High Performance Sparse Polynomial Multiplication on Chiplet Architecture

Xiangyu Liu
Supervisor: Alexander Brandt

Faculty of Computer Science
Dalhousie University

August 9, 2024

Abstract: Sparse polynomial multiplication is essential in fields such as computer algebra systems, digital signal processing, and cryptography. Efficient handling of this operation impacts the performance of related algorithms and systems, such as Gröbner basis computations (1) and public-key encryption schemes (2). Traditional advancements in polynomial arithmetic, while beneficial for dense polynomials, are less effective for sparse polynomials due to their design assumptions. To address these challenges, we introduce a novel parallel algorithm for sparse polynomial multiplication running on chiplet architectures. Chiplets, which break down a processor into smaller, individually optimized units, are ideal for parallel programming. The proposed algorithm leverages chiplets' properties to enhance performance and efficiency, distributing computational tasks efficiently and minimizing synchronization overhead.

Keywords: Polynomial Multiplication, Sparse, High Performance, Parallel

Computing, Chiplet Architecture

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Polynomial multiplication stands as a cornerstone operation in numerous fields of science and engineering, including but not limited to computer algebra systems, digital signal processing, and cryptography. The efficiency of this operation directly impacts the performance and feasibility of algorithms and systems in which they are employed. A particular challenge arises when dealing with sparse polynomials. These polynomials are prevalent in applications involving large degrees or in systems where only a few terms contribute significantly to the outcome. Handling and multiplying these sparse polynomials requires optimized approaches.

Recent advancements in polynomial arithmetic algorithms have greatly improved speed and efficiency, particularly for dense polynomials. Techniques like Karatsuba and Toom-Cook multiplication have revolutionized dense polynomial multiplication by reducing computational complexity (3). However, these methods are less effective for sparse polynomials due to their design assumptions, which lead to excessive memory usage and reduced computational efficiency when applied to sparse cases. This inefficiency is due to the unnecessary memory consumption and poor cache performance caused by a dense representation, which includes many non-contributing elements (3).

This report aims to address the challenges of sparse polynomial multiplication by designing a novel parallel algorithm specifically for chiplet architectures. Chiplets are an emerging technology that breaks down a processor into smaller, manageable pieces, or "chiplets," each of which can be individually optimized and manufactured. Moreover, chiplets are particularly suitable for parallel programming because they allow for efficient distribution of tasks across multiple processing units. The proposed algorithm leverages the unique properties of chiplets to enhance performance and efficiency in sparse polynomial multiplication. By distributing the computational tasks across multiple chiplets, the algorithm minimizes synchronization overhead and improves load balancing.

The algorithm is implemented in Julia, a high-level, high-performance programming language particularly well-suited for technical and scientific computing. Julia's capabilities in handling multi-threading and parallelism make it an ideal choice for implementing complex algorithms that require efficient computation and resource management.

2 Background

2.1 Introduction to Chiplet

Our algorithm for sparse polynomial multiplication is designed based on the chiplet architecture, which offers several advantages in terms of performance and scalability. Before delving into the specifics of our algorithm, it is essential to understand what chiplet architecture is, its motivation, and the benefits it offers.

2.1.1 What is Chiplet?

Chiplet architecture is a modern approach to processor design that involves dividing a processor into multiple smaller integrated circuits, known as chiplets. Each chiplet is a self-contained module that performs specific functions and is interconnected with other chiplets to form a complete and cohesive system. This modular design allows for greater flexibility in the manufacturing and assembly process, as different chiplets can be manufactured separately and then integrated into a single processor package. This approach contrasts with the traditional monolithic chip design, where all components are manufactured simultaneously on a single large die.

The basic concept of chiplet architecture revolves around modularity. Each chiplet can be optimized for its specific function, such as processing, memory control, or I/O operations. For

instance, CPU chiplets handle general computational tasks, while memory chiplets contain memory controllers, I/O mechanisms, and interconnects. This modularity simplifies the manufacturing process because each individual chiplet is smaller, easier to produce, has fewer defects, and is more cost-effective. In a chiplet-based architecture, multiple chiplets are connected through a standardized high-speed digital interface to form a complete integrated circuit (IC). This approach allows for creating the complex and powerful system-on-chip (SoC) by mixing and matching different functions and technologies, enabling the development of customized systems without the need to fit all components onto a single monolithic chip (11).

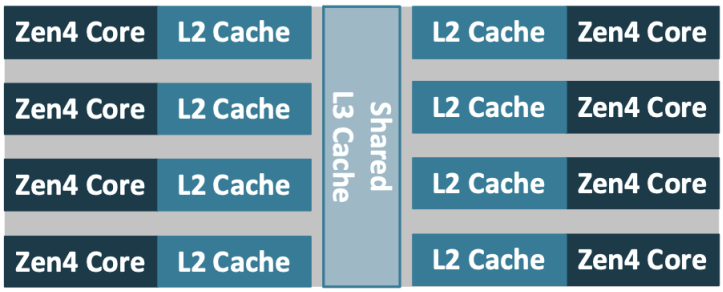

Figure 1. Eight compute cores sharing an L3 cache within a Core Complex (CCX) (12)

Figure 1 illustrates a single chiplet design within a Core Complex (CCX), where up to eight "Zen 4"-based cores share a last level cache (L3), and each core has its own level 2 cache (L2). Simultaneous Multithreading (SMT) enables a single CCX to support up to 16 concurrent hardware threads. It's worth mentioning that each chiplet has its own L3 cache, which is significantly different from traditional multi-core designs that have a single L3 cache shared across all cores. This architectural difference implies that inter-chiplet communication (i.e., communication between cores on different chiplets), is dramatically slower than expected. Consequently, parallel algorithms, other than embarrassingly parallel ones (which do not require inter-task communication), will perform poorly if executed on chiplet-based architectures due to the increased latency in inter-chiplet communication.

In contrast, Figure 2 showcases a multi-chiplet integration within the AMD EPYC 9004

IC, consisting of up to 12 core complex dies (CCDs) and a central I/O die (IOD) for 91xx-96xx models. In this case, each CCD contains a single CCX. This design reintegrates multiple chiplets with in-package interconnects to function as a single, cohesive system. This is the hardware architecture where our sparse polynomial multiplication runs in parallel.

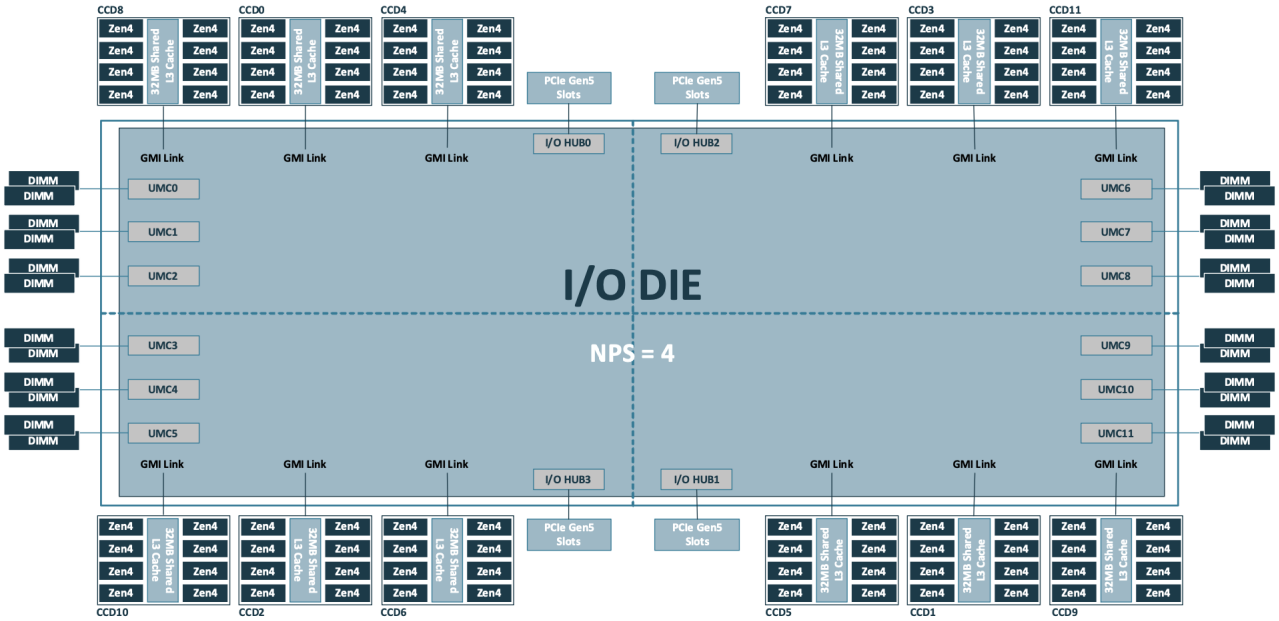

Figure 2. AMD EPYC 9004 IC multi-chiplet integration (12)

2.1.2 Motivation for Chiplets

The motivation for chiplet technology stems from the exponential increase in computational demands and the limitations of traditional monolithic CPUs. The world's computational needs have been rising rapidly, with historical performance trends showing significant growth at both component and system levels. The demand for more compute power is clear in areas, such as the explosion of machine learning (ML) and the massive computational demands associated with training and inferencing using the latest algorithms and models. For instance, the number of parameters in the largest ML models has roughly been doubling every 0.2 years (13), and the computational power required to train these models appears to be

doubling approximately every 3.4 months. Recent global challenges, such as the SARS-CoV-2019 pandemic, have highlighted the necessity for more powerful computational resources (13).

However, maintaining Moore's Law has become increasingly difficult due to may reasons. Moore's Law, proposed by Gordon Moore in 1965, predicted that the number of transistors on a chip would double approximately every two years with a minimal cost increase, leading to exponential increases in processing power (14). For decades, this prediction held true, driving rapid advancements in computer technology. However, as manufacturing technologies approach their physical and economic limits, maintaining this pace of transistor scaling has become increasingly difficult. The slowing down of Moore's Law presents significant challenges for the semiconductor industry. As transistors become smaller, issues such as heat dissipation, power consumption, and manufacturing defects become more pronounced, which constrains the ability to continue increasing performance using traditional monolithic chip designs. Chiplet architecture addresses the limitations of traditional processors by dividing them into smaller, modular units, then reintegrating them with in-package interconnects to operate as a single, logical IC. Each chiplet can also be optimized for specific functions and manufactured using the best technology for its purpose.

Economically, developing new products has become more challenging. For example, a server CPU lineup with various core counts (16, 24, 32, 48, and 64 cores) requires separate manufacturing processes for each variant, each with its own yield and cost profiles. This increases both the silicon costs and the additional upfront costs for physical design, test and debug, validation, firmware, power, and thermal management optimization (13). Given a finite engineering budget, this scenario can lead to a reduction in the number of products a company can offer, despite growing customer demand for differentiated products.

What's more, the economic rationale behind chiplet technology lies in the fact that the cost of silicon does not scale linearly with chip area. For instance, a chip with $T/2$ transistors

may cost significantly less than half the price of a chip with T transistors. Generally, if an IC with T transistors is divided into n separate chiplets, and the combined functionality of these n chiplets matches that of the original T-transistor IC, the total cost—including additional reintegration expenses (e.g., packaging)—can still be much lower than that of a monolithic T-transistor IC (13). This makes chiplet implementation economically viable. Beyond that, producing smaller chiplets reduces defects and improves yields, making the manufacturing process more cost-effective.

2.1.3 Grouping Design of Chiplets

The grouping design of chiplets refers to how different types of chiplets are organized and integrated within a single processor package to work together effectively. This approach allows for the combination of various specialized chiplets, such as CPU cores, GPUs, and memory modules, each optimized for specific tasks, to create a versatile and powerful computing system.

One key aspect of this design is heterogeneous integration. This involves the integration of different types of chiplets within a single package, allowing for a more tailored and efficient system. For example, combining CPU and GPU chiplets within the same package can optimize the system for specialized tasks that require both general-purpose processing and high-performance parallel computing. This integration ensures that each chiplet can operate at its highest efficiency to provide a huge boost in overall system performance.

High-speed interconnects play a crucial role in the effective communication between chiplets. Technologies such as silicon bridges, 2.5D packaging, and 3D stacking are commonly used to facilitate this communication (16). These interconnects enable rapid data transfer between chiplets, minimizing latency and maximizing bandwidth. For instance, in a system where multiple CPU chiplets need to communicate with memory chiplets, high-speed

interconnects ensure that data can be transferred quickly.

Scalability is another significant advantage of the chiplet architecture. The modular nature of chiplets allows systems to be scaled up or down by adding or removing chiplets, providing flexibility to meet different performance requirements and budgets. For example, in a high-performance computing environment, additional CPU or memory chiplets can be added to increase computational power and storage capacity. Conversely, for less demanding applications, fewer chiplets can be used to create a more cost-effective solution. This ability to scale makes chiplet-based systems highly adaptable and more flexible to future needs.

Redundancy and reliability are also enhanced through the use of chiplets. By incorporating redundant chiplets, systems can improve fault tolerance and ensure continuous operation in the event of a chiplet failure. For example, if one processing unit fails, another redundant chiplet can take over its tasks, maintaining system functionality without interruption. This redundancy is crucial for critical applications where downtime is not an option (15).

2.2 Introduction to Multi-threading in Julia

Multi-threading is a powerful technique in computing that allows multiple threads to execute concurrently, potentially improving the efficiency and performance of applications. In the context of high-level programming languages like Julia, multi-threading enables developers to write cleaner, more efficient code that can leverage modern multi-core processors to their fullest potential. Julia, known for its high-performance capabilities akin to languages like C, is particularly well-suited for tasks that benefit from parallel computation. Its design includes built-in support for multi-threading, making it an ideal choice for computationally intensive tasks such as polynomial multiplication.

2.2.1 Julia's Thread Pool and Task Model

In Julia, a thread pool consists of a fixed number of worker threads initialized at the start of a session. The number of threads is determined by the `JULIA_NUM_THREADS` environment variable. Once set, this number remains constant, providing a stable pool of threads for executing tasks. This fixed thread model ensures predictable performance and simplifies resource management.

Tasks, also known as coroutines or green threads, are lightweight units of work that can be paused and resumed. In Julia, tasks can be created by the function `Task()`. When a task is created, it is placed in a queue and scheduled for execution by the available threads in the pool. Idle threads pick up tasks from the queue and execute them. If all threads are busy, new tasks wait in the queue until a thread becomes available. Moreover, in Julia, tasks have the inherent ability to migrate between threads, especially when they yield. Yielding occurs when a task temporarily pauses its execution, typically to wait for I/O operations or other tasks to complete. To manage this behavior and ensure consistent execution, Julia introduced the concept of `sticky` tasks. By default, tasks are sticky, meaning they are designed to remain on the thread that scheduled them, preventing migration. This default setting enhances execution consistency. However, the sticky flag can be explicitly set to false, making the task eligible to be picked up by any thread. This adjustment allows for greater flexibility and load balancing across the available threads.

To ensure a task runs on a specific thread and to pin that thread to a specific core, you can use `set_task_tid()` along with the `ThreadPinning.jl` package (19). The `set_task_tid()` function binds a task to a specific Julia thread, ensuring that the task will only run on that thread. The `ThreadPinning.jl` package allows you to pin Julia threads to specific CPU cores by the function `pinthreads(Julia_thread_ID, core_ID)`, enhancing cache locality and reducing context switching. By combining these tools, you can achieve precise control

over task execution and optimize performance in high-performance computing environments. Here's an example:

```
function my_task_function()
    println("Task running on thread ", threadid())
end

# Create a new task
t = Task(my_task_function)

# Bind the task to thread 2
set_task_tid(t, 2)

# Pin thread 2 to core 2
using ThreadPinning
pinthreads(2, 1)

# Schedule the task
schedule(t)

# Wait for the task to complete
wait(t)
```

Listing 1. Using `set_task_tid()` and `ThreadPinning` to control task execution

In this example, a new task `t` is created and bound to thread 2 using `set_task_tid(t, 2)`. The `ThreadPinning.jl` package is then used to pin thread 2 to CPU core 2 with `pinthreads(2, 1)`, as hardware resources (cores) use 0-based indexing while Julia threads use 1-based indexing. The task is scheduled for execution and then awaited until completion.

2.2.2 @threads vs. @spawn

High-level constructs in Julia like the `@threads` and `@spawn` macros simplify distributing work across multiple threads, making parallel computing more accessible and helping avoid common multi-threading issues such as race conditions or deadlocks. The `JULIA_NUM_THREADS` environment variable defines the upper limit of threads available to the Julia process, with both `@threads` and `@spawn` operating within this setting but differing in their approach to leveraging available threads.

The `@threads` macro excels in parallelizing loop operations. It distributes loop iterations across the available threads seamlessly. When `@threads` is employed, it assigns a slice of the total iterations to each thread, ensuring all iterations are completed before proceeding, akin to a conventional loop structure. However, if the number of iterations is less than the number of available threads, not all threads will be utilized; some will remain idle. On the other hand, if the iterations exceed the number of threads, `@threads` will divide the work such that each thread gets an approximately equal share, ensuring all threads are actively engaged until their allotted iterations are processed.

In contrast, `@spawn` is the tool of choice for initiating tasks that are not necessarily tied to loop constructs. This macro allows for the creation of asynchronous tasks, akin to lightweight threads or coroutines, which are scheduled across the thread pool for execution. The key characteristic of `@spawn` is its flexibility and adaptability—it can handle any executable code block or function, and not just loop iterations. The tasks are enqueued and will commence once a thread is available, which means they might have to wait if all threads are currently busy.

Diving into the key differences, we see that `@threads` is tailored to parallelize work where the load can be statically divided at the outset, primarily loop iterations. `@spawn`, however, offers a more granular and dynamic parallelism capability, ideal for varied and asynchronous

workloads. Furthermore, after a block of `@threads`, the code will only proceed once all threads have completed their tasks, effectively synchronizing the thread completion. In contrast, tasks spawned with `@spawn` execute independently; synchronization with the main program requires explicit calls to functions like `wait` or `fetch`. Moreover, the load balancing strategy also differs between the two. With `@threads`, once a thread has completed its assigned iterations, it will not seek out more work. Meanwhile, `@spawn` allows for dynamic load balancing, with new tasks being picked up by threads as they finish their current tasks and become available again. Here are examples to illustrate the difference and address potential issues like data races (17):

```
function sum_single(a)
    s = 0
    for i in a
        s += i
    end
    return s
end

a = 1:1_000_000
println("Single-threaded sum: ", sum_single(a))
# Return: 500000500000
```

Listing 2. Naive sum function without parallelism

The function `sum_single` calculates the sum of an array a sequentially. This is the baseline single-threaded execution, summing all numbers in the array correctly.

```
function sum_multi_bad(a)
    s = 0
    Threads.@threads for i in a
        s += i
    end
    return s
end

println("Incorrect multi-threaded sum: ", sum_multi_bad(a))
# Return: 70140554652
```

Listing 3. Parallel sum using `@threads`

This version uses `@threads` without proper synchronization, leading to data races. Multiple threads try to read and write to the shared variable s simultaneously, resulting in an incorrect sum, such 70140554652. It is important to note that threading and data races lead to non-determinism, meaning this incorrect sum will vary from one execution to the next.

```
function sum_multi_good(a)
    chunks = Iterators.partition(a, length(a)    Threads.nthreads())
    tasks = map(chunks) do chunk
        Threads.@spawn sum_single(chunk)
    end
    chunk_sums = fetch.(tasks)
    return sum_single(chunk_sums)
end

# Example usage
println("Correct multi-threaded sum (@spawn): ", sum_multi_good(a))
# Return: 500000500000
```

Listing 4. Correct parallel sum using `@spawn` and avoiding data races

By using `@spawn`, we avoid data races by having each thread work on its own chunk of data. The results from each chunk are summed sequentially, ensuring correctness. If one insists on using `@thread`, an alternative solution would be the use of atomic operations on variables shared across tasks/threads. Atomic operations ensure that each read-modify-write operation on a shared variable is completed as a single, indivisible step, preventing data races.

```
function sum_multi_atomic(a)
    s = Atomic{Int}(0)
    Threads.@threads for i in a
        atomic_add!(s, i)
    end
    return s[]
end

# Example usage
println("Correct multi-threaded sum (atomic): ", sum_multi_atomic(a))
# Return: 500000500000
```

Listing 5. Correct parallel sum using `@threads` with atomic operations

This solution uses atomic operations to avoid data races. `atomic_add!` ensures that each addition to the shared variable s is performed atomically, preserving correctness.

These examples demonstrate how `@threads` can lead to issues like data races when used improperly, while `@spawn` and atomic operations allow for safe and efficient parallel execution by properly managing shared resources. In our implementation of sparse polynomial multiplication, we use `@spawn` for multi-threading, since the workload of each task is different, and a thread needs to keep waiting for tasks after it finishes the work until tasks are exhausted. More details will be explained in the next section.

3 Basic Data Structure and Algorithm for Sparse Polynomial

3.1 Polynomial Encodings: Dense vs. Sparse

Polynomial encodings play a critical role in the computational efficiency of polynomial operations. The dense polynomial encoding entails the storage of every coefficient for a range of exponents, including those that are zero. This method is typically advantageous for polynomials with a low number of variables or when most coefficients up to the highest degree are non-zero. Arrays or lists are commonly employed in dense encoding, where the index directly corresponds to the exponent and the coefficient is the value at that index, exemplified by the polynomial $3x^2 + 2x + 1$, which is stored as $[1, 2, 3]$. The array in Figure 3 encodes a polynomial $p = \sum_{i=0}^{n} a_i x^i$.

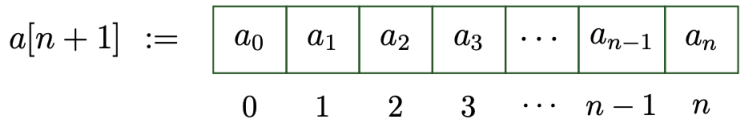

Figure 3. Array representation of a dense univariate polynomial

In this scheme, a polynomial addition/subtraction can be fairly easy. What we need to do is just iterate through both arrays and operate on the number with the same indices. With this representation, we can easily retrieve information like number of terms, the leading term, the leading coefficient, and the maximum degree.

Conversely, sparse polynomial encoding, which stores only the non-zero coefficients alongside their corresponding exponents, excels in scenarios where the polynomials exhibit a significant degree with most coefficients equating to zero, such as $x^{10000} - 1$. In this situation, one is able to encode that polynomial densely, but it is not at all efficient or wise to do so, as

it will cost us 10001 entries for storing just two useful elements. Operations in sparse representation, however, can be more complex due to the need for matching like terms, but this approach significantly conserves memory and circumvents unnecessary computations. The following is a mathematical representation of sparse polynomial. In this notation, the terms n_a are ordered decreasingly by lexicographical order. a_i is the i-th coefficient, and $e_{1i}, \ldots, e_{vi}$ are the exponents of the i-th monomial. For simplicity, a multivariate monomial is written as X^{α_i}, where X represents the sequence of variables $x_1, \ldots, x_v$ and α_i is a multi-index of exponents which is equal to $e_{1i}, \ldots, e_{vi}$.

$$\mathrm{a} = \sum_{i=1}^{n_a} a_i x_1^{e_{1i}} \cdots x_v^{e_{vi}} = \sum_{i=1}^{n_a} a_i X^{\alpha_i}$$

The choice between sparse and dense encodings hinges on the degree and the distribution of significant terms within a polynomial. Sparse encoding is notably memory-efficient for polynomials characterized by a prevalence of zero coefficients. On the other hand, dense encoding can facilitate faster arithmetic operations due to its direct indexing advantage but may be less optimal for polynomials with numerous zero elements. Sparse encoding necessitates more sophisticated algorithms to manage the non-contiguous nature of non-zero terms. In many practical applications, the polynomials of interest tend to be sparse as the systems they model are often represented with only a few significant terms. Thus, in the following sections, we will put the weight on the sparse polynomial representations and algorithms.

3.2 Data Structures for Sparse Polynomial Representation

In computational mathematics, the choice of data structure for polynomial representation is pivotal, impacting both the memory efficiency and computational speed of arithmetic operations. This section explores various data structures tailored for sparse polynomial

representation, each with its distinct trade-offs. More details about this topic can be found in (3).

Linked Lists

The linked list is a foundational and also the simplest data structure for representing sparse polynomials. It consists of nodes linked sequentially using pointers, with each node encapsulating a term of the polynomial. Consider the polynomial $13x^2y^3 + 5x^2y + 7y^3$; it is represented by a linked list with distinct nodes for each term as shown in the Figure 2 below, enabling straightforward manipulation of terms through pointer operations. This data structure is particularly advantageous for sparse polynomials due to its inherent flexibility in adding or removing terms.

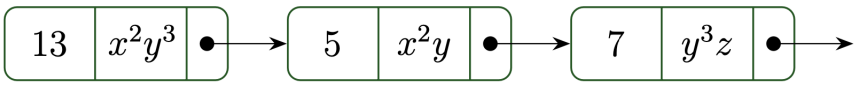

Figure 4. A polynomial encoded in a linked list

However, the linked list structure is not without drawbacks. Operations such as indexing, counting the number of terms, or identifying a specific monomial necessitate traversing the entire list, incurring an $O(n)$ time complexity and becoming less efficient for large polynomials. Furthermore, the memory overhead from pointers can introduce indirection, detracting from memory locality and rendering the representation less efficient. The memory consumed by each pointer, often 8 bytes, increases the overall memory footprint, with a substantial portion allocated to structural overhead rather than the polynomial's actual data. To address the issues of pointer storage overhead, potential indirection, and bad data locality, our objective is to compactly organize these nodes in an array.

Alternating Arrays

Another data structure is the alternating array, which stores coefficients and monomials side by side in an array format, eschewing the indirection from pointers. This design is ideal for operations that need to access both coefficients and monomials simultaneously, hence optimizing for locality. The array ensures that terms are stored in decreasing lexicographical order to maintain a canonical format, thus facilitating efficient arithmetic operations in a predictable manner without the need for searching or sorting.

Lexicographical order is a method of ordering sequences by comparing their elements pairwise from left to right. For two monomials $x^a y^b$ and $x^c y^d$, $x^a y^b$ is said to be greater than $x^c y^d$ in lexicographical order if $a > c$ or if $a = c$ and $b > d$. This definition extends naturally to more than two variables by comparing exponents pairwise from left to right, using the first inequality encountered to determine the overall ordering of the monomials.

In our alternating array approach, we employ exponent packing with long unsigned integers for compact exponent vector encoding when dealing with multivariate polynomials. More details about the exponent packing will be discussed in the next subsection. Comparing with the linked list representation, the alternating array is performing without the use of pointers, eliminating any excess overhead. Consequently, this method achieves a highly efficient representation of a set of polynomial terms.

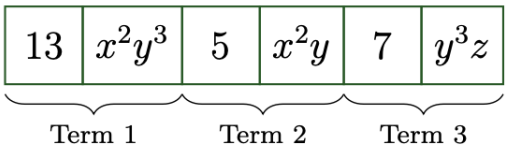

Figure 5. A polynomial encoded in an alternating array

While alternating arrays minimize memory usage and potentially enhance cache performance, they are not conducive to easy insertion or removal of terms. This limitation is

typically acceptable, given that such operations are infrequent or unnecessary for maintaining the polynomial terms once the input polynomials are given.

Exponent Packing

In the end of this subsection, we introduce a compact technique for exponent storage called *exponent packing*. It is a method used to efficiently store multiple exponents in a single machine word (we assume a single machine word of 64 bits in this report). This technique aims to improve memory usage and computational efficiency by reducing the space required to store multivariate polynomial terms.

A normal method often assigns each exponent to one machine word, which can be highly inefficient due to the presence of many leading zeros. For example, consider the following binary representation of the exponents in the term $4x^3y^5z$ as a 64-bit unsigned integer. Here, exponent 3, 5, and 1 all result in a significant number of leading zeros, with 62, 61, and 63 zeros respectively. If we store exponents separately in this manner for a polynomial term, each exponent would consume much more memory than necessary.

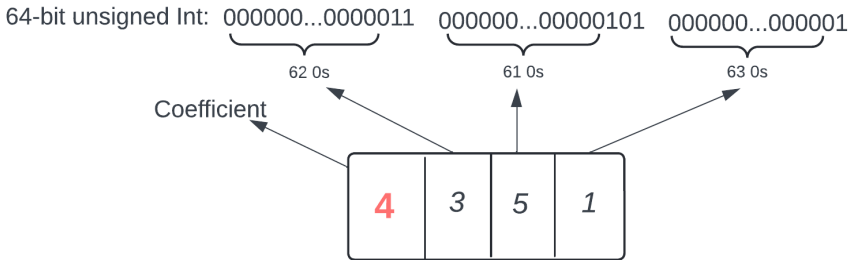

Figure 6. A representation of $4x^3y^5z$ before exponent packing

To address this inefficiency, we employ exponent packing, which encodes multiple exponents into a single machine word. This method involves dividing the single machine word into sections, with each section holding the bits of an exponent. By using bit-masks and shifts, small absolute values can be stored compactly or retrieved within the same word,

thus eliminating unnecessary leading zeros. Back to our example of $4x^3y^5z$, the exponents 5, 3, and 1 are packed into a single 64-bit word by allocating 16 bits, 20 bits, and 28 bits respectively, which results in a highly efficient use of the 64-bit word.

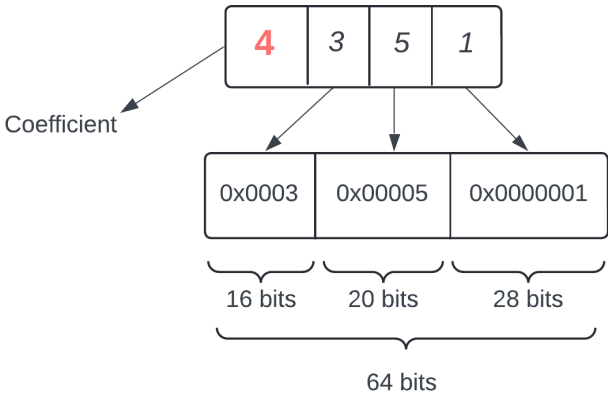

Figure 7. A representation of $4x^3y^5z$ after exponent packing

Packing exponent disproportionately allocates more bits to lower ordered variables than higher ordered variables, since the former increase much faster during the process of solving polynomial systems. Solving systems of polynomial equations is our motivating problem for our polynomial data structures and algorithm. This approach ensures that memory is used efficiently and that no bits are wasted.

The benefits of exponent packing are significant. By storing exponents compactly, memory usage is greatly reduced, and computational overhead is minimized. Operations like monomial comparison and multiplication are reduced to single machine word manipulations, resulting in faster processing times. This efficient representation allows for millions of polynomial terms to be stored within a relatively small memory footprint.

3.3 Basic Algorithms for Sparse Polynomial Multiplication

Sparse polynomial multiplication involves unique techniques tailored to handle the sparse nature of these polynomials efficiently. Consider multiplying two sparse polynomials, a and

b, with n_a and n_b number of terms, respectively. The naive approach, while straightforward, is inefficient because it involves distributing each term of one polynomial across all terms of the other. This generates all possible $n_a \times n_b$ product terms, many of which do not match, requiring additional steps to sort the product polynomial and combine like terms, as illustrated by the following example:

$$a(x) = 3x^4 + 2x^3 + 1$$

$$b(x) = x^3 + 4x + 5$$

First, distribute each term of $a(x)$ across all terms of $b(x)$:

$$(3x^4)(x^3 + 4x + 5) + (2x^3)(x^3 + 4x + 5) + 1(x^3 + 4x + 5)$$

$$= 3x^7 + 12x^5 + 15x^4 + 2x^6 + 8x^4 + 10x^3 + x^3 + 4x + 5$$

Next, sort the product polynomial:

$$3x^7 + 2x^6 + 12x^5 + 15x^4 + 8x^4 + 10x^3 + x^3 + 4x + 5$$

Finally, combine like terms:

$$3x^7 + 2x^6 + 12x^5 + (15x^4 + 8x^4) + (10x^3 + x^3) + 4x + 5$$

$$= 3x^7 + 2x^6 + 12x^5 + 23x^4 + 11x^3 + 4x + 5$$

This simple example illustrates the naive method of sparse polynomial multiplication

through the processes of distribution, sorting, and combining. The inefficiency arises because distribution generates all $n_a \times n_b$ product terms. Consequently, the sorting step incurs a cost of $O((n_a \times n_b) \log(n_a \times n_b))$. This inefficiency becomes particularly pronounced with larger polynomials, making optimization crucial for practical applications.

Heap-based multiplication

When multiplying two polynomials, the number of resulting terms is up to the product of the number of terms in each polynomial. By definition of the multiplication of monomials, the product is necessarily significantly larger in size. Memory management becomes crucial here because the size of the product can be large and the way we handle this data affects the algorithm's performance. Our goal is to minimize memory traversal and manage the term efficiently. Here, we introduce a more efficient approach called $heap - based\ multiplication$, utilizing a priority queue implemented by a binary-tree heap. The heap-based approach was first employed by Johnson in 1974 (4), and then later optimized by Monagan and Pearce in 2009 (5). This data structure facilitates efficient insertion and extraction operations, with complexities of $O(\log n)$, where n represents the number of elements within the heap. In a max heap tree, insertion adds an element to the heap, extraction (also known as pop) removes and returns the largest element, and peek retrieves the largest element without removing it. This is much better than $O(n)$ complexity of maintaining a regular sorted array.

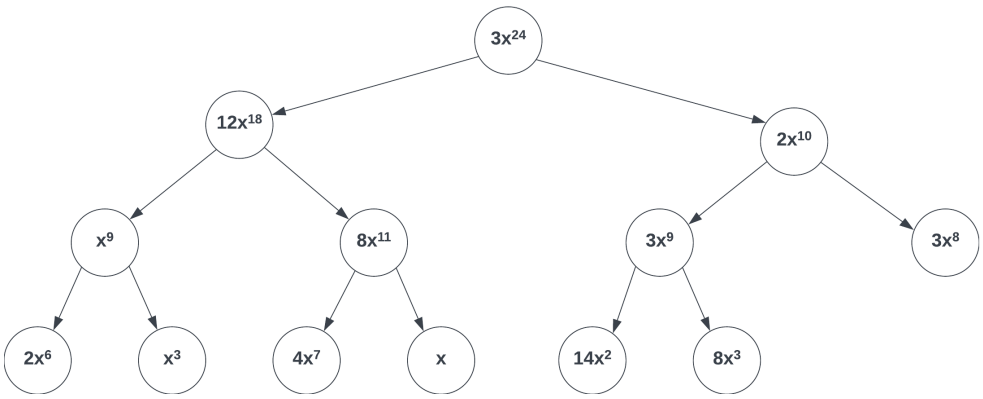

Figure 8. A max heap storing terms based on exponents

Here, we employ a new term *stream* which is generated by distributing each term in polynomial a over all terms in polynomial b. As shown below, within the product of $a \cdot b$, each line itself is a stream, including n_a streams in total (i.e., the number of terms in polynomial a). Since our input polynomial is maintained in the lexicographical order, we can confirm that the terms in each stream also retains the lexicographical order. Thus for a given α_i and β_j, we observe that $X^{\alpha_i+\beta_j}$ is always greater than $X^{\alpha_{i+1}+\beta_j}$ and $X^{\alpha_i+\beta_{j+1}}$, also, it is always greater than $X^{\alpha_{i+1}+\beta_{j+1}}$. However, we cannot guarantee that $X^{\alpha_{i+1}+\beta_j}$ is always greater than $X^{\alpha_i+\beta_{j+1}}$ or vice versa. With the help of these properties, we can generate the terms of product polynomial in the right sequence. Finding the next largest product term becomes easy by comparing the head terms of each stream, where the head of each stream is the largest term not yet consumed and committed to the product term.

$$a \cdot b = \begin{cases} (a_1 \cdot b_1)X^{\alpha_1+\beta_1} + (a_1 \cdot b_2)X^{\alpha_1+\beta_2} + (a_1 \cdot b_3)X^{\alpha_1+\beta_3} + \cdots \\ (a_2 \cdot b_1)X^{\alpha_2+\beta_1} + (a_2 \cdot b_2)X^{\alpha_2+\beta_2} + (a_2 \cdot b_3)X^{\alpha_2+\beta_3} + \cdots \\ \vdots \\ (a_{n_a} \cdot b_1)X^{\alpha_{n_a}+\beta_1} + (a_{n_a} \cdot b_2)X^{\alpha_{n_a}+\beta_2} + (a_{n_a} \cdot b_3)X^{\alpha_{n_a}+\beta_3} + \cdots \end{cases}$$

In this heap-based approach, we encode the multiplication streams as a heap. Particularly, the heap is initialized with the head of each stream, with n_a number of nodes at the very beginning. As the algorithm requires the maximum term for processing, it can be promptly retrieved from the heap's top. Afterwards, we check the stream from which the removed term came. If another term exists in that stream, we insert it into the heap. Then we peek() the heap to see if the next maximum term has the same degree with the previous extracted node, if yes, pop() it, combine the like term, and again insert its successor from that stream into the heap. We keep repeating this process until peek() doesn't return us a like-term. Then we add this singular product term into the result polynomial if the combined

coefficient is not zero. The multiplication process can thus be visualized as the merging of multiple streams, where like terms are merged into a singular result as soon as they emerge. The algorithm terminates when all nodes in the heap are exhausted. Subsequently, it yields a result polynomial that is inherently arranged in sorted order.

As a result, the heap-based approach beats the naive approach in sparse polynomial multiplication by leveraging a heap with at most n_a elements, often fitting within the CPU cache for faster access. This method condenses like terms immediately, and only inserts $X^{\alpha_i+\beta_{j+1}}$ into the heap after $X^{\alpha_i+\beta_j}$ has been extracted, avoiding the need to compute all $n_a \times n_b$ terms explicitly. Additionally, it produces terms in sorted order, eliminating the need for a final sorting step to achieve a canonical representation.

Algorithm 1, below, presents pseudocode for the heap-based multiplication. The function $heapInitialize(a, B_1)$ initializes the heap with the head of each stream. The function $heapPeek()$ retrieves the exponent vector γ of the top element of the heap and the stream index s indicating the origin of the stream. The function $heapExtract()$ removes and returns the top element of the heap. The function $heapInsert(A_i, B_j)$ inserts the product of A_i and B_j into the heap, which is the next term from the stream where the extracted element came from, if such a term exists.

Algorithm 1 HEAP-BASED_POLYNOMIAL_MULTIPLICATION
Input: $a = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$, $b = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$;
Return: $c = a \cdot b$

1: **if** $n_a = 0$ or $n_b = 0$ **then**
2: $\quad$ *return* 0
3: **end if**
4: $k \leftarrow 1; C_1 \leftarrow 0$
5: $\gamma \leftarrow \alpha_1 + \beta_1$ ▷ Maximum possible value of γ which is the head of first stream
6: $heapInitialize(a, B_1)$
7: **for** $i = 1$ to n_a **do**
8: $\quad f_i \leftarrow 1$
9: **end for**
10: **while** $\gamma > -1$ **do** ▷ $\gamma = -1$ when the heap is exhausted
11: $\quad (\gamma, s) \leftarrow heapPeek()$
12: $\quad$ **if** $\gamma \neq deg(C_k)$ and $coef(C_k) \neq 0$ **then**
13: $\quad\quad k \leftarrow k + 1$
14: $\quad\quad C_k \leftarrow 0$
15: $\quad$ **end if**
16: $\quad C_k \leftarrow C_k + heapExtract()$
17: $\quad f_s \leftarrow f_s + 1$
18: $\quad$ **if** $f_s \leq n_b$ **then**
19: $\quad\quad heapInsert(A_s, B_{f_s})$ ▷ Insert the next term in the same stream if exists
20: $\quad$ **end if**
21: **end while**
22: **return** $c = \sum_{\ell=1}^{k} C_\ell = \sum_{\ell=1}^{k} c_\ell X^{\gamma_\ell}$

Optimizations and Analysis: Pre-allocation, One-node Heap Initialization, and Chaining

When using a heap in polynomial multiplication, elements are continuously inserted and removed from the heap. If the heap grows dynamically (which is often the case in many implementations), each insertion could potentially cause a reallocation of the heap to accommodate more elements. This reallocation involves memory movement, which is the copying of the existing elements to a new, larger block of memory. Dynamic memory reallocation is expensive in terms of computational time and can lead to fragmented memory. By pre-allocating memory for the maximum number of elements that will ever be in the heap

at any one time, we can avoid these costly reallocations. In this algorithm, the maximum is determined by the number of streams, typically n_a, assuming $n_a \leq n_b$. As we continue with the multiplication, we will insert more products into the heap, but terms will also be extracted from the heap. The heap will never need to contain more than n_a elements at once, which is why we can pre-allocate this exact amount. By avoiding reallocation, the algorithm can run faster by not being interrupted by memory management operations.

In the previous introduction of heap-based multiplication, we initialize the heap with the head term of each stream, resulting in an initial heap size of n_a. To optimize this process, we can employ a strategy where the heap is initialized with a single node $X^{\alpha_1+\beta_1}$, the largest product term in the result polynomial. After extracting the root node of the heap, we insert its successor as follows: for $X^{\alpha_i+\beta_j}$, insert $X^{\alpha_i+\beta_{j+1}}$ if it exists. If $j = 1$, also insert $X^{\alpha_{i+1}+\beta_1}$ if it exists. Recall that, due to the ordering of the polynomials, $X^{\alpha_i+\beta_1}$ is always greater than $X^{\alpha_{i+1}+\beta_1}$. Therefore, the latter can be excluded from the heap until the former is extracted. As computation progresses, the heap size grows but never exceeds n_a. This approach avoids adding new elements to the heap until necessary, making the algorithm more refined by eliminating redundant initializations. Imagine, for large polynomial multiplications, such as $n_a = 1000$ and $n_b = 1000$, initializing the heap with a thousand nodes can be very heavy. However, with the one-node initialization strategy, computation can commence immediately after calculating the first node, making the heap structure look light and handy.

To further reduce the size of the heap, the *chaining* mechanism is employed for a higher level of algorithm optimization. The chaining technique, introduced by Monagan and Pearce in 2011 (6), refers to the practice of linking monomials that have the same degree. Instead of storing each of these elements as separate entries in the heap, they are linked together in a linked list structure.

When monomials of the same degree are generated during polynomial multiplication, they need to be combined by adding their coefficients. Chaining allows these terms to be

grouped together, so that, ideally, only one entry per unique degree is held in the heap, with a chain of all the like-degree terms attached. This drastically reduces the number of heap slots occupied at any one time. Here, one might ask: why chain the like terms together when you can simply condense terms right away in that node instead? Then we can get rid of the linked list structure! The answer lies in the necessity of tracking the origin of each term for the purpose of successor insertion. By condensing terms immediately, we lose the ability to trace back which streams they originated from, making it impossible to correctly manage successor terms.

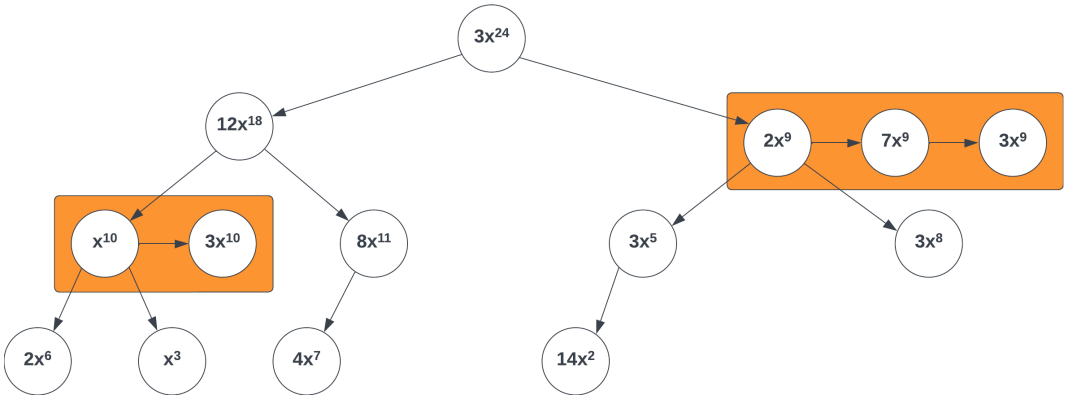

Figure 9. A max heap of product terms with chaining

With fewer elements in the heap, operations like insertions, deletions, and heapify (to maintain the heap property) become faster because there are fewer nodes to compare and move around. This reduction in elements decreases the computational complexity of these operations, leading to faster overall execution of the polynomial multiplication. Additionally, the improved data locality of the heap, due to its reduced size, can enhance cache performance on modern computer architectures.

However, it's noted that in the traversal of a binary heap, not all elements with equal monomials are guaranteed to be chained together in a single chain. For example, when a new node is inserted into the heap, it begins at the last slot on the right side of the tree and then performs a series of swaps upward (heapify) to find its correct position in the right-

hand subtree. However, the chain of nodes with the same degree might be located on the left-hand side of the tree. Consequently, this newly inserted node is not grouped with its corresponding chain. This scenario highlights the need to continue extracting chains until the maximum degree in the heap no longer matches the degree of the first extracted node in that round, ensuring that all nodes with the same degree are processed together.

4 Parallel Algorithms for Sparse Polynomial Multiplication

4.1 The SDMP Algorithm

In this section, we introduce a high-performance parallel algorithm specifically designed for multiplying sparse polynomials using multicore processors called SDMP. This method was created by Monagan and Pearce in 2009 (5). It utilizes individual cores to work on heap data structures, which take advantage of local caches for multiplication tasks. Intermediate results are written to buffers in shared cache spaces. Periodically, a global merge occurs, consuming data from the buffers and sorting terms to be committed to the final resulting polynomial. This cooperative approach improves load balancing, scalability, and can achieve superlinear speedup in practical applications.

The SDMP algorithm can be divided into three parts, with two subroutines `LOCAL_HEAP_MERGE`, and `GLOBAL_HEAP_MERGE`, and the main function `SDMP_MULTIPLICATION`. Let's begin with the most sophisticated part, `LOCAL_HEAP_MERGE`. This subroutine handles how each thread in the parallel algorithm processes a segment of the polynomial multiplication.

Local Heap Merge

In this parallel algorithm, given two polynomials a and b with a decreasing monomial order, the computational task is divided into streams (i.e., $(a_i \cdot b)$). The work is divided among the threads based on a calculation involving the number of CPU cores X and the number $t = \sqrt[3]{n_a}$. We then compute $p = min(X, t/2)$ as the number of threads to create, ensuring sufficient work for each thread. In this way, we can minimize the parallel overheads of spawning and managing threads, thereby maintaining optimal performance. Each thread is assigned with n_a/p streams and tackles every p^{th} stream starting from its own thread_ID. For example, if we have 4 threads with 16 streams, thread 1 will tackle stream 1, 5, 9, and 13; thread 2 will tackle stream 2, 6, 10, and 14; and so on, for threads 3 and 4.

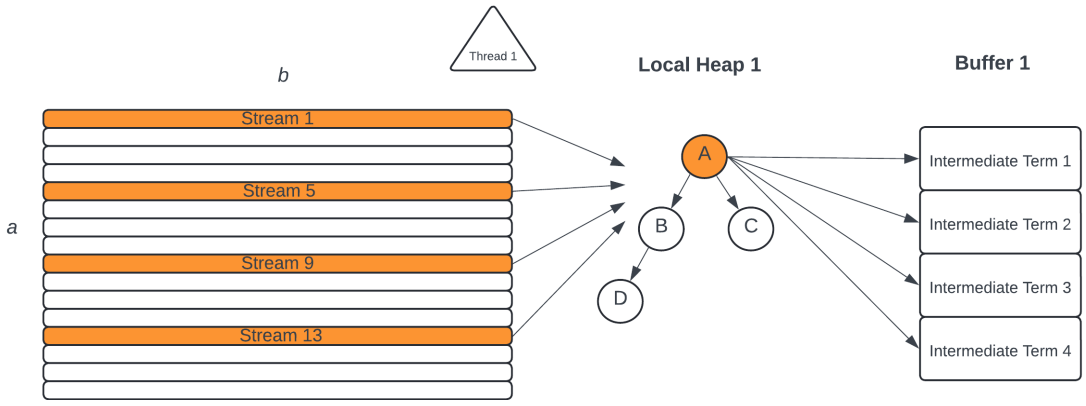

Figure 10. Local heap merge

Each thread needs to create a max binary heap for its own use. By using the one-node initialization strategy, each thread initializes the heap with the node $a_r \cdot b_1$, where $r =$ thread_ID (i.e., the maximum term for that thread). As long as the heap is not empty, extract the root of the heap (i, j, M), where i and j indicate the indices of the term being multiplied from a and b, respectively, and M indicates the monomial of the product term. Then, store the multiplied coefficient $\text{coef}(a_i) \cdot \text{coef}(b_j)$ into the variable C. Store the index pair (i, j) in an array Q, which marks the node we just extracted for later use. After that, keep peeking the heap to see if the next largest element is a like term, and accumulate the

coefficient into C if it is. After the coefficient accumulation for a unique degree, iterate over all (i, j) in Q, and insert their successors. For $a_i \cdot b_j$, insert $a_i \cdot b_{j+1}$ if it exists. If $j = 1$, insert $a_{i+p} \cdot b_1$ if it exists, ensuring that only necessary nodes are stored in the heap. Finally, write the accumulated coefficient C (if it is non-zero) with its monomial M into the buffer B.

After a few rounds of writing (C, M) into B, the thread needs to acquire the lock for the global heap, read from the buffers of every thread, and contribute to the global heap. When is the optimal time to acquire a lock? Should the thread wait until the buffer is full? The answer is no. Instead, we use a decrementing counter $k = N/p$, where N is the buffer capacity and p is the number of threads. This counter decrements each time an element is written into B. The ratio N/p ensures that the threads synchronize with a minimum frequency proportional to the number of cores. This does not limit scalability in practice, as we use at most $\sqrt[3]{n_a}$ threads. When k runs out (i.e., there are N/p elements stored in the buffer), the thread starts to acquire the lock. If the lock is acquired, the thread enters the critical section for `GLOBAL_HEAP_MERGE` and releases the lock once done. The thread then returns to computation, writing $\min(N - |B|, N/p)$ elements into its B. The value of $\min(N-|B|, N/p)$ finds the next workload for the thread between the number of empty slots left in B and the minimum synchronization frequency. If the lock is not acquired and the buffer is not full, the thread continues computation to write more product terms into the buffer. If the lock is not acquired and the buffer is full, the thread has no choice but sleeps for 10 microseconds before trying again. After each thread completely finishes its computation work, it must call $close(B)$ to signal that it has terminated instead of simply being blocked.

However, the 10 microseconds duration is relatively arbitrary and can be tuned for any particular executing system. The goal is to sleep for a very small amount of time, but still long enough to allow some OS tasks to be completed in the meantime, such as scheduling threads and unlocking the lock.

Algorithm 2 LOCAL_HEAP_MERGE
Input: $a, b \in \mathbb{Z}[x_1, \ldots, x_n]$, buffer B, thread_ID r, total number of threads p;
Return: terms of the product are written to B;
Local: heap H, momomial M, coefficient C;
Global: lock L, buffer capacity N in terms.

1: $H :=$ an empty max binary heap
2: $H1 :=$ the root of the heap
3: insert $[r, 1, \mathrm{mon}(a_r) \cdot \mathrm{mon}(b_1)] = a_r \times b_1$ into H
4: $k := N/p$ ▷ the counter used as the signal of global heap merge
5: **while** $|H| > 0$ **do**
6: $(i, j, M) := extractMax(H)$
7: $C := \mathrm{cof}(a_i) \cdot \mathrm{cof}(b_j)$
8: $Q := \{(i, j)\}$
9: **while** ($|H| > 0$ and $peek(H) = M$) **do** ▷ Combine the like term if exists
10: $(i, j, M) := extractMax(H)$
11: $C := C + \mathrm{cof}(a_i) \cdot \mathrm{cof}(b_j)$
12: $Q := Q \cup \{(i, j)\}$
13: **end while**
14: **for all** $(i, j) \in Q$ **do** ▷ insert the next nodes of the ones just been extracted
15: **if** $(j < n_b)$ **then**
16: insert $a_i \times b_{j+1}$ into H
17: **end if**
18: **if** ($j = 1$ and $i + p \leq n_a$) **then**
19: insert $a_{i+p} \times b_1$ into H
20: **end if**
21: **end for**
22: **if** $(C \neq 0)$ **then**
23: insert the term (C, M) into the buffer B
24: **end if**
25: $k := k - 1$
26: **while** $(k = 0)$ **do**
27: $k := |B|$ ▷ the number of terms stored in buffer now
28: **if** $(trylock(L))$ **then**
29: global_heap_merge($\min(k, N/p)$)
30: $release(L)$
31: **else if** $(k = N)$ **then**
32: sleep for 10 microseconds
33: **end if**
34: $k := \min(N - |B|, N/p)$ ▷ determine the workload for the next iteration
35: **end while**
36: **end while**
37: close(B)
38: **return**

Global Heap Merge

After the counter k runs out, the `GLOBAL_HEAP_MERGE` operation takes over. This subroutine is tasked with combining the terms from buffers of all threads into a single, globally ordered set. It requires careful synchronization to ensure that while one thread is merging terms into the global heap, others are not interfering or causing conflicts.

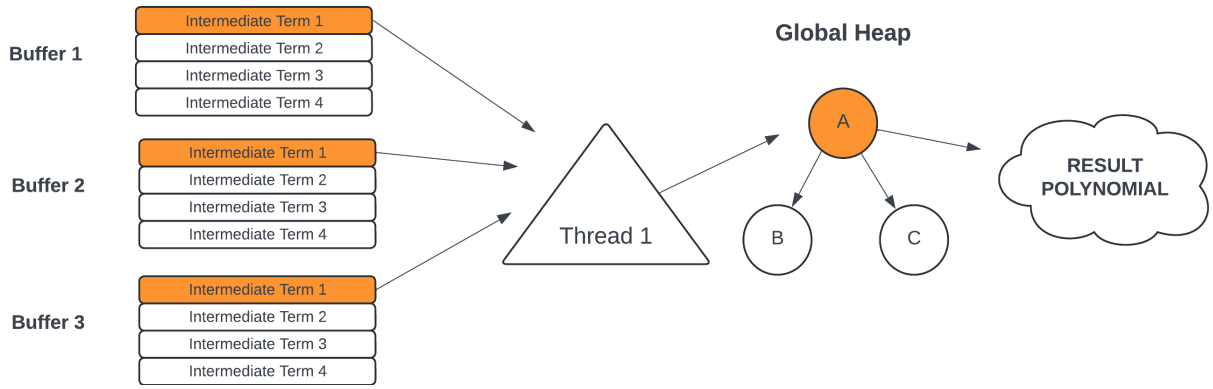

Figure 11. Global heap merge

It's important to note that `GLOBAL_HEAP_MERGE` can be called by any thread that is ready and has acquired the lock. This means all threads share the responsibility of performing the global merge. We use a global set P, which contains references to buffers, to keep track of which buffers have just been read, functioning similarly to set Q. And P is initialized with all the buffer references at the beginning. Each element in the global heap is of the form $[B, C, M]$, where C and M represent the coefficient and monomial, respectively, and B is the pointer indicating the source buffer of the result.

When `GLOBAL_HEAP_MERGE` is called for the first time, the global heap is initialized by reading an element from every buffer. If one or more buffers are empty, meaning the thread cannot read from them, the thread will abort this global merge attempt and release the lock. This is necessary because the global merge can only proceed if it has data from every thread (buffer); it cannot determine a global ordering of the terms without knowledge of

the maximum terms from each thread. The first global merge succeeds only when every buffer has at least one element, allowing the global heap to be initialized. Once an element from each buffer is inserted into the heap, the process of extracting from the heap begins, combining like terms if they exist. Each time a thread reads an element from a buffer, that buffer reference is removed from P, and only when a term is extracted from the global heap, the reference of that term's originating buffer is added back to P. Finally, the term $[C, M]$ is appended to the result polynomial c. Following this, in the next iteration of global merge, we go through P to read from buffers whose elements were just extracted and insert their next elements into the heap. This approach helps maintain the global heap at a relatively small size, equal to the number of buffers.

`GLOBAL_HEAP_MERGE` will be terminated under three situations. First, when the global heap is empty, indicating that all buffers have stopped sending terms, which is straightforward to understand. Second, if we attempt to read from a buffer and find it empty without the indication of $close(B)$, meaning it is still computing and sending elements, we abort the global heap. This design ensures the work process of each thread remains consistent. Lastly, `GLOBAL_HEAP_MERGE`, being a large while loop, will terminate when the maximum iteration i is reached, where $i = \min(N - k, N/p)$.

Algorithm 3 GLOBAL_HEAP_MERGE
Input: max iterations i; Return: terms of the result are written to c;
Local: coefficients C, monomial M, buffer B;
Global: heap G, buffer reference set P, polynomial c

```
1: while i > 0 do
2:     i := i − 1
3:     for all B in P do            ▷ Makes sure that every buffer has one term in the heap
4:         if B is not empty then
5:             extract the next term (C, M) from the buffer B
6:             insert [B, C, M] into G
7:             P := P \ {B}         ▷ Exclude buffer B in the set P then go for next buffer
8:         else if not isClosed(B) then
9:             return  ▷ Abort the global heap if a buffer is empty but is till sending terms
10:        end if
11:    end for
12:    if |G| = 0 then
13:        return
14:    end if
15:    (B, C, M) := extractMax(G)
16:    P := {B}
17:    while |G| > 0 and peek(G) = M do
18:        (B, K, M) := extractMax(G)
19:        C := C + K
20:        P := P ∪ {B}
21:    end while
22:    if C ≠ 0 then
23:        append the term (C, M) to c
24:    end if
25: end while
26: return
```

The Main Function: SDMP Multiplication

Finally, we discuss the overall process, which is also the main function of the multiplication. Before diving into the main function, some global variables must be initialized outside since they need to be shared by all other functions. These are the global heap G, buffer reference set P, lock L, buffer capacity N, and result polynomial c. It is important to note before returning the resultant polynomial c, `GLOBAL_HEAP_MERGE` needs to be called again with input $(n_a \cdot n_b)$. This step acts as a final check to ensure no terms in the global heap or

local buffer are missed. The algorithm is completed by returning c.

Algorithm 4 SDMP_MULTIPLICATION
Input: $a, b \in \mathbb{Z}[x_1, \ldots, x_n]$, monomial order $>$, number of threads p;
Return: $c = a \cdot b$;
Global: heap G, set P, lock L, buffer capacity N (in terms), result polynomial c.

```
1: G := an empty max binary heap with root element called G_1
2: P := a set of p empty buffers
3: L := an unheld lock
4: c := 0
5: lock(L)
6: for i from 1 to p do
7:     spawn localHeapMerge(a, b, P_i, i, p)
8: end for
9: release(L)
10: synchronize threads
11: GlobalHeapMerge(n_a · n_b)
12: return c
```

4.2 The TRIP Algorithm

In sparse polynomial multiplication, achieving scalability on multicore processors is challenging due to the need for efficient memory and thread management. Existing algorithms often exhibit sub-linear speedup with a limited number of cores, and performance tends to decline as more cores are added. This is especially true for algorithms designed for single-processor computers with multiple cores sharing a large cache, like SDMP. In these systems, each thread uses a private heap for sorting its results and accesses a global heap to compile the final polynomial output. However, this concurrent access requires a lock mechanism to prevent race conditions, significantly limiting scalability when many cores are involved. This lock mechanism becomes a bottleneck, reducing efficiency.

In this section, we introduce a highly scalable algorithm called TRIP, developed by Gastineau and Laskar in 2013 (9). This method is optimized for diverse hardware environments, including multicore processors, GPUs, and computer clusters. By minimizing

synchronization and avoiding locks, it achieves remarkable scalability and efficiency across various platforms.

Algorithm

TRIP is crafted to minimize synchronization or lock statements between threads to achieve scalability on many-core systems. To achieve this, we need to design a strategy to ensure that each thread computes independent terms to avoid any locks or synchronization during computation. Especially when a distributed format is used for polynomials, the division of work among threads has become more complex. This algorithm is divided into two major steps.

The preliminary step involves splitting the work between threads in a way that avoids any communication between them during the computational task. The multiplication of two polynomials is expressed as the sum of the product of their terms, which can be visualized using a matrix called the $pp - matrix$ as shown below. This is similar to the concept of streams introduced earlier, where row i of the pp-matrix encodes the exponents of stream i in the heap-based multiplication algorithm. The set of all possible $\gamma_{i,j}$ values forms an interval $[\gamma_{1,1}, \gamma_{n_a,n_b}]$. Each thread must compute independent terms, meaning it has to process all the pairwise term products of the pp-matrix that share the same value for $\gamma_{i,j}$.

$$
\begin{array}{c}
 \\
\alpha_1 \\
\vdots \\
\alpha_i \\
\vdots \\
\alpha_{n_a}
\end{array}
\begin{array}{c}
\begin{array}{ccccc} \beta_1 & \cdots & \beta_j & \cdots & \beta_{n_b} \end{array} \\
\left[\begin{array}{ccccc}
\gamma_{1,1} & \cdots & \gamma_{1,j} & \cdots & \gamma_{1,n_b} \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
\gamma_{i,1} & \cdots & \gamma_{i,j} & \cdots & \gamma_{i,n_b} \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
\gamma_{n_a,1} & \cdots & \gamma_{n_a,j} & \cdots & \gamma_{n_a,n_b}
\end{array}\right]
\end{array}
$$

Figure 12. A pp-matrix.

The key idea is to have more intervals than the number of threads. This way, a thread

that finishes quickly can work on another interval (i.e., task). By applying the following formulas, we obtain the set S^*, which ensures a well-balanced distribution of all elements used for partitioning the pp-matrix. The argument l can be adjusted for different input polynomial sizes. The greater the value of l , the more partitions there will be. However, selecting the values based on the set S^* will almost always result in duplicate $\gamma_{i,j}$ values. To address this, we remove any duplicates among the selected values. Next, we sort the set in decreasing monomial order (or increasing order if preferred) to obtain set S. The unique and sorted $\gamma_{i,j}$ values in S then become the bounds of the subintervals. These subintervals are defined as left-closed and right-open (left-closed and right-closed for the last subinterval) to ensure that all $\gamma_{i,j}$ values within the same subinterval are grouped together.

$$S_k^* = \begin{cases} \alpha_i + \beta_j & \text{for } i = 1 \text{ to } n_a \text{ step } \left\lfloor \frac{n_a}{l} \right\rfloor, \\ & \text{and for } j = j_{0,i} \text{ to } n_b \text{ step } \left\lfloor \frac{n_b}{l} \right\rfloor \\ \alpha_{n_a} + \beta_j & \text{for } j = 1 \text{ to } n_b \text{ step } \left\lfloor \frac{n_b}{l} \right\rfloor \\ \alpha_i + \beta_{n_b} & \text{for } i = 1 \text{ to } n_a \text{ step } \left\lfloor \frac{n_a}{l} \right\rfloor \end{cases}$$

$$with \begin{cases} n_s^* = (l+1)^2 \\ j_{0,i} = 1 + \left(\left\lfloor \dfrac{i}{n_a/l} \right\rfloor \mod 2 \right) \left\lfloor \dfrac{n_b}{2l} \right\rfloor \end{cases}$$

Choice of the Set S^*

This step is critical as it ensures that the workload can be distributed among the threads in such a way that no two threads need to communicate or synchronize while computing their assigned portions of the pp-matrix. This is foundational for the algorithm's parallel efficiency on shared memory systems.

The second step of the algorithm involves the main computational work: parallel process-

Algorithm 5 TRIP_MULTIPLICATION
Input: $a = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$, $b = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$, n_s^* : integer number of intervals;
Return: $c = a \cdot b$

1: // First step
2: $S^* \leftarrow$ Compute n_s^* exponents $\gamma_{i,j} = \alpha_i + \beta_j$ using an almost regular grid over the pp-matrix associated to a and b
3: $S \leftarrow$ sort S^*using the monomial order $>$
4: remove duplicate values from S
5: // S has now n_s sorted elements
6:
7: // Second step
8: Initialize an array D of n_s empty containers for the result
9: **for** $k = 1$ to $n_s - 1$ **do** ▷ compute in parallel
10: $(L_{min}, L_{max}) \leftarrow$ FindEdge(A, B, S_k, S_{k+1})
11: $D_k \leftarrow$ MergeSort(a, b, L$_{min}, L_{max}$)
12: **end for**
13: $c \leftarrow$ concatenate all containers of D using ascending order

ing of the terms of the result polynomial. In this step, each subinterval represents a task to be executed in parallel. Within each task, a thread uses a MergeSort algorithm to compute the subset of terms of the result polynomial whose exponents fall within the subinterval. While MergeSort can be any serial multiplication algorithm, it is almost always a variation of the heap-based serial algorithm. How does the thread find all its related work in the *pp*-matrix? The function *FindEdge* plays a critical role. It helps to determine the processing edges for each thread by finding the first and last columns j where $S_k \leq \gamma_{i,j} < S_{k+1}$ on each row i of the pp-matrix. That is, FindEdge computes the partition of the pp-matrix which contains all monomials in the specific subinterval. These edges are stored in two arrays, L_{min} and L_{max}, each with a length equal to the number of terms in the first polynomial, denoted by n_a. As illustrated below, for polynomials a and b of length six, one possible subinterval may be reflected on the pp-matrix as shown in the orange cluster. The corresponding L_{min} and L_{max} arrays for this orange cluster are L_{min} = [2, 1, 0, 0, 0, -1] and L_{max} = [4, 4, 3, 1, 0, -1], where -1 indicates that there is no $\gamma_{i,j}$ in the row belonging to this subinterval.

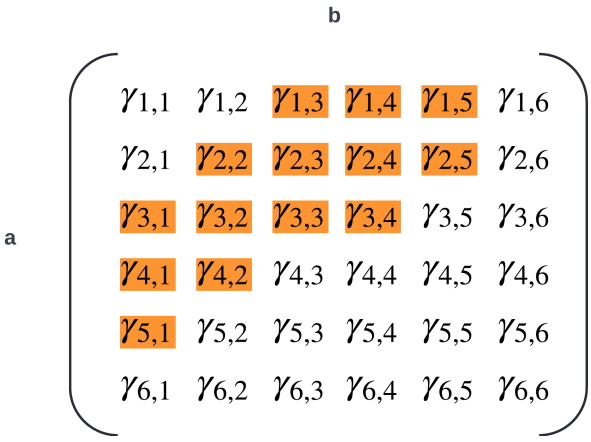

Figure 13. An example of FindEdge in pp-matrix

The time complexity for calculating these arrays is $O(n_a + n_b)$, where n_b is the number of terms in the second polynomial. This efficiency comes from the fact that the computation does not restart from the last column for each new row (we iterate the row from right to left to accommodate a decreasing monomial order), but continues from the first found column in the above row, taking advantage of the ordered structure of the pp-matrix where $\gamma_{i,j} > \gamma_{i+1,j}$. Using the same example in Figure 12, the first row is iterated from $\gamma_{1,6}$, the rightmost element, to the left, and stops at $\gamma_{1,2}$, since $\gamma_{1,1}$ is guaranteed to be not in the subinterval if $\gamma_{1,2}$ is not, there is no need to go further. The iteration on the second row starts from column $j = 5$, $\gamma_{2,5}$, since the first $\gamma_{i,j}$ found in the above row is in the second last column! We keep doing this until the whole pp-matrix is exhausted, as in some special cases, the cluster may not be connected but separated.

Each thread processes a subset of the total exponents, based on the intervals calculated. The number of intervals, n_s^*, is kept small but greater than the number of threads to ensure that the load is well-balanced across the threads. All the tasks (intervals) with their index and corresponding boundaries can be stored in a priority queue $TaskQueue$. After completing their current task, threads can dequeue the next task from TaskQueue if it is not empty. This process needs to be protected by a lock to avoid competition. Threads use their respective

L_{min} and L_{max} arrays to compute the sums of their polynomial terms $a_i b_j x^{\gamma_{i,j}}$. These are then stored in a container D at the position associated with the corresponding task index, avoiding the merging of like terms between threads, as each thread operates within its unique interval. After TaskQueue is exhausted and all threads complete their work, the result polynomial can be simply constructed by concatenating all sub-polynomials in the container D.

4.3 The Chiplet Algorithm

In this section, we will introduce a brand new algorithm for sparse polynomial multiplication, which is also the core of this report. This innovative algorithm, called the Chiplet algorithm, is designed to address the limitations of existing methods, specifically SDMP and TRIP. By combining the strengths of both SDMP and TRIP, the Chiplet algorithm leverages the advanced architecture of chiplets to achieve efficient parallel processing, reduced memory overhead, and better load balancing. This section will provide a comprehensive overview of the Chiplet algorithm, detailing its structure and explaining its primary functions, which are `CHIPLET_MULTIPLICATION`, `LEADER_THREAD_MAIN`, `LOCAL_HEAP_MERGE`, and `GLOBAL_HEAP_MERGE`.

Algorithm Overview

The algorithm begins by dividing the pp-matrix, into multiple tasks. These tasks are then assigned to different chiplets in the system. Each chiplet comprises several cores, with one core designated as the leader thread and the others as worker threads. The leader thread is responsible for managing tasks, spawning worker threads, and coordinating the computation process. Once a task is assigned, the leader thread and its worker threads collaboratively perform local heap merges to compute partial results. These partial results are then combined using a global heap merge process, producing the final output polynomial. After completing

a task, the leader thread dequeues another task from the task queue and repeats the process. This continues until all tasks are exhausted, at which point the algorithm completes.

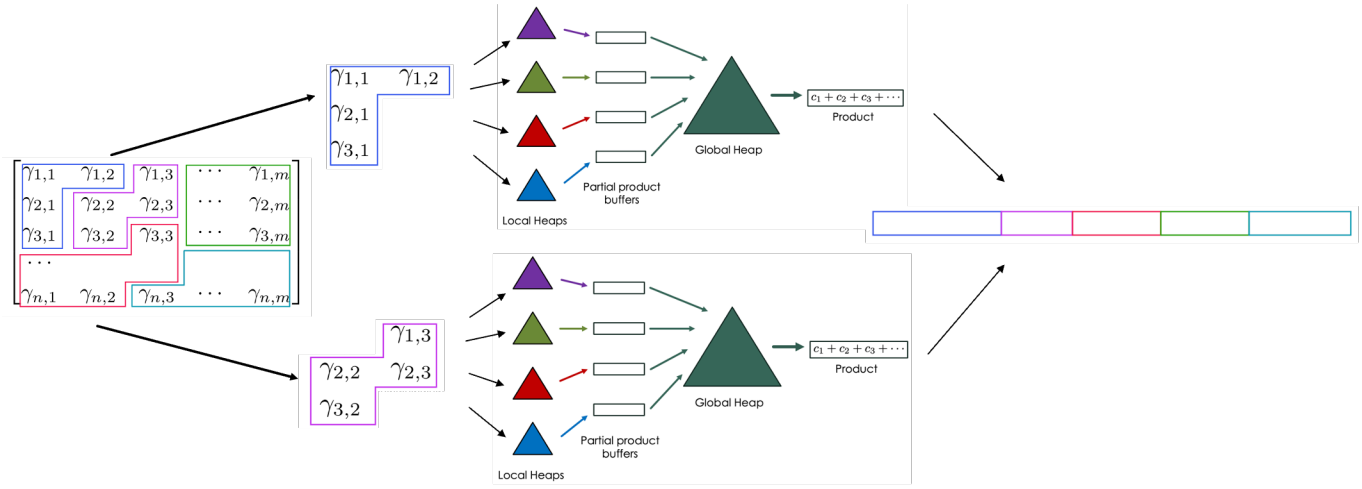

Figure 14. Chiplet algorithm overview

The Chiplet algorithm ingeniously combines the strengths of both SDMP and TRIP to enhance performance and efficiency. The SDMP algorithm is known for distributing the terms of the result polynomial across threads by assigning different streams to different threads. Using the shared cache, threads can efficiently communicate and cooperate, particularly in moving data between buffers and the global heap. However, SDMP assumes that local heaps, buffers, and the global heap all fit in a shared cache, which is not always feasible given the partitioned nature of the L3 cache on chiplet CPUs. Additionally, the explicit cooperation and synchronization around the global heap limit scalability due to mutual exclusion, which becomes more pronounced as more cores are involved.

On the other hand, the TRIP algorithm excels in task partitioning and assignment, ensuring high data affinity within each task assigned to a thread, and facilitating easy concatenation for the final result. However, TRIP faces challenges in load balancing as it requires a significantly higher number of regions compared to the number of threads. The partitioning work itself is substantial, increasing with the number of regions and input size, and there is

no guaranteed data locality between tasks, leading to competition for shared L3 cache.

The Chiplet algorithm addresses these issues by integrating the parallelism of SDMP with the dynamic task management of TRIP. By partitioning pp-matrix into tasks, the Chiplet algorithm ensures that each chiplet makes the most use of its own L3 cache. Additionally, the use of local and global heap merges optimizes memory usage and reduces computation time. This results in a highly efficient and scalable algorithm that leverages the advanced capabilities of chiplet architecture to effectively perform sparse polynomial multiplication.

Main Function: Chiplet Multiplication

The Chiplet Multiplication is the main function of the Chiplet algorithm. It orchestrates the overall process by spawning leader threads, which in turn manage worker threads to perform local heap merges and global heap merges, and finally coordinate the computation of the polynomial product.

The number of leader threads is determined by the number of L3 caches, and thus the number of CCXs (see Section 2.1) in the system. Each chiplet provides a leader thread (the first core in that chiplet) and several worker threads (the remaining cores in that chiplet). The *Leader_threads* array is initialized to identify the cores designated as leader threads. To partition the pp-matrix into manageable tasks, the function computes a set of intervals, which define the boundaries of each task. The number of intervals can be tailored based on the different polynomial input sizes by the argument l, which is discussed in the TRIP algorithm.

A concurrent queue, $TaskQueue$, is used to store these tasks, ensuring thread-safe access during the dequeue operation. Each task is represented by an $startExp$, $endExp$, and $taskID$, indicating the interval of exponents to be computed and the location of the product polynomial to be stored. The function then initializes an empty array of type sparse

polynomial, *Containers* , to store the partial results of the polynomial multiplication, with each element corresponding to a task. Thus, the length of *Containers* should be equal to *TaskQueue*.

The leader threads are spawned by the main thread based on the number of chiplets, with each leader thread pinned to a specific core (i.e., the first core index of each chiplet), because we want to ensure that there is only one leader thread per chiplet. The leader threads are then assigned to `LEADER_THREAD_MAIN`, details of which will be explained later. Once all tasks are completed, the partial results stored in Containers array are simply concatenated to form the final polynomial product. The result is returned by `CHIPLET_MULTIPLICATION`, completing the computation.

Algorithm 6 CHIPLET_MULTIPLICATION
Input: $a = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$, $b = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$, # cores in chiplet $nThread$, # chiplets N;
Return: $c = a \cdot b$

if a or b is zero **then**
    return zero polynomial b or a
end if
for i from 0 to N - 1 **do** ▷ Define leader threads
    Add ($i\times$ nThreads) to Leader_threads
end for
$S \leftarrow$ Compute n_s exponents $\gamma_{i,j} = \alpha_i + \beta_j$ using the pp-matrix for result polynomial partition
TaskQueue $\leftarrow$ Compute Length($S-1$) tuples of (*startExp, endExp, taskID*) into a thread-safe queue
Containers $\leftarrow$ Initialize an empty Vector of sparse polynomials of length (Length (*TaskQueue*))
sync block
for each leader_id **in** Leader_threads **do**
    Pin a leader thread to the core with ID = *leaderID*.
    leader_thread_main(leaderID, TaskQueue, a, b, Containers, nThreads)
end for
end sync block
$c \leftarrow$ Concatenate all sparse polynomials in Containers
return c

However, one may be concerned that the current main thread, the one spawns all leader threads, would need to call Leader Thread Main for itself, considering itself as a leader thread.

However, the main thread does not need to contribute to the computation. In the context of the Chiplet algorithm, a software thread is an abstraction managed by the operating system, independent of the specific hardware resources (cores or hardware threads). Software threads can outnumber hardware threads and can exist in an idle state. The main thread can remain idle, simply waiting for the sync block to finish before resuming its tasks. This is illustrated in the diagram below, where the main thread spawns multiple threads within a sync block and then goes idle. The spawned threads execute their tasks, and once they are finished, control passes back to the main thread, which resumes at the end of the sync block. This approach ensures that the main thread does not interfere with the computation performed by the leader and worker threads.

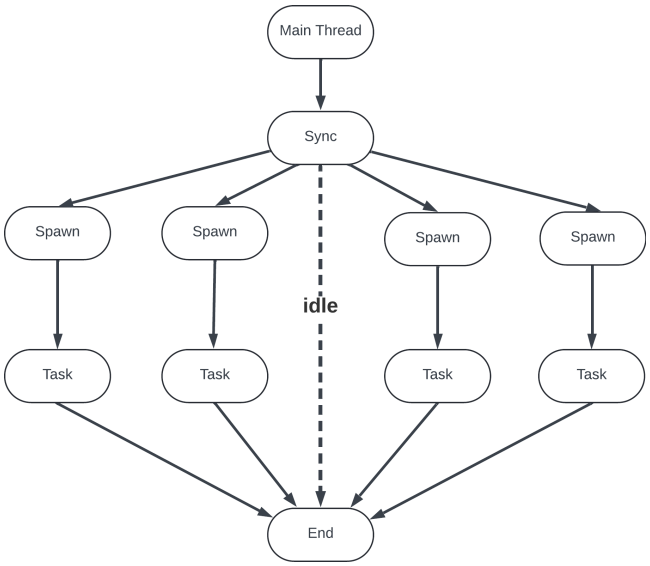

Figure 15. Thread spawning and execution flow

Leader Thread Main

The `LEADER_THREAD_MAIN` function consists of a series of tasks that each leader thread needs to complete. This function ensures efficient and parallel execution of the polynomial multiplication process within each chiplet. The leader thread initializes a set of buffers and a

global heap, with each core in the chiplet owning its buffer. The number of buffers equals the number of cores in the chiplet. The global heap, used by all threads in the chiplet, needs to be initialized with a reentrant lock to ensure thread-safe access and manipulation of shared data.

The leader thread then enters a while loop to process tasks from TaskQueue. It dequeues a task if TaskQueue is not empty. If it's empty, the loop terminates, and the leader thread exits. Otherwise, the leader thread initializes a new empty result polynomial and a global heap for the current task's computations. Next, the leader thread determines the edges of the pp-matrix by calling FindEdge based on the start and end exponents in the task. This function returns two arrays L_{min} and L_{max}, based on which, we can find some edge information, including the total number of rows of the task in the pp-matrix, the start row, and the end row, and they can be encapsulated in the $edge\_info$ structure. This structure will be passed to worker threads to help them identify their specific rows to work on during the local merge, which will be explained in more detail later.

The leader thread then spawns worker threads to perform local merge, ensuring they are pinned to the same chiplet. Each worker thread, along with the leader thread, is assigned a specific CPU core within the same chiplet to optimize performance. All threads then call `LOCAL_HEAP_MERGE`, performing the actual computation for their assigned portion of the pp-matrix using the SDMP algorithm. Similar in SDMP, once all threads in a chiplet finish their work, the leader thread needs to invoke the global heap merge again to ensures there are no remaining terms in any buffers or the global heap, acting as a double-check process. The leader thread then stores the result in the appropriate position in Containers based on taskID.

Algorithm 7 LEADER_THREAD_MAIN
Input: leaderID, TaskQueue, polynomial a, b, Containers, nThread;
Return: void (However, save intermediate product polynomials into Containers)

bufferSet ← Initialize buffers for each thread in a chiplet (size = *nThread*)
while TRUE **do**
    task ← Pop a task from TaskQueue
    if *task* is nothing **then**
        break ▷ Exit if the task queue is empty
    end if
    prodPolynomial, globalHeap ← Initialize an empty product polynomial and an empty global heap (size = *nThread*)
    (*Lmin, Lmax*) ← Find the edges of the pp-matrix based on the task
    edgeInfo ← Encapsulatation of *(Lmin, Lmax), totalRows, startRow, and endRow*
    sync block
    for *i* from *leaderID* + 1 to (*leaderID* + *nThread* - 1) **do**
        workerID = *i*
        Pin a worker thread to the core with ID = *workerId*
        local_heap_merge (a, b, workerID, edgeInfo, bufferSet, globalHeap, nThread)
    end for
    local_heap_merge (a, b, leaderId, edgeInfo, bufferSet, globalHeap, nThread)
    end sync block
    $global\_heap\_merge(n_a, n_b)$
    Containers[task.taskID] = *prodPolynomial*
end while

Local Heap Merge

The `LOCAL_HEAP_MERGE` in the Chiplet algorithm is quite similar to the one in SDMP. The primary difference lies in how the work to be performed is distributed among the threads. In SDMP, each thread works on multiple streams (the entire stream from $\gamma_{i,1}$ to γ_{i,n_b}, where i indicates the i-th row in pp-matrix). However, in the Chiplet algorithm, since the pp-matrix is partitioned into several task clusters, a thread cannot work on the entire row, but rather the subset of each row appearing within the pp-matrix partition.

The function begins by determining the buffer index for the current worker thread. This is achieved by calculating the modulo of the worker ID and the number of threads. Each worker thread has its own buffer to store intermediate results. The buffer capacity is then

obtained to determine how much data can be stored before needing to write to the global heap. The capacity of the buffer is not fixed; it should be tailored for different sizes of input polynomials to achieve optimal performance.

Then the local heap is initialized for the worker thread. In the context of a task cluster in the pp-matrix, we cannot take advantage of the property where $\gamma_{i,j} > \gamma_{i,j+1}$ and $\gamma_{i,j} > \gamma_{i+1,j}$, since the shape of the cluster is irregular, as shown in Figure 15. Consequently, we cannot initialize the heap with only one node (the head of the first row that a thread works on); instead, a thread needs to initialize the heap with the head of every row that it is responsible for. This is a compromise that has to be made for designing the Chiplet algorithm. Each thread can find the first row it needs to start with using the formula

start_row + mod((thread_id - mod(start_row, nThreads)), nThreads)

and then jump to the next row by the number of threads within a chiplet until it goes out of the bound, the end row.

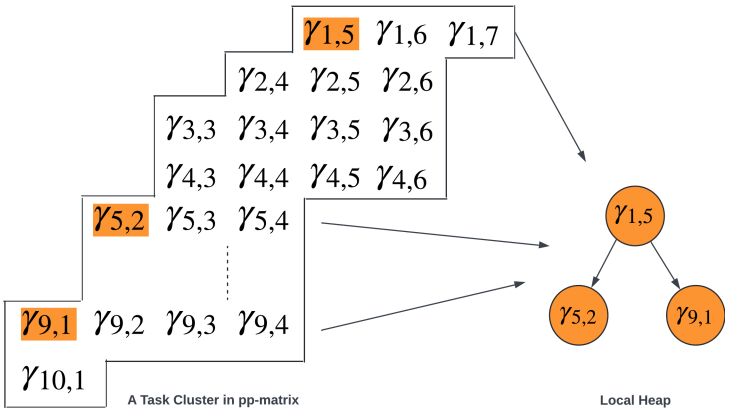

Figure 16. An example of heap initialization in chiplet with 4 cores

The algorithm then follows the same steps as the SDMP algorithm with one crucial difference. When a term is extracted from the heap, we do not blindly insert successor terms until the stream is empty (i.e., until the term $a_i \cdot b_{n_b}$ is extracted). Instead, we use the L_{max}

array to determine whether the successor is part of the cluster being processed.

Global Heap Merge

The global heap merge stays exactly the same as SDMP.

5 Implementation in Julia

5.1 Coefficient, Exponent Packing, and Sparse Polynomial

For handling coefficient types in our implementation, we utilize the `XInts.jl` package (18). This package is specifically designed to manage integers efficiently, particularly when dealing with the limitations of fixed-bit-width storage. Fixed-bit-width storage refers to the constraints of storing numerical values within a fixed number of bits, such as 32-bit or 64-bit integers, which can overflow when their maximum value is exceeded. The `XInts.jl` package employs a clever technique to handle overflow: when a 64-bit integer exceeds its storage capacity, it dynamically switches to storing a pointer to a multi-precision integer instead. This approach ensures that the integer can grow as needed without losing precision or incurring significant performance penalties.

The multi-precision integers in `XInt.jl` are implemented using GMP (the GNU Multiple Precision Arithmetic Library), which is widely recognized as the standard for multi-precision arithmetic. GMP is also the underlying library for Julia's built-in `BigInt` type, ensuring robust and efficient handling of large integers. By leveraging GMP, `XInt.jl` provides a seamless integration of multi-precision arithmetic, allowing for accurate and efficient computation even when dealing with very large numbers.

For dealing with exponent packing, the `ExpVec` structure is created to store the packed

exponents using a single unsigned integer.

```
mutable struct ExpVec{T<:Unsigned, N}
    vec::T
end
```

Listing 6. Structure of ExpVec

Here, `T` is the unsigned integer type used for storage, which can be `UInt8`, `UInt16`, `UInt32`, `UInt64`, or `UInt128`. `N` is an integer indicating the number of variables in the polynomial, and this makes the number of variables part of the type itself.

To manipulate the exponents within an `ExpVec`, various functions are implemented. These include functions to get, set, and unpack the exponents, which facilitate efficient handling of polynomial terms. The `getExp` function retrieves the exponent of a specific variable, while the `setPartialDegree` function sets the exponent of a variable, ensuring it does not exceed the maximum allowed value. The `unpackExpVec` function unpacks the exponents into a vector of integers, making them accessible for further processing.

The implementation uses bit masks and shift amounts to efficiently encode and decode the exponents. These constants are defined for various bit-widths (8-bit, 16-bit, 32-bit, 64-bit, and 128-bit) to handle different sizes of exponents. Each constant specifies the bit masks and the corresponding shift amounts needed to extract the partial degrees from a packed monomial.

For example, in the case of 8-bit exponents:

```
########################
# 8-bit exp vectors
########################
const _uint8MasksOffs::Array{UInt8,1} = Array{UInt8,1}([
    #NVAR 1
    0xff,
    0,
    #NVAR 2
    0xf0,
    4,
    0x0f,
    0,
    #NVAR 3
    0xc0,
    6,
    0x38,
    3,
    0x07,
    0,
    #NVAR 4
    0xc0,
    6,
    0x30,
    4,
    0x0c,
    2,
    0x03,
    0
]);
```

Listing 7. 8-bit Exponent Vectors

In this example, the masks and shifts are defined to handle up to four variables, with each

variable's exponent occupying specific bits within the unsigned integer. Similar definitions exist for other bit-widths, ensuring efficient packing and unpacking of exponents.

Following the `ExpVec` structure, the `Term` and `SparsePolynomial` structs are implemented to manage the coefficients and exponents of polynomial terms efficiently. These structures build upon `ExpVec` to form a comprehensive representation of sparse polynomials.

The `Term` structure represents a single term within a sparse polynomial, consisting of a coefficient and an exponent vector. Below, `C` denotes the coefficient type, which must be a subtype of `Number`, and `N` denotes the number of variables.

```
mutable struct Term{C<:Number, N}
    coef::C
    exp::ExpVec{T,N} where {T<:Unsigned}
end
```

Listing 8. Structure of Term

Finally, the `SparsePolynomial` structure encapsulates a sparse multivariate polynomial. It consists of a list of terms, each represented by the `Term` structure, and a list of variable symbols.

```
mutable struct SparsePolynomial{C<:Number,N}
    nvar::Int
    vars::Vector{Symbol}
    terms::Vector{Term{C,N}}
end
```

Listing 9. Structure of Sparse Polynomial

The `SparsePolynomial` structure also includes several utility functions to handle zero and one polynomials, check equality, and manage the internal list of terms. For instance, the functions `iszero` and `one` allow for the creation and identification of zero and one polyno-

mials, respectively. A zero polynomial has all its coefficients equal to zero, representing the additive identity, while a one polynomial has a constant term of one and all other coefficients zero, representing the multiplicative identity. The equality check function `isequal` ensures that two `SparsePolynomial` instances can be compared accurately, taking into account both the coefficients and the exponents of the terms. Additionally, there are functions to push new terms into the polynomial, condense terms with like exponents, and sort the terms in a canonical form. These utilities make it easier for `SparsePolynomial` to perform various polynomial operations efficiently.

5.2 Task Management and Thread Pinning

We use the `ThreadPinning.jl` package (19) to perform thread pinning, which enhances the performance of our parallel computations by ensuring that threads run on specific CPU cores. The following source code snippets illustrate the process of manually creating a Julia task, assigning it a thread ID, pinning the corresponding thread to a desired core, and scheduling the task for execution.

```
function Chiplet_Multiplication(a::SMP{C,N}, b::SMP{C,N}, threadConfig)
    leaders, threadConfig = parseThreadConfig(threadConfig)
    tasks = partitionPPMatrix(a,b)
    results = Vector{SMP{C,N}}(undef, length(S)-1)
    asyncTasks = Task[]
    for i in 2:length(threadConfig)
        taskfunc() = leader_thread_main(a,b,tasks,results,leaders[i],
        threadConfig[i])
        t = Task(taskfunc)
        ccall(:jl_set_task_tid, Cvoid, (Any, Cint), t, leaders[i]-1)
        t = errormonitor(t)
        schedule(t)
        push!(asyncTasks, t)
    end
    leader_thread_main(a,b,tasks,results,leaders[1],threadConfig[1])
    for task in asyncTasks
        wait(task)
    end
    return concatenatePolys(results);
end
```

Listing 10. Core Part of `Chiplet_Multiplication` Function

Due to size constraints, only the core part of the `Chiplet_Multiplication` function is displayed. The `threadConfig` parameter plays a crucial role in this implementation by determining the number of leader threads and the corresponding worker threads for each leader. Specifically, if `threadConfig` is a vector of integers, each element represents the total number of threads in a group. For instance, a `threadConfig` of `[4,4,8]` indicates a total of 16 threads, organized into two groups of 4 and one group of 8, with leader threads assigned IDs of 1, 5, and 9 respectively.

The function initializes leader threads based on the `threadConfig` parameter. For each

leader thread specified in `threadConfig`, a corresponding Julia task `t` is created using `Task()`, each executing the `leader_thread_main` function. The `ccall(:jl_set_task_tid, Cvoid, (Any, Cint), t, leaders[i]-1)` function binds `t` to the specified leader thread, ensuring that the task remains on this thread to prevent migration. Setting a task's thread ID is a low-level operation in Julia and must be done through a call to the C library. The `ccall` method performs this call taking as arguments: the function to call, the return type, argument types, and the actual arguments to pass to the called function. After this, the `schedule(t)` function schedules the task for execution, and `wait(t)` ensures that the main thread waits for all tasks to complete. It should be noted that the parameter `tasks` passed in the function `leader_thread_main`, which is distinct from the Julia task `t`, refers to the exponent intervals of the result polynomial. These intervals are used by each group of threads for the actual computations.

```
function leader_thread_main(a, b, tasks, results, leaderIdx, nThreads)
    # Pin the thread to a specific core based on leaderIdx
    pinthread(leaderIdx-1)

    # Other code is omitted for brevity
    ...
end
```

Listing 11. Implementation of the leader thread function for Chiplet Multiplication.

Upon entering the `leader_thread_main` function, the initial action of the leader thread is to pin itself to a specific core using `pinthreads(leaderIdx-1)`. This adjustment accounts for Julia threads being 1-indexed and core IDs being 0-indexed. Following this, the same task management and thread pinning techniques used for leader threads are applied to worker threads within the function. This ensures that each leader thread and its associated worker threads are grouped and pinned according to the `threadConfig`. Each thread then carries out its own local merge operation, contributing to a sub-result polynomial stored in a

designated position of the container. Detailed steps of this process are outlined in Algorithm 7.

6 Experimentation and Discussion

6.1 Experimental Setup

Experiments were conducted on a system equipped with an AMD EPYC 9554P 64-Core Processor and 12x48GB DDR5 RAM at 4.8MHz, running Ubuntu 20.04.6. All tests were executed using Julia version 1.10.2, and timing data was collected using the `Benchmark.jl` package (20).

We examine two sets of benchmarks for polynomial computations. The first set is inspired by Fateman's work (21). It features the polynomials defined as follows:

$$f = (1 + x + y + z + t)^e$$

$$g = f + 1$$

The second set, noted for its "very sparse" characteristics, is drawn from (5). This set includes the polynomials:

$$f = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^e$$

$$g = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^e$$

All results presented are the median of three trials, conducted after an initial unmeasured "warm-up" trial.

6.2 The SDMP Algorithm

The SDMP algorithm leverages the quick L3 cache interactions between threads, ensuring all buffers, local heaps, and the global heap remain within shared L3 cache to minimize slower main memory interactions. This design suits traditional multi-core architectures with a unified L3 cache. However, chiplet architectures, featuring multiple L3 caches, face increased costs in inter-cache data sharing. This effect is illustrated in Figure 17, with data collected using the core-to-core latency tool (22). Figure 17 depicts the communication latency between any two cores on the CPU. The data shows that sets of 8 sequential cores, which reside on the same chiplet and share an L3 cache, experience relatively fast communication. This is because communication within these cores occurs entirely on the same chiplet. However, when communication needs to occur across chiplets, data must be transferred through the I/O die, which significantly slows down the process due to the increased latency involved in inter-chiplet communication.

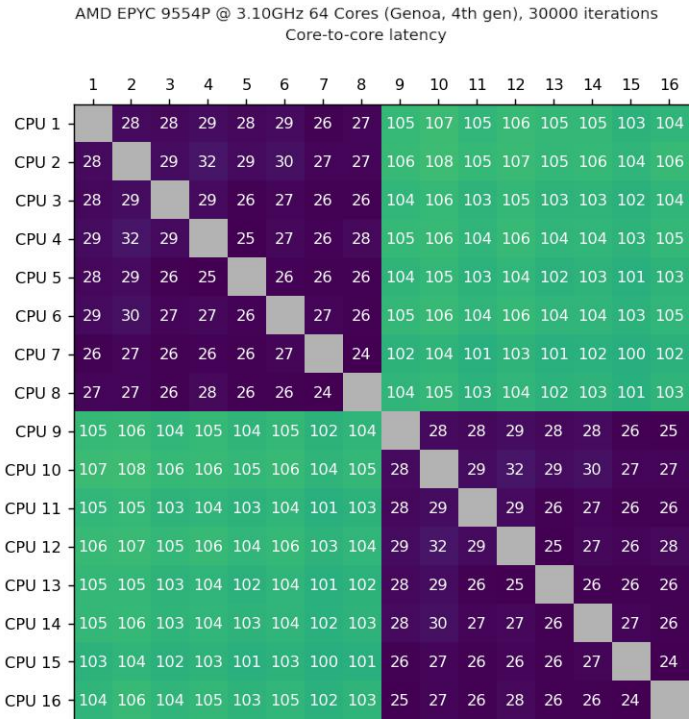

Figure 17. Core to core latency for the first 16 cores on the AMD EPYC 9554P. Note the improved communication times among cores on the same chiplet compared to across chiplets.

As demonstrated in Table 1, the SDMP algorithm's performance is assessed both with and without thread pinning across various degrees of e. Without thread pinning, the threads executing the SDMP algorithm may operate on different chiplets, significantly increasing the communication cost among these cooperating threads.

e	Time (s) Without Pinning	Time (s) With Pinning
10	0.067	0.058
14	0.271	0.258
18	0.860	0.791
22	6.825	6.511
26	28.837	27.736
30	68.793	64.534

Table 1. Execution time for the SDMP algorithm with 8 threads, in seconds, for $f \times g$ with $f = (1 + x + y + z + t)^e$ and $g = f + 1$, with and without thread pinning.

The communication overhead significantly escalates as more threads engage, due to their competition for the chiplet's interconnect, also referred to as the I/O die. Additionally, as observed in Table 2, increasing the number of threads tends to reduce overall execution speed. This slowdown is primarily driven by two factors: first, during the global merge phase of the algorithm, each thread must read data from every other thread's buffer, resulting in substantial data movement. Second, additional synchronization overhead occurs as more threads contend for the same lock on the global heap, further hindering performance. Consequently, further exploration of the SDMP algorithm was discontinued in favor of the TRIP and Chiplet algorithms.

Threads	Time (s)
8	68.793
16	66.193
24	65.787
32	71.626
40	80.270
48	88.076
56	96.009
64	104.663

Table 2. Execution time for the SDMP algorithm, in seconds, for $f \times g$ with $f = (x + y + z + 1)^{30}$ and $g = f + 1$ for various numbers of threads.

6.3 TRIP and the Chiplet Algorithm

In our comparison of the TRIP and Chiplet algorithms, we utilized the same partition value l for both, ensuring consistent pp-matrix partitioning across benchmarks. Notable improvements were observed with the Chiplet algorithm over TRIP in specific configurations, particularly for "threads = 32, $e = 14$" and "threads = 48, $e = 14$" as detailed in Table 3.

The performance gains were less pronounced in the very sparse benchmark, where the Chiplet algorithm generally underperformed compared to TRIP. Despite this, the Chiplet algorithm showed notable improvements as the size of the polynomials increased. This improvement is particularly evident in cases such as "Threads = 64, $e = 20$", as illustrated in Table 4, which highlights scenarios where the Chiplet algorithm excels with larger polynomial sizes.

The sparse benchmark is primarily memory-bound, where the limitation is the speed of memory access rather than computational throughput. This benchmark involves minimal computational tasks, such as identifying like terms and condensing them. As a result, the predominant bottleneck becomes the communication overhead associated with transferring terms across local heaps, buffers, global heaps, and managing inter-thread interactions, which

Threads	e	TRIP Time (s)	Parallel Speedup	Chiplet Time(s)	Parallel Speedup	Chiplet Speedup (TRIP/Chiplet)
16	10	0.112	0.46x	0.314	0.16x	0.36
16	14	1.037	0.47x	0.318	1.55x	3.26
16	18	1.351	3.40x	0.681	6.75x	1.98
16	22	6.477	5.36x	5.275	6.58x	1.23
16	26	21.469	7.28x	20.148	7.76x	1.07
16	30	62.867	7.96x	62.014	8.07x	1.01
32	10	0.114	0.45x	0.192	0.27x	0.59
32	14	0.966	0.51x	0.095	5.19x	10.17
32	18	0.742	6.19x	0.262	17.50x	2.83
32	22	5.915	5.87x	3.444	10.08x	1.72
32	26	20.626	7.58x	17.302	9.03x	1.19
32	30	60.867	8.22x	55.655	8.99x	1.09
48	10	0.116	0.44x	0.284	0.18x	0.41
48	14	0.582	0.85x	0.068	7.20x	8.51
48	18	0.777	5.91x	0.205	22.45x	3.80
48	22	4.693	7.40x	4.678	7.42x	1.00
48	26	22.464	6.96x	19.536	8.00x	1.15
48	30	71.483	7.00x	57.287	8.74x	1.25
64	10	0.107	0.48x	0.208	0.25x	0.51
64	14	0.557	0.88x	0.100	4.94x	5.59
64	18	0.796	5.77x	0.209	21.98x	3.81
64	22	5.347	6.49x	1.310	26.51x	4.08
64	26	23.422	6.67x	21.525	7.26x	1.09
64	30	80.406	6.22x	59.950	8.35x	1.34

Table 3. Comparing execution time and parallel speedup for the Fateman benchmark between the TRIP algorithm and our Chiplet algorithm for various values of the exponent e. Parallel speedup for both algorithms is compared to the standard heap-based serial algorithm, not simply executing either algorithm with only 1 thread. The final column shows the speedup ratio between the algorithms: TRIP over Chiplet.

significantly slows down the process.

In cases of extremely large polynomials, however, the Chiplet algorithm outperforms TRIP. We hypothesized that this is due to the differential heap sizes encountered during the multiplication of very large polynomials. In the TRIP algorithm, each thread processes a unique partition of the pp-matrix, leading to eight distinct heaps being managed within a single L3 cache on a CCD. Conversely, in the Chiplet algorithm, each L3 cache accommodates

Threads	e	TRIP Time (s)	Parallel Speedup	Chiplet Time(s)	Parallel Speedup	Chiplet Speedup (TRIP/Chiplet)
64	10	0.714	1.665x	2.480	0.479x	0.29
64	12	1.008	7.464x	7.303	1.030x	0.14
64	14	3.131	27.348x	19.252	4.448x	0.16
64	16	10.809	45.217x	51.224	9.541x	0.21
64	18	82.615	18.389x	124.661	12.187x	0.66
64	20	341.849	19.456x	185.728	35.810x	1.84

Table 4. Comparing execution time and parallel speedup for the very sparse benchmark between the TRIP algorithm and our Chiplet algorithm for various values of the exponent e. Parallel speedup for both algorithms is compared to the standard heap-based serial algorithm, not simply executing either algorithm with only 1 thread. The final column shows the speedup ratio between the algorithms: TRIP over Chiplet.

eight local heaps that together constitute a single partition of the pp-matrix. When handling extremely large polynomials, the TRIP algorithm may overwhelm the L3 cache capacity, resulting in costly cache misses. These cache misses lead to the need for accessing data from the slower, off-chip main memory, in contrast to inter-chiplet communications, which benefit from being entirely on-chip and significantly boost both speed and efficiency.

6.4 Improvements and Future Work

Further advancements in algorithmic performance are crucial, and we identify three primary areas for future exploration:

In the TRIP algorithm, the parameter l determines the number of partitions of the pp-matrix. Suggested by the original authors (9) that l be tuned once per computer; however, we argue that l should be dynamically tuned based on the polynomial size, coefficient sizes, and sparsity. Observations from Table 3 show that the parallel speedup achieved by TRIP across various thread counts and polynomial sizes remains relatively constant. This calls for further experimentation to better understand the computational demands of pp-matrix partitions and how they relate to polynomial characteristics. By understanding these dy-

namics, we can develop heuristics to adjust l optimally during runtime for each invocation of the multiplication algorithm.

Additional work is required to optimize the buffer mechanism in the SDMP algorithm that facilitates the transfer of product terms from the local to the global heap. In scenarios involving very sparse inputs, this memory-intensive step considerably impacts performance. The buffer serves as a key component in addressing the classic "produce-consumer" problem, where data is moved from local to global heaps, dominating the algorithm's runtime. Future efforts will aim to explore various producer-consumer strategies to enhance efficiency.

Last but not least, exploring different groupings of threads in the Chiplet algorithm can potentially enhance its efficiency by finding an optimal cooperation scheme. Currently, each CCD hosts all 8 cores working on a single task. This arrangement might not be optimal as it could lead to excessive competition for resources. Alternative strategies could involve configuring each CCD with smaller groups, such as two independent tasks each utilizing four threads, or four tasks each using two threads. The optimal setup is likely to depend on both the size and sparsity of the input polynomials, much like the partition value l. Further experimentation is necessary to validate these configurations.

Beyond these points, there may be additional opportunities for improvement. Continued experimentation and the development of new heuristics are recommended to further refine these approaches and evaluate their impact on enhancing computational performance.

7 Conclusions

This work presents a comprehensive examination of parallel algorithms for sparse polynomial multiplication, particularly in the context of chiplet architectures. We began by exploring the foundational concepts of chiplets and introduced multi-threading in Julia. The

report then delved into various data structures, techniques, and algorithms suitable for sparse polynomial, such as linked lists, alternating arrays, exponent packing, chaining, and heap-based multiplication, which laid the groundwork for the subsequent introduction of parallel algorithms.

In the section on parallel algorithms, we introduced SDMP, TRIP, and a newly designed algorithm specifically tailored for chiplet-based architectures. We emphasized the strengths of each algorithm, particularly how the chiplet algorithm was created by leveraging the advantages of the previous two. All of these algorithms were meticulously implemented in Julia to facilitate comparisons in the experimentation phase, along with the necessary data structures for efficient exponent packing and sparse polynomial management.

Our experimentation showed that while the chiplet algorithm demonstrated superior performance in certain scenarios, it also revealed areas for improvement. Specifically, further optimizations are required in memory movement, thread organization, and load balancing to fully leverage the advantages of chiplet architectures.

Overall, this report contributes to the ongoing research in high-performance computing by providing insights into the challenges and potential solutions for efficient sparse polynomial multiplication on modern, chiplet-based processors.

References

[1] *Gröbner basis computations*, Wikipedia, Available: `https://en.wikipedia.org/wiki/Gr%C3%B6bner_basis`

[2] F. Bao, R. H. Deng, W. Geiselmann, C. Schnorr, R. Steinwandt, H. Wu, *Cryptanalysis of Two Sparse Polynomial Based Public Key Cryptosystems*, Springer-Verlag, 2001.

[3] A. Brandt. *High Performance Sparse Multivariate Polynomials: Fundamental Data Structures and Algorithms*. The University of Western Ontario, 2024, pp. 9-65.

[4] S. C. Johnson, "*Sparse polynomial arithmetic*," ACM SIGSAM Bulletin, vol. 8, no. 3, pp. 63–71, 1974.

[5] M. Monagan and R. Pearce. Parallel Sparse Polynomial Multiplication Using Heaps. *Proceedings of the ISSAC 2009 - ACM International Symposium on Symbolic and Algebraic Computation*, 2009.

[6] M. Monagan and R. Pearce, "*Sparse polynomial division using a heap*," Journal of Symbolic Computation, vol. 46, no. 7, pp. 807–822, 2011.

[7] M. Gastineau and J. Laskar. TRIP – A Computer Algebra System Dedicated to Celestial Mechanics and Perturbation Series. IMCCE-CNRS UMR8028, Observatoire de Paris, UPMC, 2013.

[8] M. Gastineau and J. Laskar. Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. In V.N. Alexandrov et al. (Eds.), *ICCS 2006 Part II LNCS 3992*, pp. 446–453, Springer-Verlag Berlin Heidelberg, 2006.

[9] M. Gastineau and J. Laskar, "*Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems*," Lecture Notes in Computer Science, vol. 8136, pp. 100–115, 2013, Springer International Publishing.

[10] J. von zur Gathen, and J. Gerhard. *Modern Computer Algebra (3rd ed.)*. Cambridge University Press, pp. 29-43.

[11] R. P. Filho, "What is a Chiplet, and Why Should You Care?", Keysight Blog, Feb. 8, 2024. Available: https://www.keysight.com/blogs/en/tech/sim-des/2024/2/8/what-is-a-chiplet-and-why-should-you-care

[12] K. Mayo, S. Rajasekaran, I. Pasichnyk, and A. Kashyap, *High Performance Computing (HPC) Tuning Guide for AMD EPYC^{TM} 9004 Series Processors*, Publication No. 58002, Revision 1.5, Advanced Micro Devices, Inc., 2024. Available:

[13] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, *Pioneering Chiplet Technology and Design for the AMD EPYC^{TM} and Ryzen^{TM} Processor Families*, 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).

[14] C. Tardi, *What Is Moore's Law and Is It Still True?*, Investopedia, Apr. 2, 2024. Available: https://www.investopedia.com/terms/m/mooreslaw.asp

[15] Y. Han, H. Xu, M. Lu, H. Wang, J. Huang, Y. Wang, Y. Wang, F. Min, Q. Liu, M. Liu, and N. Sun, *The Big Chip: Challenge, Model and Architecture*, Fundamental Research, 2024. Available: https://doi.org/10.1016/j.fmre.2023.10.020

[16] X, Li, *High-Performance FPGA-accelerated Chiplet Modeling*, University of California, Berkeley, Technical Report No. UCB/EECS-2022-145, May 2022. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-145.html

[17] The Julia Language, *The @threads Macro*, https://docs.julialang.org/en/v1/manual/multi-threading/#The-@threads-Macro

[18] R. Fourquet, *XInts.jl: Extended Precision Integers for Julia*, GitHub repository, Available at: https://github.com/rfourquet/XInts.jl/blob/master/src/XInts.jl.

[19] C. Bauer, *ThreadPinning.jl*, GitHub repository, Available at: `https://github.com/carstenbauer/ThreadPinning.jl/blob/main/src/ThreadPinning.jl`.

[20] G. Dalle, *BenchmarkTools.jl*, GitHub repository, 2023, `https://github.com/JuliaCI/BenchmarkTools.jl/blob/master/benchmark/benchmarks.jl`

[21] R. Fateman, Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin*, 37(1):4–15, 2003.

[22] Concyclics and N. Viennot, *Core to Core Latency Measurements*, Available at: `https://github.com/nviennot/core-to-core-latency`.