. ACM (2009)
24. Monagan, M., Pearce, R.: Parallel sparse polynomial division using heaps. In: Proc. of PASCO 2010. pp. 105–111. ACM (2010)
25. Monagan, M., Pearce, R.: Sparse polynomial division using a heap. Journal of Symbolic Computation **46**(7), 807–822 (2011)
26. Monagan, M., Pearce, R.: The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. ACM Communications in Comput er Algebra **48**(3/4), 166–186 (2015)
27. Naffziger, S., et al.: Pioneering chiplet technology and design for the AMD EPYC™ and ryzen™ processor families : Industrial product. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). pp. 57–70 (2021)
28. Powell, M.D., et al.: Intel Xeon 6 product family. IEEE Micro **45**(3), 31–40 (2025)
29. Velten, M., Schöne, R., Ilsche, T., Hackenberg, D.: Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors. p. 165–175. Proc. of ICPE '22, ACM (2022)