Parallel Comba Multiplication for Many-Core Architectures (Postprint)

Authors: Huang Haoran, Xu Jiangfeng

Date: 2018-05-24T00:00:00+00:00

Abstract

To leverage the performance advantages of many-core processors and solve larger-scale problems, this study investigates the parallelization of large integer multiplication on many-core processors. Based on an analysis of the parallelism characteristics of pencil-and-paper multiplication and Comba multiplication, multiple solutions are proposed to address the load balancing issues encountered during the parallelization of Comba multiplication. Subsequently, considering the architectural features of the SW26010 processor, an improved Comba multiplication approach inspired by pencil-and-paper multiplication is adopted, with the implementation incorporating optimization techniques such as vectorization and register communication. Experimental results demonstrate that the improved parallel Comba algorithm exhibits favorable parallelism and can effectively harness the performance advantages of the SW26010 many-core processor.

Full Text

Preamble

Title: Research on Parallel Comba Multiplication for Many-Core Architectures

Authors: Huang Haoran, Xu Jiangfeng
Affiliation: College of Information Engineering, Zhengzhou University, Zhengzhou 450001, China

Abstract

To exploit the performance advantages of many-core processors and solve larger-scale problems, this paper investigates the parallelization of large integer multiplication on many-core processors. Based on an analysis of the parallelism in

written multiplication and Comba multiplication, we propose multiple solutions to address the load balancing issues faced when parallelizing Comba multiplication. Then, according to the architectural characteristics of SW26010, we select an improved Comba multiplication method derived from written multiplication, employing optimization techniques such as vectorization and register communication in the implementation. Test results demonstrate that the improved parallel Comba algorithm exhibits good parallelism and can effectively leverage the performance advantages of the SW26010 many-core processor.

Keywords: large integer multiplication; Comba algorithm; many-core processor; parallelization; load balancing

1. Introduction

Research on arithmetic operations has always been a crucial direction in the development of human science, as many scientific problems depend on accurate and fast arithmetic computations. In computing, arithmetic operations form the foundation for implementing various applications, and studying efficient algorithms is of great significance for solving scientific problems using computers. Among the four basic arithmetic operations—addition, subtraction, multiplication, and division—multiplication, particularly large integer multiplication, finds extensive applications in cryptographic systems and large-scale scientific computing. As technology advances, the scale of problems requiring solution continues to grow, along with precision requirements, leading to increasingly higher performance demands for large integer multiplication.

For decades, numerous studies on large integer multiplication have yielded many results, forming various computational methods. These algorithms can be broadly categorized into basic multiplication, divide-and-conquer multiplication, FFT (fast Fourier transform) multiplication, and number-theoretic multiplication. Basic algorithms primarily consist of traditional written multiplication and its improvements. In written multiplication, each digit of one multiplier must multiply every digit of the other, resulting in a time complexity of $O(n^2)$ for two n-digit integers. Although Comba' s algorithm, proposed in 1990, performs the same number of multiplications as written multiplication, it computes column-wise and reduces carry processing from n^2 to 2n operations, making it widely used in PC encryption systems. Basic algorithms are simplest to understand and implement.

Divide-and-conquer multiplication, FFT multiplication, and number-theoretic multiplication reduce complexity below $O(n^2)$. Divide-and-conquer is an important method for solving large-scale problems. Karatsuba used a binary recursive approach to reduce large integer multiplication complexity to O(n^1.585), with relatively simple implementation and low overhead, finding extensive use in RSA and other public-key cryptosystems. Toom extended this divide-and-conquer recursion by splitting large integers into k parts (k 2), with k varying by

multiplication scale. Later, Cook applied Toom' s ideas to accelerate computer programs, creating the Toom-Cook multiplication algorithm. This algorithm becomes extremely complex for large k values and is rarely used in practice, with typical k values below 6. When k=3, it becomes the commonly used Toom-Cook 3-Way algorithm with O(n^1.465) complexity. FFT multiplication leverages fast polynomial conversion between coefficient and point-value representations to achieve O(n log n log log n) complexity. However, FFT multiplication suffers from potential precision errors, which number-theoretic multiplication overcomes by using primitive roots from Number Theoretic Transforms (NTT) instead of unit roots from FFT. Additionally, Fürer proposed a theoretically O(n log n) multiplication algorithm in 2007—the fastest known algorithm—with numerous subsequent improvements, though its complexity limits application to ultra-large integer operations.

These algorithms employ different computational strategies with distinct advantages. In practice, no single method solves all problems. GMP, a highly performance-optimized library, strategically employs multiple algorithms—including basic, FFT, and divide-and-conquer—selecting among them based on data scale. While basic multiplication algorithms like written and Comba multiplication have higher complexity than fast algorithms, their simplicity and straightforward program flow offer significant advantages for small data scales. Moreover, basic multiplication often serves as the underlying algorithm called when divide-and-conquer algorithms recurse to smaller scales.

Currently, CPU frequency improvements have reached limits imposed by power consumption and thermal constraints, making single-CPU performance difficult to sustain. To maintain Moore' s Law, computer architecture has shifted toward multi-core and many-core structures. Multi-core processors are now ubiquitous, while many-core processors are widely used in high-performance computing. For example, the SW26010 many-core processor with independent intellectual property rights, used in China' s Sunway TaihuLight—the current top-ranked supercomputer on the TOP500 list—integrates 260 cores per chip. Research on efficiently implementing parallel large integer operations, particularly multiplication, on novel many-core architectures is crucial for exploiting processor performance advantages and solving larger-scale problems. However, current research on parallel large integer multiplication algorithms remains limited, with only a few studies exploring parallel implementation of existing algorithms on multi-core/many-core platforms, such as Jiang Lijuan et al.' s work on multi-core parallelization of Comba and Karatsuba multiplication, Xu Liang' s CUDA-based FFT multiplication implementation, and Zhao Mingxiang' s parallel Karatsuba algorithm on MIC accelerators. Yet research on algorithm parallelism analysis, improving parallelism, and solving load balancing issues during parallelization remains scarce.

This study aims to identify efficiently parallelizable basic multiplication algorithms for many-core architectures, laying the foundation for building complete large integer multiplication systems. Through analysis of parallelism in writ-

ten and Comba multiplication, we propose multiple methods for implementing efficient parallel Comba multiplication and implement them on the SW many-core processor, employing on-chip register communication and vectorization optimizations.

1.1 Written Multiplication Analysis

Written multiplication is the earliest multiplication method, characterized by its ease of understanding and implementation. Each digit of one multiplier multiplies every digit of the other, with these partial products accumulated with carries. The algorithm description is as follows:

Written Multiplication: Given two base-b integers (u_{m-1}···u_1u_0)b *and (*v{n-1}···v_1v_0)b*, their product is denoted as (*w{m+n-1}···w_1w_0)_b.

M1. [Initialization] Initialize w_{m+n-1}, w_{m+n-2}, ···, w_0 to 0, set i←0.
M2. Set j←0, k←0.
M3. [Partial product accumulation] Set t←u_i×v_j+w_{i+j}+k, then set w_{i+j}←t mod b, k←t/b.
M4. [Loop for j] Increment j; if j<n, return to M3, otherwise set w_{i+j}←k.
M5. [Loop for i] Increment i; if i<m, return to M2, otherwise algorithm terminates.

From this algorithm, the core C implementation of written multiplication is a 2-level nested loop:

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        t = u[i] * v[j] + w[i+j] + k;
        w[i+j] = t % b;
        k = t / b;
    }
    w[i+j] = k;
}
```

Analysis reveals that multiplying two m-digit and n-digit numbers requires at least mn multiplications and mn carry operations, yielding O(mn) time complexity. Specifically, when both multiplicands have n digits, complexity is $O(n^2)$.

During computation, each multiplication must propagate carries upward, making every step dependent on previous results and forcing sequential execution. In the C implementation above, each j-loop iteration depends on carry k from the previous iteration, creating a loop-carried dependency. Similarly, each i-loop iteration depends on carry w[i+j] from the previous iteration. Consequently, written multiplication can only execute sequentially, exhibiting poor parallelism.

1.2 Comba Multiplication Analysis

In 1990, Comba proposed an improved algorithm based on written multiplication. Comba multiplication computes column-wise, first merging entire columns then processing their carries, significantly reducing carry operations compared to written multiplication.

Comba Multiplication: Given two base-b integers (u_{m-1}⋯u_1u_0)*b and (v*{n-1}⋯v_1v_0)*b, their product is denoted as (w*{m+n-1}⋯w_1w_0)_b.

M1. [Initialization] Initialize w_{m+n-1}, w_{m+n-2}, ⋯, w_0 to 0, set i←0, k←0.
M2. [Calculate positions and lengths] Set y←min(i,n-1), x←i-y, s←min(m-x,y+1), t←0.
M3. [Column product] Compute t←t+u_x×v_y, then increment x and decrement y.
M4. [Loop for j] Increment j; if j<s, return to M3, otherwise compute t←t+k, k←t/b, w[i]←t mod b.
M5. [Loop for i] Increment i; if i<m+n, return to M2, otherwise algorithm terminates.

The core C implementation is:

```
for(i = 0, k=0; i < m+n; i++) {
    t = 0;
    y = min(i,n-1);
    x = i - y;
    s = min(m-x,y+1);
    for(j = 0; j < s; j++)
        t += u[x++] * v[y--];
    t += k;
    w[i] = t % b;
    k = t / b;
}
```

Analysis shows that Comba multiplication also requires mn multiplications but only m+n carry operations, reducing carry processing from quadratic to linear complexity. The time complexity remains O(mn), or $O(n^2)$ for equal-length operands.

Comba computes column-wise, where each column' s partial products merge without inter-column carry propagation. After merging a column, its carry is computed and propagated upward. In the classic implementation, the outer i-loop iterates through columns, but carry dependencies force sequential execution. The inner j-loop performs column merging as a reduction summation with iteration counts varying from 1 to n, which could parallelize efficiently. However, the i-loop' s carry dependency prevents parallel execution, limiting Comba' s parallelism.

2. Comba Multiplication Improvements

Both written and Comba multiplication exhibit poor parallelism, hindering efficient many-core implementation. However, analyzing Comba' s principle reveals that column computations are fundamentally independent and can execute in parallel. This paper focuses on improving Comba multiplication for many-core processors.

By restructuring Comba' s implementation into two steps—first computing all columns, then uniformly processing carries—we can parallelize column computations:

```
// Step 1: Merge columns
for(i = 0; i < m+n-1; i++) {
    t = 0;
    y = min(i,n-1);
    x = i - y;
    s = min(m-x,y+1);
    for(j = 0; j < s; j++)
        w[i] += u[x++] * v[y--];
}

// Step 2: Process carries column-wise
for(i = 0, k = 0; i < m+n; i++) {
    t = w[i] + k;
    w[i] = t % b;
    k = t / b;
}
```

This makes Step 1' s column computations completely independent and fully parallelizable, enhancing Comba' s parallelism. However, Step 2' s sequential carry processing from low to high columns remains inherently serial.

Although Step 1 can fully parallelize, this implementation suffers from severe load imbalance during parallel execution, as different columns require varying numbers of single-precision multiplications. Two approaches address this: (1) consider column computational loads during task partitioning to achieve better balance, or (2) further improve Comba' s implementation method.

2.1 Task Partitioning Strategy

[Figure 1: see original paper] illustrates Comba multiplication' s computation process. Excluding the highest possible carry, m+n-1 columns require computation. The first n-1 and last n-1 columns each perform fewer than n multiplications with non-uniform variation, while only the middle m-n+1 columns

perform exactly n multiplications. For task partitioning, we can pair column i with column i+m, creating m tasks with equal multiplication counts.

While this method balances loads effectively, it has drawbacks. High-level programming models like OpenMP and OpenACC cannot easily implement such fine-grained task partitioning. Lower-level models like Pthread and MPI enable fine control but require cumbersome implementations, and varying column counts per task lead to different communication volumes, complicating communication code.

2.2 Improved Comba Algorithm

Due to these limitations, we propose two improvement methods.

1) Adjusting Computation Order

Adopting the pairing strategy from task partitioning, we compute columns i and i+m within the same loop iteration:

```
// Step 1: Merge columns
for(i = 0; i < m; i++) {
    y1 = min(i,n-1);
    y2 = y1 + 1;
    x1 = i - y1;
    x2 = i + m - y2;
    s = min(n, i+1);

    for(j = 0; j < s; j++)
        w[i] += u[x1++] * v[y1--];

    for(j = 0; j < n+1-s; j++)
        w[i+m] += u[x2--] * v[y2++];
}
```

This inherits the task partitioning method' s advantages while overcoming its incompatibility with OpenMP/OpenACC, allowing developers to focus on parallel implementation rather than algorithmic deficiencies.

2) Borrowing from Written Multiplication

Combining written multiplication' s characteristics with Comba' s, we modify the computation order to traverse both multiplicands sequentially. By multiplication' s associative property, u_i and v_j' s product contributes to result w_{i+j}. After traversing both multiplicands, we accumulate column results without intermediate carry processing, which occurs after obtaining all column values.

Thus, Step 1 can be improved as:

```
// Step 1: Compute columns
for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        w[i+j] += u[i] * v[j];
```

This simplifies Step 1 into a 2-level perfectly nested loop with fixed iteration counts, forming a rectangular iteration space. The implementation offers simple control logic and facilitates optimizations like data prefetching and vectorization. However, after parallelizing the i-loop, each thread/process obtains partial column values requiring reduction, making it less suitable for systems with high communication costs but highly effective on shared-memory architectures.

---

3. Implementation on SW Many-Core Processor

3.1 SW Many-Core Processor Overview

This work targets efficient basic multiplication implementation on the domestic SW26010 many-core processor, the core component enabling Sunway TaihuLight' s high computing speed and low power consumption. SW26010 is designed for high-performance computing, with architecture shown in [Figure 2: see original paper]. The chip comprises 4 core groups, an on-chip interconnect network, and system interfaces. Each core group employs a heterogeneous many-core architecture with one main core and a 64-core accelerator array. The main core is a full-featured 64-bit RISC core supporting 32/64-bit integer operations, single/double-precision floating-point operations, and atomic operations, efficiently handling serial code segments and managing computational resources and power. The accelerator cores are also 64-bit RISC structures with streamlined instruction sets augmented with special instructions for register communication and row/column synchronization.

The 64 accelerator cores connect via an 8×8 network, supporting broadcast communication along rows and columns. The parallel implementation strategy is:

a) Partition the i-loop so each accelerator core executes m/64 iterations (assuming m divisible by 64; otherwise, pad with zeros). Each core computes partial values for product array w elements. The inner j-loop has no dependencies and exhibits continuous memory access, enabling convenient vectorization.

b) Reduce w arrays across cores. First, use on-chip register communication to reduce each row' s 8-core w arrays to the lowest-numbered core in that row (placing all results in the first column). Then reduce the first column to a single core. Finally, transfer the reduced w array from accelerator local memory to main memory via DMA.

c) The main core sequentially processes column carries. Remainder and di-

vision operations in carry processing are time-consuming. If the large integer base is a power of 2, these operations can be replaced with fast bit operations.

3.2 Parallel Comba Multiplication Implementation

A single SW26010 processor contains four core groups, the basic unit for resource allocation on Sunway TaihuLight. This work targets fast parallel large integer multiplication on a single core group. Two programming methods enable heterogeneous parallelism between main and accelerator cores: OpenACC* (compiler-directive-based, simple but less efficient) and the lower-level Athread API (enabling fine-grained control over task partitioning, data transfer, and synchronization for better efficiency). For maximum performance, we select Athread.

For parallel Comba implementation on SW26010, we choose the written-multiplication-inspired Comba algorithm due to its simplicity, low control overhead, and SW26010' s efficient accelerator-core register communication. Since Comba handles smaller operand sizes or serves as the base case in divide-and-conquer recursion, communication volumes remain small, enabling fast register-based communication.

In our SW26010 implementation, the accelerator array parallelizes Step 1 (column computation) while the main core handles Step 2 (carry processing).

---

4. Testing and Analysis

This section tests and analyzes the performance of our improved parallel Comba multiplication. Experiments run on the Sunway TaihuLight system with SW26010 processors, Raise Linux OS, and complete compilation and profiling toolchains. We use the most intuitive and scientifically common base-10 representation for implementation and testing.

Comba multiplication is typically invoked for small data sizes, so we test with 10 randomly generated datasets at hundred and thousand digit scales. compares clock cycles between parallel and serial versions, with speedup calculated as serial cycles divided by parallel cycles. The parallel version uses our improved method on SW26010, while the serial version uses classic Comba implementation. Parallel execution employs one main core controlling the accelerator array within a core group; serial execution runs on a single main core.

The results show our parallel Comba implementation significantly improves performance, achieving an average speedup of 57.75× on SW26010. Optimal speedup occurs at 1000-4000 digits—smaller sizes incur higher relative parallel startup overhead, while larger sizes increase on-chip communication overhead.

While parallel optimization for large integer multiplication has been studied extensively, platform and representation differences hinder direct comparison. For instance, Jiang Lijuan achieved 5.63× speedup for parallel Comba on an 8-core processor using OpenMP+SIMD; Vladislav Kovtun and Andrew Okhrimenko obtained 1.5× and 2× speedups with 4 threads using section and for strategies after Comba improvements. Our implementation achieves 57.75× speedup on SW26010' s 65 cores (1 main + 64 accelerators) using parallelization and vectorization, demonstrating superior parallel efficiency at scale.

These results confirm that our improved parallel Comba algorithm exhibits good parallelism, and the implementation fully leverages SW26010' s performance advantages. They also demonstrate that the domestic SW26010 many-core processor effectively handles large integer operations.

---

5. Conclusion

Motivated by architectural trends and demands for efficient large integer multiplication, this paper studied basic multiplication on many-core processors. Through parallelism analysis of written and Comba multiplication, we examined Comba multiplication in detail, analyzed its load balancing challenges during parallelization, and proposed multiple solutions. Finally, targeting the domestic SW26010 many-core processor' s architecture, we selected and optimized a written-multiplication-inspired Comba algorithm. Building upon this parallel Comba foundation, future work will investigate parallelization of other multiplication methods to construct a complete parallel large integer multiplication system.

References

[1] Wang Wei, Huang Xinming, Emmart N, et al. VLSI design of a large-number multiplier for fully homomorphic encryption [J]. IEEE Trans on Very Large Scale Integration Systems, 2014, 22 (9): 1879-1887.

[2] Moore C, O' Neill M, Hanley N, et al. Accelerating integer-based fully homomorphic encryption using Comba multiplication [C]// Processing Systems. 2014: 1-6.

[3] Karatsuba A. The complexity of computations [J]. Proceedings of the Steklov Institute of Mathematics, 1995, 211: 169-183.

[4] Lee C Y, Meher P K, Lee W Y. Subquadratic space complexity digit-serial multiplier over binary extension fields using Toom-Cook algorithm [C]// Proc of International Symposium on Integrated Circuits. 2015: 176-179.

[5] Schönhage A. Fast multiplication of polynomials over fields of characteristic 2 [J]. Acta Informatica, 1977, 7 (4): 395-398.

[6] Feng Xiang, Li Shuguo. Design of an area-efficient million-bit integer multiplier using double modulus NTT [J]. IEEE Trans on Very Large Scale Integration Systems, 2017, 25 (9): 2658-2662.

[7] Fürer M. Faster integer multiplication [J]. SIAM Journal on Computing, 2009, 39 (3): 979-1005.

[8] Covanov S, Thomé E. Fast arithmetic for faster integer multiplication [J]. Computer Science, 2015.

[9] Covanov S, Thomé E. Fast integer multiplication using generalized Fermat primes [J]. Computer Science, 2016.

[10] The GNU multiple precision arithmetic library [S/OL]. (2018-03-12) https://gmplib.org/

[11] Liao Xiangke, Xiao Nong. New high-performance computing systems and technologies [J]. Science China: Information Science, 2016, 46 (9): 1175.

[12] Fu Haoheng, Liao Junfeng, Yang Jinzhe, et al. The sunway TaihuLight supercomputer: system and applications [J]. Science China Information Sciences, 2016, 59: 1-16.

[13] Jiang Lijuan, Liu Fangfang, Zhao Yuwen, et al. Multi-core parallel of large integer multiplication Comba and karatsuba algorithms [J]. Computer Systems & Applications. 2016, 25 (11): 232-236.

[14] Xu Liang, Wang Zhen. Fast large integer multiplication based on CUDA [J]. Computer Engineering and Applications, 2013, 49 (16): 221-224.

[15] Zhao Mingxiang. The basic calculation of long integer on MIC acceleration components [D]. Guangzhou: South China University of Technology, 2016.

[16] Kovtun V, Okhrimenko A. Approaches for the performance increasing of software implementation of integer multiplication in prime fields [J]. Iacr Cryptology Eprint Archive, 2012.

Note: Figure translations are in progress. See original paper for figures.