INDIAN INSTITUTE OF TECHNOLOGY,
GANDHINAGAR

MASTERS THESIS

Accelerating Large Integer Arithmetic with Parallel Addition, Subtraction, and Vedic-Based Multiplication Using AVX512

by

Subhrajit DAS

A thesis submitted in partial fulfilment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

April, 2025

Abstract

Large operands exceeding the standard 32- or 64-bit register sizes of most modern CPUs are common in cryptography, scientific computing and other allied areas. Arithmetic operations like addition, subtraction, multiplication, division, modulo, and factorization are essential in this context. Scientific computing typically employs commercial tools like Maxima or Mathematica, while cryptography relies on optimized libraries such as GNU Multiple Precision Library (GMP) and BigInt. However, despite their fine-tuned optimizations for most general-purpose processors, they underutilize modern CPU parallelization capabilities like SIMD (Single Instruction, Multiple Data), possibly due to limited research on parallelizing core arithmetic tasks.

In this work, we address this gap by focusing on three fundamental operations: addition, subtraction, and multiplication of large integers. We propose a parallel approach for the addition and subtraction of these integers. Additionally, we introduce a hybrid multiplication technique that incorporates the ancient Indian Vedic mathematics, Urdhva-Tiryagbhyam sutra, which is traditionally aimed at faster mental multiplication. Using SIMD constructs, particularly the AVX512 intrinsics on an x86-64 Intel Rocket Lake-based CPU, we achieve an average execution time speed-up of 2.06x for addition and 2.32x for subtraction in 99.99% of cases compared to GMP on operand sizes ranging between 256 bits and 131072 bits. Even in the worst-case scenarios, our performance remains competitive with GMP, resulting in an average speed-up of 1.38x for addition and 1.49x for subtraction. Furthermore, compared to GMP, we outperformed the existing works for large number addition and subtraction on x86-64-based CPUs. However, our implementation of the multiplication algorithm using the Vedic approach currently supports only fixed 256-bit-sized operands, achieving a performance gain of 1.83 times in execution time over GMP. Notably, none of the existing works utilizing the AVX512 intrinsics have been able to outperform GMP for multiplications involving fewer than 1024-bit operands. Future research will focus on completing the hybrid multiplication implementation for variable-sized operands and fine-tuning the implementations for other architectures, such as ARM and RISC-V, using their respective SIMD constructs.

Acknowledgements

With a whole heart, I sincerely want to thank my advisor and my guide, **Abhishek Bichhawat**, who has been very supportive throughout the entire duration of my M.Tech thesis. Apart from research, I learned from him how a good professor, a good advisor, and, foremost, a good human should be. Additionally, I want to sincerely thank my co-advisor, **Yuvraj Patel**, an excellent researcher, who actively engaged with my evolving ideas and offered thoughtful perspectives throughout this thesis. I appreciate his observations and ideas, which I most often overlooked during the course of this work. Both of my advisors have been incredibly supportive and approachable, guiding me through the highs and lows of my research journey.

I also extend my thanks to the Department of Computer Science and Engineering at the Indian Institute of Technology Gandhinagar for providing me access to essential laboratory resources. Additionally, for more than half of the thesis timeline, I ran my initial experiments on CloudLab [Dup+19] CPUs, and I am truly grateful to Yuvraj Sir for providing me access to such a helpful platform. In the latter half, I switched to working with a newer CPU at the Institute, procured by Abhishek Sir. This CPU was extremely helpful as it had the necessary features that I needed.

I am deeply grateful to **my family, friends, and teachers** for their unwavering love, support, and faith. I also cherish the memory of all **my beloved pets** for the unique love and comfort they brought into my life. Without them all, I wouldn't have been able to reach this phase of my life and achieve many of my dreams. Whatever I am now is because of them.

Contents

List of Figures

List of Tables

List of Abbreviations

PML Add	Parallel Multi-Stage Large-Number Addition
PML Sub	Parallel Multi-Stage Large-Number Subraction
VBAP Mul	Vedic-based Approach for Multiplication
SIMD	Single Instruction, Multiple Data
AVX	Advanced Vector Extensions
GMP	GNU Multiple Precision Arithmetic Library

To my love and passion towards
computer science…

Chapter 1

Introduction

Large-number arithmetic, also known as arbitrary-precision or BigNum arithmetic, plays a fundamental role in the current context of cryptography [Hel79; RSA78; Mil86; JMV01; Fre10; Erb+20], scientific computing [Bai05; BB15; BBB12], and various mathematical software packages [Sag; Wol; Max; Inc22]. RSA encryption [RSA78], for example, depends on the difficulty of factoring large primes for its security. Blockchain technologies, such as Bitcoin and Ethereum, use elliptic curve cryptography [Mil86] for securing transactions operating on large numbers. Scientific computing, including fields like astrophysics and particle physics [BB15], utilizes arbitrary-precision arithmetic for calculations. These large-number or arbitrary-precision arithmetic fundamentally require operations like addition, subtraction, multiplication, division and factorization on numbers [Ski08; Knu97] spanning hundreds and thousands of bits. However, working with large operands can significantly slow down the overall performance of applications. For example, the performance of RSA encryption varies exponentially with key size. Decrypting a 2048-bit RSA key takes only a few milliseconds; however, when the key size is doubled to 4096 bits, the time required can increase nearly fivefold on modern CPUs like the Intel i7 and Apple M1 [Cof]; additionally, generating key pairs can take almost ten times longer. And as the size of the operands increases, the computation time grows exponentially.

Current solutions for performing large-number arithmetic include commercial computer algebra systems like Maxima [Max] and Mathematica [Wol], as well as various libraries such as the GNU Multiple Precision Arithmetic Library (GMP) [GNU91] and the GNU Multiple Precision Floating-Point Reliable library (MPFR) [Pro] (based on GMP) in C/C++; gmpy2 [PYP] (based on GMP), mpmath [Joh], and the built-in decimal module in Python; the built-in BigInteger [Docb] in Java; Apfloat [Tom] in C++ and Java; the BigInt [Doca] in JavaScript; num-bigint [Docc] and rug [Docd] (based on GMP and MPFR) in Rust; and various other libraries.

The GMP library [GNU91] is widely used for fast arithmetic computations on large numbers and is highly optimized for most general-purpose processors, leveraging carefully tuned assembly routines. As a result, it serves as the foundation for many other libraries supporting large-number and arbitrary precision arithmetic. However, despite its optimizations, GMP is primarily designed for single-threaded sequential execution and does not take into account modern hardware capabilities for parallel computation, possibly due to limited work in parallelizing the large-number arithmetic operations. We believe that incorporating parallel algorithms with current hardware enhancements can significantly boost the performance.

Earlier works [Coo00; GK12; KM14; GK16] have tried to introduce parallelism in large number multiplication to enhance performance, focusing on metrics like instruction count and CPU cycles. However, the actual performance improvements in terms of execution time on modern hardware remain unclear, primarily due to the absence of upgraded SIMD features in the commercial CPUs available at that time. Though some recent works [ET18; ET20; ET23] have introduced parallelism in this context, testing on some recent real-time actual CPUs and measuring in terms of execution time, they got performance improvements above 1024- to 3072-bit operand sizes against the current optimized single-threaded implementation of GMP. On the other hand, there's very limited work done [Yee19; RSS23] on accelerating large-number addition and subtraction, but hardly getting significant performance improvements as with multiplication. Parallelism in computing can be broadly categorized into two types: Task-level Parallelism and Data-level Parallelism [HP11]. Task-level parallelism can be achieved using multi-threading, where independent tasks or operations are executed concurrently across multiple CPU cores. This approach can be beneficial for accelerating computation, but it also introduces bottlenecks due to complex thread management and synchronization overhead [Lee06; Asa+06; Pau]. Multi-threading is generally useful only beyond a certain data threshold, as thread overhead can degrade performance instead of improving it; thus, a few existing works [Sam22] noticed performance benefits only for abruptly large integers (ranging between 6.4×10^6 to 6.4×10^{10} bits). Data-level parallelism, on the other hand, can be exploited using Single Instruction Multiple Data (SIMD) constructs on modern CPUs, where multiple data elements are processed simultaneously within a single instruction cycle. SIMD-based methods can achieve significant performance improvements, especially with larger vector register sizes (typically 256 bits and above), when used with parallel algorithms. However, suitable parallel algorithms have not been explored extensively for large-number arithmetic.

1.1 Problem Statement

Our focus in this work is to design, implement and optimize algorithms for large-number arithmetic using data-level parallelism, specifically through the use of SIMD constructs with x86-64 AVX512 intrinsics [Int24]. We target three fundamental arithmetic operations on large integers: addition, subtraction and multiplication.

More formally,

Input description: Given two large integers, x and y,

Problem description: Design high-performance AVX512-based parallel methods to compute $x + y$, $x - y$, and $x \times y$, minimizing each operation's computation time.

Further, we would like to compare the performance of these implementations against the widely used GMP library to assess improvements in execution speed, as GMP leverages highly optimized single-threaded assembly routines for addition, subtraction and multiplication and has not currently utilized SIMD for them [SIM].

1.1.1 Optimizing Arithmetic Operations

To address the problem statement, we have examined existing strategies for parallelizing the arithmetic operations on large numbers. We have restructured and refined some of the existing works for better performance for large integers. Additionally, we explored the use of Vedic mathematics, an ancient Indian system of mathematical techniques that offers methods for performing arithmetic operations such as multiplication, division, quotient, and factorization. Vedic mathematical approaches are typically designed for efficient mental calculations; therefore, they are often parallelizable. However, they have not been explored for parallel implementations in large-number arithmetic.

Challenges: The fundamental arithmetic operations, addition and subtraction, are inherently not parallelizable due to carry and borrow dependency; as a result, they are usually implemented sequentially. Thus, it is hard to utilize parallelism for addition and subtraction. However, keeping in mind the applications of large-number arithmetic being predominantly random numbers, such as for cryptography, we have tried to tweak the existing addition and subtraction techniques [Yee19; KS73] for faster parallel execution using SIMD constructs. On the other hand, large-number multiplications are typically implemented using various divide-and-conquer strategies [Kar63; Too63; CA69; SS71] for a smaller number of multiplications to be performed, and as a result, it is harder to compute them in parallel using SIMD constructs. However, existing works tried to leverage SIMD constructs for smaller-sized base-case multiplication utilizing a typical grade-school multiplication technique [Knu97]. In this work, we have utilized the Urdhva Tiryagbhyam sutra [MAH92] for the base-case multiplication instead of the grade-school technique, as it is more loosely coupled in nature.

Contributions: In this work, we have made the following key contributions:

- Proposed an improved addition and subtraction technique for large numbers, enabling parallel computations in most cases.
- Proposed a hybrid divide-and-conquer method incorporating the Urdhva Tiryagbhyam technique from Vedic Mathematics as base-case for large-number multiplication.
- Implemented the proposed addition and subtraction using AVX512 intrinsics, achieving an average execution time speed-up of 2.06× and 2.32× over the GNU Multiple Precision (GMP) library for operand sizes ranging from 256 to 1,31,072 bits on an Intel Rocket Lake-based CPU.
- Implemented the proposed base-case Vedic-based multiplication using AVX512 for fixed 256-bit operands and achieved a 1.83x execution time speed-up compared to GMP on the same Intel CPU.
- Additionally, introduced an approximate version of our addition and subtraction approach, which further enhances performance, achieving an average speed-up of 2.52× for addition and 2.80× for subtraction compared to GMP.

Organization of the Thesis: Chapter 2 provides background information and reviews related literature. Chapter 3 introduces the proposed technique for large number addition, including its implementation details and observations. Following this, Chapter 4 discusses the proposed techniques for large number subtraction, along with the implementation and observations. In Chapter 5, we present approximate variants of the proposed addition and subtraction approaches. The hybrid technique that incorporates Vedic mathematics for large integer multiplication is described in Chapter 6, along with the implementation and performance observations of this Vedic approach. Chapter 7 offers an overall discussion of the three operations, addressing the limitations and challenges encountered. Finally, Chapter 8 concludes the work.

Chapter 2

Background & Related Works

2.1 Large Numbers

Typical programming languages like C, C++, Java, and JavaScript have native support for variables ranging from 8-bit to 64-bit (equivalent to storing two hex-digits to 16 hex-digits), depending on the versions and the architecture. However, many real-world applications, particularly cryptography and scientific computing, may require numbers exceeding these 64-bit limits.

In this context, "large numbers" refer to values beyond the standard 64-bit-and-beyond capabilities of common programming languages required for arbitrary precision. As outlined in Problem Statement 1.1, our focus is on performing arithmetic operations on these large, randomly generated numbers as fast as possible, keeping in mind the application needs.

Use Cases: Classical cryptographic applications like Elliptic Curve Cryptography (ECC) [Mil86] typically operate on 256-bits to 521-bits [IBM], RSA [RSA78] currently utilizes operand sizes ranging between 2048 to 3072 bits [Teab; Hou], Diffie-Hellman Key Exchange [DH22] is currently recommended between 2048-bit to 3072-bit [Bar20]. However, the rise of quantum computing necessitates post-quantum cryptography (PQC) [BL17], which relies on significantly larger key sizes to ensure security against quantum attacks. For example, CRYSTALS-Kyber [Bos+18] (Key Encapsulation Mechanism) requires public keys roughly equivalent to 6144 to 12544 bits [Ala+22], CRYSTALS-Dilithium [Lyu+20] (Digital Signature) uses public keys equivalent to approximately 8704 to 20736 bits, and signatures equivalent to 16384 to 36760 bits [Ala+22], Falcon [Fou+18] (Digital Signature), a lattice-based signature, has public keys of roughly 7176 to 14936 bits and signatures of 5328 to 9832 bits [Ala+22], SPHINCS+ [Ber+15] (Digital Signature) exhibits signature sizes ranging from 62848 to 398848 bits, and public keys of 256 or 512 bits [Ala+22].

These necessitate the need to speed up the fundamental arithmetic operations like addition, subtraction, and multiplication for better performance with the usage of current hardware enhancements, as working with such large numbers can be time-consuming; even small performance gains would be for the greater good.

2.1.1 Large Number Representation

Typical cryptographic arithmetic libraries, such as GMP, adopt a limb-based representation for managing large integers. A *limb* is a fixed-size unit, typically a 32-bit or 64-bit unsigned integer, often matching the machine word size that stores a portion of the number. Multiple limbs collectively represent the full integer, stored as a contiguous array for large numbers or as a set of registers for smaller ones. The base used for limb representation determines the number of bits that can be stored within each limb. Ideally, while distributing the large numbers across 32- or 64-bit limbs, one would use 2^{32} or 2^{64} as its base for minimizing the number of limbs (i.e. if any library is utilizing hex numbers, 32-bit can hold eight hex-digits, and 64-bit can hold 16 hex-digits, effectively making the base for the whole limb as 2^{32} or 2^{64}).

However, certain cryptographic implementations, such as those used for elliptic curve cryptography (ECC), benefit from using bases other than 2^{32} or 2^{64}. This leads to an *unsaturated* representation, also called *reduced-radix*, where some bits within each limb remain unused. For instance, implementations of the P-256 elliptic curve often utilize unsaturated limbs. Conversely, *saturated* representations, or *native-radix*, utilize all available bits within each limb [RSS23]. The work by Erbsen et al. [Erb+20] highlights these unsaturated implementations and, for their implementation for ECC, they utilize *unsaturated* limbs. The work also highlights that certain representations, such as Curve25519 [Ber06], utilize *mixed-radix* bases. Instead of all limbs having a fixed number of bits utilized, also known as *uniform-radix* base, we may have an alternate pattern of bits utilized. For e.g. base $2^{25.5}$ for 32-bit Curve25519 contains 26 bits in the first limb, 25 bits in the second, 26 bits in the third, and so on. This kind of unorthodox base sometimes helps for faster performance. As highlighted by Ersben et al. [Erb+20], modular reduction by $2^{255} - 19$ is fastest when the 255th bit of a large number aligns with the first bit of a limb. In a saturated implementation, limb boundaries align with multiples of the integer size (e.g., a 64-bit saturated representation places boundaries at bits 0, 64, 128, etc.). However, in an unsaturated implementation with an unconventional base like $2^{25.5}$, limb boundaries occur at bits 0, 26, 51, 77, ... and 255. The significant speed boost in modular reduction justifies using a little extra memory to store each large number.

In our work, we have opted for 64-bit limbs to match the machine word size of modern CPUs (x86-64 based). While our addition, subtraction, and multiplication algorithms are designed to be independent of limb size and limb saturation, for addition and subtraction, we have employed saturated and uniform representation, utilizing all the 64-bits with an effective base of 2^{64}. However, for multiplication, we have used a 52-bit unsaturated and uniform representation, with an effective base of 2^{52}, to accelerate our multiplication implementation, which we have discussed in the implementation section of Chapter 6.

2.2 Addition

2.2.1 Carry Propagate Addition

A method for performing addition is through digit-by-digit carry propagation. In large-number additions, the two numbers to be added may typically contain up to 16,384 hex digits if we're handling 65,536-bit numbers. If we process each digit at once, as depicted in eq. 2.1 [Knu97], that will result in 16,384 additions between two operands and 16,383 carry propagations (ignoring the initial carry).

$$S_i = A_i + B_i + C_{i-1}, \tag{2.1}$$

$$C_i = \begin{cases} 1, & \text{if } S_i \geq B_{\max}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

In the above equations, S_i denotes the intermediate sum at position i, computed by adding the corresponding digits A_i and B_i along with the carry from the previous position, C_{i-1}. The carry for the next position, C_i, is set to 1 if the intermediate sum exceeds or equals the base limit $B_{\max}$; otherwise, it remains 0. Here, $B_{\max}$ represents the numerical base limit (e.g., 16 for hex digits).

But, most library implementations [GNU91; Docd; PYP; Docb] pack multiple digits together in a single limb instead of just a bit or digit, be it saturated or unsaturated, i.e., instead of treating each hex digit separately, we can combine multiple hex digits into a larger group, thereby reducing the number of required operations.

Example 2.2.1.1.
Consider the addition of two eight-digit hexadecimal numbers:

$$A = \texttt{7F93BC2D}, \quad B = \texttt{4E87A9F6}.$$

We perform the digit-wise addition from right to left:

$$\begin{array}{cccccccccc} & 7 & F & 9 & 3 & B & C & 2 & D \\ + & 4 & E & 8 & 7 & A & 9 & F & 6 \\ \hline & B & 1D & 11 & A & 15 & 15 & 1 & 13 \end{array}$$

Since we are working in base 16, any sum $S_i \geq 0xF$ generates a carry. Extracting the individual carries:

$$\begin{array}{ccccccccc} & B & \mathbf{D} & \mathbf{1} & A & \mathbf{5} & \mathbf{5} & 1 & \mathbf{3} \\ C: & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Here, each carry only affects its preceding digit. However, if we group the digits into two groups of four digits each, we reduce the carry propagations, as we only need to propagate just two carry instead of seven:

$$\begin{array}{cc|c} & \texttt{7F93} & \texttt{BC2D} \\ + & \texttt{4E87} & \texttt{A9F6} \\ \hline & \texttt{CE1A} & \texttt{16623} \\ \hline C: & 0 & 1 \end{array}$$

The carry from the right group (`16613`) only affects the left group if the sum exceeds four hex digits (i.e., $S_i > 0xFFFF$). This method localizes carry propagation within smaller groups, reducing the total number of carry propagations. The more digits are grouped together, the fewer additions and carry propagations.

Modern computer Instruction Set Architectures (ISA) are typically designed to process data in 32-bit or 64-bit format. For instance, a 32-bit *limb* can represent values up to $0xFFFFFFFF$. This allows us to combine eight consecutive hex digits into one 32-bit limb, and carry propagation within the limb is handled by the hardware adders themselves; we only need to account for carry if the sum overflows. This approach will reduce the number of arithmetic operations required as more digits are processed in each operation, thereby improving computational performance. For example, by grouping the digits into 32-bit limbs, the 16,384 hex-digit number is represented using only 2,048 limbs. So, now, we can perform the addition of the two 16384 hex-digit numbers with just 2048 additions and 2047 carry propagations.

Almost all libraries, including GMP, utilize the carry-propagation addition algorithm mentioned in eq. 2.1, sequentially propagating the carry for each subsequent limb addition. However, this sequential carry propagation is a major issue in terms of parallelizing the additions.

2.2.2 Carry Select Addition

In the carry propagation method, the current addition must wait for the previous addition to finish generating its carry, causing a delay. To address this issue, carry select addition [Bed62] was introduced, primarily in the circuits. Carry-select addition addresses this by pre-computing two possible results for each digit, covering both scenarios for the incoming carry C_{i-1}, which can be either 0 or 1:

- If there's no carry-in ($C_{i-1} = 0$):
  - Sum: $S_i^{(0)} = (A_i + B_i) \mod B_{max}$,
  - Carry-out: $C_{i+1}^{(0)} = (A_i + B_i)/B_{max}$,
- If there's a carry-in ($C_{i-1} = 1$):
  - Sum: $S_i^{(1)} = (A_i + B_i + 1) \mod B_{max}$,
  - Carry-out: $C_{i+1}^{(1)} = (A_i + B_i + 1)/B_{max}$.

Once the actual C_{i-1} from the previous digit is known, the correct sum and carry-out are selected:

$$S_i = S_i^{(C_{i-1})}, \quad C_{i+1} = C_{i+1}^{(C_{i-1})}. \tag{2.3}$$

By pre-computing both outcomes in advance, this approach avoids waiting for the carry to propagate, primarily in the hardware circuits. However, it incurs double the cost due to computing two possibilities for software implementation and computing them in parallel using multi-threading may incur overhead costs. Nonetheless, a long-chained dependency may still persist for software implementation to select the correct sums and carry-outs from the least to the most significant digit, limiting full parallelization.

2.3 Subtraction

Subtraction is similar to addition; instead of propagating carries, we generate borrows. We can tweak the carry propagation equation (Eq. 2.1) for subtraction as depicted in Eq. 2.4.

$$D_i = X_i - Y_i - B_{i-1}, \tag{2.4}$$

$$B_i = \begin{cases} 1, & \text{if } X_i - Y_i - B_{i-1} < 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

This still causes the same long-chained dependency as for carry-propagation addition. Similarly, the borrow-select subtraction tackles the waiting dependency of the previous borrow to be generated by pre-computing two possible results for each digit, covering both scenarios for the incoming borrow B_{i-1}, which can be either 0 or 1:

- If there's no borrow-in ($B_{i-1} = 0$):
  - Difference: $D_i^{(0)} = (X_i - Y_i) \mod B_{max}$,
  - Borrow-out: $B_i^{(0)} = \begin{cases} 1, & \text{if } X_i - Y_i < 0, \\ 0, & \text{otherwise,} \end{cases}$,
- If there's a borrow-in ($B_{i-1} = 1$):
  - Difference: $D_i^{(1)} = (X_i - Y_i - 1) \mod B_{max}$,
  - Borrow-out: $B_i^{(1)} = \begin{cases} 1, & \text{if } X_i - Y_i - 1 < 0, \\ 0, & \text{otherwise,} \end{cases}$.

2.4 Multiplication

Performing multiplication on large numbers is computationally heavier than addition or subtraction, as it incurs many more computations. Popularly, there are two approaches for computing multiplication: the simple school-book technique and divide-and-conquer-based strategies. Other approaches are not very practical for our targeted operand sizes. The following subsections discuss them in detail.

2.4.1 Grade-school

Grade school multiplication [Knu97], sometimes called schoolbook or grammar school multiplication, is the basic and conventional method of multiplying numbers by arranging a rectangle of cross-products. It's exactly like the long multiplication we do with pencil and paper.

Example 2.4.1.1.

$$\begin{array}{r} 23 \\ \times 45 \\ \hline 115 \\ 92 \\ \hline 1035 \end{array}$$

The grade-school multiplication technique for $X \times Y$ is described as follows. Let X have n digits $(X_0, X_1, \ldots, X_{n-1})$ and Y have m digits $(Y_0, Y_1, \ldots, Y_{m-1})$, in base B (e.g., $B = 10$ for decimal). For each position $k = 0, 1, \ldots, n + m - 1$:

1. Compute the sum at position k:

$$S_k = \sum_{\substack{i,j \\ i+j=k}} X_i \cdot Y_j + C_{k-1},$$

   where $C_{-1} = 0$ initially.

2. Compute the carry to the next position:

$$C_k = \lfloor S_k / B \rfloor,$$

3. Compute the digit at position k:

$$P_k = S_k \mod B.$$

The final product is $P = P_0 P_1 \cdots P_{n+m-1}$, where:

- S_k: Sum at position k,
- C_k: Carry to position $k + 1$,
- P_k: Digit at position k.

Although this method has a time complexity of $O(n \cdot m)$, where n and m represent the sizes of the operands, it is still the preferred choice in many libraries and implementations (e.g., [GNU91; Docc; Docb; ET20; ET23]) for smaller large numbers. Faster divide-and-conquer strategies, despite having better time complexity, suffer from performance degradation when dealing with moderately sized numbers. As a result, this technique proves to be more practical in those cases. Notably, GMP version 6.3.0 typically uses the grade-school multiplication method for smaller numbers and adopts divide-and-conquer strategies for larger numbers.

2.4.2 Divide-and-Conquer Multiplication Strategies

Karatsuba

The Karatsuba multiplication algorithm, as described in Knuth's book [Knu97], employs a divide-and-conquer strategy for multiplying two large numbers, X and Y. The algorithm recursively splits each number into two halves of equal length: X is divided into X_H (the high-order half) and X_L (the low-order half), while Y is divided into Y_H (the high-order half) and Y_L (the low-order half). In contrast to the traditional grade-school method, which requires four multiplications at each recursive step (specifically, $X_H \times Y_H$, $X_H \times Y_L$, $X_L \times Y_H$, and $X_L \times Y_L$), the Karatsuba algorithm reduces the total to just three multiplications: $X_H \times Y_H$, $X_L \times Y_L$, and $(X_H + X_L) \times (Y_H + Y_L)$. This optimization effectively saves one multiplication for each recursive step. The algorithm then combines these results with less costly computations, as outlined next.

The Karatsuba approach can be expressed as follows for two n-digit numbers X and Y, where n is assumed to be a power of 2 for simplicity:

- Split $X = X_H \cdot B^{n/2} + X_L$ and $Y = Y_H \cdot B^{n/2} + Y_L$, where B is the base (e.g., $B = 16$ for hexadecimal), and $n/2$ is the split point.
- Compute three recursive products:
  1. $P_1 = X_H \times Y_H$,
  2. $P_2 = X_L \times Y_L$,
  3. $P_3 = (X_H + X_L) \times (Y_H + Y_L)$.
- Combine results: $X \times Y = P_1 \cdot B^n + (P_3 - P_1 - P_2) \cdot B^{n/2} + P_2$.

The base case occurs when $n = 1$, where X and Y are single digits, and the multiplication is direct (i.e., $X \times Y$).

To save up some operations like carry-overflows while performing the addition for $(X_H + X_L) \times (Y_H + Y_L)$, GMP utilizes subtraction instead of addition, like $(X_H - X_L) \times (Y_H - Y_L)$. Their strategy [Kar] is depicted below:

For two N-limb numbers X and Y (where N is even for simplicity):

- Define $k = N/2$, and set $b = 2^{k \cdot \text{bits\_per\_limb}}$, where bits_per_limb is 64 on x86_64 CPUs.
- Split $X = X_1 \cdot b + X_0$ and $Y = Y_1 \cdot b + Y_0$, where X_0, X_1, Y_0, Y_1 are k-limb numbers.
- Compute three recursive products:
  1. $P_1 = X_1 \times Y_1$,
  2. $P_2 = X_0 \times Y_0$,
  3. $P_3 = (X_1 - X_0) \times (Y_1 - Y_0)$ (noting that if $X_1 < X_0$ or $Y_1 < Y_0$, the result may be negative).
- Combine results:
$$X \cdot Y = (b^2 + b) \cdot P_1 - b \cdot P_3 + (b + 1) \cdot P_2 \tag{2.6}$$
- Base case: If $N = \theta$, perform grade-school multiplication.

Karatsuba's time-complexity is $O(n^{1.585}))$, as on each halved-recursion, it performs three multiplications ($O(n^{log_2 3})$. For most modern CPUs, GMP utilizes Karatsuba for equal-length operands with a number of limbs between 20 and 130.

Toom-Cook

Toom-Cook multiplication [Too63; CA69; Knu97] is more of a generalized approach of Karatsuba [Kar63]. Instead of splitting the number into two equal lengths, Toom multiplications splits them into n halves; typically, three and four halves are the most popular ones.

Toom-3 Way For two N-limb numbers X and Y, Toom-3 [Too] splits them into three parts as follows:

- Represent X and Y as:
$$X = [x_2 \mid x_1 \mid x_0],$$
$$Y = [y_2 \mid y_1 \mid y_0],$$
where x_0, x_1, y_0, y_1 are k-limb pieces (roughly $N/3$ limbs each), and x_2, y_2 may be 1 or 2 limbs shorter.
- Express $X = X(b)$ and $Y = Y(b)$ as polynomials:
$$X(t) = x_2t^2 + x_1t + x_0, \quad Y(t) = y_2t^2 + y_1t + y_0$$

The product $X \cdot Y = W(b)$, where $W(t) = X(t) \cdot Y(t) = w_4t^4 + w_3t^3 + w_2t^2 + w_1t + w_0$. The coefficients w_i (each roughly b^2 in size) are determined by evaluating $W(t)$ at five points and then solving a system of equations. The final result is:

$$W(b) = w_4b^4 + w_3b^3 + w_2b^2 + w_1b + w_0$$

The algorithm proceeds as follows:

1. **Split Operands**: Divide X and Y into $[x_2, x_1, x_0]$ and $[y_2, y_1, y_0]$, with $k \approx N/3$ limbs per part (adjusting for x_2, y_2).
2. **Evaluate at Five Points**: Compute $W(t) = X(t) \cdot Y(t)$ at:
   - $t = 0$: $W(0) = x_0 \cdot y_0 = w_0$,
   - $t = 1$: $W(1) = (x_2 + x_1 + x_0) \cdot (y_2 + y_1 + y_0)$,
   - $t = -1$: $W(-1) = (x_2 - x_1 + x_0) \cdot (y_2 - y_1 + y_0)$ (use absolute values and track sign if negative),
   - $t = 2$: $W(2) = (4x_2 + 2x_1 + x_0) \cdot (4y_2 + 2y_1 + y_0)$,
   - $t = \infty$: $W(\infty) = x_2 \cdot y_2 = w_4$ (limit as $t \to \infty$, effectively $X(t) \cdot Y(t)/t^4$).
3. **Form Linear System**: Substitute the points into $W(t)$:
$$\begin{aligned} W(0) &= w_0, \\ W(1) &= w_4 + w_3 + w_2 + w_1 + w_0, \\ W(-1) &= w_4 - w_3 + w_2 - w_1 + w_0, \\ W(2) &= 16w_4 + 8w_3 + 4w_2 + 2w_1 + w_0, \\ W(\infty) &= w_4. \end{aligned}$$
4. **Solve for Coefficients**: Solve the system:
   - $w_0 = W(0)$,
   - $w_4 = W(\infty)$,
   - $w_1 = \frac{W(1) - W(-1) - 2w_4}{2}$,
   - $w_3 = \frac{W(1) + W(-1) - 2w_0 - 2w_4}{2}$,
   - $w_2 = \frac{W(2) - 16w_4 - 8w_3 - 2w_1 - w_0}{4}$.

5. **Combine**: Compute $X \cdot Y = w_4 b^4 + w_3 b^3 + w_2 b^2 + w_1 b + w_0$.

This requires five multiplications of roughly $N/3$-limb numbers, compared to nine in a base-case approach, offering time-complexity $O(n^{log_3 5}) \sim O(N^{1.465})$.

Toom-4 Way As the name suggests, Toom-4 splits up the operands into four similar-length parts:

$$X = [x_3 \mid x_2 \mid x_1 \mid x_0],$$
$$Y = [y_3 \mid y_2 \mid y_1 \mid y_0],$$

And, in a similar fashion to Toom-3, it computes the product, but with a time-complexity of $O(n^{log_4 7})$, as it performs seven multiplications in total for the four splits, effectively performing in $O(n^{1.404})$.

The higher degree Toom-n takes more pieces, resulting in fewer multiplications compared to the grade-school technique, dropping the time complexity from $O(n^2)$ to something like $O(n^{\log_n(2n-1)})$. GMP 6.3.0 uses versions like Toom-3 (3 pieces), Toom-4 (4 pieces), and even Toom-6'n'half or Toom-8'n'half for larger numbers. For reference, Toom-2 (which is equivalent to the Karatsuba method) starts at 19 limbs, Toom-3 at 125 limbs, Toom-4 at 196 limbs, Toom-6'n'half at 276 limbs, and Toom-8'n'half at 369 limbs for AMD Zen 2 architecture.

2.4.3 Fast Fourier Transformation

The idea of using the Fast Fourier Transform (FFT) for multiplication got its start with Schönhage and Strassen in 1971 [SS71], who used FFT to multiply huge integers in $O(n \log n)$ time, even faster than Toom-n. Details of FFT multiplication are not mentioned in this work. But for instance, for FFT multiplication, GMP handles the product as $x \cdot y \mod 2^N + 1$, splitting numbers into 2^k pieces of $N/2^k$-bit chunks. This requires 2^k pointwise multiplications, dropping the time complexity to $O(n^{k/(k-1)})$ for a modular result. Padding with zeros gives the full product, and GMP picks k based on size: $k = 4$ kicks in around 300–1000 limbs, while $k = 8$ takes over at 3000–10000 limbs for full products on modern CPUs.

2.5 Vedic Mathematics

Vedic Maths is an ancient Indian system of mathematical principles and techniques that evolved in India about 5,000 years ago. It was rediscovered by Indian mathematician Jagadguru Shri Bharati Krishna Tirthaji and later documented in his writings [MAH92]. The system is rooted in the ancient scriptures of India, known as the Vedas, and comprises 16 sutras (formulas) and 13 sub-sutras (corollaries). The formulae and their application are known for solving complex arithmetical operations mentally. The thirteen sub-sutras and corollaries are based on the primary sixteen sutras; specifics are not listed here. For our work, we have found the Urdhva-Tiryagbhyam sutra, the third sutra, ideal for multiplication, which we will discuss next. A summary of the sixteen sutras is mentioned in table 2.1.

No.	**Sutra**	**Usage**
1	Ekadhikena Purvena	"by one more than the previous one" for efficient multiplication or division by 2, and many more arithmetic operations.
2	Nikhilam Navatascaramam Dasatah	"all from 9 and the last from 10" for multiplication and division, especially near powers of 10.
3	Urdhva-Tiryagbhyam	"vertically and crosswise" techniques for multiplication, argumental division, and straight division.
4	Paravartya Yojayet	"transpose and apply" for division, simple equations, mergers, multiple simultaneous equations, simultaneous quadratic equations, and partial fractions.
5	Sunyam Samyasamuccaye	"when the samuccaya is the same, it is zero" for solving simple and quadratic equations.
6	(Anurupye) Sunyamanyat	"if one is in ratio, the other is zero" for solving simultaneous simple equations, quadratic equations, and simultaneous quadratic equations.
7	Sankalana-vyavakalanabhyam	"by addition and by subtraction" to solve simultaneous linear equations.
8	Puranapuranabhyam	"by the completion or non-completion" for cubic and biquadratic equations.
9	Calana-kalanabhyam	"sequential motion" (specific applications not fully listed).
10	Yavadunam	"whatever the deficiency" for squaring, cubing, square roots, and cube roots.
11	Vyastisamastih	"specific and general" for solving biquadratic and multiple simultaneous equations.
12	Sesanyankena Caramena	"remainder by the last digit" for quotient-digit computations.
13	Sopantyadvayamantyam	"the ultimate and twice the penultimate" for miscellaneous simple equations.
14	Ekanyunena Purvena	"by one less than the previous one" (specific applications not fully listed).
15	Gunitasamuccayah	"the product of the sum" (specific applications not fully listed).
16	Gunakasamuccayah	To solve differential calculus through "the factor of the sum."

TABLE 2.1: Summary of the Sixteen Vedic Sutras

2.5.1 Urdhva-Tiryagbhyam Sutra for Multiplication

Urdhva-Tiryagbhyam Sutra, also known as Urdhva-Tiryak Sutra, as mentioned in chapter 3 of the book [MAH92], is a generalized multiplication formula that applies to all cases of multiplication and is also useful for the division of two large numbers. The sutra basically simplifies to *"vertically and cross-wise"*. It divides the large integer multiplication into multiple sets of short and simple multiplications.

Example 2.5.1.1.
An instance is shown below for multiplying using the Urdhva-Tiryagbhyam technique:

$$\begin{array}{r} 29 \\ \times 36 \\ \hline \end{array}$$

Step 1: First, we take the most significant digit of both operands, then take two digits from the most significant position, and then take the least significant digit of both operands. Basically, creating prefix and suffix sets from left to right.

$$\begin{array}{ccc} 2 & 29 & 9 \\ 3 & 36 & 6 \end{array}$$

Step 2: Next, we cross-multiply the individual digits within each set that we formed in Step 1 and add the partial products within each such set.

$$(2 \times 3), \quad (2 \times 6) + (9 \times 3), \quad (9 \times 6)$$

Step 3: After Step 2, if the partial sums are greater than 9, we carry over the extra digits from right to left partial sums (i.e. from least significant set to most significant set). This step is also known as *suddhikaran*. In this case,

$$(6), \quad (39), \quad (54)$$

Next, we will propagate 5 from the right-most-hand-set and add it to the middle set, then from the middle set and so on.

$$\begin{array}{ccc} (6), & (39+5), & (4) \\ (6), & (44), & (4) \\ (6+4), & (4), & (4) \\ (10), & (4), & (4) \end{array}$$

Now, as the left-most-hand number is greater than 9 (as we are working with base 10), we'll create one extra digit.

$$(1), \quad (0), \quad (4), \quad (4)$$

Hence, the answer for the multiplication is 1044.
We take another example of three digits:

$$\begin{array}{r} 873 \\ \times 234 \\ \hline \end{array}$$

Applying Step 1, forming the prefix and suffix sets.

$$\begin{array}{ccccc} 8 & 87 & 873 & 73 & 3 \\ 2 & 23 & 234 & 34 & 4 \end{array}$$

Next, we cross-multiply the individual digits within each set as in Step 3.

$$\begin{array}{ccccc} (8 \times 2), & (8 \times 3) + (7 \times 2), & (8 \times 4) + (7 \times 3) + (3 \times 2), & (7 \times 4) + (3 \times 3), & (3 \times 4) \\ (16), & (24+14), & (32+21+6), & (28+9), & (12) \\ (16), & (38), & (59), & (37), & (12) \end{array}$$

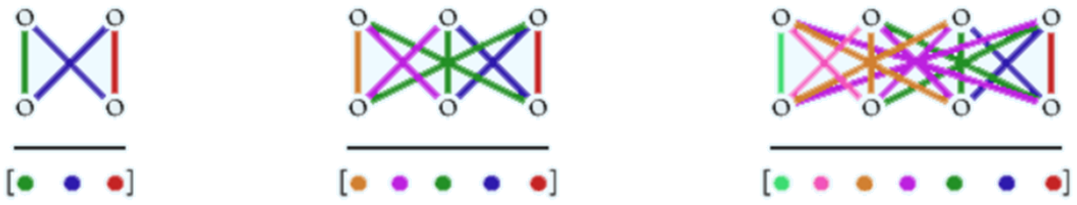

FIGURE 2.1: Urdhva-Tiryagbhyam Multiplications (set-wise) [Upa]

We then carry over extra digits from right to left:

$$\begin{array}{ccccc}
(16), & (38), & (59), & (37+1), & (2) \\
(16), & (38), & (59+3), & (8), & (2) \\
(16), & (38+6), & (2), & (8), & (2) \\
(16+4), & (4), & (2), & (8), & (2)
\end{array}$$

$$(2),\quad (0),\quad (4),\quad (2),\quad (8),\quad (2)$$

Thus, the result is 204282.

Figure 2.1 simplifies the multiplications to be performed; each color represents the multiplications within a set. Most practitioners of Vedic Mathematics often find the steps straightforward enough, with some practice, to perform all calculations for large multiplications mentally in just a few seconds. We have formalized the algorithm for Urdhva-Tiryagbhyam multiplication and have provided a detailed description here, Algorithm 1.

Example 2.5.1.2.
To compare the grade-school and Urdhva-Tiryagbhyam multiplication methods, let's look at multiplying 843 by 384.

Grade-School Multiplication: 843 $\times$ 384

$$\begin{array}{r}
843 \\
\times 384 \\
\hline
3372 \\
6744 \\
2529 \\
\hline
323712
\end{array}$$

Note: in the context of large numbers, multiplications like 843 $\times$ 4 can't be performed directly, as the size of the multiplicand 843 can be more than the native limb size of a machine. Hence, we need to break the multiplicand into small parts and add the partial products to form the resultant product. The operation count below assumes breaking up the multiplicand into individual digits; in order to form the partial multiplication result, we need to propagate the carries.
Operation counts:

- Multiplications: $3 \times 3 = 9$ (each digit of 843 multiplied by each digit of 372)
- Carry additions during multiplication: $2 + 2 + 2 = 6$
- Column-wise additions: $1 + 2 + 2 + 1 = 6$
- Column-wise carry additions: $1 + 1 + 1 + 1 = 4$

Generalized Form:

- Multiplications: $n \times n = n^2$

- Carry additions inside digit multiplications: $n \times (n-1)$
- Column-wise additions: $\sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} i = n \times (n-1)$
- Column-wise carry additions: $2n - 1$

Urdhva-Tiryagbhyam Multiplication: 843 × 384

$$\begin{array}{r} 843 \\ \times 384 \\ \hline \end{array}$$

Digit Sets:

- For 843: $8, 84, 843, 43, 3$
- For 384: $3, 38, 384, 84, 4$

Cross-Products:

$$\begin{aligned} 8 \times 3 &= 24, \\ (8 \times 8) + (4 \times 3) &= 64 + 12 = 76, \\ (8 \times 4) + (4 \times 8) + (3 \times 3) &= 32 + 32 + 9 = 73, \\ (4 \times 4) + (3 \times 8) &= 16 + 24 = 40, \\ 3 \times 4 &= 12. \end{aligned}$$

Partial Products:

$$[24, 76, 73, 40, 12]$$

After suddhikaran (Carry Propagation):

$$\begin{gathered} 2, \text{ carry } 1, \\ 40 + 1 = 41 \Rightarrow 1, \text{ carry } 4, \\ 73 + 4 = 77 \Rightarrow 7, \text{ carry } 7, \\ 76 + 7 = 83 \Rightarrow 3, \text{ carry } 8, \\ 24 + 8 = 32 \Rightarrow 2, \text{ carry } 3, \\ 3 \text{ (new digit)} \end{gathered}$$

Final Result:

$$[3, 1, 3, 7, 1, 2] = 323712$$

For additions within the sets, we have assumed the adder can perform two two-digit operand additions at a time, and some additions may produce results of three digits, which can be accounted for later by checking the wrap-around of the result (e.g. 81+64+49 would produce results of three digits).
Operation Counts:

- Multiplications: $1 + 2 + 3 + 2 + 1 = 9$
- Additions within sets: $0 + 1 + 2 + 1 + 0 = 4$
- Carry-Additions during suddhikaran: 4

Generalized Form:

- Multiplications: $n \times n = n^2$
- Additions within sets: $\sum_{i=0}^{n-1} i + \sum_{i=0}^{n-2} i = (n-1)^2$
- Carry-Additions during suddhikaran: $2n - 1$

Algorithm 1: Urdhva-Tiryagbhyam Multiplication

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base 10.
Output: Product $P = X \times Y$.
// Step 1:  Form prefix sets for both numbers
for $len \leftarrow 1$ to $n$ do
    Form prefix set $P_{X,len}$ from $X$ with length $len$;
    Form prefix set $P_{Y,len}$ from $Y$ with length $len$;    // E.g., for 873:  prefix sets
     are 8, 87, 873
// Step 2:  Form suffix sets for both numbers
for $len \leftarrow n-1$ to $1$ do
    Form suffix set $S_{X,len}$ from $X$ with length $len$;
    Form suffix set $S_{Y,len}$ from $Y$ with length $len$;    // E.g., for 873:  suffix sets
     are 73, 3
// Step 3:  Merge prefix and suffix sets into merged sets
$M_X \leftarrow [P_{X,1}, \ldots, P_{X,n}, S_{X,n-1}, \ldots, S_{X,1}]$, $M_Y \leftarrow [P_{Y,1}, \ldots, P_{Y,n}, S_{Y,n-1}, \ldots, S_{Y,1}]$;
 // E.g., for 873:  $M_X$ = [8, 87, 873, 73, 3]
// Step 4:  Compute cross-products for each pair of merged sets
for $i \leftarrow 1$ to $2n-1$ do
    $R_i \leftarrow 0$;
    $index_X \leftarrow 0$;
    $index_Y \leftarrow$ length of $M_{Y,i} - 1$;
    // Multiply and add from both ends moving inward
    while $index_X <$ length of $M_{X,i}$ and $index_Y \geq 0$ do
        $R_i \leftarrow R_i + (M_{X,i}[index_X] \times M_{Y,i}[index_Y])$;
        $index_X \leftarrow index_X + 1$;
        $index_Y \leftarrow index_Y - 1$;
// Step 5:  Perform suddhikaran (carry-over extra digits)
$carry \leftarrow 0$;
for $i \leftarrow (2n-1)$ to $1$ (right to left) do
    $R_i \leftarrow R_i + carry$;
    if $R_i \geq 10$ then
        $carry \leftarrow \lfloor R_i/10 \rfloor$;
        $R_i \leftarrow R_i \mod 10$;
    else
        $carry \leftarrow 0$;
if $carry > 0$ then
    Prepend $carry$ as an extra digit to the result;
// Step 6:  Combine digits to form the final product
Combine digits in $R$ to form the product $P$;
return $P$;
```

Operation Count Summary for n-Digit Numbers: The table 2.2 summarizes the number of operations required for multiplying two n-digit numbers. Effectively, the time complexity for Urdhva-Tiryagbhyam is $O(n^2)$ as it performs n^2 multiplications. But, if we disregard the additional carries during partial product addition of the grade-school, the Urdhva-Tiryagbhyam method minimizes operations, requiring (n-1) fewer additions than grade-school multiplication. Both algorithms are slower in terms of time complexity as compared to the divide-and-conquer methods that we have seen, but for not-so-large numbers, implementation-wise, they are typically faster due to the overhead cost of recursion.

As the Urdhva-Tiryagbhyam method natively breaks large numbers into smaller sets of individual digit multiplications (loosely coupled), it's much more effective

Operation	Grade-School	Urdhva-Tiryagbhyam
MUL	n^2	n^2
ADD	$n(n-1)$	$(n-1)^2$
Carry-ADD	$2n-1$	$2n-1$
Extra-Carry-ADD	$n \times (n-1)$	0

TABLE 2.2: Operation counts for grade-school and Urdhva-Tiryagbhyam multiplication of two n-digit numbers.

for large numbers and was popular among practitioners from ancient times for the mental multiplication of large numbers.

Usage of Viniculum Since Vedic Mathematics is intended for faster mental mathematics, we typically use the Viniculum technique for multiplying digits greater than 5. The Viniculum is similar to finding the 9's complement of a number. After applying the Vinculum to the operands, we multiply the digits as usual, but with the advantage that all digits are reduced to less than 6, making them easier to compute mentally. Afterwards, we convert the intermediate result back to its original form by applying the vinculum again, effectively normalizing it. Although this method can be faster for mental computation, depending on the numbers involved, it may not offer advantages when implemented in a computer program, as hardware circuits remain unaffected, as all the values stay within a given base. Consequently, it might not be relevant to our work.

2.6 Related Works on Parallel Arithmetic for Large Numbers

2.6.1 Addition and Subtraction

The task of parallelizing the addition operation remains challenging due to the direct dependency of carry in subsequent digit additions when adding two numbers. As a typical carry-propagation addition algorithm computes carry C_i for i^{th} place after the successful calculation of S_{i-1}, which again depends upon C_{i-1} (as depicted in Eq. 2.1), thereby creating a carry propagation dependency. This dependency may create performance bottlenecks, as each digit addition must wait for the previous carry to be generated before proceeding, which is not at all ideal for parallelizing the add operations. The alternative approach of carry-select additions is not widely used due to the heavy overhead in computing both possibilities. Consequently, many fast arithmetic libraries ([GNU91]) do not leverage parallel computation in their addition implementations for large numbers. However, we may skip the involvement of carries while adding two operands and later account for the carry. But, in certain scenarios, we may have to sequentially add the carries from the least significant digit to the most significant digit, as each digit carry adjustment may generate further carries, creating a chained sequential dependency. For an N-digit number, even if we group them with K-digits, we still need to sequentially propagate nearly $\lceil N/K \rceil$ carries.

Some progress has been made to parallelize the operations compared to Eq. 2.1. As shown by Alexander Yee's work for y-cruncher [Yee19], we can utilize the Kogge-Stone adder technique [KS73], which is based on the carry propagation method, to handle carries at a later stage. The idea is to perform the initial addition without considering carries, then detect both the carries and instances where the intermediate sum reaches the maximum value (*max-sum*) for the base (denoted $B_{\max}$). Based on this detection, we adjust the intermediate sums to account for carries where necessary (*intermediate sum adjustment*), thus avoiding the need for repeated carry propagation while doing the addition. However, some sequential operations still need to be performed on the carries and the max-sum detection values. As illustrated in Algorithm 2, after computing the intermediate sums $S_i = X_i + Y_i$ and detecting carry (C_i) and max-sum (M_i) conditions in parallel, we must sequentially process these masks from the least significant blocks to the most significant blocks. This sequential step in Phase 3, operating over m blocks, determines which S_i values require a +1 adjustment, introducing a dependency on the propagating carry that in some way stalls the desired full parallelism. Notably, the implementation mentioned by Yee [Yee19] for the algorithm 2 attempts to eliminate the sequential carry adjustment in Phase 3 of Algorithm 2. However, it does so at the expense of additional operations on the mask registers, such as XOR and masked subtraction, which may be inefficient to perform on a very wide microarchitecture, which will, in turn, increase the overall computation time. For such overhead, they claim not to see any performance improvement compared to traditional implementation with add-carry instructions.

Algorithm 2: Kogge-Stone Parallel Addition [Yee19; KS73]

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$ digits each,
         where $X_i, Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least significant blocks.
Output: Sum $S = [S_{m-1}, \dots, S_0]$.
// Phase 1: Addition (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    $S_i \leftarrow X_i + Y_i$;                                  // Add blocks, ignore overflow
// Phase 2: Detection (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    if $S_i \geq B^K$ then
        $C_i \leftarrow 1$;                                      // Carry
        $S_i \leftarrow S_i - B^K$
    else if $S_i = B^K - 1$ then
        $M_i \leftarrow 1$;                                      // Max-Sum
$C'_{i+1} = C_i, \quad \forall 0 \leq i < m-1, C'_0 = 0$;       // Left-shift Carries by 1 block
// Phase 3: Adjustment (Sequential)
$carry \leftarrow 0$;
for $i \leftarrow 0$ to $m-1$ do
    if $C'_i = 1$ or $(carry = 1 \land M_i = 1)$ then
        $S_i \leftarrow S_i + 1$;                                // Fix incorrect blocks
        if $S_i = B^K$ then
            $S_i \leftarrow 0$;
            $carry \leftarrow 1$;                                // Propagate carry
        else
            $carry \leftarrow 0$;
return $S$;
```

In contrast, the work [RSS23] presents a new addition method in Algorithm 'ProposedAdd', depicted in algorithm 3, utilizing ideas from carry-select addition [Bed62]. To tackle the long carry dependency chain, it breaks down the addition of large integers A and B into a smaller, parallel addition of 8-bit values t_i and p_i. Each digit's addition falls into one of three cases: no-carry (N, where $c_{i+1} = 0$), propagate (P, where $c_{i+1} = c_i$), or generate (G, where $c_{i+1} = 1$), based only on the operands. The t_i and p_i values are set up to match the carry behaviour of $A_i + B_i$, letting carries be figured out fast across all digits. It starts with intermediate sums $S_i = A_i + B_i$, then uses the $t + p$ addition to pull out carries, which adjust S_i to get the final sum. This skips repeated carry propagation and cuts down on data shuffling by keeping carries handy. Still, figuring out t_i and p_i and doing a cross-digit step adds extra work, making it more computationally heavy. They observed a 30% increase in speed with the latest CTIDH implementation, an 11% increase from the most recent CSIDH implementation on AVX-512 processors, and a 7% boost from Microsoft's standard PQCrypto-SIDH for SIKEp503 running on the A64FX architecture.

Everything we've talked about regarding the parallelization of large-number addition can also be applied to subtraction. Instead of propagating carry, we will generate borrows and adjust -1 instead of $+1$. Similarly, just as we checked for partial results hitting their maximum base value in addition, we will identify whether the partial results reach zero for subtraction. The addition algorithm seen in algorithm 2 can be adapted to the subtraction algorithm as depicted in algorithm 4. Similarly, the addition algorithm by [RSS23] can also be transformed into subtraction.

Algorithm 3: ProposedAdd: Proposed SIMD addition [RSS23]

```
Input  : Two $n$-digit numbers $A$ and $B$ in base $B$, where $A = [A_{n-1}, \dots, A_0]$,
         $B = [B_{n-1}, \dots, B_0]$, and $A_i, B_i < B^K$, with $A_0, B_0$ as the least significant
         digits.
Output: Sum $S = \sum_{i=0}^{n-1} 2^{i \cdot K} S_i$.
// Step 1: Compute intermediate sums (Parallel)
for $i \leftarrow 0$ to $n-1$ in parallel do
    $S_i \leftarrow A_i + B_i$;                          // Add digits, ignore carry for now
// Step 2: Set up $t_i$ and $p_i$ based on carry cases (Parallel)
for $i \leftarrow 0$ to $n-1$ in parallel do
    if $A_i + B_i < B^K$ then
        $t_i \leftarrow$ value for case $N$;                        // No-carry case
        $p_i \leftarrow$ constant for $N$
    else if $A_i + B_i = B^K - 1$ then
        $t_i \leftarrow$ value for case $P$;                       // Propagate case
        $p_i \leftarrow$ constant for $P$
    else if $A_i + B_i \geq B^K$ then
        $t_i \leftarrow$ value for case $G$;                        // Generate case
        $p_i \leftarrow$ constant for $G$
// Step 3: Compute smaller addition to extract carries
$s \leftarrow t + p$;                                // Add $t_i$ and $p_i$ across all digits
// Step 4: Pull out carries (Includes cross-digit operation)
for $i \leftarrow 0$ to $n-1$ do
    $c_i \leftarrow$ extract carry from $s_i, t_i, p_i$;                // e.g., $s_i - t_i - p_i$
// Step 5: Adjust intermediate sums with carries (Parallel)
for $i \leftarrow 0$ to $n-1$ in parallel do
    $S_i \leftarrow (S_i + c_i) \mod B^K$;                       // Final sum per digit
return $S = \sum_{i=0}^{n-1} 2^{i \cdot K} S_i$;
```

Again, Phase 3 of algorithm 4 poses the same issue as earlier, i.e. sequential operating order from the least significant block to the most significant block.

Algorithm 4: Kogge-Stone Parallel Subtraction [Yee19; KS73]

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$ digits each,
         where $X_i, Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least significant blocks.
         Also, $X \geq Y$.
Output: Sub $S = [S_{m-1}, \dots, S_0]$.
// Phase 1: Subtraction (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    $S_i \leftarrow X_i - Y_i$;                          // Subtract blocks, ignore borrow
// Phase 2: Detection (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    if $S_i < 0$ then
        $B_i \leftarrow 1$;                                  // Borrow
        $S_i \leftarrow S_i + B^K$
    else if $S_i = 0$ then
        $M_i \leftarrow 1$;                                  // Min-Sub
$B'_{i+1} = B_i, \quad \forall 0 \leq i < m-1, B'_0 = 0$;   // Left-shift Borrows by 1 block
// Phase 3: Adjustment (Sequential)
$borrow \leftarrow 0$;
for $i \leftarrow 0$ to $m-1$ do
    if $B'_i = 1$ or $(borrow = 1 \land M_i = 1)$ then
        $S_i \leftarrow S_i - 1$;                            // Fix incorrect blocks
        if $S_i = -1$ then
            $S_i \leftarrow B^K - 1$;
            $borrow \leftarrow 1$;                           // Propagate borrow
        else
            $borrow \leftarrow 0$;
return $S$;
```

2.6.2 Multiplication

In 2000, the first attempted work [Coo00] using SIMD for accelerating large-number multiplication was done using SSE2 (128-bit vectorization) using a reduced-radix representation method, achieving a speedup of approximately 10.7x compared to a naive scalar implementation on an Intel Pentium 4 processor.

In 2012, a work [GK12] implemented a multiplication program using AVX2 and reduced-radix representation for modular arithmetic, which they patched into OpenSSL. Their evaluation showed a reduction of 50% in both the number of instructions and the number of cycles compared to the original OpenSSL. Furthermore, in 2016 they also investigated [GK16] the potential of AVX-512 instructions, including AVX-512IFMA and combinations of AVX-512F, AVX-512BW, and AVX-512VL. Their instruction count analysis for fixed operand sizes (1,024 to 4,096 bits) indicated that implementations using AVX-512F, VL, and BW had approximately half the number of instructions compared to the GMP, while AVX-512IFMA reduced the instruction count by approximately ranging between one-fourth to one-eighth relative to GMP. In 2014, this work [KM14] also implemented large integer multiplication using AVX-512 instructions with a 229-radix representation. Their evaluation was on an Intel Software Developer Emulator (SDE) simulating a Knights Landing processor, which showed a 1.16x reduction in the number of instructions compared to GMP for 2,048-bit multiplication. These prior works using the AVX2 or AVX512 family primarily evaluated their results based on instruction count and cycles due to the non-commercial availability of AVX in production. However, in 2018, the work [ET18] utilizes AVX-512 intrinsics on an actual Intel Xeon Phi (Knights Landing) processor to accelerate large integer multiplication. For their implementation, they chose a *reduced-radix* representation (using 28 or 29 bits within 32-bit words) to manage carry propagation during the addition of partial products. To handle variable-length operands, they implemented a fixed-length base-case multiplication kernel(5) using grade-school multiplication(2.4.1) and repeatedly called it across the operands. Furthermore, they optimized the kernel by distributing sub-product arrays to minimize pipeline stalls caused by data dependencies, scenarios that might not be represented well through just instruction and cycle counts, as with prior works. The performance of their implementation was compared against the GMP, achieving execution times approximately 2.5x faster than GMP for operands of 1,024 bits to up to 7,168 bits. However, they have not provided their code of implementation, and we were unable to verify and compare their results on the latest CPUs.

Algorithm 5: Multiplication Kernel using AVX-512 [ET18]

```
Input  : $X = \sum_{k=0}^{m-1} x_k B^k$, a multi-precision integer with $m$ words in radix $B = 2^N$
         ($N = 28$ or 29). $Y = \sum_{k=0}^{n-1} y_k B^k$, a multi-precision integer with $n$ words in
         radix $B = 2^N$.
Output: $P = X \times Y = \sum_{k=0}^{m+n-1} p_k B^k$, the product of $A$ and $B$ in radix $B = 2^N$.
$VLEN \leftarrow 8$ ;                                // Vector Length for AVX-512
$i \leftarrow 0$;
while $i < n$ do
    $j \leftarrow 0$;
    while $j < m$ do
        $x \leftarrow [x_j, x_{j+1}, x_{j+2}, \dots, x_{j+(VLEN-1)}]$;
        for $k \leftarrow 0$ to $VLEN - 1$ do
            $p \leftarrow [p_{j+k}, p_{j+k+1}, p_{j+k+2}, \dots, p_{j+k+(VLEN-1)}]$;
            $p \leftarrow p + x \times y_{i+k}$ ;  // Using AVX-512 intrinsics (_mm512_mul_epu32 and
                                                   _mm512_add_epi64)
        $j \leftarrow j + VLEN$;
    $i \leftarrow i + VLEN$;
Convert $P$ back to $2^N$-radix representation;
return $P$;
```

In 2020, the work [ET20], by the same authors of [ET18], implemented large integer multiplication using Intel AVX-512IFMA intrinsics on a processor with Cannon Lake microarchitecture. They adopted a hybrid approach similar to GMP, vectorizing the base-case grade-school multiplication with AVX-512IFMA. The AVX-512F multiplication instructions discard the higher 64-bit of the resultant, requiring a 32-bit or smaller data representation. In contrast, AVX512-IFMA multiplication instructions offer separate access to both the lower and upper 64-bit results, though they accept only 52-bit operands. Consequently, the authors used a reduced-radix representation with 52-bit words. They reported a speedup of up to 2.97x over GMP. In 2023, another study [ET23], by the same authors, employed a similar hybrid approach as [ET20], but this time using ARM-SVE for vectorization. Due to the absence of AVX512-IFMA-like 52-bit multiplication instructions for obtaining both higher and lower 64-bit results, the authors modified the base-case grade-school multiplication technique into what they called shifted grade-school multiplication 6. They achieved performance gains of up to 36% compared to GMP when using the trad mode of the fcc compiler and up to 31% when using the clang mode of the fcc compiler for operand sizes ranging from 3,072 bits to 14,336 bits. They also noted that the SVE implementation became faster than GMP for operands larger than 3,072 bits.

However, for both works, we do not have access to their code and cannot verify and compare their results on the latest CPUs.

Algorithm 6: Shifted Basecase Multiplication Kernel [ET23]

```
Input  : $X = \sum_{i=0}^{m-1} x_i B_r^i$, a multi-precision integer with $m$ words in radix $B_r = 2^p$.
         $Y = \sum_{j=0}^{n-1} y_j B_r^j$, a multi-precision integer with $n$ words in radix $B_r = 2^p$.
         $B = 2^{\max}$, $B_s = 2^{(\max - p)/2}$, where $p < \max$ and $\max - p$ is even.
Output: $P = X \cdot Y = \sum_{k=0}^{m+n-1} p_k B_r^k$, the product of $X$ and $Y$ in radix $B_r = 2^p$.
$X' \leftarrow \sum_{i=0}^{m-1} x'_i B_r^i \leftarrow \sum_{i=0}^{m-1} x_i B_s B_r^i$ ;     // Shifted representation of $X$
$Y' \leftarrow \sum_{j=0}^{n-1} y'_j B_r^j \leftarrow \sum_{j=0}^{n-1} y_j B_s B_r^j$ ;     // Shifted representation of $Y$
$P \leftarrow 0$ ;     // Initialize product
for $j \leftarrow 0$ to $n-1$ do
    for $i \leftarrow 0$ to $m-1$ do
        $p_L \leftarrow (x_i \cdot y_j) \mod B_r$ ;     // Lower part of the product
        $p_H \leftarrow (x'_i \cdot y'_j) / B$ ;     // Higher part of the shifted product
        $P \leftarrow P + p_L B_r^{i+j} + p_H B_r^{i+j+1}$ ;     // Accumulate into $P$
return $P$;
```

The work [Zha] compares grade-school and Karatsuba algorithms for multiplying large integers, looking at execution time, complexity, and resource use. It shows Karatsuba's divide-and-conquer method, which needs fewer multiplications, beats traditional ones in speed and scalability for big numbers. Theoretical and experimental results highlight how picking the right algorithm boosts performance.

2.7 Vectorization (SIMD)

As per Flynn's taxonomy [Fly66], high-speed computing systems can be classified into four categories, namely, (i) Single Instruction Stream-Single Data Stream (SISD), (ii) Single Instruction Stream-Multiple Data Stream (SIMD), (iii) Multiple Instruction Stream-Single Data Stream (MISD) and (iv) Multiple Instruction Stream-Multiple Data Stream (MIMD).

Most of the sequential work that we perform on current CPUs is categorized into either SISD (when we typically work on a single core without any threading) or MIMD (working with multiple cores and different data streams on different cores). However, lately, most modern CPUs have started supporting SIMD capabilities as an extension feature.

In SISD systems, a single instruction is executed on a single data item at a time, as is the case with uniprocessor systems or a single core of a modern CPU without threading or vectorization. For instance, a basic arithmetic operation (e.g., adding two integers) on an Intel x86-64 CPU, when executed without parallelism, shows SISD behaviour. While less common in isolation today due to advanced features, SISD remains the foundational execution mode for non-parallel tasks. MISD involves multiple instruction streams operating on a single data stream, a rare configuration in practice. Theoretically, it could apply to systems where different algorithms process the same input for fault tolerance or redundancy, such as in some specialized pipeline architectures. However, modern general-purpose CPUs like Intel x86_64 do not natively implement MISD for commercial processors. MIMD systems execute multiple independent instruction streams on multiple data streams, offering the highest degree of parallelism. Modern multi-core Intel x86-64 CPUs, such as the Intel Core i9-13900K with 24 cores (8 performance + 16 efficiency), serve

TABLE 2.3: Evolution of SIMD Instruction Sets Across Major CPU Vendors

SIMD Version	Year	Processor	Key Features	Vendor	Source
MMX	1997	Intel Pentium P5	64-bit vector, multimedia	Intel	[Inc97]
SSE	1999	Intel Pentium III	128-bit, floating-point, graphics	Intel	[Cha99]
SSE2	2000	Intel Pentium 4	Double-precision floating-point	Intel	[SSEa]
SSE3	2004	Intel Pentium 4 Prescott	Horizontal register operations	Intel	[SSEb]
SSE4.1	2007	Intel Penryn	47 new vector instructions	Intel	[SSEc]
SSE4.2	2008	Intel Nehalem	7 new string/text instructions	Intel	[SSEc]
AVX	2011	Intel Sandy Bridge	256-bit vector processing	Intel	[AVXa]
AVX-512	2016	Intel Xeon Phi x200 (Knights Landing)	512-bit vectors, HPC/AI acceleration	Intel	[AVXb]
3DNow!	1998	AMD K6-2	64-bit SIMD for graphics	AMD	[3dN]
AVX (AMD)	2011	AMD Bulldozer	256-bit vector, software compatibility	AMD	[AVXa]
ARM SIMD	2002	ARMv6 (ARM1136)	32bit wide SIMD	ARM	[ARM]
NEON	2005	ARMv7 Cortex-A	128-bit SIMD, mobile multimedia	ARM	[ARM]
SVE	2016	ARMv8-A (A64FX)	Scalable vectors, up to 2048 bits	ARM	[Ste16]

as MIMD by running distinct threads or processes across cores. Hyper-Threading further enhances this by allowing each core to handle multiple threads.

On the other hand, SIMD systems execute a single instruction across multiple data elements simultaneously, which is ideal for data-parallel tasks. Modern CPUs, including Intel x86_64, have increasingly adopted SIMD through vector instruction extensions.

The evolution of SIMD in Intel processors began with the introduction of MMX (MultiMedia Extensions) in 1997 on the X86 Pentium P5 [REU97], supporting 64-bit vector operations. On AMD processors, SIMD started with 3DNow! in 1998, and on ARM processors, it started with ARM-v6 from ARM1136 in 2002. Table 2.3 lists out the beginning of various SIMD versions across different vendors.

2.7.1 AVX-512

For Intel and AMD's implementation of AVX-512, Figure 2.2 shows the register configuration within SIMD registers. These registers support vector widths of 128-bit (SSE-style), 256-bit (AVX2-style), or 512-bit (AVX-512). A 512-bit AVX-512 register is constructed out of four 128-bit registers, referred to as *lanes*. Each lane can be further subdivided into individual elements of 64, 32, 16, or 8 bits. Processors supporting AVX-512 provide access to 128-bit (*xmm*) registers, 256-bit (*ymm*) registers, and 512-bit (*zmm*) registers, with SIMD operations applied to the 64-bit, 32-bit, 16-bit, or 8-bit elements within these registers. Load or store operations on a 512-bit (*zmm*) vector register typically process data contiguously across all four lanes, from lane-0 to lane-3, in little-endian byte order. In our work, we have explored AVX-512 on X86-64 architecture, leveraging its 512-bit vectorization to process eight times more data per instruction than traditional 64-bit scalar registers. AVX-512 represents a set of 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions within the x86 instruction set architecture (ISA). Initially introduced by Intel in July 2013, these extensions were first integrated into the 2016 Intel Xeon Phi x200 (Knights Landing) and subsequently adopted in various AMD and Intel CPUs.

AVX-512 instruction set typically refers to a family of 512-bit vector extensions, which can be implemented independently. The family of AVX-512 instruction set [Int24] consists of the following set of independent instruction sets: **AVX-512F** serves as the foundational set of AVX-512 instructions, enabling basic 512-bit vector operations like floating-point arithmetic and data movement. **AVX-512BW** adds support

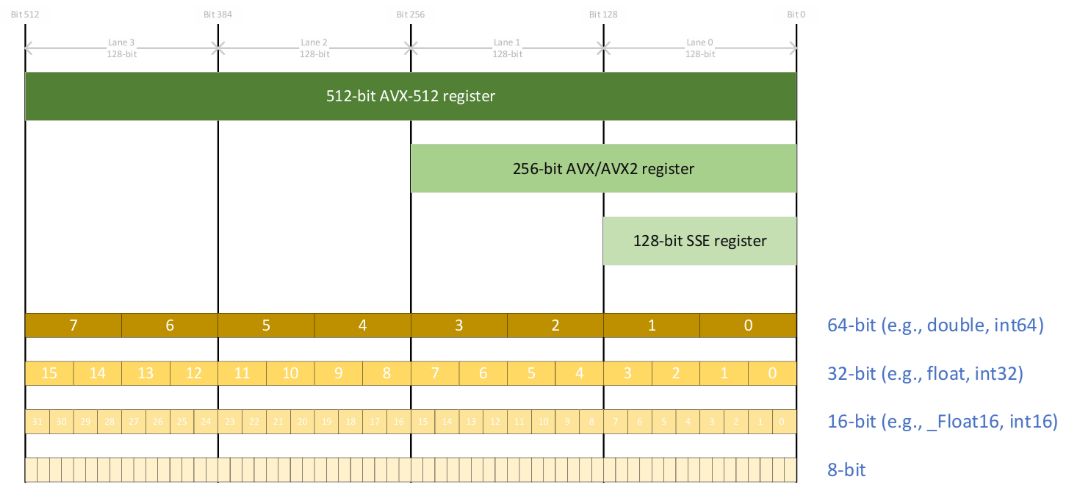

FIGURE 2.2: Layout of Various Sizes of SIMD Register and How Each Can Be Broken Down into Smaller Subgroups of Elements [Tow22]

for byte and word integer operations. **AVX-512CD** includes conflict detection capabilities. **AVX-512DQ** extends integer operations to 32-bit and 64-bit data. **AVX-512IFMA52** offers integer fused multiply-add with 52-bit precision. **AVX-512VL** supports shorter vector lengths (128-bit and 256-bit) with AVX-512 instructions. **AVX-512VPOPCNTDQ** enables population count operations on 32-bit and 64-bit integers. **AVX-512_BF16** implements bfloat16 operations. **AVX-512_BITALG** provides advanced bit manipulation features. **AVX-512_VBMI** facilitates byte-level vector manipulations. **AVX-512_VBMI2** builds on VBMI with additional data compression and expansion instructions. **AVX-512_VNNI** accelerates integer-based neural network operations. **AVX-512_VP2INTERSECT** offers vector intersection instructions. Finally, **AVX-512_FP16** adds 16-bit floating-point support. It is important to note that vendors offer different sets of AVX-512 support based on the micro-architecture; therefore, one should check the list of CPU flags for it.

We may utilize these AVX-512 instructions in two ways: writing direct assembly or using intrinsic function calls in C/C++ or even on Rust (core::arch module, nightly-only). Using intrinsic function calls might be easier for a programmer, as it provides the same effect as writing low-level code but offers better readability and availability of source code. Prior works on large-number addition, subtraction, and multiplication have utilized AVX-512F and AVX-512IFMA52 instruction sets, where most of them [ET18; ET20; Did+24] used the intrinsics, and few of them [KM14] investigated the AVX-512 instruction set using assembly for large-number arithmetic.

2.7.2 AVX-10

In 2023, Intel introduced AVX10 [Int], the successor to AVX-512, which unifies its capabilities into a single, versioned instruction set. Unlike AVX-512, which was limited to P-cores in high-end processors, AVX-10 extends its functionality to both P-cores and E-cores in all future Intel processors. AVX10.1, an early version that includes the instructions from AVX-512 at 128, 256, and 512-bit vector lengths, debuted with Granite Rapids in Q3 2024 for software pre-enabling. Applications written for AVX10.1 will be compatible with any future Intel processor (P-core or E-core) that supports AVX10.1 or higher at the same vector lengths. AVX10.2, expected to launch with Diamond Rapids in 2025–2026, will introduce new AI and data instructions, and it will require 512-bit support across all cores. While all AVX-512 instructions will remain forwards-compatible with AVX10/512, those using AVX10/256 may need to be recompiled for E-cores. Intel's ongoing specification updates are designed to promote broader adoption and simplify the development process.

2.8 Parallel Algorithm Design

By means of parallel algorithms, we fundamentally want to execute a larger task by dividing it into n small and independent subtasks and processing them independently across n different processing units at the same time, achieving a speed-up of factor n as compared to executing the sub-tasks sequentially n times in a single processing unit. However, not all tasks can be divided into independent sub-tasks. Designing a parallel algorithm may involve [Gra08]: 1) Identifying portions of work that can be computed independently. 2) Map the independent sub-tasks onto multiple processing units. 3) Reduce the work output by the processing units at various stages. Overall, we need to decompose the task into multiple independent sub-tasks. Moreover, a finer decomposition will lead to greater concurrency.

In the context of large-number addition and subtraction, we need to identify the perfect decomposition that can achieve maximum parallelism. Existing works tried to decouple the carry or borrow dependency from the main computation of add/subtract as much as possible without affecting the correctness and later accounted for them through a carry-select idea [RSS23] and creating a smaller sequential task or even running a check later-on for adjustment [Yee19], trying to increase the parallelism.

But for large-number multiplication, things are more complicated than just carry- or borrow-propagation with addition or subtraction. Fundamentally, multiplication is a more computationally heavy task in terms of CPU cycles; the first need is to reduce the number of multiplications. Current research on reducing the number of multiplication resorts to divide-and-conquer approaches like Karatsuba or Toom-Cook, where the number of multiplications is reduced in each recursion through performing extra addition or subtraction. However, the divide-and-conquer technique hampers the parallelism with the current hardware resources to a large extent, i.e. until the operand sizes become suitable enough to be processed by the parallel computing units (i.e. vector registers) [ET20; ET23], we further split the operands on the go. When the operand sizes are suitable enough, we apply the base-case multiplication algorithm using SIMD constructs. Even base-case multiplication algorithms like grade-school multiplication are not fully parallelizable due to dependency on adding the partial products, propagating carry, etc. Adding the partial product is challenging due to the fact that current SIMD capabilities limit horizontal additions across contiguous lanes and are typically much slower compared to vertical additions. Unlike addition or subtraction, where the decoupled carry/borrow phase of the algorithm is required, we can simply load the operands into the vector registers, limited only by the register widths.

Chapter 3

Parallelizing the Addition of Large Integers

In this chapter, we first present our approach for parallelizing addition on large random integers. After that, we discuss the implementation, addressing the challenges and the less successful implementations, followed by the evaluation of the final implementation.

3.1 Parallel Multi-phase Addition Algorithm

In this work, we have modified the Kogge-Stone addition algorithm (see Algorithm 2) to facilitate parallel computations in most scenarios. Instead of adopting a pessimistic approach, we take an optimistic one. The implementation by Yee ([Yee19]) of the Kogge-Stone addition algorithm seeks to identify all possible carries while checking for carries and maxed-out blocks. It then performs some operations on the carry and max detect masks and adjusts the intermediate sums. Although this implementation has successfully omitted the sequential operations to a large extent, using the max and carry block technique, they claim not to get much performance gain in comparison to typical add-carry instructions due to inefficient support for performing operations on the masked registers on x86-64 CPUs.

However, with our optimistic approach, we assert that for 64-bit limbs with a saturated base, in 99.99999999999999994578989137572% of cases, the generation of carries in the current limbs and their preceding limbs getting maxed out after the carry-independent addition will not occur for randomly generated large numbers (later described in Chapter 5). Accordingly, we have modified the algorithm to reflect this insight.

We will approach this by implementing a four-phase approach for adding large numbers:

- **Phase 1**: The limbs from both operands are added in parallel, ignoring any potential carry.
- **Phase 2**: Carries are detected across all limbs simultaneously.
- **Phase 3**: If an intermediate resultant limb generates a carry, it is added to its preceding limb in parallel, simulating the carry adjustment.
- **Phase 4**: Only in the rare case where further carries are produced do we adjust them in the Kogge-Stone technique to ensure correctness.

Algorithm 7 illustrates the process of adding two large numbers using the approach described above. The algorithm operates in four distinct phases. In our case, the additions in Phase 1 add X_i and Y_i without taking carries into account, so all the additions are independent of each other. The carries are detected independently in Phase 2 and then added back in Phase 3, mitigating the typical carry propagation stalls. However, after Phase 3, though highly unlikely, further carry propagation may still be needed; in that case, we detect the maxed-out blocks, and with the consideration of carry-generating blocks, we adjust the intermediate sums and propagate carries in Phase 4.

The adjustment of the intermediate sums after detecting the carries generated in Phase 3 can be done in several ways, such as propagating the carries sequentially or employing the Kogge-Stone adder method. We have chosen to approach the Phase 4 using the Kogge-Stone adder technique, as it simply utilizes carry positions and maxed-out blocks, which avoids the sequential adjustment of the carries, making it much faster to compute.

As we can observe, all the operations in Phase 1, Phase 2, and Phase 3 can be done in parallel, one after the other, and that is where we may exploit parallelism using vectorization. While the need for Phase 4 arises only when further carries are generated, potentially introducing some sequential operations on a smaller subset and thus a degree of performance degradation, the overall approach still remains significantly faster than purely sequential carry propagation. Thus, these four phases aggregately make the large-number addition faster than sequentially adding up the limbs and propagating the carries one by one (Eq. 2.1).

3.1.1 Proposed Addition Algorithm

Next, we describe the design of our parallel addition algorithm, named Parallel Multi-Stage Large-Number Addition (PML Add), which is shown in Algorithm 7. The algorithm operates in four phases, which were described before, building on the carry propagation chaining. Phase 1 adds the two blocks in parallel, ignoring any carry, followed by Phase 2 detecting the carries in parallel. Phase 3 then works in parallel to add the carries to the respective blocks in case of a carry generated by the preceding blocks. If, after Phase 3, further carries are generated, we adjust the carries using the Kogge-Stone Adder technique ([Yee19]) in Phase 4, which is a rare case with a large K. Although determining whether to enter Phase 4 or not in the algorithm appears straightforward and efficient, this can introduce a significant performance overhead with large operands, as we show later in Chapter 5.

In Phase 4 of the algorithm, we already have the information about the blocks that have generated the carries. We first detect the maxed-out blocks; with the shifted carry values, if we add the carries to the respective blocks, further carry will only be generated if the blocks to which carries are added are maxed out. Thus, we can add the carry and *max_mask* bits, simulating the actual scenario. This is followed by XORing the resultant mask bits with *max_mask* bits ([Yee19]). With these updated masking bits, we will add one to the respective intermediate sums from Phase 3.

Note that *C_mask* and *max_mask* contain either 0 or 1, denoting the carry or maxed-out blocks for their respective positions. Thus, these values can be stored in a much smaller space than the original numbers. Also, if we are processing the parallel parts in chunks, as with vectorization, these carry and maxed-out masks are constant in size and can fit into a typical mask register. By this method, we can avoid sequential propagation of carries. While computations combining all four phases seem computationally expensive, Phase 4 only triggers rarely, giving the parallelization benefits from Phases 1-3. However, with architectural constraints, one may mitigate the slowdown of Phase 4 by operating in chunks of elements and utilizing masked registers ([Yee19]). The efficiency of operating on the masked registers is purely micro-architecture dependent; this might adversely affect the performance, as noted in prior works ([Yee19; RSS23]). As a result, in our implementation, we typically avoid operating (e.g. adding up the masked registers) on the masked registers by putting this in Phase 4, which almost never gets executed. The final implementation of the Algorithm 7 is outlined in implementation 8.

Algorithm 7: Proposed Parallel Multi-Stage Large-Number Addition (PML Add) Algorithm

```
Input  : Two n-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$ digits each,
         where $X_i, Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least significant blocks.
         Also, $0 \leq X_i, Y_i \leq B^K - 1$.
Output: Sum $S = S_{p-1} \dots S_0$, where $S_0$ is the least significant block.
// Phase 1:  Compute Intermediate Sums (Parallel)
for $i \leftarrow 0$ to $m - 1$ in parallel do
    $S_i \leftarrow X_i + Y_i$
// Phase 2:  Detect Carries (Parallel)
$carry\_detect \leftarrow$ false;                                   // Flag for carry presence
for $i \leftarrow 0$ to $m - 1$ in parallel do
    if $S_i >= B^K$ then
        $C_i \leftarrow 1$
        $S_i \leftarrow S_i - B^K$
        $carry\_detect \leftarrow$ true;                            // Mark carry existence
$C'_{i+1} = C_i, \quad \forall 0 \leq i < m - 1, C'_0 = 0$
// Phase 3:  First Round of Carry Addition (Parallel)
if $carry\_detect = true$ then
    $carry\_detect \leftarrow false$
    for $i \leftarrow 0$ to $m - 1$ in parallel do
        if $C'_i = 1$ then
            $S_i \leftarrow S_i + 1$;                                // Add carry, modulo $2^{32}$
            if $S_i \geq B^K$ then
                $carry\_detect \leftarrow true$ $C'_i \leftarrow 1$ $S_i \leftarrow S_i - B^K$
        else
            $C'_i \leftarrow 0$
// Phase 4:  Carry Propagation (Detect Max Blocks and adjust accordingly)
if $carry\_detect = true$ then
    $C\_mask \leftarrow C'_i$ for all $i$;          // Initialize carry mask from previous phase
    // Detect maxed-out blocks and propagate carries in parallel
    for $i \leftarrow 0$ to $m - 1$ in parallel do
        $max\_mask_i \leftarrow (S_i = B^K - 1)$;            // Check if block is at max value
    // Propagate carries
    $C\_mask \leftarrow C\_mask \ll 1$;                                // Left shift carry mask
    $C\_mask \leftarrow C\_mask + max\_mask$;                              // Add max mask
    $max\_mask \leftarrow C\_mask \oplus max\_mask$;                         // XOR operation
    // Adjust sums based on propagated carries
    for $i \leftarrow 0$ to $m - 1$ in parallel do
        if $C\_mask_i = 1$ then
            $S_i \leftarrow S_i + 1$;                                   // Add incoming carry
            if $S_i \geq B^K$ then
                $S_i \leftarrow S_i - B^K$;                                    // Adjust sum
Return $S$;
```

Example 3.1.1.1.
The instance below triggers through all the phases:

```
Assuming B=10 and K=2
a: 42 11 97 78
b: 12 88 02 30
+ ------------------
s: 54 99 99 08       (phase 1, parallelly adding up)
--------------------
c:  0  0  0  1       (phase 2, detecting carries)
c:  0  0  1  0       (left shifted to account for adjustment)
--------------------
s:  54 99 99 08
c:   0  0  1  0
+ ------------------ (phase 3, adjusting the carries)
s:  54 99 00 08
--------------------
c:   0  0  1  0
Now that further carries are generated,
we will proceed with phase 4
--------------------
c:   0  1  0  0      (c=<<1)
m:   0  1  0  0      (maxed out blocks)
    ----------------
c:   1  0  0  0      (c+=m)
m:   1  1  0  0      (m XOR=c)
--------------------
s:   54 99 00 08
m:    1  1  0  0
+ ------------------
s:   55 00 00 08   (result)
```

3.2 Implementation of Proposed PML Add using AVX-512

To parallelize the large number addition, we have used SIMD constructs, specifically AVX-512 instructions, on a recent x86-64-based CPU. Instead of direct assembly instructions, we have used the AVX512 intrinsic function calls in C and relied on the compiler to generate the necessary conversion to AVX instructions for easier programming.

3.2.1 Data Representation

We have opted for a contiguous array of limbs to represent large numbers and optimize cache performance. Register-based representation can be utilized only for smaller numbers; we have chosen memory-based representation to handle much larger values and enable efficient contiguous loads and stores into and from 512-bit AVX registers. For our experiments, we typically selected a 64-bit limb size to align with the word size of modern x86-64 CPUs and reduce the total number of limbs. Minimizing the limb count helps exponentially decrease the number of operations in multiplication, as the multiplication algorithms we chose incur $O(n^x)$ time complexity.

We deployed a struct-based approach, having a total of four fields:

```
typedef struct
{
    aligned_uint64_ptr limbs; // Pointer to the limbs
    bool sign;                // Sign of the number
    size_t size;              // Size of the limbs
    bool overflow;            // Indicates of overflow after addition
} limb_t;
```

We store the absolute values in a contiguous memory block, the count of limbs in the size field and the sign of the operand in a separate sign field.

We employ a *native-radix* or *saturated* representation for addition operations, fully utilizing all 64 bits.

When it comes to limb ordering, we typically read string inputs for the operands starting from the least significant end (the right side) of the string. Then, we may store these values either from the highest index of the limb array or from the lowest index of the limb array and fill the remaining indexes of the limb array contiguously. We may use either of the representations shown below and have used them interchangeably across experiments. Specifically, in the final implementation of PML Add (Algorithm 8), we switched to storing the limbs from the lowest index.

Example 3.2.1.1.
Consider a random 256-bit hexadecimal string:

`0xEF1206754ee9451FA5F3C7B912E4D86F1B92A0D5E7C8F9346D1B5E2F9A3C7D8E`

64-bit Native-radix Representation (Highest Index)

`limbs`$_{64}$ =

i	`limbs`$_{64}[i]$
0	`0xEF1206754EE9451F`
1	`0xA5F3C7B912E4D86F`
2	`0x1B92A0D5E7C8F934`
3	`0x6D1B5E2F9A3C7D8E`

64-bit Native-radix Representation (Lowest Index)

`limbs`$_{64}$ =

i	`limbs`$_{64}[i]$
0	`0x6D1B5E2F9A3C7D8E`
1	`0x1B92A0D5E7C8F934`
2	`0xA5F3C7B912E4D86F`
3	`0xEF1206754EE9451F`

Observation: Our initial implementation represented numbers using base-10, i.e., as decimals, and to accommodate 32 or 64-bit limbs, we utilized an unsaturated representation accordingly (e.g., for 32-bits, we were storing up to 9 decimal digits into each 32-bit limb and for 64-bits we were storing 19 decimal digits). However, this approach of representation was computationally costly, and to extract high or low bits, we had to perform division and modulo operations. In contrast, with a hexadecimal representation, we could use shifting and logical operations, which are more efficient than performing division and modulo.

3.2.2 Implementations

Before obtaining our final addition implementation, we had two sub-optimal implementation approaches, which we briefly discussed first, followed by the final version:

Sub-optimal Implementations

First Version: Decimal-based unsaturated-radix of 64-bit limbs

Initially, we went for a decimal-based representation of strings and packed them within 18 digits into chunks of 64-bit limbs. The idea is similar to the approach we discussed (Algorithm: 7), but the only difference is that we were handling the last phase sequentially.

Approach:

1. Loading operands from the starting address of memory in chunks of 512-bits
    (a) Add them
    (b) Check for carry and generate a carry mask of 8 bits for eight 64-bit limbs
    (c) Subtract $1e^{18}$ from sum in-case of carry
    (d) Store the intermediate result back in memory
    (e) Store the carry mask in an array
2. Left-shift the carry mask array by one position
3. Loading the intermediate sums from memory in chunks of 512-bits
    (a) Add 1 based on the left-shifted carry mask value
    (b) Check again for carry and store it in a carry mask array
    (c) Subtract $1e^{18}$ from sum in-case of carry
    (d) Store the result in memory
4. Check if new carries are generated; if yes, handle them sequentially

Although with our test cases, the last phase was never triggered, this implementation did not give any speed-up as compared to GMP, and performance was much more degraded for higher-sized operands. Also, this decimal-based representation did not fully utilize the bits, as it had to be an unsaturated representation. Also, performing addition on unsaturated limbs does not actually wrap around. Consequently, we had to manually subtract the resultant from the max-base to adjust to the correct resultant, incurring more cost in comparison to wrapping around.

Second version: Hexadecimal-based saturated-radix Addition of 64-bit limbs

Instead of using decimal-based representation, we moved to hexadecimal-based representation to utilize the limbs fully and eliminate the manual adjustments in case of overflows. We implemented Algorithm 7 by splitting it into separate independent phases. To take advantage of AVX512 for parallel addition, we're limited to adding two 512-bit vectors at a time (two sets of eight 64-bit values). For each addition, we load eight numbers from one set (say X) and eight from another (say Y) into two AVX512 registers, perform the addition, and get our intermediate sums. Since AVX512 only gives us a handful of registers to work with, we can't hold all the sums in them at once. After each addition, we store the intermediate sums back in memory to free up space for the next set of numbers. Here's how it breaks down across the phases:

- **Phase 1**: We load the operands in chunks of eight pairs, add them using AVX512 and save the intermediate sums to memory. Repeat this for all the chunks remaining.
- **Phase 2**: We pull those intermediate sums back from memory (again, eight at a time), run the necessary computations, like checking for overflows or making adjustments, and store the updated sums back to memory.
- **Carry Shift**: Next, we shift the carry-bit array (tracking overflows) one position to the left. This step is tricky because it's tough to parallelize with AVX512, so it might have to happen one element at a time sequentially.

- **Phase 3**: We load the adjusted sums from Phase 2 back into registers, still in chunks of eight, do more computations, and save the results to memory unless Phase 4 kicks in.

This approach boosted the performance a bit but was still much slower compared to GMP across 65536 bit-sized operands. Mainly, the big issues for not getting performance improvements are:

1. **Too Much Memory Back-and-Forth**: We're constantly loading and storing intermediate sums between memory and registers across Phases 1 to 3. This ping-ponging slows things down since memory access, even if it is cached, takes way longer than working in registers.
2. **Stuck on the Carry Shift**: Shifting the carry-bit array on the memory doesn't play nice with parallel processing, creating a bottleneck that downgrades AVX512's speed benefits. As a result, we relied on the typical ALU operations for shifting the carry bit array:

   Example 3.2.2.1.

   ```
   for (int i = 0; i < n-1; --i){
        carry_arr[i] = carry_arr[i+1];
   }
   ```

Solution: To tackle the memory-back-and-forth issue, we tweaked the implementation slightly by removing the redundant loads and stores. Also, we started processing the limbs in chunks of eight from the least significant end (highest index) toward the most significant. This also allows us to propagate carries directly through the computation without unnecessarily writing intermediate results back to memory. A key improvement lies in how we handle carry bits: instead of storing the final carry from each chunk and reloading it later, we extract it immediately and shift it into the carry mask for the next phase's addition.

Also, we tried to use masking load and stores for an operand size of 256-bit so that it does not include unintended data to be loaded or stored in memory. However, we saw performance degradation with masking load and stores during the implementation. Hence, we switched to using AVX2 (256-bit vectorization) only for 256-bit operands, avoiding the performance issue with masked load and stores.

Final Implementation of PML Add

Finally, to implement the PML Add (algorithm 7), we utilized 64-bit native-radix limbs in hexadecimal representation. The implementation approach is depicted here: algorithm 8.

For parallelization, we can process two sets of eight 64-bit elements using the AVX-512 registers. We begin by loading eight limbs from both operands into the AVX512 registers, and then we add them in parallel. Next, we detect probable carries through a simple wrap-around check and generate an 8-bit mask. We extract the carry bit value from lane-3's second value and store it in *c_out*, as it needs to be added to the first element in lane-0 in the next round of addition. To manage the carries, we shift the 8-bit *carry_mask* one bit to the left. We then combine this shifted mask with any previous *c_in* (which starts as 0; the current round's *c_out* becomes the next round's *c_in*. If any carry was detected, we add 1 to all the elements inside the lanes of the intermediate sum according to the *carry_mask*. We check for probable carries again. Finally, we store the eight intermediate sums from this round in memory at the same index from which the operands were read. Rarely, even when Phase 4 is triggered, we are handling the carry propagation using the carry and maxed-out logic. We then modify our index pointer by eight positions and repeat the same set of operations while moving towards the remaining elements. One thing to note: for the final implementation of addition and subtraction, we tweaked the ordering of the elements stored; earlier, we were storing from the highest index, and now we have switched to storing from the lowest index. Thus, we move from lowest to highest.

Algorithm 8: Outline of implementation of PML Add using AVX512

```
Data: Arrays x and y of 64-bit elements, output array s, size n (multiple of 8)
Result: $s = x + y$
c_in ← 0, c_out ← 0 ;                                  // Initialize carry variables
for $i \leftarrow 0$ to n - 8 step 8 do
    c_in ← c_out ;                            // Propagate carry from previous round
    // Phase 1: Parallel Addition
    x_vec ← Load eight 64-bit elements from x[i] ;
    y_vec ← Load eight 64-bit elements from y[i] ;
    r_vec ← x_vec + y_vec ;                         // Add eight elements in parallel
    // Phase 2: Carry Detection
    carry_mask ← Generate 8-bit mask where r_vec < x_vec ;   // Detect carries
    c_out ← carry_mask » 7 ;                                 // Extract highest carry
    carry_mask ← carry_mask « 1 ;                                      // Shift left
    carry_mask ← carry_mask | c_in ;                     // Combine with input carry
    // Phase 3: Adjust Carries
    carry_vec ← Set 1s in lanes based on carry_mask ;       // Create carry vector
    r_vec_new ← r_vec + carry_vec ;                                // Apply carries
    mask_new ← Generate 8-bit mask where r_vec_new < r_vec ;         // Check new
     carries
    r_vec ← r_vec_new ;
    // Phase 4: Handle Further Carries
    if mask_new is non-zero then
        mask_new ← mask_new « 1 ;                                   // Shift carries
        eq_mask ← Generate 8-bit mask where r_vec = all-ones ; // Detect all-ones
        mask_new ← mask_new + eq_mask ;                            // Combine masks
        c_out ← c_out | (mask_new » 8) ;                        // Update carry out
        sub_mask ← mask_new XOR eq_mask ;           // Compute subtraction mask
        r_vec ← Subtract all-ones from r_vec in lanes where sub_mask is set ;
         // Adjust result, with the help of wrap around
    // Store Results
    Store eight 64-bit elements from r_vec to s[i] ;
result.carry ← c_out ;                                         // Store final carry
```

In Phase 3, instead of preparing Ones inside *carry_vec*, we can also opt for subtracting the values with maximum-base based on the *carry_mask*, which will minimize the number of operations (similar to the Kogge-Stone technique). Specifically, we can subtract an all-ones vector from `r_vec` in lanes where the `carry_mask` is set. We performed the final implementation exactly.

Compilers, including the Intel Compiler, typically avoid generating complex sequences of mask instructions. Instead, they prefer to move data into general-purpose registers for processing and then return it to mask registers. Therefore, when programming in C, it is essential to ensure that the compiled code does not move masks back to general-purpose registers, as this can degrade performance.

Initially, we declared the carry input (*c_in*) and carry output (*c_out*) as unsigned 8-bit integer variables. However, we defined the *carry_mask* as an *__mmask8* variable type. Although the code to shift the *carry_mask* (*carry_mask* $<<$ *1*) was generated using AVX registers, the operation for *carry_mask*$|$(*c_in* $>>$ *7*) was performed with general-purpose registers. This happened because *c_in* was declared as a *uint8* type, leading the compiler to prefer avoiding the complexity of converting it into AVX registers.

Additionally, storing the result of *carry_mask* and *c_out* as a *uint8* type was also executed using general-purpose registers.

Example 3.2.2.2.
A snippet of code that faces the above-mentioned issue:

```
some_func(result, a, b, uint8 c_in, uint8 *c_out) {
        ...
        ...
        (*c_out) = carry_mask & 1;
        carry_mask <<= 1;
        carry_mask |= (c_in >> 7);
        ...
        ...
}
```

For this reason, we did not see any speed-ups, but we matched the performance with GMP. In order to tackle this, we tried to tweak the variable declarations of *c_out* and *c_in* into *__mmask16* to avoid the involvement of general-purpose registers, and instead of using typical OR operation, we used *_mm512_kor*(), which computes the bitwise OR of 16-bit masks provided to it on the AVX registers. However, shifting operations like *carry_mask* $<<$ *1* and *c_in* $>>$ *7* have to be handled by the general-purpose registers, and we can't do much about it currently as we don't have any available instructions that can directly shift bits inside the mask registers.

Example 3.2.2.3.
Below is the optimized code snippet instance:

```
    some_func(result, a, b, __mmask16 c_in, __mmask16 *c_out) {
        ...
        ...
        (*c\_out) = carry_mask;
        carry_mask <<= 1;
        c_in = (c_in >> 7);
        carry_mask = _mm512_kor(carry_mask, c_in);
        ...
        ...
}
```

Also, after all rounds are over, we check for the *c_out*. If its least significant bit is one, we pass the information about the carry-out, too, so that the sum can be adjusted properly. This carry-out information is not needed for subtraction operations.

One benefit of processing eight elements and finishing each of the needed operations on these elements is that the Phase 4 trigger event is limited only to these eight elements. In another way, even if the rarest situation where the current limb generating carry and the preceding limb is maxed out, i.e. Phase 4 is triggered, with the algorithm 7 we may have to sequentially move from the least significant end to the most significant end (the problem with the second version of the sub-optimal implementation. However, with this implementation, the adjustment in phase 4 is just limited to the current eight limbs without affecting the next round of limbs. Unless each round consists of the operands that lead to Phase 4 triggering, we only process and adjust carry for some of the rounds, thereby gaining performance gains. By doing all of these, we have got the speed-up now, finally!, as compared to GMP.

Following AVX512 intrinsics (from AVX512-F and AVX512-VL) were utilized for the PML Add implementation:

1. *_mm*512*_load_si*512: For loading the operands from memory into AVX512 registers
2. *_mm*512*_add_epi*64: For adding up two sets of eight 64-bit elements loaded into two AVX512 register
3. *_mm*512*_cmplt_epu*64*_mask*: For generating the 8-bit carry mask using wrap-around check; performs less-than checks of eight 64-bit values of two AVX512 register
4. *_mm*512*_mask_set*1*_epi*64: To produce eight 64-bit 1's based on carry_mask to be added to intermediate sum
5. *_mm*512*_kor*: To compute OR between two 16-bit mask registers
6. *_mm*512*_kand*: To compute AND between two 16-bit mask registers
7. *_mm*512*_store_si*512: To store AVX512 register content into memory

The programming approach implements the PLM Add using a macro-based AV-X512 vectorized kernel, which does all the operations by taking the operands and previous round carry-in using AVX512 and returns the sum as well as the carry-out for the next round. We repeatedly call this kernel for all the operands from the least significant end towards the most significant end, passing eight limbs at a time and the carry-out of the previous round.

The current implementation assumes that both operands are equal-sized and are of at least 512 bits, and the number of limbs is a multiple of 8. If we are working with 128-bit or 256-bit, instead of using AVX512, we can use AVX2 for 256-bit and SSE2 for 128-bit and adjust the implementation accordingly. Also, if the limb count is not a multiple of eight, we may use a combination of AVX256, AVX2 and manual 64-bit to take care of the remainder of element additions. Figure 3.1 shows the overview of the AVX512 Lane operations for eight limbs (512-bits) using PML Addition, showing operations of the three phases (Phase 4 is not shown for simplicity).

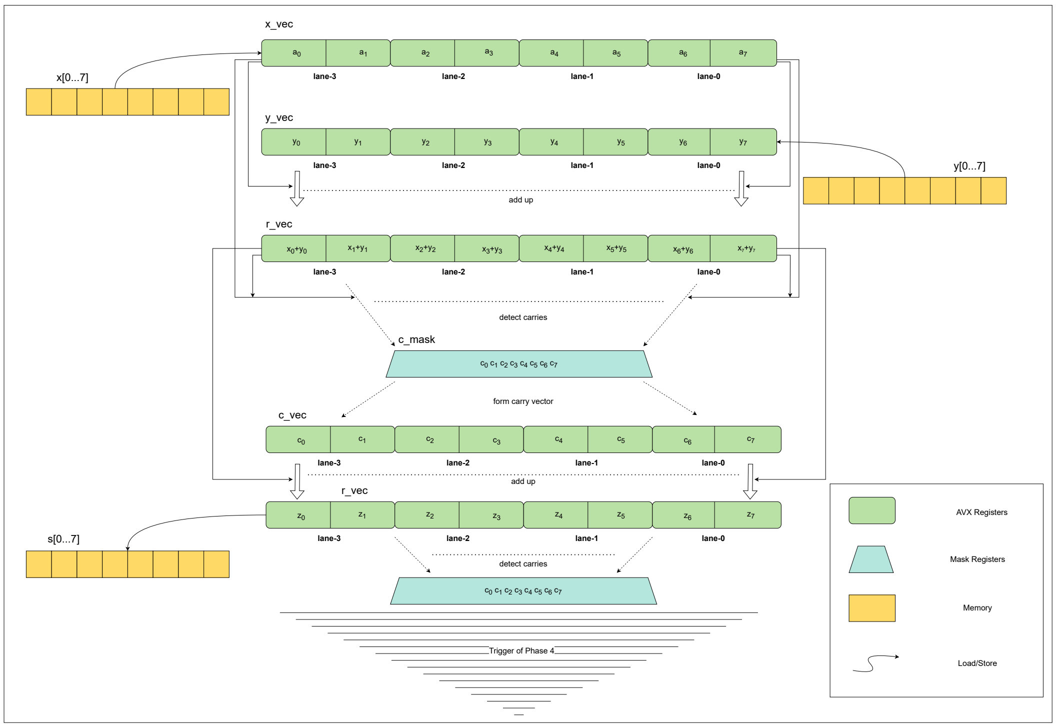

FIGURE 3.1: Overview of AVX Lane operations for eight limbs using PML ADD

3.3 Evaluation

The core questions we are trying to answer here are the following:

1. Are the computations yielding correct results?
2. Are we getting any performance improvements, in terms of execution time, instruction count, and CPU Cycles, over GMP?
3. How much performance gain do we get compared to GMP for addition with different data sizes?
4. How much performance gain are we observing with vectorization as compared to a non-vectorized baseline implementation?

For the vectorized C code implementation of addition (PML Add), we have utilized the GCC compiler with the following optimization flags:

```
-mavx512f -mavx512vl -O1
```

Also, we compiled the baseline variant using the O1 flag. The implemented codes are available in the GitHub repository at https://github.com/iamsubhrajit10/Large-Number-Arithmetic-Operations.

The correctness check and benchmarking were conducted on an x86-64-based Intel Xeon E-2314 CPU, with four cores and 32 GB RAM, clocked at 2.80 GHz, based on Rocket Lake microarchitecture, equipped with the necessary AVX-512 instruction set support, running on Fedora 41 OS with Red Hat 14.2.1-7 GCC compiler.

3.3.1 Correctness

To validate the correctness of our vectorized addition implementation while achieving performance gains, we generated multiple sets of 1,00,000 random test cases using the random numbers produced by the Mersenne Twister random number generator [Wik] seeded with a random integer between 1 and 4,294,967,295 for each operand size implemented via Python's random module and the `gmpy2` ([PYP]) library. These test cases were stored as strings in files, formatted as `<operand1, operand2, result>`, with results precomputed for validation. We created multiple such sets of 1,00,000 test cases over different operand sizes ranging from 256 bits to 1,31,072 bits, spanning different time frames over approximately three months, to mitigate potential biases in the pseudo-random number generator. For each such set, we read the files from our C code, computed the product of `operand1` and `operand2` using our implementation, and verified the outputs against the precomputed `result` values, ensuring correctness across all test scenarios. We read string inputs from the files and converted our results back to a string for comparison.

All computations we performed for addition operations using PML Add yielded correct results (compared to pre-computed values using the gmpy2 library). Notably, phase 4 of the algorithm (ref algorithm: 7) was never executed with the set of lakhs-and-lakhs of randomly generated test cases over different bit sizes, primarily because of the rare chance, i.e. $1/2^{64}$ (refer to Section 5.1). However, for completeness, we manually passed through some test cases where the phase would trigger, and we got matched with the pre-computed results.

3.3.2 Performance Compared to GMP and Baseline

The work by [RSS23] did not provide their source code or detailed implementation specifics for addition, making it challenging to replicate and compare their performance gains with our implementation. On the other hand, the implementation of Kogge-Stone Adder by Yee for y-cruncher [Yee19] claimed not to outperform sequential add-carry instructions for addition on large numbers. Consequently, we opted to benchmark our performance primarily against the GMP, which offers one of the fastest single-core, non-SIMD ([SIM]), leveraging carefully architecture-tuned optimized assembly routines. We benchmarked against the latest available GMP version 6.3.0. We installed the relevant GMP developer packages on the test system and invoked the corresponding functions of the GMP API from C code.

We acknowledge that hand-written architecture-fine-tuned assemblies are hard to beat with code generated by a compiler that works for generalized cases across architectures. To get a better idea of the performance gains through vectorization over different operand sizes, we also compared our vectorized implementation against a non-vectorized single-threaded implementation of the same code.

In our benchmarking process, we concentrate on measuring the individual additions for various bit sizes. We recorded each addition's execution time, throughput and some other performance metrics across different bit sizes on the Intel Xeon E-2314 CPU. Our results were compared with existing alternatives from GMP and our non-vectorized baseline code. We initialized the variables from the strings and did not consider initialization in the measurements. We measured only the timing of the function call that computes the addition; in our case, $pml\_add()$ for PML Add and $baseline\_add()$ for baseline, and for GMP, we called the $mpz\_add()$ function from three separate C programs. We used GMPbench's strategy to calibrate the

CPU Speed and extract the timings and throughput accordingly. The measurement technique is briefly described in Appendix A.

As we already mentioned, phase 4 of the PML Add was never triggered with the extensive set of randomly generated test cases. To get an overview of the performance for inputs triggering phase 4, we separately passed through some test cases that trigger phase 4 always for every chunk of eight 64-bit elements for the operands. Thus, the comparison in the following sections contains a total of four models: Baseline (non-vectorized) implementation of PML Add, GMP's Add, PML Add, and PML Add Worst. The PML Add Worst is the version that, after the computation of phases 1 to 3, triggers phase 4 and processes the computations within that block. In contrast, the usual PML Add takes all the random inputs and never gets triggered into phase 4. We randomly read the string operands from the randomly generated test case files.

Execution Time: Figure 3.2 shows the best measured execution time of our final implementation of PML Add (8) in comparison with GMP's Addition function call, the baseline Addition and PML Add Worst over data sizes ranging from 256-bit to 131072-bit on the Intel Xeon E-2314. The red line depicts GMP_ADD timings in ns, teal green is for PML_ADD, lighter green denotes the PML Addition Worst Case, and blue highlights the execution timings for the baseline non-AVX version. We also listed out the speed-ups inside the plot. The X-axis denotes the bit sizes, and the Y-axis is for the Execution Time in nanoseconds, and is on a log_{10} scale. We plotted the timings for 256-, 512-, 1024-, 2048-, 4096-, 8192-, 16384-, 32768-, 65536- and 131072-bit sized operands.

With PML Add, we achieve a 2.24x speed-up over GMP with 65,536-bit operands and consistently maintain over 1.8x speed-up for operands larger than 2,048 bits, reaching near 2x speed-up beyond 8,192 bits. In contrast, when compared to the baseline implementation, we record over 6x speed-up for 65,536-bit and 1,31,072-bit operands and consistently achieve a near 5x speed-up for operands larger than 4,096 bits. However, it is worth noting that, even when phase 4 is triggered as with PML Add (Worst Case), we are still faster than GMP except for 1,31,072 bits. In comparison to GMP, with PML Add (Worst Case), we achieve the highest performance gains for 512-bit, a 2.11x speed up, and the performance gains slowly reduce. However, on average, we are still getting 1.38x speed up. Thus, for most possible cases, working with large random number additions, we can safely say we can achieve nearly 2.06x speed up on average in comparison to GMP, and in rare cases, we may be matching GMP or be slightly higher.

Throughput: We also measured the throughput of our PML Add implementation (including Worst Case) against the GMP, using GMPbench's technique mentioned in Appendix A. The throughput of PML Add is in line with the execution time. We're achieving up to 2.24 times the throughput of GMP Add for 65536-bit and constantly ranging above 1.8x for larger operands. And a consistent throughput gain of over 5x in comparison with the baseline. For 8192 bits, we see the highest throughput gain, 6.69x. Notably, the PML Add variant, which goes through phase 4, is still faster than GMP for most operands and catching up with GMP for larger operands.

Table 5.3 in Chapter 5 summarizes the throughput comparison for various operand sizes across all the models.

Performance Metrics: In addition to execution time and throughput, we also recorded some other performance metrics of our implementation (both regular and worst case) compared to GMP and the baseline, mainly: Ticks and User Instruction count. We have not mentioned the page faults as the largest operand size, i.e. 131072-bit, can easily be referred through one page. Also, with the AVX512 approach, there are not many comparable cache benefits, as all the variants use a similar contiguous array structure for limb storage and fetch the limbs in a similar fashion. We used $perf\_event\_open$ and RDTSC(P) ticks (refer to A) to measure these metrics during actual runs, rather than through simulators, to gain better insights into real-life workloads. We have listed out the performance metrics: For regular, non-phase-4 trigger cases, on average, PML Add implementation has nearly 37% less user instruction counts than GMP Add and nearly 75% less than the baseline. But for worst cases, PML Add (Worst Case line) has nearly 4% fewer user instructions than GMP and 65% fewer user instruction counts than baseline.

On the other hand, PML Add has nearly 55% fewer tick counts when compared to GMP and nearly 84% fewer tick counts than the baseline implementation. In worst cases, it still has fewer ticks, ranging between 15% and 27% improvement. Notably, both the tick count and user instruction counts are proportionally in line with the execution time, except that for 1,31,072 bits, we were seeing a degradation in the performance gain in execution time, but both user instruction counts and the ticks do not indicate any performance degradation (similar user instruction counts and 1.4x performance gains in terms of ticks).

In chapter 5, the Table 5.4 summarizes the user instruction count comparison, and Table 5.2 summarizes the ticks comparison for all the models on Intel Xeon E-2314.

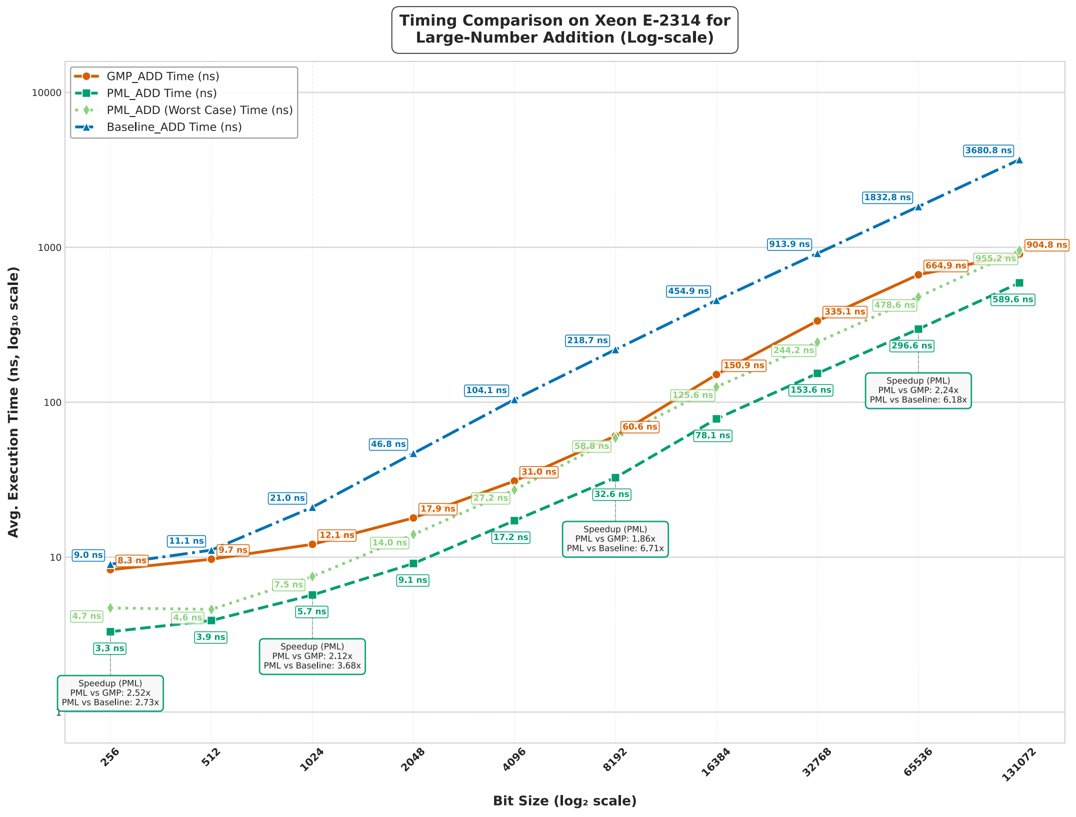

FIGURE 3.2: Timing comparison of PML Add, PML Add Worst, Baseline Add, and GMP Add for various data sizes on Intel Xeon E-2314

Chapter 4

Parallelizing the Subtraction of Large Integers

The content of the chapter is as follows: We first present our approach for parallelizing subtraction on large random integers. After that, we discuss the final implementation, followed by the evaluation of it.

4.1 Parallel Multi-phase Subtraction Algorithm

. Subtraction on large numbers is similar to performing addition. However, the order of the operands might be crucial. In this work, we assume the first operand, let's say X, is always greater than or equal to the second operand, say Y, while performing $X - Y$. However, in cases where X is smaller than Y, we can swap the numbers and return the difference between the two numbers and extra information containing the sign of the result, an approach similar to GMP's subtraction implementation.

4.1.1 Proposed Subtraction Algorithm

We propose a parallel subtraction algorithm, named Parallel Multi-Stage Large-Number Subtraction (PML Sub), shown in algorithm 9, which operates in four phases, similar to the addition algorithm we proposed. In Phase 1, we simultaneously subtract Y_i from X_i without accounting for borrow or underflow. Then, in Phase 2, we generate the borrows and adjust the underflowed subs, followed by left-shifting the borrow values. Phase 3 adjusts the borrows in the respective positions in parallel. In rare cases, Phase 4 eventually adjusts the intermediate subs if any borrow was generated during Phase 3.

4.2 Implementation of Proposed PML Sub using AVX-512

For parallelization of large number subtraction, we utilized the exact *saturated* 64-bit limb-based data representation of addition and used the AVX-512 C intrinsics for the final implementation. We also had a similar set of two sub-optimal implementations for subtraction as with addition. We only listed out the final implementation approach here. Instead of carry propagation, we generated borrow propagation and accordingly designed the implementations. We have only shown the final 64-bit hex-based PML Sub implementation 10.

Algorithm 9: Proposed Parallel Multi-Stage Large-Number Subtraction (PML Sub)

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$
         digits each, where $X_i$, $Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least
         significant blocks. Also, $0 \leq X_i, Y_i \leq B^K - 1$ and $X > Y$.
Output: Sub $S = S_{m-1} \ldots S_0$, where $S_0$ is the least significant block.
// Phase 1: Compute Intermediate Subs (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    $S_i \leftarrow X_i - Y_i$
// Phase 2: Detect Borrows (Parallel)
$borrow\_detect \leftarrow$ false;                          // Flag for borrow presence
for $i \leftarrow 0$ to $m-1$ in parallel do
    if $S_i < 0$ then
        $B_i \leftarrow 1$
        $S_i \leftarrow S_i - B^K$
        $borrow\_detect \leftarrow$ true;                   // Mark borrow existence

$B'_{i+1} = B_i, \quad \forall 0 \leq i < m-1, B'_0 = 0$
// Phase 3: First Round of Borrow Adjust (Parallel)
if $borrow\_detect = true$ then
    $borrow\_detect \leftarrow false$
    for $i \leftarrow 0$ to $m-1$ in parallel do
        if $B'_i = 1$ then
            $S_i \leftarrow S_i - 1$
            if $S_i = -1$ then
                $borrow\_detect \leftarrow true$
                $B'_i \leftarrow 1$
                $S_i \leftarrow B^K - 1$
            else
                $B'_i \leftarrow 0$
// Phase 4: Borrow Propagation (Detect Min Blocks and Adjust Accordingly)
if $borrow\_detect = true$ then
    $B\_mask \leftarrow B'_i$ for all $i$ ;                  // Initialize borrow mask
    for $i \leftarrow 0$ to $m-1$ in parallel do
        $min\_mask_i \leftarrow (S_i = 0)$ ;                 // Check if block is at min value
    $B\_mask \leftarrow B\_mask \ll 1$ $B\_mask \leftarrow B\_mask + min\_mask$
     $min\_mask \leftarrow B\_mask \oplus min\_mask$ ;        // XOR operation
    for $i \leftarrow 0$ to $m-1$ in parallel do
        if $B\_mask_i = 1$ then
            $S_i \leftarrow S_i - 1$ if $S_i < 0$ then
                $S_i \leftarrow S_i + B^K$
return $S$;
```

Algorithm 10: Outline of implementation of PML Sub using AVX512

```
Data: Arrays x and y of 64-bit elements, output array s, size n (multiple of 8) and
      $x > y$
Result: $s = x - y$
b_in ← 0, b_out ← 0 ;                              // Initialize borrow variables
for $i ← n - 8$ downto 0 step 8 do
    b_in ← b_out ;                           // Propagate borrow from previous round
    // Phase 1: Parallel Subtraction
    x_vec ← Load eight 64-bit elements from x[i] ;
    y_vec ← Load eight 64-bit elements from y[i] ;
    r_vec ← x_vec - y_vec ;                  // Subtract eight elements in parallel
    // Phase 2: Borrow Detection
    borrow_mask ← Generate 8-bit mask where y_vec > x_vec ;// Detect borrows
    b_out ← borrow_mask ;                               // Keep borrow for next round
    borrow_mask ← borrow_mask » 1 ;          // Shift right (little-endian order)
    borrow_mask ← borrow_mask | b_in « 7 ;      // Account for borrow-in to most
     significant lane
    // Phase 3: Adjust Borrows
    borrow_vec ← Set 1s in lanes based on borrow_mask ;
    r_vec_new ← r_vec - borrow_vec ;                         // Adjust for borrows
    mask_new ← Generate 8-bit mask where r_vec_new > r_vec ;       // Check new
     borrows
    r_vec ← r_vec_new ;
    // Phase 4: Handle Further Borrows
    if mask_new is non-zero then
        mask_new ← mask_new « 1 ;                                // Shift borrows
        eq_mask ← Generate 8-bit mask where r_vec = all-zeros ;          // Detect
         all-zeros
        mask_new ← mask_new + eq_mask ;                          // Combine masks
        b_out ← b_out | (mask_new » 8) ;                      // Update borrow out
        sub_mask ← mask_new XOR eq_mask ;             // Compute subtraction mask
        r_vec ← Add all-ones into r_vec in lanes where sub_mask is set ; // Adjust
         result with the help of wrap around
    // Store Results
    Store eight 64-bit elements from r_vec to s[i] ;
```

In Phase 3, instead of preparing Ones inside the *borrow_vec*, we can choose to add the values with maximum-base based on the *borrow_mask*, similar to what we mentioned in addition, which will minimize the number of operations (similar to the Kogge-Stone technique). That is, add all-ones into `r_vec` in lanes where `borrow_mask` is set. We performed the final implementation exactly.

In contrast to addition, subtraction does not need to pass on the borrow-out information, as we are always assuming X is greater than or equal to Y.

Instead of using the $\_mm512\_add\_epi64$ in PML Sub, we used the $\_mm512\_sub\_epi64$ flag from AVX512-F, as compared to PML Add, for compilation.s

The implementation of the PLM Sub follows a programming approach similar to that of addition. It utilizes a macro-based vectorized kernel, specifically designed for an AVX-512 type, with operands grouped into equal-sized multiples of 8 64-bit limbs. For the 256-bit subtraction, we have employed AVX2. It is important to note that this implementation calculates the absolute difference between x and y, assuming that x is greater than y. If x is less than y, we swap the pointers for x and y. In addition to the result, we indicate that the difference is negative using the *limb_t* structure.

4.3 Evaluation

Similar to Addition, the core questions we are trying to answer here are the following:

1. Are the computations yielding correct results?
2. Are we getting any performance improvements, in terms of execution time, instruction count, and CPU Cycles, over GMP?
3. How much performance gain do we get compared to GMP for subtraction with different data sizes?
4. How much performance gain are we observing with vectorization as compared to a non-vectorized baseline implementation?

For the vectorized C code implementation of subtraction (PML Sub), we use the same set of GCC compiler flags:

```
-mavx512f -mavx512vl -O1
```

Also, we compiled the baseline variant using the O1 flag. The correctness check and benchmarking were conducted on the same Intel Xeon E-2314 CPU with the same configuration as mentioned in chapter 3 for addition.

4.3.1 Correctness

To validate the correctness of our PML Sub implementation, we generated multiple sets of 100,000 random test cases using Python's random module and the gmpy2 library, similar to addition. These tests were conducted over a range of operand sizes, from 256 bits to 1,31,072 bits. We ensured the correctness of the outputs by comparing them against precomputed values using gmpy2 for each test case, confirming correctness across all scenarios. Additionally, for the subtraction operation, phase 4 of the algorithm was never executed for any test case. For thoroughness, we manually curated some test cases that would trigger this phase and found that the results matched the pre-computed values.

4.3.2 Performance Compared to GMP and Baseline

Similar to Addition, we opted to benchmark our performance primarily against the latest GMP version 6.3.0. In total, the performance comparison had a total of four models: PML Sub, PML Sub Worst (phase 4 triggering cases), Baseline Sub and GMP Sub. We measured and compared the execution time, throughput, ticks, user instruction count and L1D Cache miss rate among these four models.

Execution Time: We compared the execution time of our implementation (10) of PML Sub and PML Sub (Worst Case) with GMP Sub and the baseline implementation. Figure 4.1 shows the best measured execution times for various operand sizes. In comparison to GMP, for the random implementation, we are achieving a speed-up ranging from 1.92x to 2.97x for various data sizes across CPUs. We are observing the highest performance gains of up to 2.97x the performance for 256 bits and consistently exceeding 2x for most operands. In contrast, relative to baseline implementation, we achieve speed-ups ranging from 2.40x to 5.31x and are constantly above

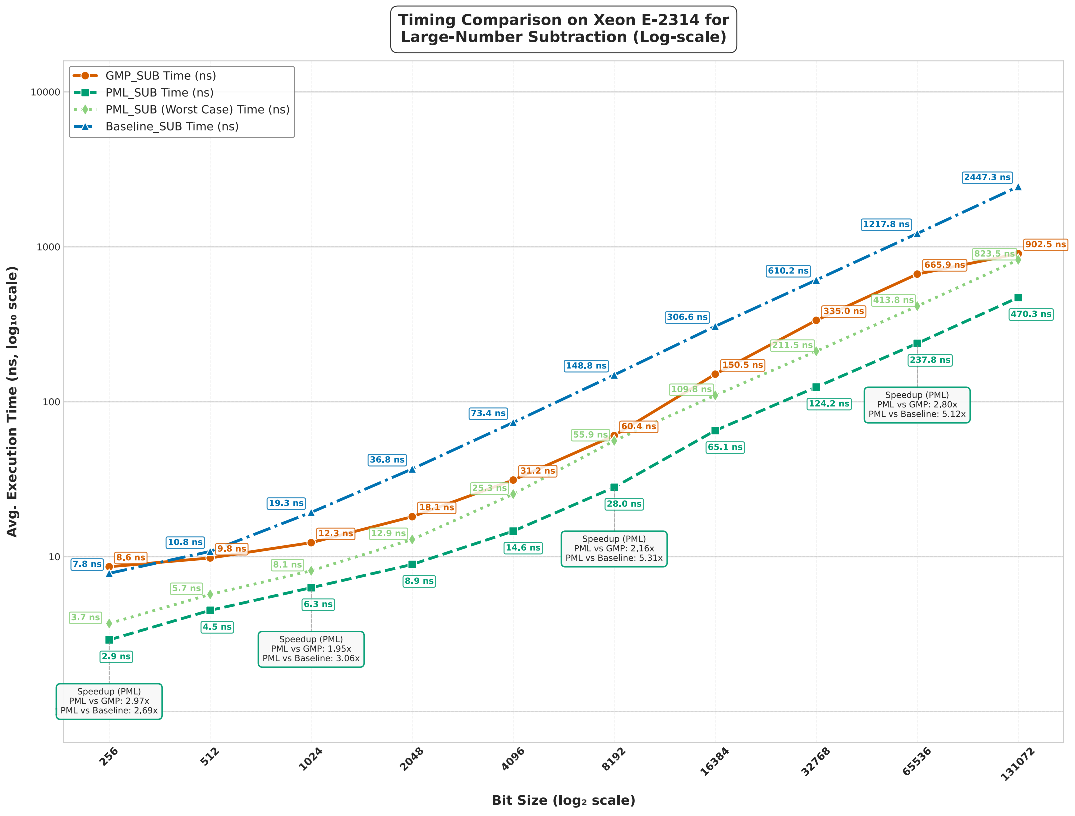

FIGURE 4.1: Timing comparison of PML Sub with GMP Sub for various data sizes on Intel Xeon E-2314

4x from 2048-bit operand sizes. With the Phase 4 test cases, PML Sub is still comparable to GMP, ranging between 1.08x and 2.32x. As the operand size increases, the performance becomes closer to GMP.

Performance Metrics: Compared to GMP Sub, we see almost a 45% reduction in user instruction counts for PML Sub for random test cases. And in the worst cases, we see nearly 15% reduction in user instruction counts. Similarly, ticks are nearly 64% less with PML Sub compared to GMP Sub for random cases and nearly 36% reduction for worst cases.

We have listed out the statistics of performance metrics for subtraction for all the models in Chapter 5; Table 5.4 shows the user instruction count comparison, and Table 5.2 shows the ticks comparison.

Throughput: Table 5.3 in Chapter 5 shows the throughput comparison on different operand sizes. Throughputs are in line with the execution time, similar to addition. Compared with GMP, we observe the highest throughput gain of 3x, and the throughput ranges above 2x for larger operands.

Chapter 5

Approximate Parallel Addition and Subtraction for Large Integers

5.1 Chained Carry Propagation

In most cases of adding two large numbers using PML Add (Algorithm 7 in Chapter 3), when we group a larger number of bits within each limb (using a large K), the entire computation can often be completed by Phase 3, making Phase 4 unnecessary. After initially adding all the limbs in Phase 1, further carry propagation should only occur if we add 1 to intermediate sums that have reached their maximum in Phase 3. We refer to this as *chained carry propagation*, as it may require additional adjustments depending on the maxed-out blocks. In Phase 4, we are simply propagating the carries that arise from adding 1 to the maxed-out limbs.

We now argue that, for most large-number arithmetic use cases, the occurrence of Phase 4, or chained carry propagation, is rare. This is primarily because large number additions are typically done with random numbers, making it uncommon for the sum of two limbs X_i and Y_i to equal $B^K - 1$ (the maximum base value), especially when K is sufficiently large.

5.1.1 Likelihood of Chained Carry Propagation

Let's illustrate the scenario potentially leading to chained carry propagation with an example.

Example 5.1.1.1.

```
Assume we are working with B=10, K=2.
Limb indices:     0   1   2
Number X:        45  23  87
Number Y:        11  76  56
---------------------------
Phase 1 (Initial Sum S = X + Y mod 100):
S:               56  99  43
Phase 2 (Calculate & Shift Carries from Phase 1):
Carries (C):      0   0   1   (from 56, 99, 143)
Shifted (C):      0   1   0   (carry into next limb)
Phase 3 (Add Shifted Carries S = S + C):
S:               56  99  43
C:          +     0   1   0
---------------------------
Result (S):      56  00  43
New Carries(C):   0   1   0
---------------------------
We propagate further carries in Phase 4.
```

We can observe that in order to trigger Phase 4, i.e., chained carry propagation, certain conditions would be needed: a limb S_p produces a carry after Phase 1 and its preceding limb S_{p-1} should be equal to $B^K - 1$, where K is the limb size and B is the base of the digits. Below proof examines how likely these conditions are to coincide.

Proof:

Let:

$$M = \{S_{p-1} = B^K - 1\}, N = \{S_p \geq B^K\}$$

where $S_i = X_i + Y_i$, and X_i, Y_i are independent, uniformly random variables over $[0, B^K - 1]$, with B^K possible values per limb.

Probability of M**:**
Consider $S_{p-1} = X_{p-1} + Y_{p-1} = B^K - 1$
Number of pairs: For $X_{p-1} = 0$ to $B^K - 1$, $Y_{p-1} = B^K - 1 - X_{p-1}$, giving B^K pairs.
Total pairs: $(B^K)^2$

$$\Pr[M] = \frac{B^K}{(B^K)^2} = \frac{1}{B^K}$$

Probability of N**:**
We compute $\Pr[N] = \Pr[S_p \geq B^K]$, where $S_p = X_p + Y_p$, with X_p and Y_p independent and uniformly distributed over $[0, B^K - 1]$.
Range of S_p: Since $X_p, Y_p \in \{0, 1, \ldots, B^K - 1\}$, the sum S_p ranges from 0 to $2B^K - 2$.
Total number of pairs: There are $(B^K)^2$ possible pairs (X_p, Y_p).
Favourable pairs: We need $X_p + Y_p \geq B^K$.
For each $X_p = x$:

$$Y_p \geq B^K - x \quad \text{and} \quad Y_p \leq B^K - 1$$

The number of Y_p satisfying this is:

$$\begin{cases} 0 & \text{if } x = 0, \\ x & \text{if } 1 \leq x \leq B^K - 1 \end{cases}$$

Thus, the total number of favourable pairs is:

$$\sum_{x=1}^{B^K-1} x = \frac{(B^K - 1)B^K}{2}$$

Probability calculation:

$$\Pr[N] = \frac{\frac{(B^K-1)B^K}{2}}{(B^K)^2} = \frac{B^K - 1}{2B^K}$$

Event $M \cap N$**:**
Sequential propagation (Phase 4) occurs when $M \cap N$ holds, a saturated S_{p-1} and a carry from limb S_p, potentially causing further carries in a carry-adjustment phase.
Since X_i, Y_i are independent across limbs, M and N are independent.

$$\Pr[M \cap N] = \Pr[M] \cdot \Pr[N] = \left(\frac{1}{B^K}\right)\left(\frac{B^K - 1}{2B^K}\right) = \frac{B^K - 1}{2(B^K)^2}$$

Conditional Probability:

$$\Pr[M \mid N] = \frac{\Pr[M \cap N]}{\Pr[N]} = \frac{\frac{B^K-1}{2(B^K)^2}}{\frac{B^K-1}{2B^K}} = \frac{1}{B^K}$$

> Thus, given a carry from S_p, the chance $S_{p-1} = B^K - 1$ remains $\frac{1}{B^K}$, a event that decreases exponentially with K, making sequential propagation unlikely in cryptographic applications with large K and randomly generated numbers. Hence, the probability of not needing sequential propagation for the number patterns is approximately 0.99999999999999999994578989137572 if we group 64 bits or 16 hexadecimal digits together ($1/2^{64}$ or $1/16^{16}$ = 5.42e-20).

It's important to note that this rare chained carry propagation scenario we observed during addition also applies to subtraction with sufficiently large K. Instead of encountering a rare situation where a limb is maxed out, we may have a rare occurrence of a limb being zero after the Phase 1 subtraction. Consequently, the likelihood of *Chained Borrow Generation* happening is also $1/B^K$. As a result, triggering Phase 4 of the PML Sub is quite rare as well.

Based on this proof, we can safely say only with $1/2^{64}$ probability that Phase 4 gets triggered for the PML Add and PML Sub. In other words, when working with 64-bit saturated bases for PML Add and PML Sub, there is merely a 5.42×10^{-18} % chance that the current limb addition or subtraction generating a carry or borrow, along with its preceding limb operation will be maxed out or result in zero.

However, the conditional check to enter the Phase 4 code block is executed for all operands in PML Add and PML Sub to ensure correctness across all cases. Consequently, if certain applications like machine learning, image processing, scientific computing and computer vision can operate with approximate calculations ([Jia+20; AKL18]), they may entirely skip Phase 4 when large-number addition and subtraction are required. By doing so, they can avoid executing two key computations for every 512-bit data chunk: checking whether a carry or borrow has been generated after Phase 3 and determining whether to enter Phase 4. As a result, for use cases where approximation is acceptable, performance can further improve with larger operands.

We will approach this by implementing a three-phase operation:

- **Phase 1**: The limbs of both operands will be added or subtracted in parallel without considering any potential carries or borrows.
- **Phase 2**: Carries or borrows will be detected simultaneously across all limbs.
- **Phase 3**: If any intermediate limb results in a carry or borrow, it will be added to or subtracted from its preceding limb in parallel, simulating the necessary adjustments for the carry or borrow.

Note that, although we are approximating here, for a given a and b, we are not modifying or transforming the original a and b. Their computation with this approach will always yield the same result; thus, this is deterministic. With this approach, we may rarely arrive at a computation where some resulting limb will have a difference of value one from the intended correct value and in even rarer cases, like fully chained propagation across all the limbs, the difference one will be in terms of wrap-around.

Example 5.1.1.2.
Long Chained Carry Propagation: Consider the following worst-case scenario: In this case, most of the blocks are maxed out after Phase 3, leading to carry propagation from the least significant limb to the most significant limb.

- Let $a = 999\dots9999$
- Let $b = 000\dots0001$

Suppose we group $k = 2$ digits together. We can represent the numbers as:

$$a' = \underbrace{99\,99\,\dots\,99\,99\,99}_{\text{Grouped representation}}$$

$$b' = \underbrace{00\,00\,\dots\,00\,00\,01}_{\text{Grouped representation}}$$

The Phase 1 addition yields:

$$\text{Partial Sum} = 99, 99, 99, \dots, 99, 99, 100$$

Phase 2 detects the below carry values:

$$\text{Carries} = 0, 0, 0, \dots, 0, 0, 1$$

We will add up the carries to the preceding limbs (Phase 3):

$$\text{Intermediate Sum} = 99, 99, 99, \dots, 99, 00, 00$$

$$\text{Carries} = 0, 0, 0, \dots, 0, 1, 0$$

After Phase 3, we begin to detect carries again. In this situation, we need to propagate the carry in a chained manner repeatedly until no more carries are generated. However, as we have observed regarding the likelihood of chained propagation, when K is sufficiently large, the probability of encountering maxed-out limbs becomes rare. As a result, this type of long-chained carry propagation is much rarer in large-number addition use cases. Nevertheless, in such cases, Phase 4 in the original PML Add will still manage to perform the remaining carry propagation with some amount of performance reduction.

Next, we describe the Parallel Multi-phase Approximate Addition and Subtraction algorithms alongside their implementation with AVX-512, followed by the observations on performance.

We omitted the carry checking after phase 3, followed by the phase 4 adjustment block from the original proposed PML Add algorithm 7. Algorithm 11 depicts our approximation approach for large number addition. Similarly, the implementation of the PML Add (Approx) (Algorithm 12) omits the carry check after phase 3 and phase 4 adjustment of the original implementation of PML Add (implementation 8). Thus, the implementation successfully eliminates most of the operations on the masked registers.

5.2 Parallel Multi-phase Approximate Addition

Algorithm 11: Proposed Parallel Multi-Stage Large-Number Approximate Addition (PML Add (Approx)) Algorithm

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$ digits each,
         where $X_i$, $Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least significant blocks.
         Also, $0 \leq X_i, Y_i \leq B^K - 1$.
Output: Sum $S = S_{p-1} \dots S_0$, where $S_0$ is the least significant block.
// Phase 1:  Compute Intermediate Sums (Parallel)
for $i \leftarrow 0$ to $m - 1$ in parallel do
    $S_i \leftarrow X_i + Y_i$
// Phase 2:  Detect Carries (Parallel)
$carry\_detect \leftarrow$ false;                          // Flag for carry presence
for $i \leftarrow 0$ to $m - 1$ in parallel do
    if $S_i >= B^K$ then
        $C_i \leftarrow 1$
        $S_i \leftarrow S_i - B^K$
        $carry\_detect \leftarrow$ true;                   // Mark carry existence
$C'_{i+1} = C_i, \quad \forall\, 0 \leq i < m - 1, C'_0 = 0$
// Phase 3:  First Round of Carry Addition (Parallel)
if $carry\_detect = true$ then
    for $i \leftarrow 0$ to $m - 1$ in parallel do
        if $C'_i = 1$ then
            $S_i \leftarrow S_i + 1$;                      // Add carry, modulo $2^{32}$
Return $S$;
```

Algorithm 12: Outline of Implementation of PML Add (Approx) using AVX512

```
Data: Arrays x and y of 64-bit elements, output array s, size n (multiple of 8)
Result: $s = x + y$
c_in ← 0, c_out ← 0 ;                                     // Initialize carry variables
for $i \leftarrow 0$ to n - 8 step 8 do
    c_in ← c_out ;                                        // Propagate carry from previous round
    // Phase 1:  Parallel Addition
    x_vec ← Load eight 64-bit elements from x[i] ;
    y_vec ← Load eight 64-bit elements from y[i] ;
    r_vec ← x_vec + y_vec ;                               // Add eight elements in parallel
    // Phase 2:  Carry Detection
    carry_mask ← Generate 8-bit mask where r_vec < x_vec ; // Detect carries
    c_out ← carry_mask » 7 ;                              // Extract highest carry
    carry_mask ← carry_mask « 1 ;                         // Shift left
    carry_mask ← carry_mask | c_in ;                      // Combine with input carry
    // Phase 3:  Adjust Carries
    carry_vec ← Set 1s in lanes based on carry_mask ;     // Create carry vector
    r_vec_new ← r_vec + carry_vec ;                       // Apply carries
    r_vec ← r_vec_new ;
    // Store Results
    Store eight 64-bit elements from r_vec to s[i] ;
result.carry ← c_out ;                                    // Store final carry
```

5.3 Parallel Multi-phase Approximate Subtraction

Similar to PML Add (Approx), we omitted the borrow checking after phase 3, followed by the phase 4 adjustment block from the original proposed PML Sub algorithm 9. Our approximation approach for large number subtraction is described in algorithm 13.

Implementation outline 14 shows our implementation approach for PML Sub (Approx) (Algorithm 13) using AVX512 intrinsics.

Algorithm 13: Proposed Parallel Multi-Stage Large-Number Approximate Subtraction (PML Sub (Approx))

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base $B$, split into $m$ blocks of $K$
         digits each, where $X_i, Y_i$ are the $i$-th blocks and $X_0, Y_0$ are the least
         significant blocks. Also, $0 \leq X_i, Y_i \leq B^K - 1$ and $X > Y$.
Output: Sub $S = S_{m-1} \dots S_0$, where $S_0$ is the least significant block.

// Phase 1: Compute Intermediate Subs (Parallel)
for $i \leftarrow 0$ to $m-1$ in parallel do
    $S_i \leftarrow X_i - Y_i$
// Phase 2: Detect Borrows (Parallel)
$borrow\_detect \leftarrow$ false;                        // Flag for borrow presence
for $i \leftarrow 0$ to $m-1$ in parallel do
    if $S_i < 0$ then
        $B_i \leftarrow 1$
        $S_i \leftarrow S_i - B^K$
        $borrow\_detect \leftarrow$ true;                 // Mark borrow existence

$B'_{i+1} = B_i, \quad \forall 0 \leq i < m-1, B'_0 = 0$
// Phase 3: First Round of Borrow Adjust (Parallel)
if $borrow\_detect = true$ then
    $borrow\_detect \leftarrow false$
    for $i \leftarrow 0$ to $m-1$ in parallel do
        if $B'_i = 1$ then
            $S_i \leftarrow S_i - 1$
return $S$;
```

Algorithm 14: Outline of Implementation of PML Sub (Approx) using AVX512

```
Data: Arrays x and y of 64-bit elements, output array s, size n (multiple of 8) and
      $x > y$
Result: $s = x - y$
b_in ← 0, b_out ← 0 ;                                    // Initialize borrow variables
for $i \leftarrow n$ - 8 downto 0 step 8 do
    b_in ← b_out ;                                // Propagate borrow from previous round
    // Phase 1: Parallel Subtraction
    x_vec ← Load eight 64-bit elements from x[i] ;
    y_vec ← Load eight 64-bit elements from y[i] ;
    r_vec ← x_vec - y_vec ;                          // Subtract eight elements in parallel
    // Phase 2: Borrow Detection
    borrow_mask ← Generate 8-bit mask where y_vec > x_vec ;// Detect borrows
    b_out ← borrow_mask ;                                    // Keep borrow for next round
    borrow_mask ← borrow_mask » 1 ;                    // Shift right (little-endian order)
    borrow_mask ← borrow_mask | b_in « 7 ;       // Account for borrow-in to most
     significant lane
    // Phase 3: Adjust Borrows
    borrow_vec ← Set 1s in lanes based on borrow_mask ;
    r_vec_new ← r_vec - borrow_vec ;                                // Adjust for borrows
    r_vec ← r_vec_new ;
    // Store Results
    Store eight 64-bit elements from r_vec to s[i] ;
```

5.4 Evaluation

For the approximate versions of addition and subtraction, the core questions we are trying to answer here are the following:

1. Are the computations yielding correct results for most cases?
2. Are we getting any performance improvements, in terms of execution time, instruction count, and CPU Cycles, over the regular variant of PML?
3. How much performance gain do we get compared to GMP and regular PML with different data sizes?

For the vectorized C code implementation, we use the same set of GCC compiler flags:

```
-mavx512f -mavx512vl -O1
```

Also, we compiled the baseline variant using the O1 flag.

The correctness check and benchmarking were conducted on the same Intel Xeon E-2314 CPU with the same configuration as mentioned in chapter 3 for addition.

5.4.1 Correctness

Luckily, the implementation of PML Add (Approx) and PML Sub (Approx) passed all the lakhs-and-lakhs of the test cases we generated for the original PML Add and PML Sub implementation, possibly due to the rarity of occurring test cases that generate chained dependency. However, when such cases may arise, our computation will hold near computations, a difference of value one, from the intended ones.

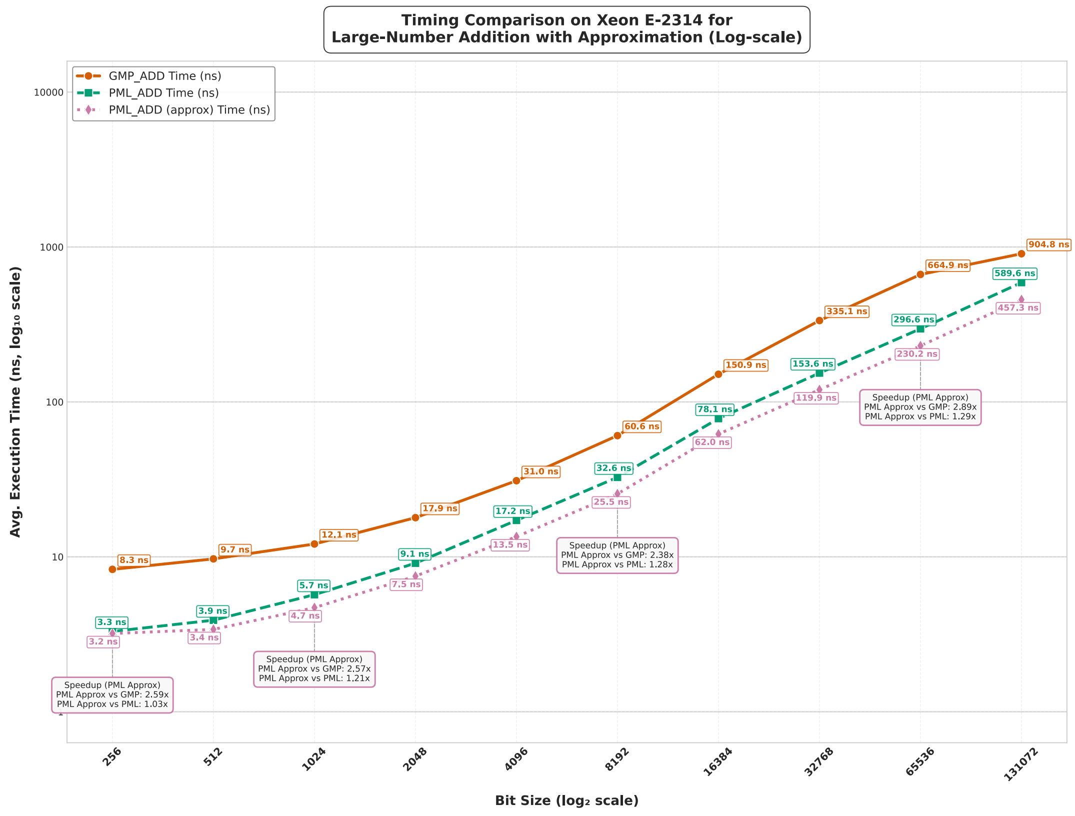

FIGURE 5.1: Timing comparison of PML Add (Approx) with GMP and PML Add for various data sizes on Intel Xeon E-2314

5.4.2 Performance Compared to GMP and PML Add

Below, we compare the performance metrics of the approximate versions with those of the GMP and the original versions.

Execution Time: In terms of execution time, with approximate additions, we get an average of 2.51x speed-up compared to GMP and a 1.23x speed-up compared to the original addition implementation. To put it in contrast, the original addition is 2.06x faster on average than the GMP. Figure 5.1 shows the best execution time comparison of the three models. On the other hand, the execution time speed-up for the approximate version of subtraction, as demonstrated in Figure 5.2, is 2.80x as compared to GMP and 1.21x faster than the original version of subtraction. We also notice that the speed-ups for the approximate versions are higher for larger operands, as we execute fewer carry or borrow checks for every 512-bit of data we process.

Performance Metrics: We have also listed the performance metric comparison regarding user instruction count and ticks. The ticks for the original addition implementation are near 335, whereas the approximate version consumes 266 ticks on average for operand sizes ranging between 256 bits and 131072 bits — a reduction of 24%. In terms of user instruction count, we observe an average 18% reduction for the same set of operand sizes between the approximate version (1140 user instructions) and the original version of addition (934 user instructions). Table 5.4 and 5.2 show the comparison of these performance metrics for addition across all the models. On

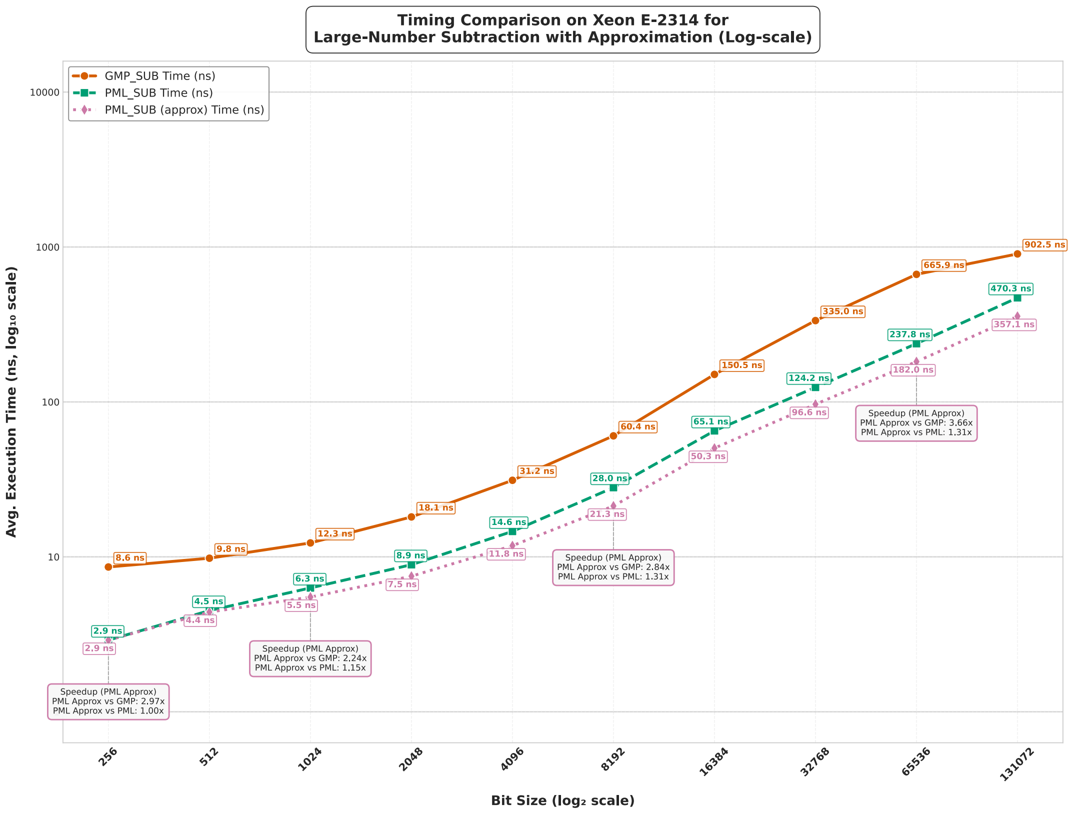

FIGURE 5.2: Timing comparison of PML Sub (Approx) with GMP and PML Sub for various data sizes on Intel Xeon E-2314

the other hand, for subtraction, we observe an average of 20% reduction for ticks and 28% for user instruction counts with the approximate version compared to the original PML Sub implementation (refer Tables: 5.4 and 5.8).

Throughput: Similarly, we have listed out the throughput comparison of the three models for addition and subtraction in Tables 5.3 and 5.6.

Table 5.1 presents a summary of statistics for all the models compared regarding addition across various operand sizes.

Below we have listed tables that precisely measure the respective values using the methods specified in Appendix A. We conducted ten individual measurements for each operand size across each model and have presented the statistics, including the minimum, maximum, average, and standard deviation for each value listed.

TABLE 5.1: Summarized Execution Time Statistics of Addition by Operand Size Across Models (in ns)

Size	Baseline				GMP				PML Add (Worst)				PML Add				PML Add (Approx)			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	9.0	12.7	10.6	1.24	8.3	8.6	8.5	0.14	4.7	4.7	4.7	0.00	3.3	3.3	3.3	0.00	3.2	3.2	3.2	0.00
512	11.1	12.7	12.2	0.77	9.7	9.8	9.8	0.04	4.6	6.3	4.9	0.59	3.9	3.9	3.9	0.00	3.4	3.4	3.4	0.00
1024	21.0	23.0	21.8	0.91	12.1	13.8	12.4	0.50	7.5	9.7	7.9	0.71	5.7	5.7	5.7	0.00	4.7	4.7	4.7	0.00
2048	46.8	49.2	47.5	1.02	17.9	18.1	18.0	0.07	14.0	15.1	14.1	0.34	9.1	12.2	10.0	1.06	7.5	7.7	7.5	0.06
4096	104.1	106.3	105.2	1.15	31.0	32.2	31.3	0.38	27.2	28.2	27.3	0.31	17.2	19.0	17.7	0.66	13.5	13.9	13.6	0.12
8192	218.7	221.0	219.6	1.16	60.6	61.0	60.8	0.15	58.8	61.5	59.6	0.73	32.6	33.1	32.7	0.15	25.5	26.1	25.8	0.23
16 384	454.9	457.8	456.1	1.35	150.9	151.2	151.0	0.09	125.6	126.5	125.9	0.24	78.1	78.9	78.3	0.28	62.0	62.4	62.2	0.13
32 768	913.9	922.4	916.2	2.44	335.1	672.7	369.3	106.61	244.2	429.5	263.0	58.50	153.6	154.3	153.8	0.21	119.9	120.3	120.1	0.11
65 536	1832.8	1838.7	1836.5	1.86	664.9	1005.4	700.1	107.27	478.6	480.8	479.8	0.69	296.6	297.8	297.4	0.34	230.2	230.7	230.4	0.13
131 072	3680.8	3696.4	3687.2	4.88	904.8	921.9	914.0	5.39	955.2	1153.7	996.7	81.89	589.6	594.3	591.7	1.61	457.3	459.1	457.9	0.56

TABLE 5.2: Summarized Ticks Statistics of Addition by Operand Size Across Models

Size	Baseline				GMP				PML Add (Worst)				PML Add				PML Add (Approx)			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	25	31	29	2.64	23	24	24	0.42	13	16	14	1.02	10	10	10	0.01	9	9	9	0.00
512	31	36	34	2.48	27	28	28	0.09	13	13	13	0.03	11	12	11	0.42	10	11	10	0.30
1024	59	70	62	4.15	34	35	35	0.33	22	22	22	0.04	16	23	18	2.84	13	15	13	0.51
2048	132	145	136	4.77	50	52	51	0.49	40	43	40	1.06	26	35	27	2.87	21	22	22	0.34
4096	293	300	296	3.38	87	89	88	0.47	77	79	77	0.80	48	52	49	1.11	38	38	38	0.08
8192	615	627	619	3.88	171	172	171	0.36	168	193	173	9.11	92	96	93	1.71	72	74	72	0.85
16 384	1281	1295	1284	4.94	425	429	426	1.33	354	357	355	1.18	220	222	221	0.66	172	176	174	1.35
32 768	2573	2862	2620	93.58	943	947	945	1.29	687	691	688	1.44	433	434	433	0.24	338	342	339	1.21
65 536	5160	5408	5196	75.40	1872	1881	1876	2.83	1347	1355	1351	2.74	836	841	838	1.29	648	650	649	0.54
131 072	10 361	10 416	10 382	16.86	3781	3804	3794	8.02	2682	2718	2698	11.48	1658	1678	1669	6.63	1287	1293	1289	1.68

TABLE 5.3: Summarized Throughput Statistics of Addition by Operand Size Across Models (in million OP/s)

Size	Baseline				GMP				PML Add (Worst)				PML Add				PML Add Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	78.90	110.80	95.71	11.21	116.05	120.04	117.63	2.04	213.20	213.31	213.25	0.04	253.95	253.97	253.96	0.01	253.95	253.99	253.97	0.01
512	77.10	90.46	81.13	6.42	101.95	102.68	102.25	0.33	157.93	206.55	198.61	16.96	232.77	232.80	232.79	0.01	285.67	285.87	285.77	0.06
1024	43.35	47.49	45.84	1.96	72.16	82.23	80.61	3.06	102.05	121.53	119.05	6.19	174.46	174.55	174.53	0.02	209.21	213.25	212.83	1.27
2048	20.30	21.30	20.99	0.44	55.15	55.73	55.50	0.20	66.08	70.91	70.42	1.53	81.66	108.66	100.22	9.78	130.40	132.73	132.48	0.73
4096	9.39	9.60	9.50	0.11	31.19	32.21	32.01	0.34	35.24	36.57	36.42	0.42	52.19	58.05	56.57	2.16	72.11	74.56	74.15	0.76
8192	4.52	4.57	4.55	0.02	16.41	16.48	16.45	0.03	16.24	16.69	16.61	0.13	30.16	30.60	30.55	0.14	38.61	39.45	39.08	0.36
16 384	2.18	2.20	2.19	0.01	6.62	6.62	6.62	0.00	7.87	7.93	7.93	0.02	12.65	12.79	12.75	0.05	16.00	16.23	16.16	0.07
32 768	1.08	1.09	1.09	0.00	1.49	2.98	2.83	0.47	2.33	4.09	3.91	0.56	6.47	6.50	6.49	0.01	8.29	8.31	8.30	0.00
65 536	0.54	0.54	0.54	0.00	1.00	1.50	1.45	0.16	2.08	2.09	2.08	0.00	3.35	3.36	3.36	0.00	4.33	4.33	4.33	0.00
131 072	0.27	0.27	0.27	0.00	1.09	1.10	1.09	0.01	0.87	1.05	1.01	0.08	1.68	1.69	1.69	0.00	2.18	2.19	2.18	0.00

TABLE 5.4: Summarized User Instructions Statistics of Addition by Operand Size Across Models

Size	Baseline				GMP				PML Add (Worst)				PML Add				PML Add Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	96	123	115	13.04	113	113	113	0.00	50	50	50	0.00	34	34	34	0.00	25	25	25	0.00
512	134	161	148	14.23	130	130	130	0.00	56	56	56	0.00	48	48	48	0.00	41	41	41	0.00
1024	237	264	248	13.94	164	164	164	0.00	90	90	90	0.00	70	70	70	0.00	59	59	59	0.00
2048	443	470	457	14.23	232	232	232	0.00	158	158	158	0.00	114	114	114	0.00	95	95	95	0.00
4096	855	882	866	13.94	368	368	368	0.00	294	294	294	0.00	202	202	202	0.00	167	167	167	0.00
8192	1679	1706	1690	13.94	640	640	640	0.00	566	566	566	0.00	378	378	378	0.00	311	311	311	0.00
16 384	3327	3354	3335	13.04	1184	1184	1184	0.00	1110	1110	1110	0.00	730	730	730	0.00	599	599	599	0.00
32 768	6623	6650	6639	13.94	2272	2272	2272	0.00	2198	2198	2198	0.00	1434	1434	1434	0.00	1175	1175	1175	0.00
65 536	13 215	13 242	13 226	13.94	4448	4448	4448	0.00	4374	4374	4374	0.00	2842	2842	2842	0.00	2327	2327	2327	0.00
131 072	26 399	26 426	26 410	13.94	8800	8800	8800	0.00	8726	8726	8726	0.00	5658	5658	5658	0.00	4631	4631	4631	0.00

TABLE 5.5: Summarized Execution Time Statistics of Subtraction by Operand Size Across Models (in ns)

Size	Baseline				GMP				PML Sub (Worst)				PML Sub				PML Sub Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	7.8	7.8	7.8	0.00	8.6	8.7	8.6	0.05	3.7	4.7	4.3	0.52	2.9	3.6	3.1	0.36	2.9	3.9	3.5	0.52
512	10.8	11.2	10.9	0.19	9.8	11.0	9.9	0.36	5.7	6.6	6.2	0.36	4.5	5.8	5.2	0.67	4.4	5.4	5.1	0.45
1024	19.3	20.2	19.5	0.26	12.3	12.5	12.4	0.08	8.1	9.0	8.5	0.42	6.3	8.1	7.3	0.92	5.5	6.5	5.8	0.47
2048	36.8	37.5	36.9	0.24	18.1	18.4	18.1	0.09	12.9	14.2	13.2	0.39	8.9	10.0	9.5	0.53	7.5	8.4	8.0	0.29
4096	73.4	73.8	73.6	0.18	31.2	32.1	31.4	0.36	25.3	26.7	25.5	0.41	14.6	16.5	15.2	0.75	11.8	12.4	12.0	0.30
8192	148.8	149.9	149.2	0.26	60.4	61.0	60.6	0.18	55.9	59.2	56.8	0.96	28.0	29.2	28.3	0.49	21.3	23.3	21.6	0.72
16 384	306.6	307.8	307.2	0.43	150.5	151.3	150.8	0.25	109.8	112.1	110.1	0.69	65.1	67.8	66.3	0.89	50.3	52.2	51.0	0.68
32 768	610.2	612.0	611.1	0.65	335.0	336.2	335.3	0.42	211.5	221.3	213.2	3.02	124.2	126.5	125.8	0.83	96.6	684.8	156.3	185.6
65 536	1217.8	1222.1	1219.7	1.54	665.9	667.2	666.6	0.46	413.8	423.0	417.7	4.11	237.8	240.3	238.5	1.10	182.0	184.0	182.3	0.60
131 072	2447.3	2459.3	2455.5	4.07	902.5	913.0	907.8	3.87	823.5	843.4	832.8	8.98	470.3	475.8	471.5	1.57	357.1	361.3	358.8	1.21

TABLE 5.6: Summarized Throughput Statistics of Subtraction by Operand Size Across Models (in million OP/s)

Size	Baseline				GMP				PML Sub (Worst)				PML Sub				PML Sub Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	127.92	128.67	128.44	0.36	115.07	116.11	115.63	0.42	199.54	253.46	221.09	27.80	253.95	348.44	310.64	48.77	232.80	310.36	263.82	40.05
512	89.15	91.62	90.87	1.18	91.08	102.41	100.52	3.37	144.73	167.66	151.75	10.91	172.11	212.88	188.62	20.83	164.28	199.33	174.82	16.91
1024	49.25	51.77	50.99	0.74	80.20	80.95	80.50	0.35	104.00	116.83	110.49	6.67	126.02	148.60	135.53	11.15	139.67	164.32	154.45	12.72
2048	26.66	27.07	27.00	0.14	54.34	55.30	55.03	0.29	70.16	76.44	74.83	1.98	99.70	110.24	104.20	5.18	115.20	130.74	118.36	6.51
4096	13.53	13.59	13.55	0.03	31.18	32.06	31.81	0.35	37.10	39.41	39.12	0.72	58.59	67.68	65.40	3.08	75.47	84.17	80.83	3.67
8192	6.66	6.71	6.69	0.01	16.43	16.52	16.48	0.03	16.83	17.64	17.47	0.25	34.10	35.70	35.15	0.59	42.74	46.85	45.99	1.41
16 384	3.25	3.26	3.25	0.00	6.61	6.64	6.63	0.01	8.90	9.09	9.05	0.05	14.78	15.28	15.08	0.17	19.23	19.70	19.55	0.13
32 768	1.63	1.64	1.64	0.00	2.97	2.98	2.98	0.00	4.51	4.72	4.68	0.06	7.90	8.02	7.93	0.05	1.46	10.30	9.32	2.76
65 536	0.82	0.82	0.82	0.00	1.50	1.50	1.50	0.00	2.37	2.42	2.39	0.02	4.16	4.20	4.19	0.02	5.42	5.47	5.47	0.02
131 072	0.41	0.41	0.41	0.00	1.10	1.11	1.10	0.00	1.18	1.21	1.20	0.01	2.10	2.12	2.12	0.01	2.77	2.79	2.78	0.01

TABLE 5.7: Summarized Ticks Statistics of Subtraction by Operand Size Across Models

Size	Baseline				GMP				PML Sub (Worst)				PML Sub				PML Sub Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	22	22	22	0.03	24	24	24	0.08	10	13	11	1.29	8	10	9	1.05	8	11	10	1.41
512	30	32	31	0.61	27	28	28	0.19	16	18	17	0.94	13	18	15	2.11	13	15	13	0.93
1024	54	59	56	1.40	35	37	35	0.76	23	25	24	1.16	19	23	21	1.68	15	17	16	1.07
2048	104	105	104	0.47	51	51	51	0.11	37	38	37	0.49	25	28	26	1.17	21	24	23	0.95
4096	207	208	207	0.52	88	91	88	1.12	71	80	73	2.52	41	45	43	1.53	33	39	35	1.86
8192	419	420	420	0.56	170	171	171	0.33	158	162	159	1.62	79	85	81	2.72	60	65	60	1.50
16 384	863	866	864	1.15	424	428	425	1.44	309	328	312	5.73	185	189	187	1.76	139	144	142	1.45
32 768	1717	1723	1720	1.79	944	1937	1044	313.68	596	670	614	22.56	350	1062	424	224.34	271	276	273	1.83
65 536	3426	3449	3436	7.87	1872	2017	1890	44.68	1164	1190	1180	10.28	669	678	674	3.30	512	548	518	10.81
131 072	6873	6977	6913	28.26	3786	10 095	4469	1979.32	2316	2371	2343	23.33	1325	1475	1343	46.68	1007	1012	1009	1.97

TABLE 5.8: Summarized User Instructions Statistics of Subtraction by Operand Size Across Models

Size	Baseline				GMP				PML Sub (Worst)				PML Sub				PML Sub Approx			
	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std	Min	Max	Avg	Std
256	88	95	92	3.61	117	117	117	0.00	42	49	44	3.38	35	42	39	3.69	31	38	34	3.61
512	134	141	138	3.61	134	134	134	0.00	66	73	69	3.61	55	62	59	3.61	50	57	53	3.61
1024	239	246	245	2.21	168	168	168	0.00	96	103	99	3.61	74	81	77	3.61	65	72	69	3.61
2048	449	456	453	3.61	236	236	236	0.00	156	163	160	3.69	112	119	113	2.95	95	102	100	3.38
4096	869	876	874	3.38	372	372	372	0.00	276	283	280	3.69	188	195	191	3.61	155	162	159	3.61
8192	1709	1716	1713	3.69	644	644	644	0.00	516	523	520	3.69	340	347	344	3.61	275	282	278	3.61
16 384	3389	3396	3392	3.61	1188	1188	1188	0.00	996	1003	1000	3.69	644	651	648	3.61	515	522	519	3.61
32 768	6749	6756	6753	3.61	2276	2276	2276	0.00	1956	1963	1960	3.69	1252	1259	1256	3.69	995	1002	998	3.61
65 536	13 469	13 476	13 473	3.61	4452	4452	4452	0.00	3876	3883	3878	3.38	2468	2475	2473	3.38	1955	1962	1958	3.61
131 072	26 909	26 916	26 913	3.69	8804	8804	8804	0.00	7716	7723	7720	3.69	4900	4907	4903	3.61	3875	3882	3879	3.69

Chapter 6

Accelerating the Multiplication of Large Integers

Moving on, this chapter presents our multiplication approach for accelerating the multiplication of large integers. We first discuss our proposed hybrid algorithm for multiplying variable-sized large operands, followed by an algorithm for a fixed-size base-case Vedic-based approach for multiplication to be integrated into the hybrid approach. The subsequent discussion focuses on the implementation of this base-case Vedic method, including the challenges faced and less effective strategies. Finally, the chapter concludes with an evaluation of the resulting implementation.

6.1 Parallel Hybrid-Multiplication

Parallelizing multiplication is tough for large numbers. As the size of the operands increases, the number of multiplications increases exponentially $O(n^2)$ with a grade-school or Urdhva-like approach, unlike the addition or subtraction, which linearly increases with the operand size. For that reason, we typically need to utilize divide-and-conquer-based strategies like Karatsuba and Toom-Cook or FFT for larger numbers, which minimizes the number of multiplications to a large extent (see Table 6.1). However, for a smaller number of limbs, even though asymptotically slower, Grade-school or Urdhva-Tiryagbhyam can be implemented faster than the others.

For this work, we propose a hybrid approach that is similar to some existing works ([ET20; ET23]). However, instead of using the Grade-School as the base-case multiplication, we utilize the Vedic approach, Urdhva-Tiryagbhyam. Since the Vedic approach inherently breaks the operands into smaller individual digit sets, similar to decomposing them into parallel sub-tasks, it reduces the number of additions (refer to Table 2.2) for settling partial-products. Larger numbers can be recursively split into two parts until they reach a certain threshold, after which we apply the base-case parallel Urdhva-Tiryagbhyam kernel.

TABLE 6.1: Asymptotic Comparison of Number of Multiplications

Limbs (n)	Grade/Urdhva ($O(n^2)$)	Toom-2 ($O(n^{1.585})$)	Toom-3 ($O(n^{1.465})$)	Toom-4 ($O(n^{1.404})$)
4	16	9 (1.78×)	8 (2.00×)	7 (2.29×)
8	64	27 (2.37×)	21 (3.05×)	19 (3.37×)
16	256	81 (3.16×)	58 (4.41×)	49 (5.22×)
32	1024	243 (4.21×)	160 (6.40×)	130 (7.88×)
64	4096	729 (5.62×)	443 (9.25×)	343 (11.94×)
128	16384	2187 (7.49×)	1222 (13.41×)	909 (18.02×)
256	65536	6562 (9.99×)	3373 (19.43×)	2405 (27.25×)
512	262144	19688 (13.31×)	9313 (28.15×)	6365 (41.19×)
1024	1048576	59064 (17.75×)	25709 (40.79×)	16845 (62.25×)

6.1.1 Base-case Parallel Urdhva-Tiryagbhyam

To parallelize multiplication with the Urdhva-Tiryagbhyam method, we propose an approach named Vedic-Based Approach for Multiplication (VBAP Mul) that leverages contiguous memory and concurrent operations for the multiplications. The multiplicands and multipliers (in order with vertically and cross-wise) are organized into two arrays, enabling parallel computation of intermediate products, followed by set-wise summation and carry-over of higher portions (suddhikaran).

Note that the original Urdhva-Tiryagbhyam multiplication technique [MAH92] is designed for single digits and operates in base 10. However, we can apply this technique to any base B (as demonstrated in algorithm 15). In the implementation, we can use the base B^k and adjust the limb size accordingly, without affecting the overall result.

Below, we have shown our overall approach for computing multiplication using the Vedic technique.

Approach

1. **Operand Preparation**: Store all multiplicands and multipliers in two contiguous arrays, M_x and M_y, respectively. The elements are ordered to align with the multiplication sets defined in Step 3 of our Urdhva-Tiryagbhyam algorithm (see Algorithm 1).

2. **Parallel Multiplication**: Compute intermediate products concurrently using $M[i] = M_x[i] \times M_y[i]$ for all i.

3. **Set-Wise Addition**: Group the intermediate products by their corresponding sets and sum them.

4. **Carry Propagation**: Propagate carry-overs from the least significant set to the most significant set, adjusting the results accordingly.

An example of the approach is shown below.

Example 6.1.1.1.
Consider the multiplication: 843×384
We will do the following steps:
Step 1: Define Digit Sets

- For 843: $\{8, 84, 843, 43, 3\}$
- For 384: $\{3, 38, 384, 84, 4\}$

Step 2: Parallel Multiplication To enable parallelism, we define two arrays:

- $M_x = [8, 8, 4, 8, 4, 3, 4, 3, 3]$
- $M_y = [3, 8, 3, 4, 8, 3, 4, 8, 4]$

Each pair $(M_x[i], M_y[i])$ corresponds to a term in the cross-products. Compute the intermediate products in parallel:

$$M = [24, 64, 12, 32, 32, 9, 16, 24, 12]$$

Thus, most of the heavy computation (multiplications) can be done in parallel.
Step 3: Group and Sum by Sets Group the elements of M according to their sets and compute the sums:

$$M = [24, \underbrace{64, 12}_{\text{sum}=76}, \underbrace{32, 32, 9}_{\text{sum}=73}, \underbrace{16, 24}_{\text{sum}=40}, 12]$$

Resulting in:

$$M = [24, 76, 73, 40, 12]$$

Performing the sums within the sets is challenging to parallelize because it resembles horizontal reduction, which is more difficult to parallelize for smaller numbers, especially in the base case. However, for larger numbers, where the intermediate products are 128 bits or more, we can apply our PML Add techniques to speed up individual additions. Unfortunately, this does not allow us to perform the overall additions within the sets in parallel.

Step 4: Propagate Carries Propagate exceeding portions from the least significant set to the most significant set:

- Initial: $[24, 76, 73, 40, 12]$
- $12 = \underbrace{1}2$: Carry 1 to 40
- $40 + 1 = 41 = \underbrace{4}1$: Carry 4 to 73
- $73 + 4 = 77 = \underbrace{7}7$: Carry 7 to 76
- $76 + 7 = 83 = \underbrace{8}3$: Carry 8 to 24
- $24 + 8 = 32 = \underbrace{3}2$: Carry 3 to a new digit

After propagation:

$$M = [3, 2, 3, 7, 1, 2]$$

The carry-overs must be sequentially propagated and added, which may limit their parallelization. However, when implementing this on modern x86-64 systems, we can mitigate latencies associated with the carry-overs by using fused multiply-add instructions. This approach was also incorporated into our final base-case VBAP Mul implementation. Concatenating the digits results in: 323712.

The algorithm for the Vedic-Based Approach for Multiplication is outlined in Algorithm 15. Building on the approach mentioned before, the algorithm begins by creating the arrays for the multiplicand and multiplier by examining the prefix and suffix sets in step 1. In step 2, it then multiplies the corresponding digits independently. Step 3 involves adding the intermediate products according to the sets, followed by performing the carry-over, known as *suddhikaran*, based on the resulting sums. If we examine the process, we can see that the multiplications in step 2 can be executed in parallel. The formation of the multiplicands and multipliers can also be determined independently by using a precomputed map, without needing to form sets. The only limitation to parallelization arises from the steps of summing within the sets and handling the carry-overs between them.

Algorithm 15: Proposed Vedic-Based Approach for Multiplication (VBAP Mul)

```
Input  : Two $n$-digit numbers $X$ and $Y$ in base B.
Output: Product $P = X \times Y$.
$index_x \leftarrow 0, index_y \leftarrow 0$;                // Iterators for $M_x$ and $M_y$
// Step 1: Form prefix and suffix digit sets and insert digits
for $len \leftarrow 1$ to $n$ do
    Form prefix set $P_{X,len}$ from $X$ with length $len$ and extract individual digits;
    for each digit $d$ in $P_{X,len}$ do
        $M_x[index_x] \leftarrow d$;
        $index_x \leftarrow index_x + 1$;
    Form prefix set $P_{Y,len}$ from $Y$ with length $len$ and extract individual digits;
    for each digit $d$ in $P_{Y,len}$ do
        $M_y[index_y] \leftarrow d$;
        $index_y \leftarrow index_y + 1$;
for $len \leftarrow n - 1$ to 1 do
    Form suffix set $S_{X,len}$ from $X$ with length $len$ and extract individual digits;
    for each digit $d$ in $S_{X,len}$ do
        $M_x[index_x] \leftarrow d$;
        $index_x \leftarrow index_x + 1$;
    Form suffix set $S_{Y,len}$ from $Y$ with length $len$ and extract individual digits;
    for each digit $d$ in $S_{Y,len}$ do
        $M_y[index_y] \leftarrow d$;
        $index_y \leftarrow index_y + 1$;
// Step 2: Compute intermediate products in parallel
for $i \leftarrow 0$ to $index_x - 1$ do
    $M[i] \leftarrow M_x[i] \times M_y[i]$;          // Perform multiplications concurrently
// Step 3: Group and sum intermediate products by sets
Define set boundaries based on original prefix/suffix pairings;
for each set $S_k$ in $M$ do
    $R_k \leftarrow$ sum of all $M[i]$ within set $S_k$;
// Step 4: Perform suddhikaran (carry-over extra digits)
$carry \leftarrow 0$;
for $k \leftarrow$ number of sets to 1 (right to left) do
    $R_k \leftarrow R_k + carry$;
    if $R_k \geq B$ then
        $carry \leftarrow \lfloor R_k / B \rfloor$;
        $R_k \leftarrow R_k \mod B$;
    else
        $carry \leftarrow 0$;
if $carry > 0$ then
    Prepend $carry$ as an extra digit to the result;
// Step 5: Combine digits to form the final product
Combine digits in $R$ to form the product $P$;
return $P$;
```

6.1.2 Hybrid Approach: KVBAP Mul

Using the VBAP approach as the base case and Karatsuba overall, we aim to perform the multiplication faster than the grade-school-based base case approach. While splitting the operands into two parts using the Karatsuba method, if the number of limbs is less than or equal to five (i.e. 256-bits for 52-bit radix), we will utilize the VBAP Mul approach. Algorithm 16 depicts the Hybrid approach: Karatsuba-VBAP (KVBAP) Multiplication.

Proposed Hybrid Algorithm

Algorithm 16: Proposed Hybrid Karatsuba-VBAP (KVBAP) Multiplication

```
Input  : Two $N$-limb numbers $X$ and $Y$ in base $B$, where each limb is bits_per_limb
         bits (e.g., 64 on x86_64).
Output: Product $P = X \times Y$.
// Define split point and base
$k \leftarrow \lfloor N/2 \rfloor$;
$b \leftarrow 2^{k \cdot \text{bits\_per\_limb}}$;                        // Base for splitting
// Base case:  Use VBAP Multiplication
if $N \leq 5$ then
    $P \leftarrow \text{VBAP\_Mul}(X, Y)$;                    // Solve base case with VBAP
    return $P$;
// Recursive case:  Split operands
Split $X$ into $X_1 \cdot b + X_0$, where $X_0$ and $X_1$ are $k$-limb numbers;
Split $Y$ into $Y_1 \cdot b + Y_0$, where $Y_0$ and $Y_1$ are $k$-limb numbers;
// Compute recursive products
$P_1 \leftarrow \text{Hybrid\_Karatsuba\_VBAP}(X_1, Y_1)$;            // High part:  $X_1 \times Y_1$
$P_2 \leftarrow \text{Hybrid\_Karatsuba\_VBAP}(X_0, Y_0)$;             // Low part:  $X_0 \times Y_0$
$P_3 \leftarrow \text{Hybrid\_Karatsuba\_VBAP}(X_1 - X_0, Y_1 - Y_0)$;      // Cross term, handle
 negatives
// Combine results
$P \leftarrow (b^2 + b) \cdot P_1 - b \cdot P_3 + (b + 1) \cdot P_2$;            // Karatsuba combination
return $P$;
```

To simplify implementation, similar to GMP, we use subtraction operations instead of addition operations within the Karatsuba algorithm to reduce the likelihood of intermediary carry-outs. In this work, we successfully implemented the base-case VBAP multiplication, and we are currently in the early stages of implementing the hybrid approach. The implementation specifics of base-case VBAP Mul are listed at implementation 20.

6.2 Implementation of Proposed 256-bit VBAP Mul using AVX512-IFMA

6.2.1 Data Representation

We adopt a 52-bit *reduced-radix* or *unsaturated* to accelerate large number multiplication representation. The final implementation of the VBAP Mul (20) assumes limbs are stored from the highest index while reading the strings in little-endian order, unlike addition and subtraction, which assume limbs are stored from the lowest indices. Based on the implementation need, we may utilize either approach.

Example 6.2.1.1.
Consider a random 256-bit hexadecimal string:

```
0xEF1206754ee9451FA5F3C7B912E4D86F1B92A0D5E7C8F9346D1B5E2F9A3C7D8E
```

52-bit Reduced-Radix Representation (Highest Index)

$\texttt{limbs}_{52} =$

i	$\texttt{limbs}_{52}[i]$
0	0x0EF1206754EE9
1	0x451FA5F3C7B91
2	0x2E4D86F1B92A0
3	0xD5E7C8F9346D1
4	0xB5E2F9A3C7D8E

52-bit Reduced-Radix Representation (Lowest Index)

$\texttt{limbs}_{52} =$

i	$\texttt{limbs}_{52}[i]$
0	0xB5E2F9A3C7D8E
1	0xD5E7C8F9346D1
2	0x2E4D86F1B92A0
3	0x451FA5F3C7B91
4	0x0EF1206754EE9

We had four implementation versions before obtaining our final version based on the hybrid approach. We first discuss these four versions, followed by the final implementation outline 20.

6.2.2 Sub-optimal Implementations

First Version: VBAP_MUL using decimal bases, grouping four decimal digits into 32-bit limbs, unsaturated representation using AVX512F

Initially, we began working with four decimal digits, as multiplying four decimal digits would result in eight digits, which is lower than the capability to handle nine decimal digits safely with the 32-bit limb ($2^{32} - 1 = 4,29,49,67,295$). However, large operands like 32768-bit multiplications with Urdhva-Tiryagbhyam would result in a lot of additions within the sets, and we had to handle the additions within the set with a 64-bit accumulator. To extract the higher and lower four digits, we used division and modulo operators for each set.

Algorithm 17: VBAP_MUL using decimal bases, grouping four decimal digits into 32-bit limbs, unsaturated representation

```
Input: num1, num2: Two numbers with $n$ digits each
Output: product: Array of $2n-1$ limbs representing the product
// Divide the two numbers into n/4 limbs each, group into four digits from
   least significant digit
set_index ← 0, max_index ← $2n-2$;
for set_index ← 0 to max_index do
    $p$ ← 0;
    start ← max(0, set_index − $n$ + 1);
    end ← min(set_index, $n$ − 1);
    $p$ ← MULTIPLY_and_ADD(num1[start : end], num2[end : start]) ;
     // Cross-multiply and add resultants
    product[set_index] ← $p$ mod 10000;
    carry[set_index + 1] ← $p$/10000;
$c$ ← 0;
for $i$ ← $2n-2$ to 0 do
    $p$ ← product[$i$] + carry[$i$] + $c$;
    product[$i$] ← $p$ mod 10000;
    $c$ ← $p$/10000;
return product
```

In terms of performance, it was 30 to 300 times slower than GMP across various operand sizes. This can primarily be attributed to two factors: (1) half-saturated limbs, which double the number of limbs and quadruple the number of multiplications due to the $O(n^2)$ algorithm, and (2) the overhead associated with costly division and modulo operations for each set.

Second Version: VBAP_MUL using saturated 64-bit hexadecimal representation, AVX512F

To mitigate the division and modulo operations for extraction, we resorted to hexadecimal representation of the values utilizing fully saturated radix within 64-bit limbs. But as AVX512-F does not provide any instructions to get higher 64-bits of the 128-bit resultant of multiplication for multiplying two 64-bit operands, we had to partition the 64-bit into two smaller 32-bit sub-limbs (Similar to what GMP does). The outline of the approach is mentioned here: outline 18. We have analysed the timings of each of the utility functions (refer to Table 6.3), and we have observed that as the number of limbs increases, the timing for only doing multiplication using AVX512 increases drastically (due to $O(n^2)$ running time). For a smaller limb size (256-bit), the time for only doing the multiplications is 5ns compared to the overall time of GMP, 30ns. But the extra overheads of accumulating, adding, and carrying over adjustments sum up to 108ns (3x slow). And for 2048 bits, overall VBAP_MUL (64-32, AVX512F) is almost 10x slower.

Algorithm 18: Outline: VBAP_MUL (64-bit Limbs, split-into 32-bit sub-limbs, AVX-512F)

```
Input: $n1, n2$, len: two numbers in hex, length in 64-bit limbs
Output: result: product, length $2 \times$ len
max_idx $\leftarrow 4 \times$ len $\times$ len;
mul_tmp_1, mul_tmp_2 $\leftarrow$ arrays of length max_idx;
result $\leftarrow$ array of length max_idx;
accumulate_muls($n1, n2$, len, mul_tmp_1, mul_tmp_2) ; // split 64-bit limbs into 32-bit sub-limbs
multiply_muls(max_idx, mul_tmp_1, mul_tmp_2, result) ;   // 32-bit multiplications
add_within_limbs(max_idx, result) ;                      // add within sets
adjust_inner_limbs(max_idx, result) ;                    // adjust within sets
remove_intermediary_zeros(max_idx, result) ;             // compact result
add_limbs(len, max_idx, result) ;                        // outer adds
final_result $\leftarrow$ adjust_limbs($2 \times (2 \times$ len $- 1) - 1$, result) ;   // outer adjusts
return final_result
```

Example 6.2.2.2.
The overall approach for the second version can be seen below for 16-bit numbers split into two 8-bit numbers:

```
n1: f548 8543
n2: 6b0d 9410
After accumulate_muls():
mul_tmp_1: f5 f5 48 48 f5 f5 48 48 85 85 43 43 85 85 43 43
mul_tmp_2: 6b 0d 6b 0d 94 10 94 10 6b 0d 6b 0d 94 10 94 10
After multiply_muls():
result: 6667 0c71 1e18 03a8 8da4 0f50 29a0 0480 3797 06c1 1c01
0367 4ce4 0850 26bc 0430
After add_within_limbs():
result: 6667 2a89 0000 03a8 8da4 38f0 0000 0480 3797 22c2 0000
0367 4ce4 2f0c 0000 0430
After adjust_inner_limbs() and remove_intermediary_zeros():
result: 6691 8ca8 8ddc f480 37b9 c567 4d13 1030
result: (6691 8ca8) (8ddc f480 + 37b9 c567) (4d13 1030)
After add_limbs():
result: 6691 8ca8 c596 b9e7 4d13 1030
After adjust_limbs():
result: 6692 523f 6fa 1030
```

Limbs	Bits	VBAP_MUL (64-32,AVX512F)	GMP Mul
4	256	0.108	0.030
8	512	0.448	0.076
16	1024	1.583	0.247
32	2048	8.058	0.787

TABLE 6.2: Timing (in μs) comparison VBAP_MUL (64-bit Limbs, split-into 32-bit sub-limbs, AVX512-F) with GMP on Intel Xeon E-2314

Limbs	Bits	acc	mul	add_within	adjust_within	remove_zeros	add	adjust
4	256	0.020	0.005	0.003	0.025	0.000	0.026	0.005
8	512	0.093	0.023	0.015	0.084	0.021	0.098	0.010
16	1024	0.299	0.110	0.058	0.326	0.090	0.367	0.026
32	2048	2.894	1.087	0.229	1.287	0.366	1.393	0.056

TABLE 6.3: Timing(in μs) of different utility functions for VBAP_MUL(64-32,AVX512F) on Intel Xeon E-2314

Our optimization strategy to lower the total timing of the multiplication can be one of the following:

1. Reduce the number of multiplications
2. Switch to 32-bit based limb format to avoid sub-limbs additions and adjustments
3. Avoid accumulation of operands into memory and directly load into the AVX registers
4. Get rid of removing zeros
5. Optimize the add and carry-over adjustment operations

Third Version: Base-Case 256-bit VBAP_MUL using saturated 32-bit hexadecimal representation

To optimize the second version further, we tried to address the possible optimizations in this version.

Reduce the number of multiplications: The number of multiplications can be reduced in two ways: Use a bigger base for operands. However, using AVX512-F, it is not feasible to have more than 32 bits as an effective base, as using more than 32 bits of data would result in a loss of the higher bits (the lower 64 bits would be returned). The only other way remaining is to use a divide-and-conquer-based approach (like Karatsuba), which reduces the number of multiplications. However, Karatsuba adds overhead for smaller numbers, and vectorizing Karatsuba is a bit tricky due to its recursion. Hence, we switched to the Hybrid Approach: For a threshold larger than a specific one, use Karatsuba, and when the threshold is reached, use Urdhva-Tiryagbhyam. The only task remaining now is to find a suitable size where we can get benefits using Urdhva-Tiryagbhyam and AVX512. If we analyze the cost only for multiplying using AVX512 (table 6.3), we can see that for 256-bit, multiplication is taking 5ns, and the overall timing for GMP is 30ns. Hence, we have a lot of space remaining (25ns) to beat GMP for 256 bits if we can optimize the other utility functions. Thus, that's how we chose 256-bit Urdhva-Tiryagbhyam as the base case.

Switch to 32-bit based limb format to avoid sub-limbs additions and adjustments: As switching to 32-bit based limb format does not increase or decrease the number of multiplications (because 64-bit was further divided into 32 bits), we can easily do that and save up the extraction of higher and lower parts. In turn, we can save up the sub-limb additions and carry-overs.

Avoid accumulation of operands into memory and directly load into the AVX registers: As we decided to go on with only 256-bit for VBAP_MUL, serving as the base case for our hybrid approach, we can manually place the required operands in exact order inside the AVX512 registers. Thereby saving up the cost of the accumulation part.

Get rid of removing zeros: This was an unnecessary operation, and we can easily get rid of it.

Optimize the add and carry-over adjustment operations: It is hard to optimize the add and carry-over adjustments, as these are the necessary operations. Using AVX512-F, we could not find other ways to optimize it algorithmically. However, we did some implementation optimization.

Algorithm 19: Outline: Base-case VBAP_MUL (32-bit Limbs, AVX-512F)

```
Input: num1, num2: Arrays of 32-bit unsigned integers (limbs)
Output: res: Array of 32-bit unsigned integers (product)
ACCUMULATE_MULTIPLY_AVX(num1, num2, res) ;  // Directly Accumulate and
 multiply using AVX-512
ADD_LIMBS(res) ;                            // Sum partial products into limbs
ADJUST_LIMBS(res) ;                          // Propagate carries across limbs
return res
```

With 256-bit operands, the ACCUMULATE_MULTIPLY_AVX implementation processes two arrays, num1 and num2, each containing eight limbs of 32-bit unsigned integers (denoted $a_0, a_1, \ldots, a_7$ and $b_0, b_1, \ldots, b_7$). These are loaded into 512-bit AVX-512 registers as four lanes split into eight elements of 64-bit integers, with each 32-bit limb zero-extended to 64-bit.

We need a total of 64 multiplications ($n^2 = (8^2)$). At a time, we can process 512 bits, thus eight multiplications can be performed at once. Hence, the multiplication proceeds in 8 groups, each computing eight partial products by permuting and multiplying selected limbs as per the order of the Urdhva-Tiryagbyam technique:

The eight groups and their lane contents are as follows:

- **Group 1**: Multiplies $(a_1b_2, a_0b_3, a_2b_0, a_1b_1, a_0b_2, a_1b_0, a_0b_1, a_0b_0)$, stores at res[0..7].
- **Group 2**: Multiplies $(a_0b_5, a_4b_0, a_3b_1, a_2b_2, a_1b_3, a_0b_4, a_3b_0, a_2b_1)$, stores at res[8..15].
- **Group 3**: Multiplies $(a_2b_4, a_1b_5, a_0b_6, a_5b_0, a_4b_1, a_3b_2, a_2b_3, a_1b_4)$, stores at res[16..23].
- **Group 4**: Multiplies $(a_3b_4, a_2b_5, a_1b_6, a_0b_7, a_6b_0, a_5b_1, a_4b_2, a_3b_3)$, stores at res[24..31].
- **Group 5**: Multiplies $(a_4b_4, a_3b_5, a_2b_6, a_1b_7, a_7b_0, a_6b_1, a_5b_2, a_4b_3)$, stores at res[32..39].
- **Group 6**: Multiplies $(a_6b_3, a_5b_4, a_4b_5, a_3b_6, a_2b_7, a_7b_1, a_6b_2, a_5b_3)$, stores at res[40..47].
- **Group 7**: Multiplies $(a_5b_6, a_4b_7, a_7b_3, a_6b_4, a_5b_5, a_4b_6, a_3b_7, a_7b_2)$, stores at res[48..55].
- **Group 8**: Multiplies $(a_7b_7, a_7b_6, a_6b_7, a_7b_5, a_6b_6, a_5b_7, a_7b_4, a_6b_5)$, stores at res[56..63].

(Note: data inside AVX512 across lanes are loaded in little-endian byte order) An example lane configuration is shown in Figure 6.1 for Group 1.

For 256-bit base-case multiplication using this technique, it took 35ns compared to GMP's 30ns on Intel Xeon E-2314, which was still slower than GMP but much faster than the second version (from 108ns to 35ns).

Therefore, with AVX512-F, we could not reduce the number of multiplications. Also, if we analyze the total timing contribution, the ACCUMULATE_MULTIPLY_AVX is still only taking nearly 8ns on Intel Xeon E-2314; the rest are from add and adjust. The add and adjust functions are not efficiently vectorizable for 256 bits, as this needs horizontal sum SIMD, which is not optimized properly yet. Consequently, we had to resort to doing these operations on the ALU, significantly increasing the overall execution time.

Fourth Version: Base-Case 256-bit VBAP_MUL using unsaturated 52-bit hexadecimal representation using AVX512-IFMA

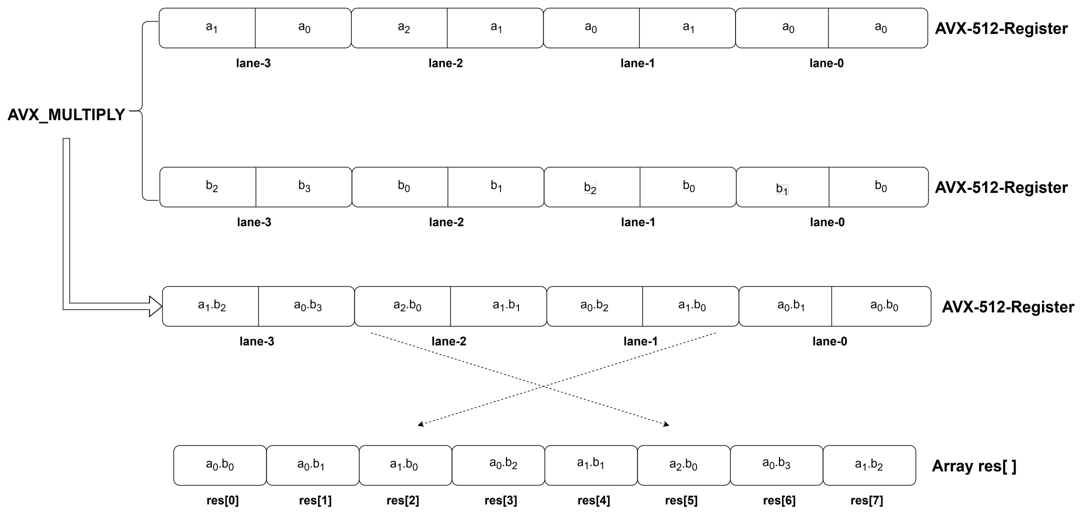
AVX_MULTIPLY
a_1 a_0 a_2 a_1 a_0 a_1 a_0 a_0
AVX-512-Register
lane-3 lane-2 lane-1 lane-0
b_2 b_3 b_0 b_1 b_2 b_0 b_1 b_0
AVX-512-Register
$a_1.b_2$ $a_0.b_3$ $a_2.b_0$ $a_1.b_1$ $a_0.b_2$ $a_1.b_0$ $a_0.b_1$ $a_0.b_0$
AVX-512-Register
$a_0.b_0$ $a_0.b_1$ $a_1.b_0$ $a_0.b_2$ $a_1.b_1$ $a_2.b_0$ $a_0.b_3$ $a_1.b_2$
Array res[ ]
res[0] res[1] res[2] res[3] res[4] res[5] res[6] res[7]

FIGURE 6.1: Example of AVX Lanes containing the operands while multiplying and storing (Group-1)

We tried to look for AVX512 flags other than AVX512F that may contain any useful way to get the higher part access while using 64-bit operands. We found that AVX512IFMA-52 can be used to get the higher and lower bits of the resultant, but we need to resort to a 52-bit base for the limbs. However, we get the higher bits and lower bits by applying multiplication twice while loading the operands only once. But that would be better than using 32-bit, as it reduces the number of limbs, in turn reducing the number of multiplications, additions and adjustments.

We can utilize the following two intrinsics from AVX512IFMA-52:

1. `_mm512_madd52hi_epu64`: This intrinsic takes three operands: __m512i a, __m512i b, and __m512i c. It multiplies packed unsigned 52-bit integers in each 64-bit element of b and c, producing a 104-bit intermediate result. Then, it adds the high 52-bit unsigned integer from the intermediate result to the corresponding unsigned 64-bit integer in a, storing the results in the destination (dst).

2. `_mm512_madd52lo_epu64`: This intrinsic also takes three operands: __m512i a, __m512i b, and __m512i c. Similar to the first intrinsic, it multiplies packed unsigned 52-bit integers in each 64-bit element of b and c to form a 104-bit intermediate result. However, it adds the low 52-bit unsigned integer from the intermediate result to the corresponding unsigned 64-bit integer in a and stores the results in the destination (dst).

For 256 bits, we require five limbs to store in a 52-bit format with 64-bit limbs. We will load five limbs only once and permute the AVX data to prepare the exact operands for multiplication. Hence, a total of 25 multiplications are required for five limbs. But to get the higher and lower 52 bits, we require multiplying them twice: a total of 50 multiplications. Next, we need to add the higher and lower partial products separately, corresponding to the set order (note we also need to account for carries in higher partial products that may be generated in the lower partial product additions). Then, the summed-up higher products are added to the lower products according to the carry-over adjustments. This approach for multiplying two 256-bit operands took 28ns on Intel Xeon E-2314, compared to GMP, which took 30ns (Faster than the third version, taking 35ns). The permuting and multiplying inside the AVX register only took 4ns; the rest are from the sequential addition and carrying over.

Example 6.2.2.3.
An instance of the fourth version outline is shown with 128-bit operands:

```
For 128-bits:
a = 0xab32ef0112f0987afe01fabc12349f24
b = 0xab21fe1024ab5c2e234f867c664f3abe
(52-bit unsaturated format, 64-bit limbs)
a = 0000000000ab32ef 0000112f0987afe0 0001fabc12349f24
b = 0000000000ab21fe 0001024ab5c2e234 000f867c664f3abe
If we are accumulating:
m1: 0000000000ab32ef 0000000000ab32ef 0000112f0987afe0
    0000000000ab32ef 0000112f0987afe0 0001fabc12349f24
    0000112f0987afe0 0001fabc12349f24 0001fabc12349f24
m2: 0000000000ab21fe 0001024ab5c2e234 0000000000ab21fe
    000f867c664f3abe 0001024ab5c2e234 0000000000ab21fe
    000f867c664f3abe 0001024ab5c2e234 000f867c664f3abe
But, we will directly load this inside the AVX512 registers
using permutation.
Using AVX512-IFMA using mullo52:
r_lo_52: 7271c1125822 b246cd9db568c 37a886cec6040 b310b3a5af362
         9c2e460937980 92e1925c589b8 9da8456ad4840 e3f1334761b50
         736cae33844b8
Using AVX512-IFMA using mulhi52:
r_hi_52:            0           acbb4          b7cb        a61ebd
         11566b66d70        152bee  10ac88797012  1ff456c7f06b
         1ebb39c039737
Next, we add the lower parts and the higher parts according
to set ordering:
r_lo_52: 7271c1125822 (b246cd9db568c + 37a886cec6040)
         (b310b3a5af362 + 9c2e460937980 + 92e1925c589b8)
         (9da8456ad4840 + e3f1334761b50) 736cae33844b8
r_hi_52: 0 (acbb4 + b7cb) (a61ebd + 11566b66d70 + 152bee)
         (10ac88797012 + 1ff456c7f06b) 1ebb39c039737
Resulting into:
r_lo: 7271c1125822 e9ef546c7b6cc 1e2208c0b3f69a 1819978b236390
736cae33844b8
r_hi: 0 b837f 1156771b81b 30a0df41607d 1ebb39c039737
If r_lo[i] more than 13-hex digits, add 1 to r_hi[i] and
adjust r_lo[i]
r_lo: 7271c1125822 e9ef546c7b6cc (1)e2208c0b3f69a (1)819978b236390
736cae33844b8
r_hi: 0 b837f 1156771b81b 30a0df41607d 1ebb39c039737
==
r_lo: 7271c1125822 e9ef546c7b6cc e2208c0b3f69a 819978b236390
736cae33844b8
r_hi: 0 b837f 1156771b81c 30a0df41607e 1ebb39c039737
Carry-over adjustment:
Perform r_lo[i] += r_hi[i+1]:
r_lo: 7271c1125822 e9ef546c7b6cc e2208c0b3f69a 819978b236390
736cae33844b8
r_hi:        b837f   1156771b81c  30a0df41607e 1ebb39c039737
+     ----------------------------------------------------------------
sum:  7271c11ddba1 ea00aae396ee8 e52a99ff55718 a054b2726fac7 736cae33844b8
     (that's our multiplication result)
```

Final Implementation using AVX512-IFMA

To further reduce the sequential operations, we tried to look into the fused multiplication and addition, using which we can pass some values to be added while computing the products within the same cycles $(p = a + b * c)$ using the AVX512-IFMA52. We can pass some higher partial products directly while computing the lows as we analyze the adds within the sets and carry-overs for 256-bit (52-bit limbs). Figure 6.2 shows the intuition behind doing such a technique. For a five-limb multiplication (assuming each limb contains 8 bits for simplicity), we would typically perform the vertical and cross-wise multiplication followed by additions, as depicted in (0) in the figure. However, after performing addition within the sets, we typically carry over the exceeding bits of a current set to its preceding set as part of the suddhikaran process. That means the higher ends of the resultant limbs are added to the lower limbs of their preceding limbs.

By exploiting this behaviour, we may hide the latency for the carry-over process using the fused-multiply-addition technique, which performs $(p = a + b * c)$ within the same cycles. Simply, what we aim to do is to compute the highs beforehand. Then, pass on the highs to the fused addition while computing the low products $(a + (b * c))$. We have marked the highs that can be added while computing the products using the pointed-arrowed lows in Figure 6.2-(1) for five limb multiplications. Note that some of the highs can't be added in the same cycles of FMA as other highs are already being added to the desired lows and require adding them up after the FMA is done, and those are marked with dotted arrows, and (2) shows the process of adding up the remaining highs. Now, the only thing remaining is to add up the intermediate results by their sets and propagate any minimal carry that has been generated due to the within-set additions. Marked processes (3) and (4) exactly perform that. Note that this carry-propagation is not the same as the usual suddhikaran, where highs are added to the lows. In the typical suddhikaran process, the highs are added to the lows; we are still required to propagate any carry generated to the preceding lows, and that can be taken as the same process of this carry propagation. By this technique, we are able to hide the latency of adding the highs to the lows using the FMA.

To represent 256-bit in 52-bit reduced radix format, we require exactly five limbs that perform a similar thing, except taking 52 bits instead of 8 bits as shown in the example in Figure 6.2.

For base case multiplication, we're working with 256 bits; five limbs are needed for each of the two operands. Means a total of 25 multiplications. However, through AVX512IFMA, we can process eight 64-bit (with 52-bit radix) at once, which means we would require a total of three rounds of AVX512IFMA-52 multiplication followed by a single remaining multiplication to be performed either on the usual ALU or 128-bit register. Also, the remaining last multiplication does not contribute to any other highs or lows, as can be seen in Figure 6.2. Table 6.4 shows the high indices that need to be added to the corresponding low indices while performing the fused multiply and add. The corresponding highs of the asterisk-marked low indices have to be added after the fused multiplication is over, as one of their preceding highs is already being added during FMA. For example, the value at high index-1 is added to the value at low index-0, so after the FMA operation, the value at high index-2 will also be added to low index-0.

To effectively compute the first set of low indices (0-7), we first need to account for their dependency on the high indices (0-11). This requires us to calculate two sets

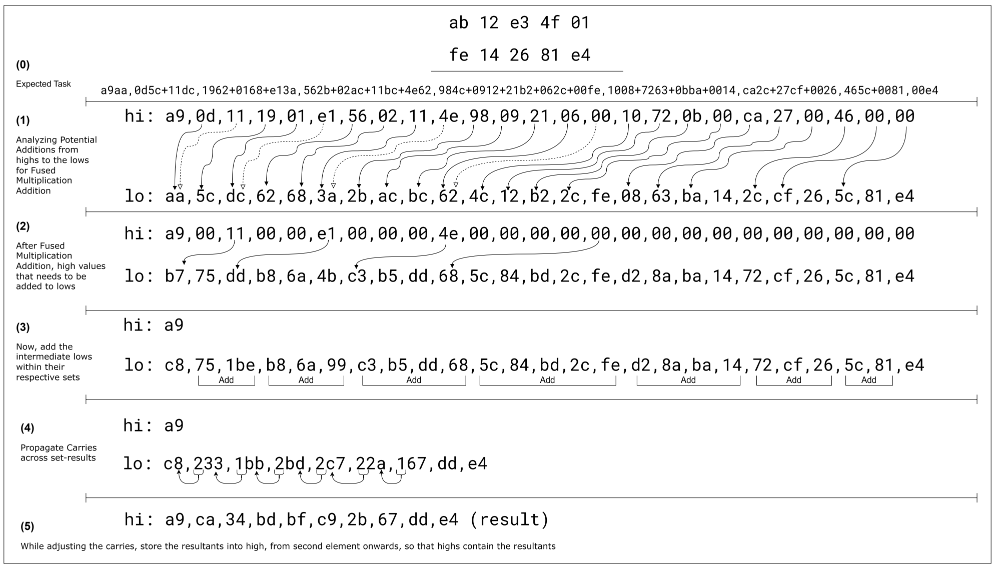
ab 12 e3 4f 01
fe 14 26 81 e4
(0)
Expected Task
a9aa,0d5c+11dc,1962+0168+e13a,562b+02ac+11bc+4e62,984c+0912+21b2+062c+00fe,1008+7263+0bba+0014,ca2c+27cf+0026,465c+0081,00e4
(1)
Analyzing Potential Additions from highs to the lows for Fused Multiplication Addition
hi: a9,0d,11,19,01,e1,56,02,11,4e,98,09,21,06,00,10,72,0b,00,ca,27,00,46,00,00
lo: aa,5c,dc,62,68,3a,2b,ac,bc,62,4c,12,b2,2c,fe,08,63,ba,14,2c,cf,26,5c,81,e4
(2)
After Fused Multiplication Addition, high values that needs to be added to lows
hi: a9,00,11,00,00,e1,00,00,00,4e,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
lo: b7,75,dd,b8,6a,4b,c3,b5,dd,68,5c,84,bd,2c,fe,d2,8a,ba,14,72,cf,26,5c,81,e4
(3)
Now, add the intermediate lows within their respective sets
hi: a9
lo: c8,75,1be,b8,6a,99,c3,b5,dd,68,5c,84,bd,2c,fe,d2,8a,ba,14,72,cf,26,5c,81,e4
Add
Add
Add
Add
Add
Add
Add
(4)
Propagate Carries across set-results
hi: a9
lo: c8,233,1bb,2bd,2c7,22a,167,dd,e4
(5)
hi: a9,ca,34,bd,bf,c9,2b,67,dd,e4 (result)
While adjusting the carries, store the resultants into high, from second element onwards, so that highs contain the resultants

FIGURE 6.2: Example of Proposed Urdhva-Tiryagbhyam Five-Limb Fused-Multiplication-Addition Technique

High Index	0	1	2	3	4	5	6	7	8	9	10	11
Low Index	-	0	0*	1	2	2*	3	4	5	5*	6	7

High Index	12	13	14	15	16	17	18	19	20	21	22	23	24
Low Index	8	9	9*	10	11	12	13	15	16	17	19	20	22

TABLE 6.4: High-Low Index Mapping for five limbs IFMA AVX Multiplication for Basecase 256-bit VBAP MUL

of high indices: the first set (0-7) and the second set (8-15), before we can proceed with computing the first set of low indices.

Next, for the second set of low indices (8-15), there is a dependency on high indices (12-19). Thus, we need to compute both the second set of high indices (8-15) and the third set (16-23). After calculating the third and final set of high indices (16-23), we can assign the respective indices from the second and third sets of high indices to their corresponding low indices during the fused multiply-add (FMA) operation. Followed by passing the required indices from the third set of highs to the corresponding third set of lows for preparation. After the FMA is over, we will add the asterisk-marked high indices to the corresponding low indices of the FMA computed values.

Once all the high indices have been added to the low indices, we have already performed both the multiplication and the carry-over phase. The only remaining task is to add the lows as per the set ordering, propagate any extra carry generated from the low indices to their preceding low indices and store the results from the high indices (1-9). It's important to note that high index 0 already contains the desired value, with the exception of any carry that needs to be added from the first low index.

Before proceeding, we need to understand the lane contents to be prepared for multiplications:

$$\begin{array}{rl}
 & a_0a_1a_2a_3a_4 \\
\times & b_0b_1b_2b_3b_4 \\
\hline
 & a_0b_0 \\
 & a_0b_1 + a_1b_0 \\
 & a_0b_2 + a_1b_1 + a_2b_0 \\
 & a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 \\
 & a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 \\
 & a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 \\
 & a_2b_4 + a_3b_3 + a_4b_2 \\
 & a_3b_4 + a_4b_3 \\
 & a_4b_4
\end{array}$$

The three groups and their lane contents would be as follows:

- **Group 1**: Multiplies $(a_1b_2, a_0b_3, a_2b_0, a_1b_1, a_0b_2, a_1b_0, a_0b_1, a_0b_0)$
- **Group 2**: Multiplies $(a_1b_4, a_4b_0, a_3b_1, a_2b_2, a_1b_3, a_0b_4, a_3b_0, a_2b_1)$
- **Group 3**: Multiplies $(a4b3, a3b4, a4b2, a3b3, a2b4, a4b1, a3b2, a2b3)$

Left-over a_4b_4 can be multiplied in the usual general-purpose registers.

The implementation outline (Algorithm 20) depicts the final approach for computing 256-bit base-case multiplication using VBAP MUL and FMA techniques. In the code implementation for the base case 256-bit VBAP Mul, arranging the ordering of the intrinsic calls is crucial, as there are heavy read-after-write (RAW) dependencies among them. In our final implementation, we tried many permutations and combinations to arrange the intrinsic calls of load, stores and other operations in a way that minimizes the pipelined stalls due to conflicts.

For a detailed overview of the AVX Lane operations, you may see Figure 6.3.

Algorithm 20: Implementation Outline of 256-bit Base-Case Multiplication with VBAP MUL and IFMA

```
Input: $a[0\dots4]$, $b[0\dots4]$ (256-bit operands split into 5 limbs each)
Output: $high[0\dots9]$ (result of $a[0\dots4] \times b[0\dots4]$)
1. Load $a[0\dots4]$ into base_1 (lanes 0 to 4) and $b[0\dots4]$ into base_2 (lanes 0 to 4)
2. Permute base_1 to get Group 1 multiplicands: $a\_vec_0 \leftarrow \langle a_1, a_0, a_2, a_1, a_0, a_1, a_0, a_0 \rangle$
3. Permute base_2 to get Group 1 multipliers: $b\_vec_0 \leftarrow \langle b_2, b_3, b_0, b_1, b_2, b_0, b_1, b_0 \rangle$
4. $hi\_vec_0 \leftarrow$ fused_mul_add_hi$(0, a\_vec_0, b\_vec_0)$          // Highs 7-0
5. Store $hi\_vec_0[7\dots0]$ to $high[0\dots7]$
6. Permute base_1 to get Group 2 multiplicands: $a\_vec_1 \leftarrow \langle a_1, a_4, a_3, a_2, a_1, a_0, a_3, a_2 \rangle$
7. Permute base_2 to get Group 2 multipliers: $b\_vec_1 \leftarrow \langle b_4, b_0, b_1, b_2, b_3, b_4, b_0, b_1 \rangle$
8. $hi\_vec_1 \leftarrow$ fused_mul_add_hi$(0, a\_vec_1, b\_vec_1)$          // Highs 15-8
9. Store $hi\_vec_1[7\dots0]$ to $high[8\dots15]$
10. $hi\_add_{low_0} \leftarrow$
   $\langle hi\_vec_1[3], hi\_vec_1[2], hi\_vec_1[0], hi\_vec_0[7], hi\_vec_0[6], hi\_vec_0[4], hi\_vec_0[3], hi\_vec_0[1] \rangle$

11. $lo\_vec_0 \leftarrow$ fused_mul_add_lo$(hi\_add_{low_0}, a\_vec_0, b\_vec_0)$
12. Store $lo\_vec_0[7\dots0]$ to $low[0\dots7]$
13. Permute base_1 to get Group 3 multiplicands:
   $a\_vec_2 \leftarrow \langle a_4, a_3, a_4, a_3, a_2, a_4, a_3, a_2 \rangle$
14. Permute base_2 to get Group 3 multipliers: $b\_vec_2 \leftarrow \langle b_3, b_4, b_2, b_3, b_4, b_1, b_2, b_3 \rangle$
15. $hi\_vec_2 \leftarrow$ fused_mul_add_hi$(0, a\_vec_2, b\_vec_2)$          // Highs 23-16
16. $hi\_add_{low_1} \leftarrow$
   $\langle hi\_vec_2[3], 0, hi\_vec_2[2], hi\_vec_2[1], hi\_vec_2[0], hi\_vec_1[7], hi\_vec_1[5], hi\_vec_1[4] \rangle$
17. $lo\_vec_1 \leftarrow$ fused_mul_add_lo$(hi\_add_{low_1}, a\_vec_1, b\_vec_1)$
18. $hi\_add_{low_2} \leftarrow \langle 0, 0, 0, hi\_vec_2[7], hi\_vec_2[6], 0, hi\_vec_2[5], hi\_vec_2[4] \rangle$
19. $lo\_vec_2 \leftarrow$ fused_mul_add_lo$(hi\_add_{low_2}, a\_vec_2, b\_vec_2)$
20. Store $lo\_vec_1[7\dots0]$ to $low[8\dots15]$
21. $prod \leftarrow a[4] \times b[4]$
22. begin // Remaining Additions
    $low[22] \leftarrow low[22] + (prod >> 52)$
    $low[0] \leftarrow low[0] + high[2]$
    $low[2] \leftarrow low[2] + high[5]$
    $low[5] \leftarrow low[5] + high[9]$
    $low[9] \leftarrow low[9] + high[14]$
23. begin // Within-set additions and carry propagation
    $high[9] \leftarrow prod \& 0xFFFFFFFFFFFFF$; for $i \leftarrow 8$ to $1$ do
        $sum \leftarrow carry$; switch $i$ do
            case 8 do
                $sum += low[22] + low[23]$
            case 7 do
                $sum += low[19] + low[20] + low[21]$
            case 6 do
                $sum += low[15] + low[16] + low[17] + low[18]$
            case 5 do
                $sum += low[10] + low[11] + low[12] + low[13] + low[14]$
            case 4 do
                $sum += low[6] + low[7] + low[8] + low[9]$
            case 3 do
                $sum += low[3] + low[4] + low[5]$
            case 2 do
                $sum += low[1] + low[2]$
            case 1 do
                $sum += low[0]$
        $high[i] \leftarrow sum \& 0xFFFFFFFFFFFFF$; $carry \leftarrow sum >> 52$;
    $high[0] += carry$;
24. $high[0\dots9]$ contains the result of $a[0\dots4] \times b[0\dots4]$
```

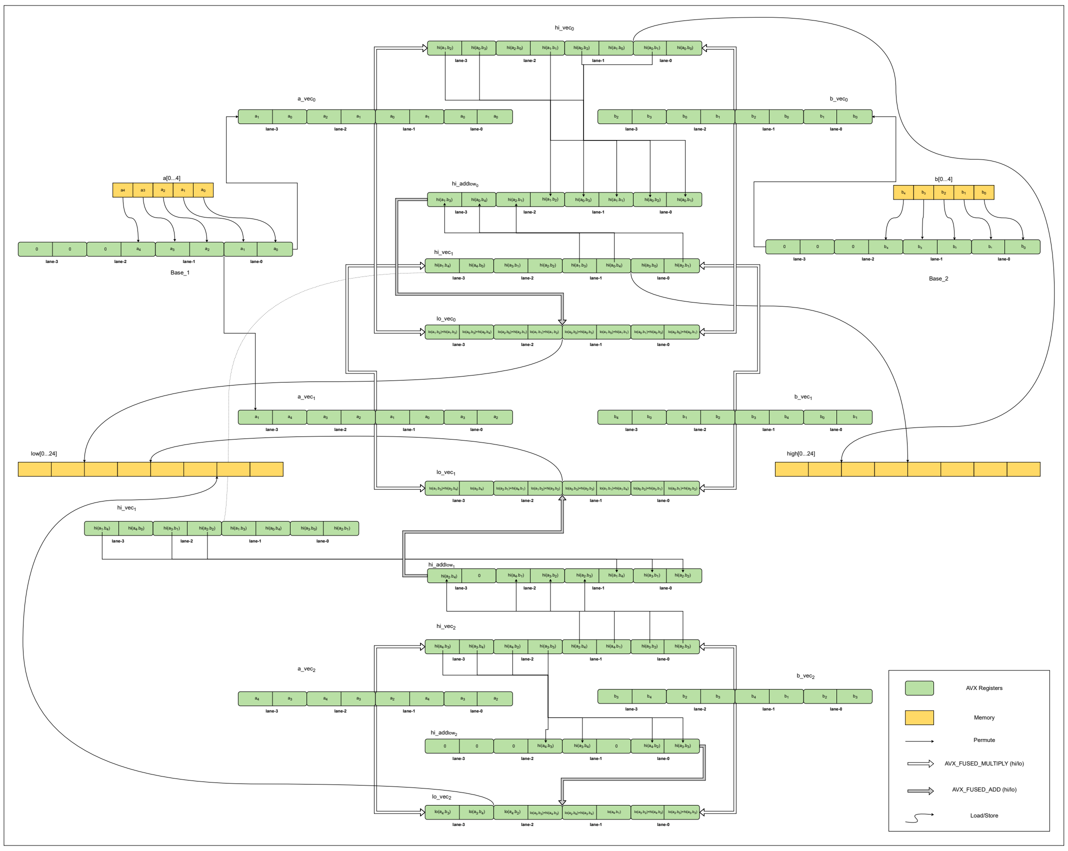

FIGURE 6.3: Overview of AVX Lane operations for five limb VBAP MUL using IFMA

6.3 Evaluation

In the hybrid KVBAP Mul approach, we implemented only the 256-bit base case VBAP multiplication. Therefore, the entire evaluation is based solely on 256-bit operand sizes. Similar to Addition and Subtraction, the core questions we are trying to answer here are the following:

1. Are the computations yielding correct results for 256-bit multiplication?
2. Are we getting any performance improvements, in terms of execution time, instruction count, and CPU Cycles, over GMP?
3. How much performance gain do we get compared to GMP for 256-bit multiplication?
4. How much performance gain are we observing with vectorization as compared to a non-vectorized baseline implementation?

For the vectorized multiplication (Base case 256-bit VBAP Mul), we have used the following set of flags:

```
-mavx512f -mavx512ifma -O1
```

For the base case 256-bit VBAP Mul, the GCC compiler with the O2 flag rearranged the intrinsic calls to an undesired order, leading to probable pipeline stalls in the AVX ALU. In contrast, the O1 flag kept the ordering intact, getting the optimal performance. The correctness check and benchmarking were conducted on the same x86-64-based Intel Xeon E-2314 CPU.

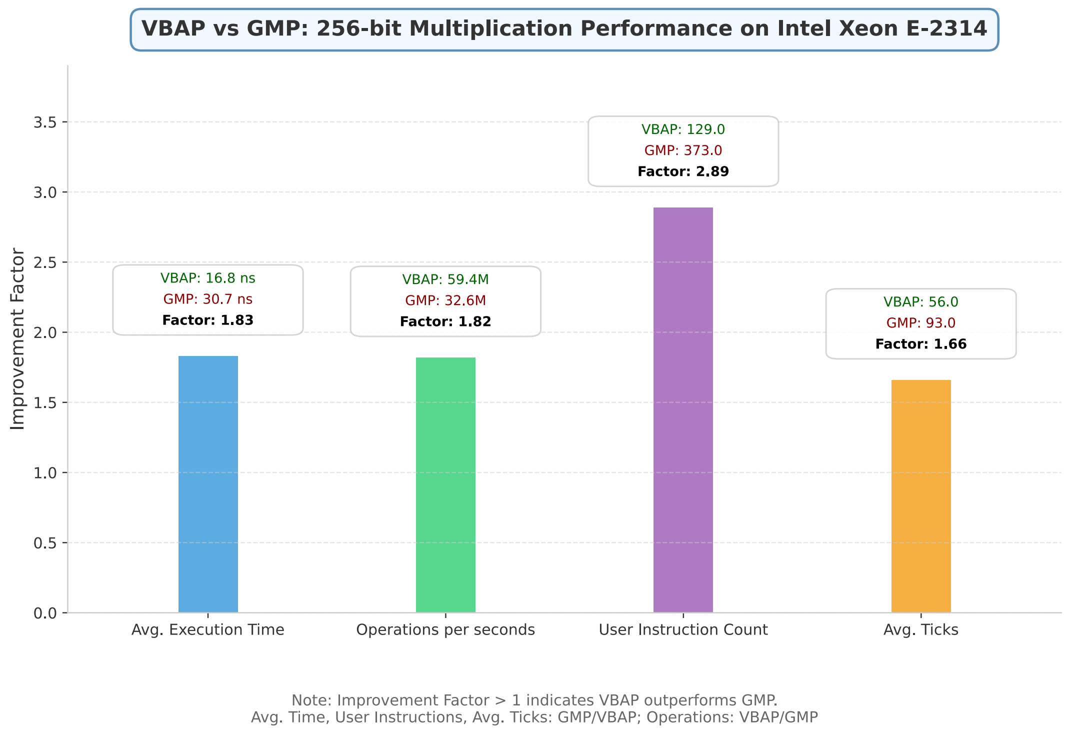

FIGURE 6.4: VBAP vs. GMP: Performance Gains on Intel Xeon E-2314 (256-bit Mul)

6.3.1 Correctness

To validate the correctness of our base case 256-bit VBAP Mul, we generated multiple sets of 1,00,000 random test cases using a similar technique as with addition and subtraction implemented via Python's random module and the `gmpy2` ([PYP]) library. All the computations with the VBAP Mul matched the pre-computed results of the gmpy2 library.

6.3.2 Performance Compared to GMP

The works by [KM14; GK16; ET18; ET20; ET23] obtained their performance gains only beyond 1024-bit operand sizes and are typically slow below 1024-bits when compared to GMP. Consequently, we opted to benchmark our performance primarily against the GMP. Similar to addition and subtraction, we benchmarked against the latest available GMP, version 6.3.0, invoking the corresponding function of GMP (mpz_mul()) from C code.

We reported the average execution time (in nanoseconds), average operations per second, average tick counts, and user instruction counts for both multiplication functions. Figure 6.4 illustrates the improvement factors for each category.

In terms of average execution time, GMP takes 31.7 ns, while VBAP performs better at 16.8 ns, making VBAP 1.83 times faster than GMP. The throughput for GMP is approximately 32.6 million operations per second (OP/s), compared to VBAP's 59.4 million OP/s, which is 1.82 times higher than GMP. With VBAP, only 129 user instructions are executed, whereas GMP requires about 373 user instructions, resulting in 2.89 times fewer instructions for VBAP, a reduction of 65% in the user instruction count. In terms of average ticks, VBAP shows an improvement of 1.66 times over GMP, with VBAP taking 56 ticks compared to GMP's 93 ticks.

Chapter 7

Discussion and Limitations

7.1 Discussion

Addition and Subtraction : In terms of average execution time for large integer additions, we observed a notable performance improvement when compared to the GMP library, with an average speed-up of 2.06x and a median of 2.02x. When compared to the baseline, we achieved an average speed-up of 5.14x and a median of 5.38x. Even in the worst rare cases, we still achieve an average of 1.38x and a median of 1.35x performance gain over GMP. For subtraction, we see an average performance improvement of 2.32x and a median improvement of 2.17x over GMP, an average of 4.26x and a median of 4.81x improvement over baseline. In worst cases, we are still seeing an average of 1.49x and a median of 1.46x improvement over GMP.

The performance gain we achieved for addition and subtraction, in rare cases, is reduced due to phase 4 trigger events (i.e. current limb addition generating a carry and its preceding block is maxed or zeroed out). However, for millions of test cases, we could not find any such cases triggering into phase 4 for each of the limb sizes ranging from 256-bit to 1,31,072-bit, generated using the random number generator. We have to manually curate the inputs so that they get triggered into phase 4, and our implementation is still considerably faster than GMP in such cases, too.

We are seeing a superior performance in subtraction compared to addition, and this can be noted in all the metrics. For instance, on average, across the operand sizes for the regular cases, PML Add is taking 335 ticks compared to 270 ticks by PML Sub and 1151 user instructions compared to 1006 user instructions. In worst cases, on average across operands, PML Add is observing 540 average ticks and 1762 user instructions compared to PML Sub's 469 average ticks and 1569 user instructions. Although we simply followed a similar implementation strategy for addition and subtraction, we still see a mismatch in execution time and user instruction counts for both PML and baseline. We are confused about its reason, with one of the possibilities being compiler-generated code, as GMP for both operations stays with similar metrics. Current benchmarks only cover bit sizes of {256, 512, 1024, 2048, 8192, 16384, 32768, 65536, 131072}, and we haven't shown odd bit sizes. Our implementation, however, is structured around the number of limbs, as AVX512 processes 512 bits at once (i.e. eight 64-bit limbs). This means we achieve peak performance when the number of limbs is a multiple of eight. For cases where the number of limbs isn't a multiple of eight, we can handle them efficiently using four versions of the ADD/SUB macro: single-limb operations can be done manually with general-purpose registers; two-limb operations can leverage SSE2 (128-bit) using the same PML Add or Sub logic; four-limb operations can utilize AVX2 (256-bit) for PML Add/Sub; eight-limb operations can directly run with AVX512. Any remaining odd limbs are processed using combinations of these macros. In our initial experiments with a 512-bit hybrid KVBAP MUL using IFMA (52-bit limbs), we integrated these four addition and subtraction macros into Karatsuba, and it worked flawlessly, delivering comparable performance gains.

For the approximate variants of addition and subtraction, we tested our code implementation using an extensive set of randomly generated test cases, and it produced correct results across all the test cases. If one's application for large number addition and subtraction mostly uses random operands and can accept slightly approximate computations in very rare cases, they could achieve further performance boosts compared to GMP: an average speed-up of nearly 2.52x for addition and 2.8x for subtraction. In addition to offering superior performance, the approximation approach also decreases the number of instructions to be executed by 18% for addition and 28% for subtraction, on average, compared to the standard non-approximate variant. This reduction in instructions can lead to greater energy savings as well. One of the motivations behind existing works on approximate calculations, as noted in previous studies ([Jia+20; AKL18]), is energy savings, and this principle is applicable in this case as well.

In comparison to existing works, Alexander Yee's technique ([Yee19]) for y-cruncher claimed not to show much performance gains due to the micro-architectural slowdown of operating on mask registers. They noted that their method incurs considerable overhead, making it difficult to outperform a chain of add-with-carry instructions on x64 systems; thus, our technique puts the majority of the operations on the masking registers inside phase 4, which almost never triggers. On the other hand, the work by Ren et al. [RSS23] did not provide their code in their paper, and the implementation details were not sufficient to recreate their implementation. However, they did not compare their work with GMP; rather, they tried to infuse it in the existing libraries: a 30% speed-up from the latest CTIDH implementation, an 11% speed-up from the latest CSIDH implementation in AVX-512 processors, and a 7% speed-up from Microsoft's standard PQCrypto-SIDH for SIKEp503 on A64FX.

Multiplication: For multiplication, we only computed for fixed 256-bit (or five 52-bit limb format), and we are working on variable size multiplications (below or equal to 256-bit use base case approach; otherwise, use hybrid).

We observed 65% fewer user instructions executed by the VBAP Mul as compared to GMP, providing above 1.80 times performance gain in terms of execution time and throughput. While analyzing timings inside the VBAP Mul, the mostly computationally heavy AVX part (highs and lows computation inside the AVX512 registers, including loads and stores) attributed just about 5.4ns and the rest of the 11.4ns were contributed by the remaining non-AVX code that utilized the general-purpose registers (for adding up the lows and propagating the carry-overs) and the latency to switch between AVX registers and general-purpose registers. This observation is further corroborated by analyzing the assembly instructions generated from the compiled C code. Out of the total number of instructions, AVX512 instructions made up only 42% of the total, whereas the serial parts represented 56%. Additionally, when we look at the timing magnitudes, 42% of the instructions contributed to 32% of the total execution time, while the following 56% of serial parts accounted for 56% of the timing.

Notably, we observed that the GCC compiler with certain optimization flags like O2, O3 and Ofast rearranges the inlined intrinsic calls in an undesired way for AVX512. Thus, the compiled code with O2 or higher flags degrades the performance, as the original high-level C code for VBAP Mul is written with the RAW dependencies in mind. O1 worked best as it hardly rearranged the order of the intrinsic calls, providing the best performance among the O flags.

Existing implementations using intrinsics [KM14; GK16; ET18; ET20; ET23] saw performance gains above 1,024 bits to 3,072 bits and did not see any benefit below 1024 bits, but we are seeing performance gains even on 256 bits.

Future with AVX10: Unlike AVX-512, which is limited to P-cores in Intel's high-end processors, AVX10 introduces 512-bit SIMD capabilities to both P-cores and E-cores across all future Intel processors, broadening its applicability. AVX10.1, scheduled for release with the already launched Granite Rapids in Q3 2024, will enumerate AVX-512 instructions at 128, 256, and 512-bit vector lengths for software pre-enabling. This ensures that applications will run on any future processor supporting AVX10.1 or higher while maintaining compatibility with these vector lengths. A unified AVX10 with at least 256-bit vectors will be supported by all Intel processors. With ongoing updates to specifications and intrinsics (including support in GCC), AVX10 retains AVX-512's zmm (512-bit) registers, offering potential microarchitectural performance improvements for a wide range of applications. Although all the intrinsics we used in the AVX-512 implementation for addition, subtraction, and multiplication are natively supported in AVX10 through GCC (including the most recent version, 15), the performance of the code remains specific to the microarchitecture. Due to the current timeline of our work, we could not acquire an AVX-10 CPU in a bare-metal form. We anticipate similar or improved performance from the upcoming x86-64-based CPUs that support AVX-10.

Overall: GMP utilizes micro-architecture fine-tuned assembly instructions rather than relying solely on the compiler to generate assembly code from C. This approach allows them to take advantage of several newer instructions that can capitalize on the capabilities of the latest CPUs, which a general-purpose compiler might not optimize for to avoid complications. However, we were still able to get performance benefits with GCC-compiled code with AVX512 intrinsics, and we believe that writing micro-architecture fine-tuned assembly routines utilizing the AVX512 instructions could improve performance further. From a performance standpoint, we have achieved several benefits through the use of AVX512. Specifically, we can process eight 64-bit elements simultaneously for addition and subtraction. Additionally, our modified algorithm attempts to decompose dependencies into multiple phases to enhance parallel processing. For multiplication, we utilized IFMA (Integer Fused Multiply-Add), allowing us to process eight 52-bit elements while also passing other prepared values to be added within the same cycle. However, regarding cache performance, AVX typically fetches data from the memory hierarchy. Since we have aligned our data representation exactly with GMP (GNU Multiple Precision Arithmetic Library), we did not take advantage of cache benefits compared to GMP.

Suggestions for Micro-architectural Changes for Better Performance: When working with AVX registers, the best approach is to perform all computations using these registers and then switch back to the general-purpose registers. This strategy helps avoid the latency penalty associated with transitioning between AVX registers and general-purpose registers. However, due to the current micro-architectural design, during the addition and subtraction operations, we often need to manipulate the masking bits stored in the mask registers. This operation requires transferring these values to general-purpose registers, which incurs additional latency. It would be more efficient if we could operate on these masks directly within the AVX registers or utilize a separate set of mask registers within the AVX ALU. This change could help minimize the latency penalty.

Currently, x86-64-based CPUs do not have any AVX512 instructions that can horizontally add up certain values inside the AVX registers efficiently. If that becomes available, we can drastically reduce the timings of multiplication.

Also, movement of data across lanes is quite complex, as we do not have a single set of intrinsics with AVX512 that can perform that efficiently. The programmer needs to manually perform them with a combination of multiple intrinsics, which may be inefficient sometimes, if not written properly.

One would hope that SIMD registers can perform computations at their full vector size. For instance, when working with AVX512, we could load 512 bits and carry out an addition on all 512 bits simultaneously. The underlying hardware would automatically manage carry propagation, minimizing the need for manual software implementation within the 512 bits.

Some Lessons Learned: **Write the code wisely:** Even with a single if statement within a loop, it can drastically affect the performance. We may use LIKELY or UNLIKELY constructs, if the operations inside the loop are large enough or switch to logical operations for some small set of operations inside a loop, if applicable. Additionally, the types and sizes of variables can affect your code's performance. Optimization flags such as O1, O2, or higher may sometimes degrade performance, so it's essential to test with all available flags and choose the one that optimizes your code effectively. We have observed that with GCC using O2 optimization, it can sometimes rearrange AVX intrinsics in a way that leads to conflicts or pipeline stalls. **Benchmarking the code:** Sometimes, with a bug in the code, the compiler omits some of the unreachable basic blocks within your code, which may increase the performance. Cross-check with the generated assembly to verify its correctness. Certainly, try to use standard tools for profiling; GNU standards would be preferred, and some tools could even be provided by the hardware vendor. And, do not profile your code on a VM instance, at least for parallelized codes, try to get a bare-metal CPU.

7.2 Limitations

Our benchmarking was conducted on a single CPU (Intel Xeon E-2314, based on the Rocket Lake microarchitecture) due to limited hardware availability. It is possible that different microarchitectures might yield different behaviours. For example, during addition and subtraction, Phase 4 relied heavily on masked registers, which could perform more slowly on other architectures. Nevertheless, our implementation may match the performance of GMP for those cases. Note that these mask register-based computations are predominantly placed within conditional if-blocks that are rarely executed (in our tests, they were never executed).

We could not compare our performance metrics with certain prior works [RSS23] due to missing implementation details and the unavailability of corresponding code. Moreover, there's very limited research on parallelizing addition and subtraction operations, largely because of their chained dependency.

Our SIMD vectorization analysis focused exclusively on AVX512. In the context of multiplication, the VBAP MUL was implemented with a fixed number of limbs (five) without generalizing the implementation to accommodate an arbitrary number of limbs.

We did not integrate addition and subtraction operations into any existing cryptographic library to test under real workloads. This is mainly because these operations are used indirectly (for instance, within multiplication algorithms like Karatsuba and other modulo arithmetic operations) rather than directly in use cases like cryptography tools. We have begun implementing the hybrid KVBAP Multiplication approach, where we aim to use the PML Add and Sub in each recursion for addition and subtraction. Our initial experiments indicated some performance benefits from this approach.

Chapter 8

Conclusion and Future Scope

Nevertheless, we benefit from using vectorization techniques combined with well-designed parallel algorithms for large-number addition, subtraction, and multiplication, consistent with existing works' findings. Our implementation often outperforms previous methods for large-number addition and subtraction. While earlier studies observed speed-ups over GMP only for multiplication on operands larger than 1024 bits, our base case implementation using the Vedic approach showed improvements even at 256 bits. With a fully optimized implementation, we anticipate further performance gains.

In future, we plan to extend the parallel algorithms to additional SIMD instruction sets, such as the upcoming AVX10 on Intel (with planned support for Granite Rapids and Diamond Rapids, promising to integrate all the sets of AVX512 instructions into one simplified set with 512+ bits support) Neon, SVE, and SVE2 on ARM, as well as RVV on RISC-V. For instance, SVE (Scalable Vector Extension) and RVV (RISC-V Vector Extension) offer scalable, implementation-defined vector sizes compared to the fixed sizes typical of x86-64 machines, which may enable further optimizations.

In this work, we only targeted addition, subtraction, and multiplication on large numbers, but there are a few more foundational operations, such as division, modulo arithmetic, and factorization. Vedic mathematics has an exhaustive set of sutras in this context, and no prior study has analyzed their efficacy for large-number arithmetic. For instance, the same Urdhva-Tiryagbhyam technique is also applicable to division. For higher-order Toom-Cook multiplication, we came across solving a system of equations, and Vedic mathematics has sutras, including Paravartya Yojayet and Sankalana-vyavakalanabhyam for solving a system of equations, which might be explored there for solving large-number multiplication itself.

We may also explore efficient multi-threading for sufficiently larger operands (unlike the earlier works on operand sizes ranging between 6.4×10^6 to 6.4×10^{10} bits) to evaluate its potential benefits. Although we initially experimented with Pthreads and OpenMP on the Intel Xeon E-2314, we observed no performance gains for this work's targeted operand sizes, resulting in slower execution. However, a more refined implementation might yield positive results.

We are in our initial phase of the hybrid KVABP implementation for variable-sized operands, with results expected in the near future. One may also investigate offloading the parallel computations to a GPU to evaluate the performance; we have not yet tried that out with our algorithms, but there are a few works [EW11] for multiplication that yielded the benefits with GPU, but on 255K bits to 24.512M bit operand sizes.

We are also trying to address the inefficiencies in the 256-bit base case VBAP multiplication, specifically the serial parts involving the addition of lower segments and the adjustment of carry-overs. These limitations are primarily attributed to current hardware capabilities in handling conditional horizontal additions within AVX registers. We have not explored the other SIMD instruction sets and whether we can benefit from using them or not. Also, we may consider utilizing SIMD assembly instructions instead of intrinsics. A carefully tuned, assembly-level approach tailored to specific microarchitectures could potentially provide substantial speed-ups.

Bibliography

[3dN] Wiki 3dNow. *3DNow! - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/3DNow!. [Accessed 16-03-2025].

[AKL18] Haider A.F. Almurib, Thulasiraman Nandha Kumar, and Fabrizio Lombardi. “Approximate DCT Image Compression Using Inexact Computing”. In: *IEEE Transactions on Computers* 67.2 (2018), pp. 149–159. DOI: 10.1109/TC.2017.2731770.

[Ala+22] Gorjan Alagic et al. “Status report on the third round of the NIST post-quantum cryptography standardization process”. In: (2022).

[ARM] ARM. *DSP & SIMD*. https://web.archive.org/web/20160318193125/https://www.arm.com/products/processors/technologies/dsp-simd.php. [Accessed 16-03-2025].

[Asa+06] Krste Asanovic et al. “The landscape of parallel computing research: A view from berkeley”. In: (2006).

[AVXa] Wiki AVX. *Advanced Vector Extensions - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions. [Accessed 16-03-2025].

[AVXb] Wiki AVX512. *AVX-512 - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/AVX-512. [Accessed 16-03-2025].

[Bai05] D.H. Bailey. “High-precision floating-point arithmetic in scientific computation”. In: *Computing in Science & Engineering* 7.3 (2005), pp. 54–61. DOI: 10.1109/MCSE.2005.52.

[Bar20] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. https://doi.org/10.6028/NIST.SP.800-57pt1r5. [Accessed 13-03-2025]. 2020.

[BB15] David H. Bailey and Jonathan M. Borwein. “High-Precision Arithmetic in Mathematical Physics”. In: *Mathematics* 3.2 (2015), pp. 337–367. ISSN: 2227-7390. DOI: 10.3390/math3020337. URL: https://www.mdpi.com/2227-7390/3/2/337.

[BBB12] D.H. Bailey, R. Barrio, and J.M. Borwein. “High-precision computation: Mathematical physics and dynamics”. In: *Applied Mathematics and Computation* 218.20 (2012), pp. 10106–10121. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2012.03.087. URL: https://www.sciencedirect.com/science/article/pii/S0096300312003505.

[Bed62] O. J. Bedrij. “Carry-Select Adder”. In: *IRE Transactions on Electronic Computers* EC-11.3 (1962), pp. 340–346. DOI: 10.1109/IRETELC.1962.5407919.

[Ber+15] Daniel J Bernstein et al. “SPHINCS: practical stateless hash-based signatures”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 368–397.

[Ber06] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer. 2006, pp. 207–228.

[BL17] Daniel J Bernstein and Tanja Lange. “Post-quantum cryptography”. In: *Nature* 549.7671 (2017), pp. 188–194.

[Bos+18] Joppe Bos et al. “CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 353–367.

[CA69] Stephen A Cook and Stål O Aanderaa. “On the minimum computation time of functions”. In: *Transactions of the American Mathematical Society* 142 (1969), pp. 291–314.

[Cha99] Lin Chao. *Intel Technology Journal Q2*. https://www.intel.com/content/dam/www/public/us/en/documents/research/1999-vol03-iss-2-intel-technology-journal.pdf. [Accessed 16-03-2025]. 1999.

[Cof] Neil Coffey. *RSA key lengths — javamex.com*. https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml. [Accessed 12-03-2025].

[com] The kernel development community. *Transparent Hugepage Support 2014; The Linux Kernel documentation — kernel.org*. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html. [Accessed 17-03-2025].

[Con24] Linux perf wiki Contributors. *perf: Linux profiling with performance counters — perfwiki.github.io*. https://perfwiki.github.io/main/. [Accessed 20-03-2025]. 2024.

[Coo00] Intel Cooperation. *Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications*. Tech. rep. Technical Report, 2000.

[DH22] Whitfield Diffie and Martin E Hellman. “New directions in cryptography”. In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 2022, pp. 365–390.

[Did+24] Laurent-Stéphane Didier et al. “Truncated multiplication and batch software SIMD AVX512 implementation for faster Montgomery multiplications and modular exponentiation”. In: *IACR Communications in Cryptology* 1.3 (Oct. 7, 2024). ISSN: 3006-5496. DOI: 10.62056/a3txl86bm.

[Doca] Mozilla JS Docs. *BigInt - JavaScript | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt. [Accessed 12-03-2025].

[Docb] Oracle Java Docs. *BigInteger (Java Platform SE 8) — docs.oracle.com*. https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html. [Accessed 12-03-2025].

[Docc] Rust Docs. *num_bigint - Rust — docs.rs*. https://docs.rs/num-bigint/latest/num_bigint/. [Accessed 12-03-2025].

[Docd] Rust Docs. *rug - Rust — docs.rs*. https://docs.rs/rug/latest/rug/. [Accessed 12-03-2025].

[Dup+19] Dmitry Duplyakin et al. “The Design and Operation of CloudLab”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[Erb+20] Andres Erbsen et al. "Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises". In: *SIGOPS Oper. Syst. Rev.* 54.1 (Aug. 2020), pp. 23–30. ISSN: 0163-5980. DOI: 10.1145/3421473.3421477. URL: https://doi.org/10.1145/3421473.3421477.

[ET18] Takuya Edamatsu and Daisuke Takahashi. "Acceleration of Large Integer Multiplication with Intel AVX-512 Instructions". In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2018, pp. 211–218. DOI: 10.1109/HPCC/SmartCity/DSS.2018.00059.

[ET20] Takuya Edamatsu and Daisuke Takahashi. "Accelerating Large Integer Multiplication Using Intel AVX-512IFMA". In: *Algorithms and Architectures for Parallel Processing*. Ed. by Sheng Wen, Albert Zomaya, and Laurence T. Yang. Cham: Springer International Publishing, 2020, pp. 60–74. ISBN: 978-3-030-38991-8.

[ET23] Takuya Edamatsu and Daisuke Takahashi. "Efficient Large Integer Multiplication with Arm SVE Instructions". In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. HPCAsia '23. Singapore, Singapore: Association for Computing Machinery, 2023, pp. 9–17. ISBN: 9781450398053. DOI: 10.1145/3578178.3578193. URL: https://doi.org/10.1145/3578178.3578193.

[EW11] Niall Emmart and Charles Weems. "High Precision Integer Multiplication with a GPU". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1781–1787. DOI: 10.1109/IPDPS.2011.336.

[Fly66] M.J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[Fou+18] Pierre-Alain Fouque et al. "Falcon: Fast-Fourier lattice-based compact signatures over NTRU". In: *Submission to the NIST's post-quantum cryptography standardization process* 36.5 (2018), pp. 1–75.

[Fre10] Gerhard Frey. "The arithmetic behind cryptography". In: *Notices of the AMS* 57.3 (2010), pp. 366–374.

[GK12] Shay Gueron and Vlad Krasnov. "Software implementation of modular exponentiation, using advanced vector instructions architectures". In: *Arithmetic of Finite Fields: 4th International Workshop, WAIFI 2012, Bochum, Germany, July 16-19, 2012. Proceedings 4*. Springer. 2012, pp. 119–135.

[GK16] Shay Gueron and Vlad Krasnov. "Accelerating big integer arithmetic using Intel IFMA extensions". In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE. 2016, pp. 32–38.

[GMP] GMPbench. *GMPbench results — gmplib.org*. https://gmplib.org/gmpbench. [Accessed 21-03-2025].

[GNU91] GNU Project. *The GNU MP Bignum Library — gmplib.org*. https://gmplib.org/. [Accessed 03-03-2025]. 1991.

[Gra08] Ananth Grama. *An Introduction to Parallel Computing: Design and Analysis of Algorithms, 2/e*. Pearson Education India, 2008.

[Hel79] Martin E. Hellman. "The Mathematics of Public-Key Cryptography". In: *Scientific American* 241.2 (1979), pp. 146–157. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24965269 (visited on 03/12/2025).

[Hou] Mike Housch. *The Current Encryption Landscape: The Need For 3072-Bit Keys — forbes.com*. https://www.forbes.com/councils/forbestechcouncil/2024/02/23/the-current-encryption-landscape-the-need-for-3072-bit-keys/. [Accessed 12-03-2025].

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[IBM] Linux on IBM Systems. *Common Cryptographic Architecture (CCA): ECC key token — ibm.com*. https://www.ibm.com/docs/en/linux-on-systems?topic=formats-ecc-key-token. [Accessed 13-03-2025].

[Inc22] The MathWorks Inc. *Symbolic Math Toolbox*. Natick, Massachusetts, United States, 2022. URL: https://in.mathworks.com/products/symbolic.html.

[Inc97] Intel Inc. *Intel Introduces The Pentium Processor With MMX Technology; Technology — intel.com*. https://www.intel.com/pressroom/archive/releases/1997/dp010897.htm. [Accessed 16-03-2025]. 1997.

[Int] Intel. *Intel® Advanced Vector Extensions 10.1 (Intel® AVX10.1) Architecture Specification — intel.com*. https://www.intel.com/content/www/us/en/content-details/848455/intel-advanced-vector-extensions-10-1-intel-avx10-1-architecture-specification.html. [Accessed 30-04-2025].

[Int24] IntelIntrins. *Intel® Intrinsics Guide — intel.com*. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. [Accessed 05-03-2025]. 2024.

[Jel] Jakub Jelínek. *Vectorization optimization in GCC | Red Hat Developer — developers.redhat.com*. https://developers.redhat.com/articles/2023/12/08/vectorization-optimization-gcc. [Accessed 17-03-2025].

[Jia+20] Honglan Jiang et al. "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications". In: *Proceedings of the IEEE* 108.12 (2020), pp. 2108–2135. DOI: 10.1109/JPROC.2020.3006451.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[Joh] Fredrik Johansson. *mpmath - Python library for arbitrary-precision floating-point arithmetic — mpmath.org*. https://mpmath.org/. [Accessed 12-03-2025].

[Kar] GMP Karatsuba. *Karatsuba Multiplication (GNU MP 6.3.0) — gmplib.org*. https://gmplib.org/manual/Karatsuba-Multiplication. [Accessed 14-03-2025].

[Kar63] Anatolii Karatsuba. "Multiplication of multidigit numbers on automata". In: *Soviet physics doklady*. Vol. 7. 1963, pp. 595–596.

[KM14] Anastasis Keliris and Michail Maniatakos. “Investigating large integer arithmetic on Intel Xeon Phi SIMD extensions”. In: *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2014, pp. 1–6.

[Knu97] Donald Ervin Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997.

[KS73] Peter M. Kogge and Harold S. Stone. “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”. In: *IEEE Trans. Comput.* 22.8 (Aug. 1973), pp. 786–793. ISSN: 0018-9340. DOI: 10.1109/TC.1973.5009159. URL: https://doi.org/10.1109/TC.1973.5009159.

[Lee06] E.A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

[Lyu+20] Vadim Lyubashevsky et al. “Crystals-dilithium”. In: *Algorithm Specifications and Supporting Documentation* (2020).

[MAH92] BHARATI KRSNA TIRTHAJI MAHARAJA. *Vedic Mathematics Bharati Krishna Tirth Ji Maharaj : Bharati Krishna Tirth Ji Maharaj : Free Download, Borrow, and Streaming : Internet Archive — archive.org*. https://archive.org/details/vedic-mathematics-bharati-krishna-tirth-ji-maharaj/page/n7/mode/2up. [Accessed 05-03-2025]. 1992.

[Man22] Intel Developers Manual. *Explicit Vector Programming – Best Known Methods — intel.com*. https://www.intel.com/content/www/us/en/developer/articles/training/explicit-vector-programming-best-known-methods.html. [Accessed 17-03-2025]. 2022.

[man24a] Linux man-pages. *getrusage(2) - Linux manual page — man7.org*. https://www.man7.org/linux/man-pages/man2/getrusage.2.html. [Accessed 20-03-2025]. 2024.

[man24b] Linux man-pages. *perf_event_open(2) - Linux manual page — man7.org*. https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html. [Accessed 20-03-2025]. 2024.

[Max] Maxima. *Maxima – GPL CAS based on DOE-MACSYMA — maxima.sourceforge.io*. https://maxima.sourceforge.io/. [Accessed 12-03-2025].

[Mil86] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology — CRYPTO ’85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 978-3-540-39799-1.

[Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel White Paper. [Accessed 21-03-2025]. 2010.

[Pau] McKenney Paul E. *Is Parallel Programming Hard, And, If So, What Can You Do About It? — cdn.kernel.org*. https://cdn.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html. [Accessed 12-03-2025].

[Pro] GNU Project. *The GNU MPFR Library — mpfr.org*. https://www.mpfr.org/. [Accessed 12-03-2025].

[PYP] PYPI. *gmpy2 2.2.1 — pypi.org*. https://pypi.org/project/gmpy2/. [Accessed 12-03-2025].

[REU97] REUTERS. *Makers Unveil PCs With Intel&apos;s MMX Chip — archive.nytimes.com*. https://archive.nytimes.com/www.nytimes.com/library/cyber/week/010997intel.html. [Accessed 16-03-2025]. 1997.

[Rig20] Erik Rigtorp. *Using huge pages on Linux — rigtorp.se*. https://rigtorp.se/hugepages/. [Accessed 17-03-2025]. 2020.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[RSS23] Pengchang Ren, Reiji Suda, and Vorapong Suppakitpaisarn. “Efficient Additions and Montgomery Reductions of Large Integers for SIMD”. In: *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*. 2023, pp. 48–59. DOI: 10.1109/ARITH58626.2023.00034.

[Sag] SageMath. *SageMath Mathematical Software System - Sage — sagemath.org*. https://www.sagemath.org/. [Accessed 12-03-2025].

[Sam22] Vivien Samuel. “Parallel integer multiplication”. In: *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2022, pp. 100–107. DOI: 10.1109/PDP55904.2022.00024.

[SIM] GNU MP SIMD. *Assembly SIMD Instructions (GNU MP 6.3.0) — gmplib.org*. https://gmplib.org/manual/Assembly-SIMD-Instructions. [Accessed 12-03-2025].

[Ski08] Steven S Skiena. *The algorithm design manual*. Vol. 2. Springer, 2008.

[SS71] Arnold Schönhage and Volker Strassen. “Fast multiplication of large numbers”. In: *Computing* 7 (1971), pp. 281–292.

[SSEa] Wiki SSE2. *SSE2 - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/SSE2. [Accessed 16-03-2025].

[SSEb] Wiki SSE3. *SSE3 - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/SSE3. [Accessed 16-03-2025].

[SSEc] Wiki SSE4. *SSE4 - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/SSE4. [Accessed 16-03-2025].

[Ste16] Nigel Stephens. *The Scalable Vector Extension for Armv8-A — community.arm.com*. https://community.arm.com/arm-community-blogs/b/servers-and-cloud-computing-blog/posts/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture. [Accessed 16-03-2025]. 2016.

[Teaa] GCC Team. *Other Built-in Functions Provided by GCC*. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. [Accessed 17-03-2025].

[Teab] SSL Support Team. *New Minimum RSA Key Size for Code Signing Certificates - SSL.com — ssl.com*. https://www.ssl.com/blogs/new-minimum-rsa-key-size-for-code-signing-certificates/. [Accessed 13-03-2025].

[Tom] Mikko Tommila. *Apfloat - Arbitrary precision library for Java and C++, applets and calculator — apfloat.org*. http://www.apfloat.org/. [Accessed 12-03-2025].

[Too] GMP Toom-3. *Toom 3-Way Multiplication (GNU MP 6.3.0) — gmplib.org*. https://gmplib.org/manual/Toom-3_002dWay-Multiplication. [Accessed 14-03-2025].

[Too63] Andrei L Toom. *The complexity of a scheme of functional elements realizing the multiplication of integers, published in Soviet Math (translations of Dokl. Adad. Nauk. SSSR), 4*. 1963.

[Tow22] Daniel Towner. *Intel Advanced Vector Extensions 512 (Intel AVX-512) - Permuting Data Within and Between AVX Registers*. `https://builders.intel.com/docs/networkbuilders/intel-avx-512-permuting-data-within-and-between-avx-registers-technology-guide-1668169807.pdf`. [Accessed 16-03-2025]. 2022.

[Upa] Upavidhi. *Vedic Mathematics: Sūtra 3. ūrdhva tiryagbhyāṃ — web.archive.org*. `https://web.archive.org/web/20230605040310/http://upavidhi.com/sutra/urdhva-tiryagbhyam`. [Accessed 15-03-2025].

[Wik] WikiMersenne. *Mersenne Twister - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Mersenne_Twister`. [Accessed 03-04-2025].

[Wol] Wolfram. *Wolfram Mathematica: Modern Technical Computing — wolfram.com*. `https://www.wolfram.com/mathematica/`. [Accessed 12-03-2025].

[Yee19] Alexander Yee. *Integer Addition and Carryout — numberworld.org*. `http://numberworld.org/y-cruncher/internals/addition.html`. [Accessed 03-03-2025]. 2019.

[Zha] Jun Zhang. *Efficiency of large integer multiplication algorithms: A comparative study of traditional methods and Karatsuba&apos;s algorithm | Applied and Computational Engineering — ewadirect.com*. `https://www.ewadirect.com/proceedings/ace/article/view/13474`. [Accessed 05-03-2025].

Appendix A

Measurement Techniques

A.1 Timing Measurements

A.1.1 RUSAGE

In Linux systems, `getrusage` [man24a] is a system call that provides resource usage statistics for the calling process. It is used to measure the resources used by the process, like CPU time, memory usage, etc. We used this system call to measure the CPU time used by our code. POSIX.1 specifies `getrusage` but defines only the fields `ru_utime` and `ru_stime`. For our benchmarking purposes, along with other tools PERF, RDTSCP, and Timespec (`clock_gettime`), we also utilized `ru_utime` and `ru_stime` to measure user CPU time and system CPU time, respectively.

Example A.1.1.1.

```
// Function to measure CPU time in microseconds as a long double
static inline long double cputime()
{
    struct rusage rus;
    getrusage(RUSAGE_SELF, &rus);
    return rus.ru_utime.tv_sec * 1000000.0L + rus.ru_utime.tv_usec;
}
```

A.1.2 RDTSC

To measure hardware ticks/CPU cycles accurately while operating on the actual CPU (not emulators) without much overhead (which tools like PERF add), we initially used RDTSC ticks. However, it is well-known that RDTSC does not provide accurate measurements in cases of code cross-contamination due to out-of-order execution. To address this, we explored proper benchmarking techniques using RDTSC. We found a white paper by Intel that explains how to measure ticks accurately using a combination of CPUID, RDTSC, and RDTSCP instructions. You can find the white paper here: [Pao10]. Below, we have mentioned the code that we utilized for measuring start and ending functions based on the white paper.

Example A.1.2.1.

```
static inline unsigned long long measure_rdtsc_start()
{
    unsigned cycles_low, cycles_high;
    unsigned long long ticks;
    asm volatile("CPUID\n\t"
                 "RDTSC\n\t"
                 "mov %%edx, %0\n\t"
                 "mov %%eax, %1\n\t" :
```

```
                 "=r"(cycles_high), "=r"(cycles_low)::
                 "%rax", "%rbx", "%rcx", "%rdx");
    ticks = (((unsigned long long)cycles_high << 32) | cycles_low);
    return ticks;
}

static inline unsigned long long measure_rdtscp_end()
{
    unsigned cycles_low, cycles_high;
    unsigned long long ticks;
    asm volatile("RDTSCP\n\t"
                 "mov %%edx, %0\n\t"
                 "mov %%eax, %1\n\t"
                 "CPUID\n\t" : "=r"(cycles_high), "=r"(cycles_low)::
                               "%rax","%rbx", "%rcx", "%rdx");
    ticks = (((unsigned long long)cycles_high << 32) | cycles_low);
    return ticks;
}
```

We used this benchmarking methodology throughout our work to report relative ticks.

A.1.3 Timespec

In addition to `getrusage` and RDTSC-based methods, we employed the `clock_gettime` function with the `CLOCK_MONOTONIC_RAW` clock to measure elapsed wall-clock time with high precision for our benchmarking. This system call, available on POSIX-compliant systems like Linux, provides timing measurements in nanoseconds via the `struct timespec`, which includes seconds (`tv_sec`) and nanoseconds (`tv_nsec`) fields. We selected `CLOCK_MONOTONIC_RAW` to ensure a monotonically increasing clock unaffected by system clock updates or adjustments, offering a reliable measure of real-time elapsed between two points in our code execution. The following functions were implemented to capture and compute time differences: `get_timespec` retrieves the current timestamp, and `diff_timespec_us` calculates the difference between two timestamps in microseconds, handling nanosecond underflows to maintain accuracy. This approach complements our CPU-focused measurements by providing a precise wall-clock perspective, which we used alongside `PERF`, `RDTSC`, and `getrusage` to evaluate performance comprehensively.

Example A.1.3.1.

```
static inline struct timespec get_timespec()
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
    return ts;
}


static inline long diff_timespec_us(struct timespec start,
                                    struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec - start.tv_nsec) < 0)
    {
        temp.tv_sec = end.tv_sec - start.tv_sec - 1;
        temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
    }
    else
    {
        temp.tv_sec = end.tv_sec - start.tv_sec;
        temp.tv_nsec = end.tv_nsec - start.tv_nsec;
    }
    // return in microseconds
    return (temp.tv_sec * 1000000000 + temp.tv_nsec) / 1000;
}
```

A.1.4 Measuring Execution Time and Number of Operations

To get a stable and trustworthy average execution time for micro-benchmarking, we have used the averaging technique from GMPbench [GMP].

To get the average execution time:

Example A.1.4.1.

```
#define TIME(t, func)                        \
  do                                         \
  {                                          \
    long int __t0, __times, __t, __tmp;      \
    __times = 1;                             \
    {                                        \
      func;                                  \
    }                                        \
    do                                       \
    {                                        \
      __times <<= 1;                         \
      __t0 = cputime();                      \
      for (__t = 0; __t < __times; __t++)    \
      {                                      \
        func;                                \
      }                                      \
      __tmp = cputime() - __t0;              \
    } while (__tmp < 250000);                \
    (t) = (double)__tmp / __times;           \
  } while (0)
```

The macro `TIME(t, func)` computes the average CPU time required to execute a specified function or code block `func` by calibrating the CPU speed to ensure accurate timing. It begins with a single execution of `func` as a warm-up to account for initial overhead and then enters a loop where the iteration count `__times` doubles with each cycle until the total elapsed CPU time, measured using `cputime()`, which extracts CPU time using `rusage`, exceeds 250 milliseconds. Within this loop, `func` is executed repeatedly for the current iteration count, and once the threshold is met, the macro computes the average CPU time per execution by dividing the total elapsed time by the number of iterations. Note, not just `cputime()`, for verification across all the timing measurement techniques, we replaced `cputime()` with RDTSC- and Timespec-based measurement functions. But for reporting the stats, we used to `cputime()`.

Based on the computed average time using `TIME`, we have again resorted to GMPbench with the following code to determine a suitable iteration count and compute the performance metric:

Example A.1.4.2.

```
TIME(t, func(z, x, y));

niter = 1 + (unsigned long)(1e4 / t);

t0 = cputime();
for (i = niter; i > 0; i--)
{
  func(z, x, y);
}
ti = cputime() - t0;

ops_per_sec = 1000.0 * niter / ti;
f = 100.0;
for (decimals = 0;; decimals++)
{
  if (ops_per_sec > f)
    break;
  f = f * 0.1;
}

printf("RESULT: %.*f operations per second\n", decimals, ops_per_sec);
```

A.2 Code Profiling

A.2.1 PERF

PERF [Con24] is a powerful performance analysis tool in Linux, widely used for profiling applications and understanding performance metrics. It can capture a range of data points, such as CPU cycles, user/kernel instructions, page faults, cache misses, and more.

In our work, we have leveraged PERF to gain deep insights into how our code behaves under different conditions. Instead of relying solely on the usual perf through the shell, we're also using the `perf_event_open` [man24b] system call to measure performance at a more granular level. This allows us to target specific code fragments and gather precise performance data, offering more flexibility and control over the profiling process.

The $perf\_event\_open$ system call does not reliably reflect the L1D read counts for AVX instructions, possibly due to the separation of AVX execution units from the main ALU. However, it accurately tracks L1D misses, likely because miss events are handled through the main CPU pipeline. Given that both the baseline and vectorized implementations operate on similar data elements with comparable access patterns, we use the L1D read counts from the baseline as a reference for analysis. Additionally, to account for performance measurement overhead, we subtract the perf overhead by measuring the stats over an empty function call.

Appendix B

Miscellaneous Optimization Techniques

B.1 Auto-Vectorization

Vectorization in a C/C++ program can be explicit or implicit. For explicit vectorization, we use SIMD intrinsic calls like AVX-512 intrinsics [Int24] directly in the code. Alternatively, implicit vectorization [Jel] of loops avoids manual intrinsics by setting up the code for compilers like GCC or ICX to auto-vectorize with the right flags. This needs three things: proper compiler flags, aligned data allocation, and alignment hints to the compiler. In GCC, auto-vectorization requires $-O2$ or higher flags. By default, it uses 128-bit vectors; for 256-bit or 512-bit vectorization, add $-mavx2$ or $-mavx512f$ flags. Since AVX-512 includes multiple flag variants, each must be specified separately in GCC. However, in some cases, auto-vectorized code by the compiler may not be as efficient as a programmer can achieve using careful explicit vectorization, but it can still be a helpful feature for many users.

B.2 Aligned Memory Allocation for Vectorization

Intel documentation [Man22] recommends 16-byte alignment for SSE, 32-byte for AVX and AVX2, and 64-byte for AVX-512. It suggests using $\_mm\_malloc(ptr, < alignment - size >)$ and $\_mm\_free(ptr)$ for memory management. Afterwards, using the $\_\_assume\_aligned$ attribute to inform the compiler of data alignment. Misaligned data may prevent vectorization. To verify vectorization in GCC, use the $-ftree - vectorizer - verbose = 2$ flag for detailed output or the $-fopt - info - vec$ flag for a vectorization report.

Aligned allocation is key for performance in both implicit and explicit vectorization. With AVX-512F aligned loads or stores, the starting memory address must be 64-byte aligned to avoid segmentation faults. If not properly aligned, unaligned loads/stores can be used, but this may degrade performance.

B.3 Transparent Huge Pages (THP)

Huge pages reduce TLB misses with sizes larger than the default 4 KiB. Linux transparent huge page [com] (THP) support lets the kernel automatically promote regular memory pages into huge pages. Linux's Transparent Huge Page (THP) support [com] lets the kernel promote regular pages to huge pages automatically. We can advise the kernel to allocate 2 MiB or 1 GiB huge pages (depending on system support) using *madvise* calls from C/C++ code. However, after allocation, the kernel won't recognize it as a THP until the memory is initialized or accessed. The basic idea of using THP for this work would be that if we were to work with a big workload of large numbers requiring a lot of memory, we may try to pre-allocate a large chunk using THP, which can help avoid frequent page faults and TLB misses within the memory segment. Some tutorials and the effectiveness of THP have been listed on this blogpost [Rig20].

B.4 Branching Optimization

We can avoid *if* or conditional branching inside loops since it slows execution a lot. When some branches rarely occur, we can hint the compiler with *LIKELY* and *UNLIKELY* macros, using GCC's *__builtin_expect* [Teaa], to improve branch prediction. These macros provide a hint to the compiler about the expected outcome of a condition, allowing it to optimize instruction ordering for better performance. For example, consider a scenario where a function call happens only in rare error conditions:

Example B.4.0.1.

```
/* Define macros for branch prediction */
#define LIKELY(x)   __builtin_expect(!!(x), 1) // Condition is usually true
#define UNLIKELY(x) __builtin_expect(!!(x), 0) // Condition is rarely true

/* Process array elements with rare error logging */
void process_array(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        if (UNLIKELY(arr[i] < 0)) {
            /* Rare function call of log_error */
            log_error("Negative value detected", arr[i]);
        }
        arr[i] = arr[i] * 2;  // Normal case: double the value
    }
}
```