2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00016

Cambricon-P: A Bitflow Architecture for Arbitrary Precision Computing

Yifan Hao[1,3], Yongwei Zhao[1,3], Chenxiao Liu[1,2,3], Zidong Du[1,3], Shuyao Cheng[1,2,3], Xiaqing Li[1], Xing Hu[1], Qi Guo[1], Zhiwei Xu[1,2], Tianshi Chen[3]
[1]*SKL of Processors, Institute of Computing Technology, CAS, Beijing, China*
[2]*University of Chinese Academy of Science, Beijing, China*
[3]*Cambricon Technologies Co., Ltd., China*
Email: {haoyifan, zhaoyongwei, liuchenxiao, duzidong, chengshuyao, lixiaqing, huxing, guoqi, zxu}@ict.ac.cn, tchen@cambricon.com

***Abstract*—Arbitrary precision computing (APC), where the digits vary from tens to millions of bits, is fundamental for scientific applications, such as mathematics, physics, chemistry, and biology. APC on existing platforms (e.g., CPUs and GPUs) is achieved by decomposing the original data into small pieces to accommodate to the low-bitwidth (e.g., 32-/64-bit) functional units. However, such fine-grained decomposition inevitably introduces large amounts of intermediates, bringing in intensive on-chip data traffic and long, complex dependency chains, so that causing low hardware utilization.**

To address this issue, we propose Cambricon-P, a bitflow architecture supporting monolithic large and flexible bitwidth operations for efficient APC processing, which avoids generating large amounts of intermediates from decomposition. Cambricon-P features a tightly-integrated computational architecture for processing different bitflows in parallel, where full bit-serial data paths are deployed. The bit-serial scheme still needs to eliminate the dependency chain of APC for exploiting parallelism within one monolithic large-bitwidth operation. For this purpose, Cambricon-P adopts a carry parallel computing mechanism, which enables recursively transforming the multiplication into smaller inner-products that can be performed in parallel between bit-indexed IPUs (Inner-Product Units). Furthermore, to improve the computing efficiency of APC, Cambricon-P employs a bit-indexed inner-product processing scheme, namely BIPS, to eliminate intra-IPU bit-level redundancy. Compared to Intel Xeon 6134 CPU, Cambricon-P achieves 100.98× performance on monolithic long multiplication, and 23.41×/30.16× speedup and energy benefit over four real-world APC applications on average. Compared to NVidia V100 GPU, Cambricon-P also delivers the same throughput, as well as 430×/60.5× lesser area and power, respectively, on batch-processing multiplications.

***Keywords*-arbitrary precision; bitflow architecture; bit-serial**

I. Introduction

Arbitrary precision computing (APC), indicating computing numbers with an arbitrary number of digits, is critical for many scientific computing applications, such as supernova simulation, climate modeling, coulomb N-body atomic system simulations, planetary orbit calculations, Schrodinger Solutions [5]–[7], [9], [18], [24]. These applications require data with hundreds, thousands, or even millions of digits, which can not be implemented *directly* on the common fixed-bitwidth functional units (e.g., 32/64-bit integer and floating-point) in modern CPUs and GPUs. As a result, the fixed-bitwidth units fail to *directly* perform arbitrary-precision computing in terms of both data representation range and precision [40].

Traditionally, APC is performed through sophisticated decomposition algorithms such as FFT [31], Karatsuba [37], and Toom-Cook [15]. The basic idea behind these algorithms is to recursively split large bitwidth operands into hundreds-bit pieces (a.k.a., limbs), where arithmetic operations (e.g., 'schoolbook multiplication') are applied to further decompose these hundreds-bit limbs into 32/64-bit ones to fit the functional units in CPUs/GPUs. More important, similar decomposition exists in most of the APC operations, such as division, square root, etc.

However, such decomposition prevents CPUs and GPUs from high computing efficiency due to large amounts of calculation intermediates. Experimental results show that the hardware utilization of CPU is only 19.1%, and GPU is even less than 0.001% over four representative APC workloads (i.e., Pi [13], Frac [32], zkcm [49], and RSA [12]). The reason behind this result is two-fold: 1) A large number of intermediates are produced when decomposing APC operands into small-bitwidth limbs, which inevitably causes intensive on-chip data traffic between computing units and register files (or buffers). 2) There are long, complex computational dependency chains between these intermediates, which undermines the parallelism. It is measured that such calculation intermediates can be significantly reduced by decomposing the APC operands into coarse-grained limbs (i.e., with larger-bitwidths). For instance, in our measurement, 7.68× less intermediates are generated when decomposing a 1,000,000-bit Karatsuba multiplication into 1024-bit limbs than 32-bit limbs (223.71 MB vs. 1.72 GB). Therefore, to reduce the amount of intermediates, it is necessary to propose monolithic large bitwidth functional units to avoid decomposing APC operands into small-bitwidth limbs.

Bit-serial functional units seem to be a promising solution, because of their low area and power costs compared with a naive arithmetic logical unit (e.g., 512-bit multiplier). But

Corresponding Author: Qi Guo (guoqi@ict.ac.cn)

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00016

the bit-serial scheme still need to address two problems: 1) the long, complex dependency chain, and 2) the bit-level redundancy. On the one hand, the dependency chain is caused by gathering results of limbs with difference significance. For example, in a schoolbook multiplication, the final result depends on *carry-bits* from products of limbs, as these products are overlapped in digits. More important, even *carry-bits* of the product from the least significant limb may change the final result (detailed illustrated in Section III). Such dependency chains obstruct processing different limbs in a single APC operand in parallel. However, existing methods fail to develop parallelism within a single APC operation, causing their low hardware utilization for APC applications with large bitwidth and few operands (such as Pi [13]). On the other hand, there are repetitive or sparse bit-level MAC operations, namely *bit-level redundancy*, when APC operations (such as multiplication) are processed in a bit-serial manner. So existing methods focusing only on bit-level sparsity (e.g., Bit-Tactical [42]) need to be further improved in computing efficiency.

In this paper, for high performance and energy efficiency of APC, we propose Cambricon-P, a bitflow architecture supporting monolithic large and flexible bitwidth operations with optimized parallelism. On the one hand, to eliminate the dependency chain for better parallelism, Cambricon-P propose a *monolithic-multiplication-friendly* method, including two key techniques: 1) An *inner-product transformation* decomposes a monolithic large multiplication into small inner-products to be performed simultaneously on bit-indexed IPUs (Inner-Product Units) as well as exploiting inter-IPU data reuse. 2) And a *carry parallel computing mechanism* in GUs (Gather Units) gathers all partial-sums (outcomes of these inner-products) in parallel. In such *inter-IPU parallelism* manner, Cambricon-P could exploit the parallelism within one monolithic APC multiplication, so that improve the performance of APC. On the other hand, to further improve the computing efficiency, Cambricon-P propose the bit-indexed IPU which employs a *bit-indexed inner-product processing scheme*, namely BIPS, to eliminate intra-IPU bit-level redundancy. In details, the IPU uses the input bitflows (e.g., the operand $\vec{y}$) to index fixed bit-patterns that are converted from another input data (e.g., the operand $\vec{x}$) for accumulation, which can exploit sparse and repetitive MAC operations for further reducing unnecessary computations.

We conduct detailed experiments to evaluate Cambricon-P on both key APC operations and four representative APC workloads (i.e., Pi [13], Frac [32], zkcm [49], and RSA [12]). Experimental results show that compared to Intel Xeon 6134 CPU, Cambricon-P achieves 100.98× performance on monolithic long multiplication, and achieves 23.41×/30.16× speedup and energy benefit on average over four real-world APC applications. Compared to NVIDIA V100 GPU on batch-processing multiplications, Cambricon-P still achieves the same throughput with 430×/60.5× lesser area and power respectively.

This paper makes the following contributions:

- **Detailed analysis of APC workloads.** We present a detailed breakdown and analysis of how existing APC applications perform on CPU and GPU platforms.
- **Bitflow architecture.** We propose a novel bitflow architecture for accelerating APC.
- **Inter-IPU parallelism.** We propose a transformation mechanism to recursively transforms the original APC computations into parallel smaller inner-products and a carry parallel computing mechanism to eliminate the complex data dependencies.
- **Intra-IPU bit-level redundancy.** We propose the bit-indexed IPU with the bit-indexed inner-product processing scheme, namely BIPS, that exploits bit-level redundancy including sparse and repetitive computations.
- **Thorough evaluation of Cambricon-P.** We conduct a thorough evaluation of Cambricon-P on both key APC operations and representative APC workloads. Experimental results well demonstrate that Cambricon-P significantly outperforms CPUs and GPUs in terms of performance, power consumption and area.

II. Background and Motivation

In this section, we introduce the basis of APC and their implementation on general CPUs/GPUs. We then analyze their inefficiency and discuss the challenges of designing an efficient architecture for APC.

A. Primer on APC

APC plays a key role in a number of scientific computation, where data precision varies from hundreds to millions of the bits, such as Diophantine equation ($\sim$ 200 bits) [9], Ising theory($\sim 1,000$ bits) [6], Log-Tan Integral Identity ($\sim 50,000$ bits) [7], and Computational Number Theory ($\sim 7,000,000$ bits) [18]. More important, since one tiny disturbance/error can lead to a highly deviated result (i.e., classical Coulomb N-body atomic system simulation [24]), such digit requirement can be even infinite for some precision-sensitive computation.

Today's APC applications are commonly implemented on CPUs or GPUs with specialized libraries (e.g., GMP [27] or XMP [16]) to accommodate to fixed-bitwidth hardware functional units. For instance, GMP is designed to be the most mature and representative CPU library for APC, and it served as the code-base integrated in a series of GMP-based libraries (e.g., MPFR [23], PARI/GP [53],CGAL [20], and MPIR [26]).

We present the software stack of GMP and GMP-based libraries in Figure 1. As shown in Figure 1, APC applications are composed of operators with different levels, and these operators are built hierarchically from the basic arithmetic operations of natural numbers. From the bottom up: 1) The library for natural numbers (GMP MPN) [27] is implemented for long addition, subtraction, bit-shifting, logical operations,

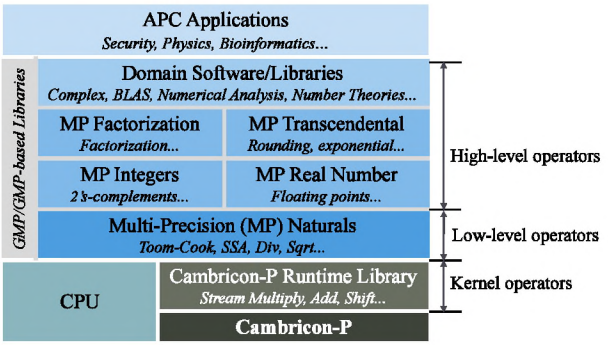


Figure 1. The software stack for APC implementation, where high-level functionalities are built hierarchically from long natural numbers multiplication and addition (i.e., low-level operators). GMP and GMP-based libraries are committed to hierarchically building and deploying these low-level operators. Cambricon-P replaces the CPU for accelerating the key part (i.e., kernel operators, such as *Multiply, Add, and Shift*) in low-level operators with the help of Cambricon-P runtime library (i.e., MPApca, introduced in Section V-C).

and *more importantly*, for fast algorithms of multiplication, including *Karatsuba algorithm* (a.k.a. *Toom-Cook 2-way algorithm*), *Toom-Cook k-way algorithms*, and *Schönhage-Strassen algorithm (SSA)*. 2) Then, the fast algorithms for multiplication are used to implement division and square root of naturals [61]. See Table I for these low-level operators. 3) Based on the library for naturals, libraries for integers (GMP MPZ) and real numbers (GMP MPF) are developed, with the ability to deal with signs and exponents. 4) Further, libraries for rationals (GMP MPQ) introduce fast factorization algorithms, and MPFR [23] presents high-level functions with error analysis, e.g. transcendental. These high-level functions are decomposed to low-level operators via iterative methods or divide-and-conquer methods, such as Newton-Raphson [59], AGM [50], and binary-splitting [11]. 5) Built upon these libraries, software/libraries with domain-specific functionalities, including complex numbers, BLAS and algebras for number theories, are developed for certain domains, such as mathematics, security, bioinformatics, quantum informatics and celestial physics. By the way, XMP library on GPUs shares the analogical methodology.

To describe how common APC algorithms are constructed hierarchically, we select Chudnovsky algorithm [13] (Algorithm 1), the fastest known algorithm to calculate the digits of π, as an example. The formula of π calculation in the first line of Algorithm 1 is binary-split with $P()$, $Q()$, and $R()$. These three functions are built with naturals addition, subtraction, and multiplication. Specifically, π is obtained via floating-point square root and division. The floating-point is processed by MPF with little overhead, where the divisions and square roots are decomposed to naturals, which are then performed with Karatsuba's algorithms [61] with naturals multiplication and addition. To further increase the acceleration, factorization can be optionally leveraged to simplify the fraction before dividing.

Algorithm 1 Computing π's digits (Chudnovsky algorithm [13] with binary-splitting)

$$\pi = 10005^{-\frac{1}{2}} \lim_{q\to\infty} \frac{4270934400Q(0,q)}{P(0,q)+13591409Q(0,q)},$$

where
$P(b-1,b) = (-1)^b(13591409+545140134b)R(b-1,b)$,
$Q(b-1,b) = 10939058860032000b^3$,
$R(b-1,b) = (2b-1)(6b-5)(6b-1)$,

and for $a < m < b$,
$P(a,b) = P(a,m)Q(m,b)+P(m,b)R(a,m)$,
$Q(a,b) = Q(a,m)Q(m,b)$,
$R(a,b) = R(a,m)R(m,b)$.

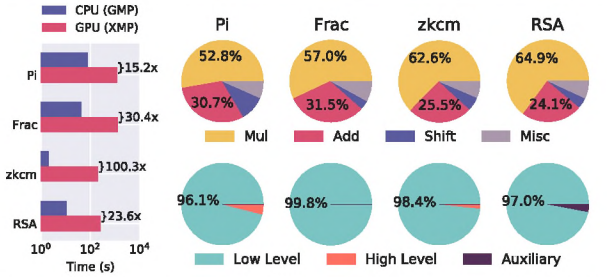


Figure 2. Performance profiling of four APC applications on both CPU and GPU. *To the left:* averagely, general purpose APC runs 32.2× slower on GPU (V100+XMP) than CPU (single thread Xeon 6134+GMP). *To the right:* run time breakdown on CPU showing 97.8% time consumed by low-level operators and 87.2% time consumed by kernel operators (*Multiply, Add, Shift*), averagely.

B. The Inefficiency on Existing Hardware

Although CPUs/GPUs with GMP/XMP libraries have been widely used in many APC applications, they fail to perform APC efficiently. We run a set of experiments to investigate the performance of Intel Xeon 6134 CPU with GNU GMP 6.2 [27] and NVIDIA Tesla V100 GPU with NVIDIA XMP 1.0 [16] over four typical APC applications (i.e., π's digits (*Pi*) [13], fractal rendering (*Frac*) [32], quantum information (*zkcm*) [49], and RSA encryption (*RSA*) [12]) respectively.

Table I
LOW-LEVEL OPERATORS IN APC AND THEIR FAST ALGORITHMS [27], [40] (n: the number of bits; m: the hyper-parameter in the complexity of *Division*, which depends on the specific multiplication algorithm applied in it, where $1 \le m < 2$)

Functions	Complexity	Functions	Complexity
Multiplication		Addition	$O(n)$
Schoolbook	$O(n^2)$	Subtraction	$O(n)$
Karatsuba	$O(n^{1.585})$	Negation	$O(n)$
Toom-3	$O(n^{1.465})$	Comparison	$O(n)$
Toom-4	$O(n^{1.404})$	Division	
Toom-6	$O(n^{1.338})$	Schoolbook	$O(n^2)$
SSA	$O(n\log n\log\log n)$	Karatsuba	$O(n^m\log n)$

We estimate the hardware utilization of CPU/GPU by the ratio of measured GMP/XMP performance to the ideal/peak performance, and get experimental results as follows. Despite excluding the SIMD execution units (SSE/AVX) in CPU and tensor units (Tensor Core) in GPU, the utilization of CPU is only 19.1% of a single core, and the utilization of GPU is even less than 0.001%. The performance profiling results are presented in Figure 2 (left). It is shown that general purpose APC runs 32.2× slower on GPU (V100+XMP) than CPU (single thread Xeon 6134+GMP), which is consistent with the hardware utilization results.

Such low hardware utilization on CPUs or GPUs is caused by the low parallelism for APC. For CPUs, GMP is not optimized for multi-threads [27]. Recent AVX-512 IFMA/VBMI extensions [29], landed in Intel IceLake CPUs in 2021, provide the SIMD extension. But it still can only basically process APC multiplication in limited-threads. For GPUs, APC applications cannot be properly processed in parallel across CUDA threads, which makes the parallelism even worse than CPUs. Besides XMP [16], recent libraries, such as CGBN [48] and CAMPARY [35], is specially designed for batch-processing scenarios, and thus can not work for general purpose APC, especially applications with large-bitwidth and few operands (e.g., *Pi*).

C. The Performance Bottleneck of APC

We explore the bottleneck and analyze the reason behind the inefficiency of APC, especially multiplication, on CPUs/GPUs.

Performance profiling. To identify the hot-spot operators that dominate the total runtime (or hardware utilization) of APC, we break down those four APC applications on the Xeon CPU. As shown in Figure 2 (right), low-level operators take the bulk of total runtime in four APC applications on a CPU (96.1%, 99.8%, 98.4% and 97% respectively). On average, low-level operators consumed 97.8% of the total runtime, while high-level operators (i.e., processing signs and floating points) or auxiliary functions (i.e., memory management and I/O) only consumes 2.2% of the total runtime. Moreover, it can be observed that, in Figure 2 (right), the operators involved in *Multiply*, *Add*, and *Shift* occupies 87.2% of the total runtime. Particularly, the *Multiply* occupies more than half. Therefore, we primarily focus on the performance boosting of the low-level operators, especially on *APC multiplication*.

Performance bottleneck. To understand the computational characteristic of *APC multiplication*, we use an idealized LRU model to investigate the performance bottleneck. Figure 3(*a*) shows a typical cache hierarchy design (AMD Zen3 [3]) with capabilities and bandwidths labeled. Figure 3(*b*) shows the bandwidth utilization of *APC Multiply*, *Random Access* ($n \log_2 n$ uniformly distributed accesses to n-elements) and single-precision *Matrix Multiply* respectively. Regarding the Random Access, bandwidths at all hierarchies are reasonably

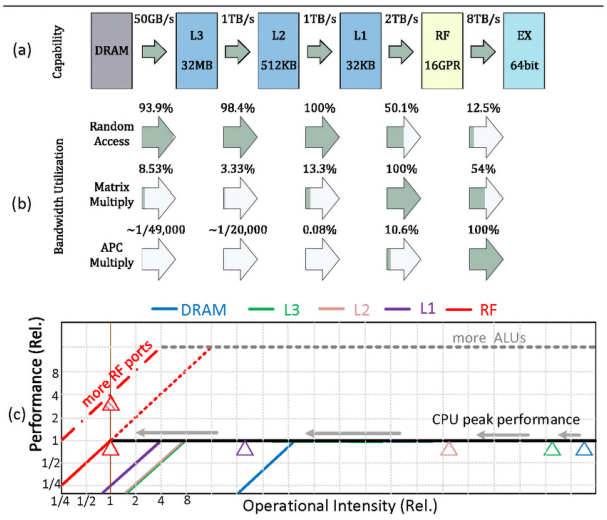
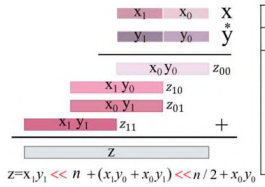


Figure 3. (a) The hardware capability of a conventional memory hierarchy. (b) The bandwidth utilization at each level for Random Access, Matrix Multiplication and APC Multiply. (c) The *roofline* for APC Multiply, showing the rapidly decline of operational intensity from remote to near hierarchies, so that the capacity of register bandwidth limits its performance.

utilized, and the bottleneck is lying at remote hierarchies (i.e., DRAM and L2/L3 caches). Regarding the Matrix Multiply, the utilization is concentrated at the near-end hierarchy (i.e., between L1 cache and register files), showing the effect of high *data locality*. However, APC Multiply is completely stuck at the nearest hierarchy (i.e., register files), while the remote hierarchies are almost idle.

The performance bottleneck of APC Multiply is due to the extremely high data locality, derive from the excessive intermediates generated by the fine-grained decomposition of APC operands. High data locality and high *decomposability factor* [34] are two sides of the same coin, the later means when decomposed into finer granularity, the amount of I/O operations increases dramatically. At near-end hierarchies (e.g., RF), the capacity is reduced thus the APC operation is forced into smaller-bitwidth. Along with more data spilled to far-end hierarchies, excessive intermediates are generated, swapped back and forth repeatedly. An example is shown in Figure 4. APC operands are often decomposed into thousands if not millions of limbs recursively, causing several orders of magnitudes more traffic pressure at the near-end (1/49000 utilization of DRAM bandwidth vs. full register file utilization in the example), challenging the conventional architecture design.

In addition, we analyze the bottleneck with the *Roofline* model [57]. As shown in Figure 3(*c*), when inspecting the memory bandwidth (blue line), the APC multiplication is profiled as a compute-dominated application (blue mark, hitting the black line). However, increasing the peak per-



Op	Input Bits	Output Bits	Total Bits
$z = x * y$	n, n	2n	**4n**
$z_{00} = x_0 * y_0$	n/2, n/2	n	2n **20n**
$z_{01} = x_0 * y_1$	n/2, n/2	n	2n
$z_{10} = x_1 * y_0$	n/2, n/2	n	2n
$z_{11} = x_1 * y_1$	n/2, n/2	n	2n
$z_0 = z_{01} + z_{10}$	n, n	n	3n
$z_1 = z_{00} + z_{11}$	n, n	2n	4n
$z = z_0 + z_1$	n, 2n	2n	5n

Figure 4. Schoolbook multiplication. An n-bit multiplication is decomposed into four $\frac{n}{2}$-bit multiplications and three additions, but intermediates lead to 5× larger total accessed bits than directly performing n-bit multiplication ($20n$ vs. $4n$). Moreover, the final result z depends on carries from z_{00}, z_{10}, and z_{01}.

实验验证，性能限制，不是由ALU决定的，是由带宽决定的测试后，发现分解后的位宽越大，所需要的中间体越少，容易打破性能瓶颈

formance (gray dashed lines) by adding more Arithmetic-Logic-Units does not improve the attained performance. So we inspect on-chip bandwidths (L3: green, L2: pink, L1: violet, RF: red) to seek the reason for such disability. It could be observed that, as the bandwidth increases, the operational intensity decreases faster (marks moving to the left) and eventually hits the bandwidth limit at the RF-level (assuming the RF keeps up with the peak performance). Such decreasing tendency of operational intensity is consistent with the *decomposability factor* theory above. Therefore, to improve the attained performance, both the peak performance and the RF bandwidth must be lifted (the red dot-dashed line). Since adding ports to the RF is much more expensive than piling up ALUs, the result is likely to be memory-bounded (red shadowed mark).

Inspiration. On the basis of the previous analysis, we propose a hypothesis that large-bitwidth functional units with coarse-grained decomposition could reduce the amount of intermediates. A straightforward experiment verifies this hypothesis: in our measurement, 7.68× less data are generated when decomposing a 1,000,000-bit Karatsuba multiplication into 1024-bit limbs than 32-bit limbs (223.71 MB vs. 1.72 GB). Therefore, a monolithic large-bitwidth multiplier is a promising solution to break the performance bottleneck by avoiding decomposing APC multiplications into small-bitwidth limbs.

III. Challenges of Designing an Efficient Architecture

On the basis of such inspiration, we try to design monolithic large-bitwidth functional units for accelerating APC, especially multiplication.

A direct method is that implementing the large-bitwidth ALU (Arithmetic Logical Unit), such as a 512-bit multiplier. However, whether using *Dadda* or *Wallace* schemes, such a large-bitwidth ALU would suffer from impractical fan-outs and congestion with current CMOS technologies. For example, a 512-bit integer multiplier takes an unacceptable area of $0.16 mm^2$ and costs 521.67× more energy, 189.36× more area, 5.74× slower than a 32-bit integer multiplier, under 16 nm technology. Moreover, even employing the large-bitwidth ALU, it cannot handle the varying bitwidth, which

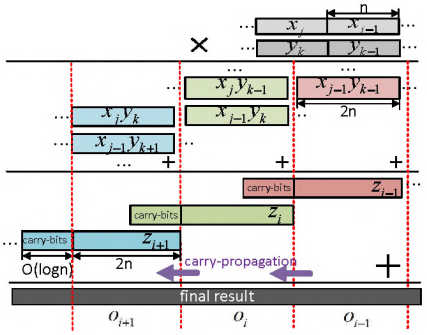
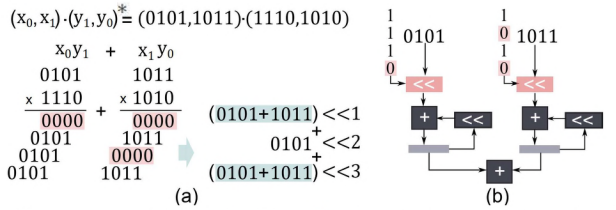


Figure 5. The long, complex dependency chain in APC multiplication. Before computing, operands (i.e.,x and y) are decomposed into n-bit limbs respectively. It reveals that the i-th product of limbs (i.e.,z_i) depends on *carry-bits* of the former product(i.e.,z_{i-1}), and propagates its *carry-bits* to the latter product(i.e.,z_{i+1}).

is required by different APC operations to achieve high efficiency.

To avoid such expensive hardware overhead, the bit-serial multiplier, which supports flexible bitwidth operands and has been widely used in recent DLPs (Deep Learning Processors) [19], [36], [42], [44], [54]–[56], seems to be a superior solution. However, these bit-serial schemes can not be extended to APC directly for their low hardware utilization, caused by the following two problems: 1) parallelism—*the long, complex dependency chain*, and 2) computing efficiency—*the bit-level redundancy*.

Long, complex dependency chain. With the leverage of bit-serial multipliers which support large-bitwidth monolithic multiplications, we can reduce intermediates by decomposing APC operands into larger limbs. But it is still challenging to process multiple limbs in parallel as there exist the long, complex dependency chains. For multiplication, each bit in the *output* depends on 1) the equal significant bits in *inputs* directly, 2) and the less significant bit in *inputs* indirectly because of the carry-propagation. In the same way, the carry-bits in the product of less significant limbs would affect former products and the final result. As shown in Figure 5, the result of i-th product (namely z_i) is the sum of massive multiplications of limbs (namely x_j,y_k), i.e. $z_i = \sum_{j+k=i} x_j y_k$. z_i brings up to $O(\log n)$ bits of carries, where n equals to the bitwidth of each limb. Given these possible carries, the result of the i-th part cannot be determined until the $(i-1)$-th part is finished. In the worst case, an unexpected carry propagated from the $(i-1)$-th part may invert the i-th part result, and affect the carry from the i-th to the $(i+1)$-th part, and even affect up to the most significant part through the dependency chain. Therefore, without an elaborate approach of handling carry-propagation, a monolithic APC multiplication can only be performed inefficiently in such a sequential manner, causing the low hardware utilization, like today's CPUs and



∗ This corresponds to 'IP1' in Figure 7 and the 'bit-indexed inner-product' in Figure 8.

Figure 6. (a) Bit-level redundancy in the MAC operation (*Red:* bit-sparsity. *Green:* repeated computations). (b) Bit-serial MAC unit, supporting bit-sparsity by skipping the zero bits.

降低B级冗余，减少0拍

GPUs.

Bit-level redundancy. Furthermore, higher computing efficiency (i.e., providing higher performance under less area and power budget) should also be acquired for bit-serial schemes. For MAC operations, bit-level redundancy includes bit-sparsity where zero-valued bits exit inside each multiplication, and repeated computations where same additions exit crossing different multiplications, as shown in Figure 6 (a). Generally, bit-sparsity can be easily supported in existing bit-serial designs [4], [42] by skipping the zero bits, as shown in Figure 6 (b). However, identifying and eliminating repeated computations is not an intuitive task, which would be resolved in our proposal.

IV. Our Proposal

According to the former analysis, the bit-serial computing scheme, supporting monolithic large-bitwidth operations (esp. multiplications), is a pragmatic solution for reducing intermediates. On the basis of such scheme, we aim at solving two problems discussed above: 1) breaking the long, complex dependency chain for parallelism, and 2) eliminating the bit-level redundancy for computing efficiency.

A. Inter-IPU Parallelism

Different from previous methods, which are only applicable to batch-processing scenarios, *Inter-IPU parallelism* can exploit parallelism within one monolithic APC operation by enabling two key techniques: *inner-product transformation* and *carry parallel computing mechanism*.

Inner-product transformation. We first transform a monolithic large multiplication into a bunch of small inner-products of limbs.

$$\begin{aligned} x\cdot y &= \sum_{i=0}^{n_x-1}\sum_{j=0}^{n_y-1} 2^{iL}x_i\cdot 2^{jL}y_j \\ &= \sum_{t=0}^{n_x+n_y-2} 2^{tL} \sum_{j=0}^{min(n_y-1,t)} x_{t-j}y_j \\ &= \sum_{t=0}^{n_x+n_y-2} 2^{tL}\vec{x}*\vec{y}\,(t) \end{aligned} \tag{1}$$

As shown in equation (1), a schoolbook multiplication (i.e., $x\cdot y$) is equivalent to two limb-vectors' (i.e., $\vec{x}=(x_0,\ldots,x_{n_x-1})$ and $\vec{y}=(y_0,\ldots,y_{n_y-1})$) polynomial-convolution (i.e., recursively accumulate inner-product partial-sums after shifting tL digits) after decomposition, where L is the bit-width of each limb (i.e., x_i, y_i) and n_x, n_y is the number of limbs. For example, as shown in Figure 7(a), one APC multiplication are transformed into 5 inner-products. Moreover, the limb-vector $\vec{y}=(y_0,y_1)$ can be reused between 3 inner-products (i.e., IP1/IP2/IP3 in Figure 7(a)), and the limb-vector $\vec{x}=(x_0,x_1,x_2,x_3)$ can be partially reused between these inner-products (e.g., x_1 could be reused between IP1 and IP2). Processing all inner-products of limbs in parallel is preferred to leverage such data locality.

Carry parallel computing mechanism. Although all inner-products of limbs can be calculated in parallel, the calculated partial-sums (e.g., the outcome of IP1) cannot be gathered simultaneously because of the dependency chain. As shown in Figure 7(b), each partial-sum is outputted by one IPU in a bitflow manner. The *overlap-bits* between adjacent bitflows causes the propagation of multiple *carry-bits* during the gathering process, thus leading to the dependency chain (more detailed illustration has been shown in Figure 5). So it is inspired that, we should try to reduce the length of such *overlap-bits* by gathering them *timely*, so as to eliminate the dependency. In fact, when each *partial-sum* outputs $2L$-bits (namely aligned *partial-sums*), the adjacent $2L$-bits aligned *partial-sums* only have L *overlap-bits*, and the non-adjacent ones do not have any *overlap-bits*. As shown in Figure 7(c), the first step of gathering these aligned *partial-sums* is cutting the accumulation into several parts, making each part only contains two L-bits *summands* and the *carry-bits* from the former part (i.e., C_{in}). In each part of such accumulation, the key observation is that two L-bits *summands* have at most 1 *carry-bit* (i.e. C_{out}) for the next addition.

$$\begin{aligned} &partial\_sum_{i-1}^{(low\ L\ bits)} + partial\_sum_{i}^{(high\ L\ bits)} + C_{in} \\ &\le \left(2^L-1\right)+\left(2^L-1\right)+1\ or\ 0 \\ &\le 2^{(L+1)}-1 \end{aligned} \tag{2}$$

To illustrate this, we list the calculations of each part in the first line of Equation (2):

- For $i=0$, 0-th part contains only the least significant L-bits of *partial-sum0*, so C_{out} must be 0 (i.e. C_{in} of 1-st part is equal to 0).
- For $i=1$, $C_{in}=0$, two L-bits *summands* can obtain $L+1$-bits sum at most. Lower L-bits of the sum can be directly outputted as the final result. While the highest bit is regarded as the *carry-bit* sent to the next part (i.e., C_{out}), due to the overlap with next part *summands*.
- For $i=2$, C_{in} = 0 or 1, Equation (2) proves that the upper-bound of sum's bitwidth is $L+1$-bits. Therefore, the resulting C_{out} is still at most 1-bit. Moreover, all subsequent parts are the same as $i=2$.

将部分积分成多个部分
对每一部分，将其进位看做0或者1

the pattern matrix K, the index matrix Bcoi, and the digit-weight vector C

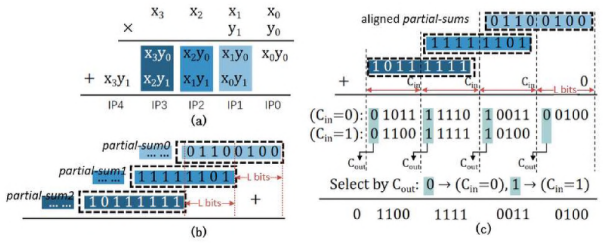


Figure 7. Leveraging the inter-IPU parallelism of APC multiplication: (a) Inner-product transformation, where a monolithic large multiplication (i.e., $x \cdot y$) is decomposed into several small inner-products (e.g., IP1). (b) Gathering aligned *partial-sums* (which have *L overlap-bits* between adjacent ones). (c) Carry parallel computing: adding up *partial-sums* in advance, so that keeping the *carry-bits* within one bit, enabling to calculate all parts of accumulations in parallel by considering two possible cases of the *carry-bits* (i.e., 0 or 1).

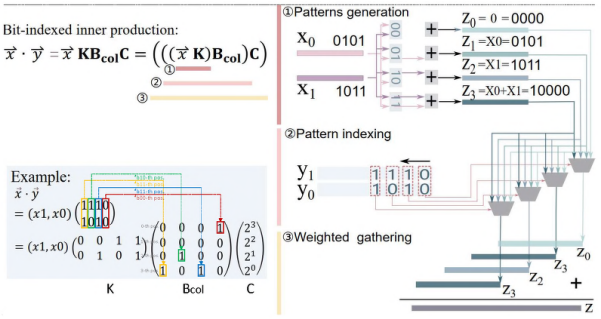


Figure 8. Exploiting the intra-IPU bit-level redundancy by the bit-indexed inner-product processing scheme (BIPS).

Therefore, we should timely gather bitflows of partial-sums before their output-bits beyond $2L$. To concurrently process all aligned *partial-sums*, as shown in Figure 7(c), we propose the *carry parallel computing mechanism*, i.e., computing all two possible values (i.e. 1 or 0) of C_{in} in advance when gathering. After each aligned *partial-sum* is summed up and get its 1-bit C_{out}, the correct value of the next aligned *partial-sum* can be selected. The following parts of *partial-sums* can also be aligned and gathered by such mechanism, so that the final result can be outputted in parallel without the obstruction of the dependency chain.

In short, we decompose the large monolithic APC multiplication into batch-processed small inner-products of limbs by the *inner-product transformation*, and then gather all partial-sums (outcomes of these inner-products) in parallel by the *carry parallel computing mechanism*. In such *inter-IPU parallelism* manner, we could exploit the parallelism within one monolithic APC multiplication, thus improving the performance of APC.

B. Intra-IPU Bit-level Redundancy

To further improve the hardware efficiency, we propose a bit-indexed inner-product processing scheme (BIPS) for eliminating *intra-IPU bit-level redundancy* (i.e. sparse or repetitive operands, as shown in Figure 6) in a single bit-serial inner-product operation.

Bit-indexed inner-product processing scheme. Using the two-data vectors (consistent with the data in Figure 6) as an example, Figure 8 shows the basic processing flow of the proposed BIPS. Roughly, the inner-product of $\vec{x} \cdot \vec{y}$ is performed via the form of $\vec{x}KB_{col}C$, where $\vec{y}$ is decomposed into three parts: the *pattern matrix K*, the *index matrix* B_{col}, and the *digit-weight vector C*. By decomposing elements from $\vec{y}$, digits in $\vec{y}$ can be represented with binary matrices, as shown in Figure 8(*left*). K is the *pattern matrix* with a size of $q \times 2^q$, where q is the number of elements in

K的每一列都是可能的结果

$\vec{x}$ or $\vec{y}$ ($q = 2$ in this example) and each column of K is a possible value of a q-bit binary number, thus 2^q patterns in total. So the elements of K is only determined by q, and it is feasible to fix K in hardware circuits after selecting q. B_{col} is the *index matrix* with a size of $2^q \times p_y$, where p_y is the bitwidth of $\vec{y}$ ($p_y = 4$ in this example) and each column of B_{col} indicates which pattern in K should be selected to get original bits of $\vec{y}$. So the position of '**1**' in each one-hot column vector (from right to left) of B_{col} is equal to the binary number consisting of corresponding q-bits in $\vec{y}$ (from LSB to MSB), as shown in Figure 8(*left*) 'Example'. Therefore, in the real hardware process, $\mathbf{B_{col}}$ does not need to be actually generated or stored. The *digit-weight vector C*, with the length of p_y, records the significance of original bits in $\vec{y}$. As shown in Figure 8(*right*), the entire computation is achieved through three stages: *patterns generation*, *pattern indexing*, and *weighted gathering*.

- *Patterns generation:* This stage computes the $\vec{x}K$, where the result $\vec{z} = \vec{x}K$ contains all the possible combinations of elements in $\vec{x}$ (i.e., patterns of $\vec{x}$).
- *Pattern indexing:* This stage computes the $\vec{z}B_{col}$, where the arithmetic unit access the position of '**1**' in each one-hot column vector of B_{col} by reading $\vec{y}$ from LSB to MSB *directly*. This operation is actually selecting certain elements in $\vec{z}$ according to each position of '**1**'.
- *Weighted gathering:* This stage accumulates indexed patterns based on weights in C which are powers of 2. So indexed patterns are shifted one by one for accumulation.

In this way, Cambricon-P avoids the repeated computations by first processing inputs as patterns for indexing (e.g., $x_0 + x_1$ = 'b0101+'b1011 is calculated only once, instead of twice in original bit-serial MAC units), so that more tidy and efficient circuits could be implemented in the functional units. Moreover, both $\vec{x}$ and $\vec{y}$ can be serialized as bitflows, which is critical for APC when the bitwidths of operands comes to very large, as it avoids the requirement of large size on-chip storage for operands.

Benefit analysis. We analyze the potential benefit of BIPS by measuring the quantity of reduced computations. We

bops定义为x+y max(px,py)
xy定义为 px*py

pe被设计用于执行内核APC操作。CC协调所有组件以一起工作。CMA将pe连接到有限责任公司。AT集成了所有PEs的结果

use *bops* (i.e., binary-operations) to measure the quantity of computations. The definition of the metric *bops* is— for x, y with the bitwidth of p_x, p_y, we define the *bops* of addition $x+y$ as $max(p_x, p_y)$, and multiplication xy as $p_x \cdot p_y$. Then, we count the number of *bops* required to compute an inner-product operation $\vec{x} \cdot \vec{y}$ (where $\vec{x}, \vec{y}$ have q elements with bitwidths of p_x, p_y) through BIPS. 1) For *pattern generation*, by skipping zeros and reusing partial-sums (e.g., x_0+x_1 can be reused for $x_0+x_1+x_2$), the *bops* of $\vec{z} = \vec{x}K$ is $(2^q - q - 1) \cdot p_x$ at most. 2) For *pattern indexing*, as each column in B_{col} is a one-hot vector, calculating $\vec{\hat{z}} = \vec{z}B_{col}$ just need selection operations without any *bops*. 3) For *weighted gathering*, the calculation $\vec{\hat{z}}C$ is simply achieved by shifting and accumulating with $p_y(p_x+q)$ *bops* at most. For $p_x, p_y \gg q$ in common, the total *bops* of BIPS is roughly $(2^q - 1 + p_y)p_x$. While the *bops* of straightforward bit-serial scheme is qp_xp_y, and we can get the ratio of their *bops* is $\lambda = \frac{1}{q}(1 + \frac{2^q-1}{p_y})$. For $p_y = 32$ and arbitrary p_x, used in this paper's hardware, $\lambda_{min} = 0.367$ when $q = 4$. In other words, under such configuration, decomposing an APC multiplication into inner-products with 4 elements would have the least quantity of computations (i.e., *bops*)—36.7% of the straightforward bit-serial scheme. Therefore, we process 4 bitflows in parallel in the later architecture design.

V. Architecture Design

In this section, we introduce the detailed architecture of Cambricon-P, including overall integration, Cambricon-P Processing Elements (PE), controller, datapath, and software.

A. System Integration

There are three typical integration schemes that can be applied for specialized accelerators to work with the host CPU: 1) *LLC-integration* integrates the accelerator tightly to the CPU core where the Last Level Cache (LLC) is shared; 2) *SoC-integration* integrates the accelerator by connecting to the NoC in CPU SoC; 3) *IO-integration* integrates the accelerator as I/O peripherals. In this paper, we adopt the LLC-integration scheme for Cambricon-P. The reason is two-fold. First, LLC-integration can efficiently reduce the interaction cost between Cambricon-P and CPU. Cambricon-P interacts with the host CPU intensively when performing entire APC applications. By data sharing of LLC, integrating Cambricon-P into the LLC can provide the least cost of such intensive interaction. Second, the large buffer of LLC can improve the efficiency of memory accesses on Cambricon-P. There are millions of bits in high-precision data of APC, and the large capacity of LLC can be leveraged to buffer the high precision data, thus avoiding repeated memory accesses. Otherwise, Cambricon-P is required to equip the costly large buffer. Moreover, the granularity of Cambricon-P is sufficiently large to alleviate the anti-memory-wall, and will not incur excessive loads on the LLC even without the buffering of L1/L2.

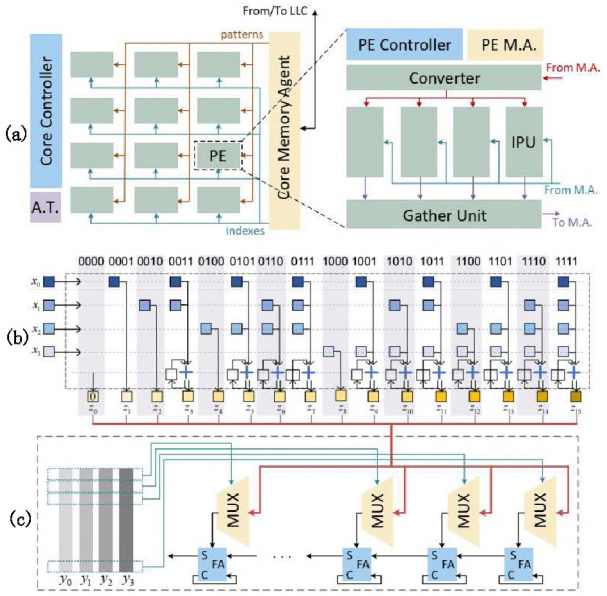


Figure 9. (a) The overall architecture of Cambricon-P and Cambricon-P PE. (b) The functionality of Converter (in implementation, repeated additions are saved by reusing previous results, e.g., z_{15} can be computed using the results of $z_3 = x_0 + x_1$ and $z_{12} = x_2 + x_3$). (c) The architecture of IPU in PE (z_i from Converter, and y from PE M.A.).

B. Cambricon-P Architecture

1) Overall: With LLC-integration scheme, Cambricon-P is integrated into the CPU SoC and serves as a co-processor to the host CPU. Figure 9(a) (left) shows the overall architecture of Cambricon-P. Cambricon-P consists of multiple Processing Elements (N_{PE} PEs), a Core Controller (CC), a Core Memory Agent (CMA), and an Adder Tree (AT). The PEs are designed to perform kernel APC operations. The CC coordinates all the components to work together. The CMA connects PEs to the LLC. The AT integrates the results of all PEs.

Roughly, Cambricon-P adopts recursive decomposition for control [60], multiple bitflows for datapaths, carry parallel computing and inner-product batch-processing for parallelism, and bit-indexed inner-product units for computation efficiency. a) Regarding the control, when Cambricon-P receives orders (instructions) from the CPU to perform an arbitrary-precision inner-product, the CC decomposes the inner-product into N_{PE} small pieces (still inner-products) evenly and maps them to N_{PE} PEs for computing. Similarly, each PE decomposes the small piece of inner-product into smaller inner-products. b) Regarding the datapaths, Cambricon-P works with multiple bitflows, where each input operand is streamed into PEs from the CMA with 1 bit per cycle, multiple input operands are streamed in parallel (multiple bitflows), and the outputs are streamed out to the CMA in a bit-serial manner. c) Regarding the parallelism, each PE implements a Gather Unit (GU) to resolve the dependency chain by carry parallel computing,

关于控制，当Cambricon-P收到来自CPU的指令（指令）来执行任意精度的内积时，CC将内积均匀地分解为Npe小块（仍然是内积），并将它们映射到Npepe进行计算。类似地，每个PE将小块内积分解为更小的内积，b)对于数据路径，Cambricon-P处理多个比特流，其中每个输入操作数从CMA流到PE，多个输入操作数并行流（多个位流），输出以位串行方式流到CMA，c)关于并行性，每个PE实现一个收集单元（GU），通过携带并行计算来解决依赖链

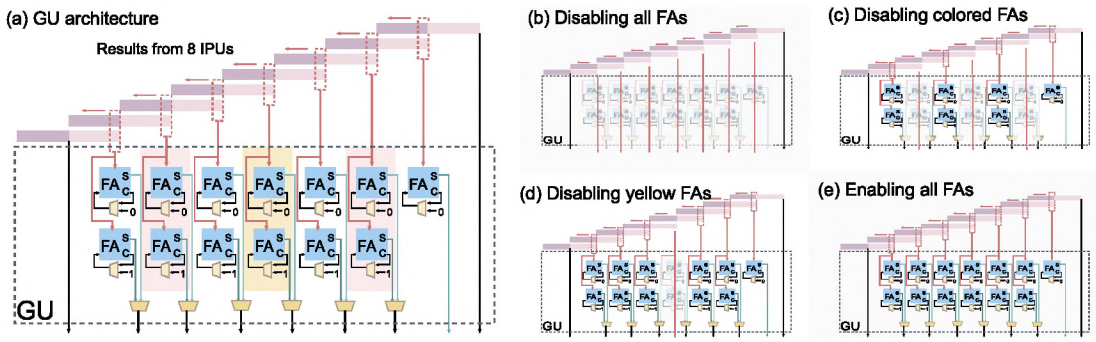


Figure 10. (a) The Gather Unit (8 IPUs). (b) Disabling all FAs: all PEs generate separate results. (c) Disabling FAs in the colored background: results of every 2 IPUs are combined. (d) Disabling FAs in the yellow background: results of every 4 IPUs are combined. (e) Enabling all FAs: results of all PEs are combined.

and all PEs exploit data reuse in the polynomial-convolution computing by the inner-product batch-processing. d) Regarding the computation efficiency, Cambricon-P implements multiple bit-indexed inner-product units (IPUs) to leverage the intra-IPU bit-level redundancy.

2) Cambricon-P PE: In Cambricon-P, each PE is designed to perform one bit-indexed inner-product. Figure 9(a) (right) shows the architecture of one PE, which contains 5 functional components: a PE controller (PEC), a PE Memory Agent (PEMA), a Convertor, multiple inner-product units (N_{IPU} IPUs), and a Gather Unit (GU).

Converter. The Converter performs the *patterns generation* stage, where one input vector of inner-product is converted into all possible combinations (i.e., patterns). Figure 9(b) shows the architecture of the Convertor ($n = 4$ for example), which receives a four-data vector, i.e., $\vec{x} = (x_0, x_1, x_2, x_3)$, and converts it to a sixteen-data vector, i.e., $\vec{z}$. Therefore, the sixteen data in $\vec{z}$ contains all the possible combinations of the four data in $\vec{x}$, $2^4 = 16$ possibilities in total. Particularly, the Convertor works in a bit-serial manner, where four bitflows of the $\vec{x}$ stream in and sixteen bitflows stream out. The bandwidth required is as small as 4-bits per cycle. The generated patterns will be broadcasted to all the IPUs in PE for later computations.

IPU. IPUs are designed as a homogeneous architecture to perform the *indexed accumulation* stage and part of the *weighted gathering* stage. Figure 9(c) shows the detailed architecture of one such IPU. All IPUs share the same patterns ($\vec{z}$), but each has individual indexes. Each IPU receives its indexes, i.e., $\vec{y}$, from the PEMA. Based on indexes, each multiplexer in IPU selects one bitflow from $\vec{z}$ and allows that bitflow to pass through. The passed results of multiplexers are accumulated through a bit-serial accumulator. Therefore, the IPU generates 1-bit *partial-sum* of the inner-product per cycle.

GU. The GU finishes the *weighted gathering* stage. Figure 10 shows the architecture of the GU, where 8 IPUs ($N_{IPU} = 8$) are plotted for clarity. The GU gathers the output (i.e. *partial-sum*) bitflows from IPUs and transmits integrated results to the Adder Tree (AT) for generating the final result. As shown in Figure 7(c), the bitflows from IPUs are aligned and *timely* gathered to exploit carry parallel computing mechanism. In such executing manner, where both $carry = 1$ and $carry = 0$ scenarios are executed first for later selecting the correct result, GU can generate the final results much faster than naive and direct accumulation. For a GU supporting N_{IPU} IPUs, Moreover, for naive accumulation need process overlapped bitflows (which are not *timely gathered*) by a costly circuit, GU appears to be more hardware efficient.

Moreover, to flexibly support possible decomposition of patterns and indexes, GU is required to combine results from certain IPUs, including every 1, 2, 4, 8, 16, and all 32 IPUs. The naive and direct accumulation requires a costly adder tree to achieve different combinations. In our implementation, such combinations can be simply achieved by disabling different FAs in GU. In Figure 10, enabling all FAs allows combining all the IPUs, disabling FAs in color backgrounds (red+yellow) allows combining every two adjacent IPUs, and disabling FAs only in the yellow background allows combining every 4 IPUs. Therefore, by disabling different FAs, GU can achieve different combinations of IPU results.

3) Bitflow Controls: Cambricon-P employs a two-level controller, i.e., the core controller (CC) and the PE Controller (PEC). Controllers of both levels decompose and map the workload (which takes the inner-product form) into the next level of units, and the decomposed sub-workloads also takes the inner-product form, constitutes the fractal controlling scheme [60]. The Memory Agents (MAs), which are in charge of the bitflow management, is also a two-level architecture, including the PE memory agent (PEMA) and the core memory agent (CMA), in charge of bitflows inside PE and Cambricon-P respectively.

Data are prefetched into and read from the LLC as cache lines, then dispatched in block (4 flows, each of 32-bit length) onto the core-level internal data bus. The data block is saved in PEMAs and consumed over time till the next data block arrives. The CC determines which PE to receive the data

朴素积累和直接积累需要一个昂贵的加法器树来实现不同的组合

block from the core data bus, and the PEC determines which IPU to receive the data block from the PE data bus.

Bitflows inside PE. Each PE receives 4 bitflows from each input, 8 bitflows in total. One input ($\vec{x}$) serves as the *patterns* and is converted into 16 bitflows, which will later be indexed by the other input (i.e., $\vec{y}$ as *indexes*) in IPUs. Each IPU fetches the 4 bitflows starting from different positions and all the IPU work simultaneously (see Figure 9(c)). Note that the bitflows of indexes can belong to different input vectors, therefore with GU configured into different modes (Figure 10), IPUs in PE can perform various numbers of inner-products of the same $\vec{x}$ but with different $\vec{y}$.

Bitflows in Cambricon-P. The *indexes* are shared among vertical PEs while the *patterns* are shared among the horizontal PEs. *Indexes* and *patterns* are multi-casted via corresponding buses respectively from CMA to PEs. As both *indexes* and *patterns* can belong to either the same vector or different vectors, Cambricon-P can be configured to perform a monolithic inner-product, or different numbers of small inner-products. High-level operators, e.g., *convolution* and *matrix multiplication* are also directly supported with such flexibility. It is worth noting that when performing a monolithic inner-product, PEs are activated in sequence to align the timing of result bits across Cambricon-P. Therefore we can save FIFOs and registers from the AT, which integrates the computational results of PEs periodically.

index垂直共享，patterns水平共享

C. Working with host CPU

As a co-processor, Cambricon-P focuses on the essential operators (esp. multiplication) with natural numbers only, while host CPU helps to process other trivial parts, as shown in Figure 1. In this section, we discuss how Cambricon-P functionalities are implemented with the help of the host CPU.

Operators and fast algorithms. For Cambricon-P, we build a runtime library, MPApca, which realizes both the essential operators—including addition, subtraction, multiplication, and bit-shifts—and several high-level operators. Such operators are determined by conducting clustering analysis of widely-used APC applications. Regarding the addition, MPApca scatters and maps the addends into different PEs to perform parallel addition, and leverage the chained Gather Units to deal carries afterward. Subtraction is the same as an addition in the hardware's perspective, while MPApca inverses the bitflow of the subtrahend and provides an initial carry bit to the start of the Gather Unit chain. Regarding the multiplication, MPApca implements several fast multiply algorithms (Toom-{2,3,4,6} and SSA). MPApca (as well as GMP) selects at runtime which fast multiply algorithm is used by comparing the bitwidth of operands to compile-time tuned thresholds. Bit-shifts are translated into timing delays or advancements with no extra overhead. Several high-level operators are also provided in MPApca including polynomial convolution, division, square root, and *Montgomery reduction* [47], etc., composed with inner-product, addition, subtraction, shift, and multiplication.

Negative, rational, real, and complex data. In line with common practices, Cambricon-P supports only naturals on hardware. The signs, fractions, exponents, and imaginaries are managed from the host CPU with negligible overhead. Note that the negatives are supported via sign-magnitude instead of 2's complementary in common APC libraries as well as in MPApca. This is to avoid the additional costs on computing with sign-extended leading 1s.

VI. Methodology

A. Configurations

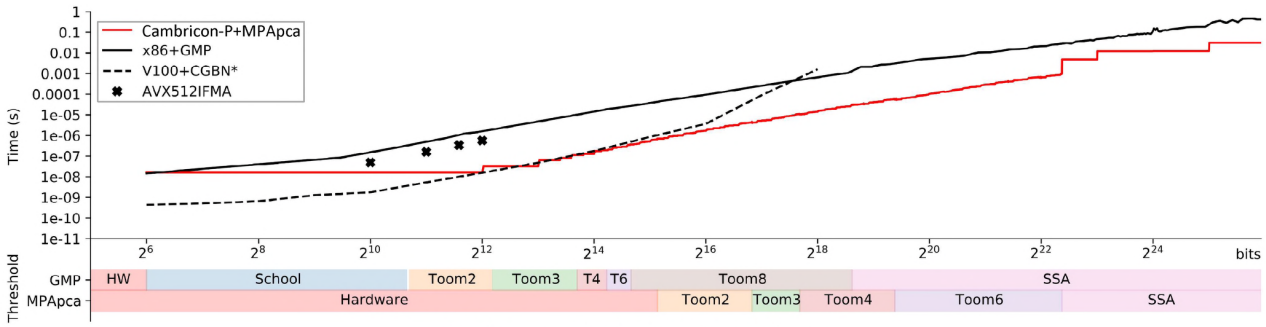
CPU. The CPU baseline is the Intel Xeon 6134 CPU, which is equipped 768 GB main memory with 119.21 GB/s bandwidth. We enabled the turbo frequency to achieve the best single-core performance and disabled SMT. Scalar instructions on the single-core provide a peak performance of 11.1Gops@INT64. The time and energy consumption of CPU is measured with `sprof` command and Intel SoC Watch, respectively. Regarding the time measurement, `sprof` only measure CPU time cost by shared libraries (i.e. GMP/MPFR), excluding other overheads, e.g. system calls, I/O, `malloc` and value initializing. Regarding the energy costs, to measure the energy consumed only by running APC, we measure the energy consumption of both idle CPU and busy CPU that runs benchmarks, where the difference between the two readings are taken as the final energy costs. To avoid random errors, we take the average of multiple measurements until the error at 95% confidence level is below 1%.

We also measure the AVX512IFMA implementation open-sourced by researchers from Intel Haifa labs [29], which utilized instructions recently shipped with Ice Lake CPUs, i.e., VPMADD52LUQ, VPMADD52HUQ, VPERMI2B, etc. These instructions enabled the packed full 52-bit multiplication and convenient horizontal carry-propagation, achieving the state-of-the-art SIMD performance over APC scenarios.

GPU. The GPU baseline is the NVIDIA V100 [17], which has a peak performance of 125 Tops/s 16-bit floating-point computations leveraged by the CGBN [48] library. Since CGBN is designed to work with batched multiplication, we measure the amortized time consumption of a single multiplication over a batch size of 10,000. We measure the performance and power using the official `nvprof` tool [10].

Table II
Benchmarks.

Benchmark	Note
Pi	Computing N digits of π with Algorithm 1.
Frac	Rendering Mandelbrot zooming with Perturbation theory [32].
zkcm [49]	Simulating quantum computers with complex matrices.
RSA [12]	Cryptosystem.



∗ Amortized over a batch of 10,000 multiplications.

Figure 11. The time costs of multiplication of N-bit natural numbers on Cambricon-P and baseline systems.

Accelerators. We also re-implemented several accelerators (the state-of-the-art accelerator DS/P [38] and the well-known Bit-Tactical [42]) with the same technology and the same theoretical throughput to compare their area and power.

Cambricon-P. To obtain the hardware characteristics, Cambricon-P is implemented in Verilog RTL and synthesized, placed&routed with Synopsys tools under TSMC 16 nm technology. Memory compiler [30] is used to instantiate on-chip buffer and DESTINY [46] to model behaviors of the main memory. The hardware design is verified with CPU results by using VCS and Verdi. To obtain the performance of Cambricon-P, we implement a cycle-accurate simulator with hardware characteristics from layout, whose behavior is also calibrated with the hardware design.

To evaluate the time and energy consumption of applications on Cambricon-P, we override the operators in GMP/MPFR libraries with MPApca[1]. The MPApca library, which is validated with GMP, is linked to our simulator of Cambricon-P to collect exact time and energy consumption spent on Cambricon-P. Please note that the energy consumption of LLC is also collected for Cambricon-P.

B. Benchmarks

We compare the performance of multiplication of two N-bit long naturals as the most essential metrics. CPU+GMP and Cambricon-P+MPApca are plotted over the $64 \sim 64,000,000$-bit range, and V100+CGBN and AVX512IFMA are plotted over their applicable ranges. Amortized costs are shown for CGBN.

We also select 4 representative APC applications, i.e., *Pi* [13], *Frac* [32], *zkcm* [49], and *RSA* [12], as benchmarks to be evaluated in this paper, see Table II. These benchmarks are evaluated with varying precisions within the applicable range, and we report the comparison between Cambricon-P and CPU on multiple precision. All the benchmarks are linked to GNU GMP [27] (version 6.2.1) and/or GNU MPFR (version 4.1.0), compiled with GNU GCC [14] (version 11.1). All benchmarks are single-threaded and affined to a fixed CPU core, therefore we compare the Cambricon-P to one CPU core in all the following experiments unless otherwise specified. Error range under 95% confidence level is marked as the shadow. V100+CGBN and AVX512IFMA are not evaluated due to their incompatible programming interfaces.

[1]It is worth noting that although we tune MPApca at best effort, it is still very elementary compared to GMP: it lacks many fast multiply algorithms (Toom-{3/2, 4/3, 4/2, 5/3, 5/4, 6/3, 6H, 8, 8H}), fine-grained policy to apply SSA, and optimization-oriented optional low-level operators (e.g. *AddMul*, *MulLo*, *DivExact*). The result may be not showing the full advantage of Cambricon-P due to software limitations.

VII. Experimental Results

A. Hardware Characteristics

We implemented Cambricon-P in TSMC 16 nm technology, with 256 PEs where each PE contains 32 IPUs. Cambricon-P has an area of $1.894\,mm^2$, which only takes $\sim 2.3\%$ of a core complex die (assuming AMD Zen3 [3] integration), or $\sim 56\%$ of a CPU core. Therefore, Cambricon-P only adds a negligible area cost to the whole CPU SoC. Cambricon-P has a power consumption of 3.644 W under a 2 GHz clock frequency.

B. Fast Multiplication

Figure 11 shows the time costs of the multiplication of two N-bit natural numbers on Cambricon-P and baselines, including Xeon 6134 CPU, V100 GPU, and AVX512IFMA. To achieve better performance on APC multiplication, both the GMP/GMP-based and MPApca libraries select a certain *fast algorithm* routine for computing, based on the bitwidth of operands. The thresholds for different fast multiplications are predefined and tuned in compile-time. We first examine the relative advantages of Cambricon-P over CPU on different bitwidths when performing multiplication with these tuned *fast algorithms*.

- With the arbitrary bitwidth supported of Cambricon-P, the MPApca library no longer needs the schoolbook multiplication. The ranges of fast multiply algorithms are also delayed accordingly, saving multiplicative constant factors in the time complexity. Cambricon-P hardware can

efficiently process the multiplication of up to $N = 35904$, which fully covers the bitwidth range of GMP's schoolbook, Toom-{2,3,4,6H}, achieving up to 100.98× speedup. This is the source of advantages of Cambricon-P.

- The implementation of MPApca's Toom-Cook algorithms is not optimal, MPApca also lacks support for Toom-8 thus using Toom-6 instead. Therefore, we expect the speedup will drop a little bit in the ranges of Tooms. Within the ranges of Tooms, Cambricon-P can keep 18.06× ∼ 67.78× speedup.
- The SSA implementation in MPApca lacks a fine-grained policy. SSA requires the bitwidth of inputs being $2^k (k \in \mathbb{Z}^+)$ to keep the optimal time complexity (i.e. $O(n \log n \log\log n)$). MPApca always pads the bitwidth of inputs to the next 2^k and do calculations on the paddings, introducing big zigzags into the time versus bitwidth curve, while GMP use a fine-tuned lookup-table to select which policy to apply [25], resulted in a smoother curve. Within the range of SSA, Cambricon-P can keep 3.87× ∼ 14.89× speedup.

We also compare Cambricon-P with other baseline systems in Figure 11 and Table III. Compared with V100+CGBN, Cambricon-P achieves almost identical performance with 430× smaller area and 60.5× lower power. The source of efficiency is mainly from larger granularity ($4096 \times N$ where N is arbitrary, versus V100's 16×16 in major), and also better computation efficiency (due to bit-level redundancy exploited via bit-indexed inner-product). In addition, CGBN only processes in batches, while Cambricon-P can also accelerate monolithic long multiplication resulted in better generality. This is because carry parallel computing enabled PEs in Cambricon-P to concatenate into a monolithic multiplier. Compared with AVX512IFMA units, the state-of-the-art SIMD solution, Cambricon-P achieves 35.6× performance with comparative area (3.48× larger) and power (3.64× lower). Compared with DS/P [38], the state-of-the-art accelerator solution, Cambricon-P costs 3.06× smaller area and 2.53× lower power under iso-throughput comparison[2]. The advantage over Bit-Tactical [42] is even greater.

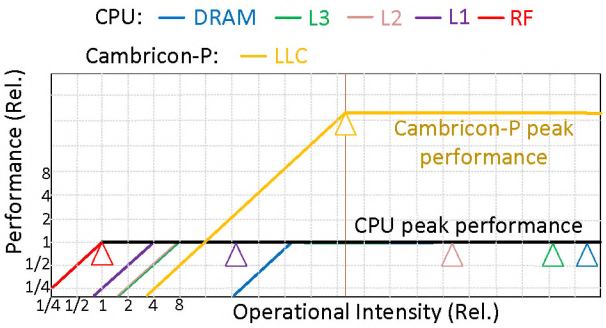


Figure 12. The roofline for APC multiplication on Cambricon-P (and the comparison against CPU).

Moreover, we find the utilization of memory bandwidth is far from upper-bound for all systems for experiments in Table III. Because the operational intensity [58] of 4096×4096 multiplication is equal to 4 imul(64)/byte, which is relatively high comparing against common applications/benchmarks (i.e., 0.17 ∼ 1.64 flops/byte) [34], [57].

Roofline for Cambricon-P. To further understanding the experimental results, we also draw a *roofline* for APC multiplication on Cambricon-P, shown in Figure 12. Comparing against CPU (under an ideal/100% hardware utilization), the larger multiplication granularity (i.e., bitwidth of limbs) in Cambricon-P guarantees higher operational intensity, so that making full use of more abundant arithmetic units (i.e., IPUs), leading to the improvement of performance. In addition, when obtaining the experimental results in this figure, we force the Memory Agent of Cambricon-P idle in 50% run-time cycles to guarantee the cost of maintaining CPU memory ordering and coherence. Therefore, the bandwidth of Cambricon-P is one grid (50%) lower than CPU LLC(L3) in the figure.

[2]We compare p.p.a. under the condition of iso-throughput (i.e., aligned with Cambricon-P), since DS/P and Bit-Tactical cannot efficiently scale-up.

Table III
COMPARISON OF CAMBRICON-P AND BASELINE SYSTEMS OVER 4096 × 4096-BIT MULTIPLICATION.

	Cambricon-P	SkyLake-X (GMP [27])	V100 (CGBN [48])	AVX512IFMA [29]	DS/P [38]	Bit-Tactical [42]
Technology	TSMC 16 nm	Intel 14 nm	TSMC 12 nm	Intel 10 nm	TSMC 16 nm	TSMC 16 nm
Area (mm^2)	1.89	∼ 17.98†	815	∼ 0.54†	5.80	7.12
(Rel.)	1	9.49†	430	0.29†	3.06	3.76
Power (W)	3.64	7.43	220.58	13.26	9.20	18.29
(Rel.)	1	2.04	60.50	3.64	2.53	5.02
Time (s)	1.60×10^{-8}	7.59×10^{-2}	1.56×10^{-8}**	5.70×10^{-7}	-	-
(Rel.)	1	4.74×10^{5}	0.98**	35.60	1*	1*
Bandwidth (GB/s)	512 (LLC)	128 (L1D)	900 (HBM)	128 (L1D)	-	-
(Rel.)	1	0.25	1.76	0.25	1*	1*

† Estimated based on die photos. * Theoretical performance/bandwidth (keep iso-throughput with Cambricon-P).
** Amortized over a batch of 100,000 multiplications.

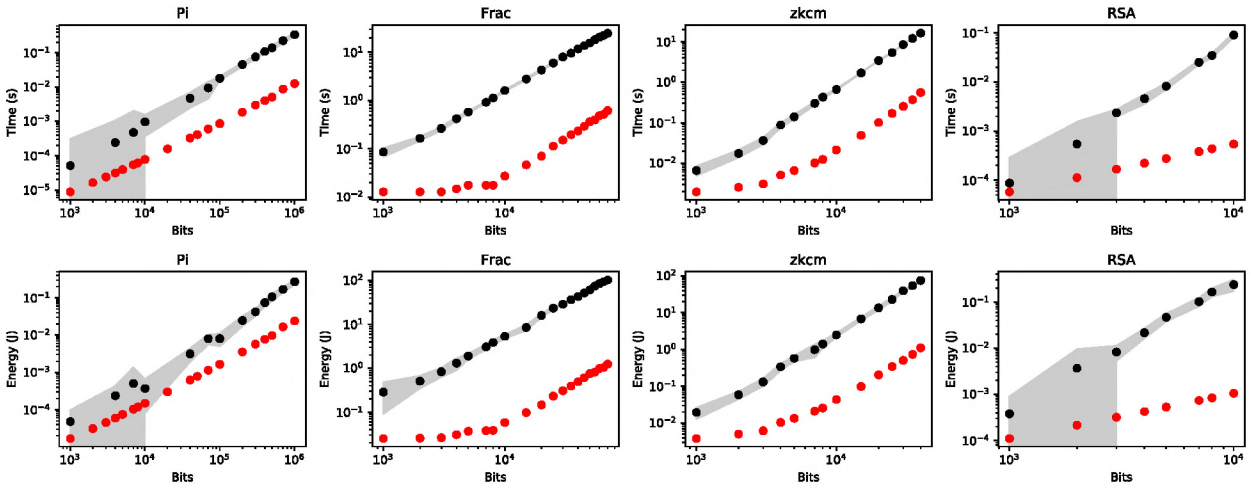


Figure 13. Time (top row) and energy (bottom row) cost of benchmark applications on CPU and Cambricon-P, within a range of precision (X-axis). Black: CPU+GMP, with 95% confidence interval shown as the shadow. Red: Cambricon-P+MPApca.

C. Applications

Figure 13 (top row) compares the performance results of Cambricon-P and the baseline CPU on the benchmark applications. Overall, Cambricon-P is 23.41× faster than CPU on the average of four applications of all sample points. More specifically, over the CPU, Cambricon-P can achieve

- 5.82× ∼ 16.65×, 11.22× on average for Pi,
- 6.71× ∼ 63.92×, 38.62× on average for Frac,
- 3.38× ∼ 34.97×, 21.30× on average for zkcm, and
- 1.51× ∼ 166.02×, 21.94× on average for RSA.

Particularly, Cambricon-P outperforms the baseline CPU most on the large RSAs, where the gap grows with the bitwidth, since RSA is composed of Montgomery reductions (implemented by pairs of multiply and add operations) and squares, the time proportion of multiplicative operations grows rapidly with bitwidth, which best fit Cambricon-P's architecture. The least speedup is from Pi since the binary-splitting method introduced many small-bitwidth multiplications that are hard to accelerate.

The energy efficiency of Cambricon-P versus the baseline CPU is in line with the performance results, Cambricon-P achieves 30.16× energy benefits overall.

VIII. Related Works

Bit-serial computing. Bit-serial computing is resurgent in the field of computer architecture for the recent development of deep learning processors. The major advantage taken from bit-serial computing is the flexibility for various bitwidth [19], [36], [42], [44], [54]–[56], instead of the chip area efficiency for early bit-serial architecture [28], [52]. MLWeaving [56], Stripes [36], and Bit-Tactical [42] leverage the bit-serial computing for deep neural networks with low-precision/quantized data (below 16 bits). Shapeshifter [43] further extends the bit-serial computing for a finer-grain than layer. Li [44] exploits computational redundancy in *iterative computations* (Jacobi/Newton's solvers) rather than general proposes. However, these bit-serial schemes can not be extended to APC directly for their low hardware utilization, caused by the unresolved dependency chain of intermediates and the unidentified repetitive MAC operations.

Hardware platforms for APC. As CPUs only contain limited computing power, there is a clear trend towards large parallel hardware platforms, including GPUs and customized architecture— recent works try to leverage the GPU by manually optimization with CUDA implementations for a specified subset of APC applications. For example, a multiple-precision arithmetic library is built especially for chaotic dynamical, which targets mainly applications deployed on NVIDIA GPU [35]. Isupov et al. [33] proposes a residue number system (RNS) based library for heterogeneous CPU-GPU architectures to enable effective parallelism of arithmetic operations. However, these works do not achieve comparable performance as CPUs on general APC applications (especially those applications with fewer operands that could hardly be batch-processed). Efforts on customized architecture are also made. Bocco et al. [8] proposes an FPGA prototype, namely SMURF, that supports long floating-point data computing having up to 512-bit mantissa. It is mainly concerned with the efficient control of numerical methods (such as newton-Raphson method) on the coprocessor (rather than how to design a high efficient APC units), which is orthogonal to our work. Feinberg et al. [21] uses a memristive crossbar together with a GPU for matrix multiplication but with only 32-bit fixed-point data. Koenig et al. [41] proposes an accelerator to compute dot products with up to 4288 bits. Li et al. [44] build an iterative solver on FPGA,

namely Architect, to accelerate iterative applications (e.g., Jacobi/Newton solvers for equations) with multiple precisions. The insight of Architect is to avoid recalculating digits across different iterations (not for a single operands).

These efforts either fail on processing arbitrary precision or are unable to be used for arbitrary applications, while Cambricon-P can process general APC applications with high efficiency.

IX. Conclusions

While APC is important to many fields, massive intermediates in decomposition brings in intensive on-chip data traffic and long, complex dependency chains, so that causing low hardware utilization. Reducing intermediates and their data traffic requires large-bitwidth monolithic multipliers but the dependency chain exacerbates the design difficulties. In this paper, we propose Cambricon-P, an accelerated bit-serial architecture to efficiently process APC. Cambricon-P exploits inter-IPU parallelism for breaking dependency chains and intra-IPU bit-level redundancy for further computing efficiency. Compared to Intel Xeon 6134 CPU, Cambricon-P achieves 100.98× performance on monolithic long multiplication, and achieves 23.41×/30.16× speedup and energy benefit on average over four real-world APC applications. Compared to NVidia V100 GPU on batch-processing multiplications, Cambricon-P also achieves the same throughput with 430×/60.5× lesser area and power.

Despite extending APC to ripe fields like Homomorphic Encryption (HE) [22], [39], [51], Celestial Orbit Calculation [1], [2], and Protein Structure Prediction [45], from the perspective of architecture, our future works will focus on end-to-end acceleration of APC applications, including FFT, IFFT integration.

Acknowledgment

This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA0700902), the NSF of China (under Grants 61925208, 62102398, 62002338, 61732020, U19B2019), Strategic Priority Research Program of Chinese Academy of Science (XDB32050200), Beijing Academy of Artificial Intelligence (BAAI) and Beijing Nova Program of Science and Technology (Z191100001119093), CAS Project for Young Scientists in Basic Research (YSBR-029) and Youth Innovation Promotion Association.

References

[1] A. Abad and A. Elipe, "Evolution strategies for computing periodic orbits," *Mathematics and Computers in Simulation*, vol. 146, pp. 251–261, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378475414002018

[2] A. Abad, R. Barrio, and A. Dena, "Computing periodic orbits with arbitrary precision," *Phys. Rev. E*, vol. 84, p. 016701, Jul 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.84.016701

[3] *Software Optimization Guide for AMD Family 19h Processors (PUB)*, Advanced Micro Devices, 2020, available from https://www.amd.com/system/files/TechDocs/56665.zip.

[4] J. Albericio, P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, "Bit-Pragmatic Deep Neural Network Computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 1–16.

[5] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, 2012.

[6] D. H. Bailey, J. M. Borwein, and R. E. Crandall, "Integrals of the ising class," *Journal of Physics A: Mathematical and General*, vol. 39, no. 40, p. 12271, 2006.

[7] D. H. Bailey, J. M. Borwein, V. Kapoor, and E. W. Weisstein, "Ten problems in experimental mathematics," *The American Mathematical Monthly*, vol. 113, no. 6, pp. 481–509, 2006.

[8] A. Bocco, Y. Durand, and F. De Dinechin, "Smurf: Scalar multiple-precision unum risc-v floating-point accelerator for scientific computing," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, 2019, pp. 1–8.

[9] A. R. Booker and A. V. Sutherland, "On a question of mordell," *Proceedings of the National Academy of Sciences*, vol. 118, no. 11, 2021. [Online]. Available: https://www.pnas.org/content/118/11/e2022377118

[10] T. Bradley, "Gpu performance analysis and optimisation," *NVIDIA Corporation*, 2012.

[11] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *J. ACM*, vol. 23, no. 2, p. 242–251, Apr. 1976. [Online]. Available: https://doi.org/10.1145/321941.321944

[12] M. Calderbank, "The rsa cryptosystem: History, algorithm, primes," *Chicago: math. uchicago. edu*, 2007.

[13] D. V. Chudnovsky and G. V. Chudnovsky, "Approximations and complex multiplication according to ramanujan," in *Pi: A Source Book*. Springer, 1988, pp. 596–622.

[14] G. C. Collection, "GCC 8 Release Series Changes, New Features, and Fixes," https://gcc.gnu.org/gcc-8/changes.html, 2019.

[15] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, 1969.

[16] N. Corp, "xmp: Cuda accelerated(x) (m)ulti-(p)recision library."

[17] N. Corporation, "NVIDIA Tesla V100 GPU Architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2018.

[18] R. Crandall and C. B. Pomerance, *Prime numbers: a computational perspective*. Springer Science & Business Media, 2006, vol. 182.

[19] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018. [Online]. Available: http://arxiv.org/abs/1805.03718

[20] A. Fabri and S. Pion, "Cgal: The computational geometry algorithms library," in *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2009, pp. 538–539.

[21] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 367–382.

[22] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption (extended version)," 09 2021.

[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, pp. 13–es, 2007.

[24] A. M. Frolov and D. H. Bailey, "Highly accurate evaluation of the few-body auxiliary functions and four-body integrals," *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 36, no. 9, p. 1857, 2003.

[25] P. Gaudry, A. Kruppa, and P. Zimmermann, "A GMP-based implementation of Sch$^{\acute{}}$onhage-Strassen's large integer multiplication algorithm," 2007. [Online]. Available: https://hal.inria.fr/inria-00126462

[26] B. Gladman, W. Hart, J. Moxham *et al.*, "Mpir: Multiple precision integers and rationals," 2016.

[27] T. Granlund and G. D. Team, *GNU MP 6.0 Multiple Precision Arithmetic Library*. London, GBR: Samurai Media Limited, 2015.

[28] V. Gregory and B. Dellande, *MC14500B industrial control unit handbook*. Motorola Semiconductor Products Inc., 1977.

[29] S. Gueron and V. Krasnov, "Accelerating big integer arithmetic using intel ifma extensions," in *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, 2016, pp. 32–38.

[30] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–6.

[31] W. G. Hawkins, "Fft interpolation for arbitrary factors: a comparison to cubic spline interpolation and linear interpolation," in *Proceedings of 1994 IEEE Nuclear Science Symposium-NSS'94*, vol. 3. IEEE, 1994, pp. 1433–1437.

[32] C. Heiland-Allen, "Perturbation techniques applied to the mandelbrot set." [Online]. Available: https://mathr.co.uk/mandelbrot/perturbation.pdf

[33] K. Isupov, A. Kuvaev, M. Popov, and A. Zaviyalov, "Multiple-precision residue-based arithmetic library for parallel cpu-gpu architectures: data types and features," in *International Conference on Parallel Computing Technologies*. Springer, 2017, pp. 196–204.

[34] H. Jia-Wei and H. T. Kung, "I/o complexity: The red-blue pebble game," in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '81. New York, NY, USA: Association for Computing Machinery, 1981, p. 326–333. [Online]. Available: https://doi.org/10.1145/800076.802486

[35] M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker, "Campary: cuda multiple precision arithmetic library and applications," in *International Congress on Mathematical Software*. Springer, 2016, pp. 232–240.

[36] P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Stripes: Bit-Serial Deep Neural Network Computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–6.

[37] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[38] M. Karlsson and M. Vesterbacka, "Digit-serial/parallel multipliers with improved throughput and latency," in *2006 IEEE International Symposium on Circuits and Systems*, 2006, pp. 4 pp.–.

[39] S. Kim, J. Kim, M. J. Kim, W. Jung, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," 2021.

[40] D. E. Knuth, *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[41] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 114–121.

[42] A. D. Lascorz, P. Judd, D. M. Stuart, M. Mahmoud, K. Siu, and A. Moshovos, "Bit-Tactical : A Software / Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 749–763.

[43] A. D. Lascorz, S. Sharify, I. Edo, D. M. Stuart, O. M. Awad, P. Judd, M. Mahmoud, M. Nikolic, K. Siu, Z. Poulos *et al.*, "Shapeshifter: Enabling fine-grain data width adaptation in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 28–41.

[44] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides, "architect: Arbitrary-precision hardware with digit elision for efficient iterative compute," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 516–529, 2019.

[45] V. Marx, "Method of the year: Protein structure prediction," *Nature methods*, vol. 19, no. 1, pp. 5–10, 2022.

[46] S. Mittal, R. Wang, and J. Vetter, "Destiny: A comprehensive tool with 3d and multi-level cell memory modeling capability," *Journal of Low Power Electronics and Applications*, vol. 7, no. 3, p. 23, 2017.

[47] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[48] Nvidia, "CGBN: CUDA Accelerated Multiple Precision Arithmetic (Big Num) using Cooperative Groups," https://github.com/NVlabs/CGBN.

[49] A. SaiToh, "Zkcm: A c++ library for multiprecision matrix computation with applications in quantum information," *Computer Physics Communications*, vol. 184, no. 8, pp. 2005–2020, 2013.

[50] E. Salamin, "Computation of pi using arithmetic-geometric mean," *Mathematics of computation*, vol. 30, no. 135, pp. 565–570, 1976.

[51] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sánchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 173–187. [Online]. Available: https://doi.org/10.1145/3470496.3527393

[52] K. Tanigawa, K. Umeda, and T. Hironaka, "Comparison of bit serial computation with bit parallel computation for reconfigurable processor," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2010, pp. 388–393.

[53] *PARI/GP version* `2.11.2`, The PARI Group, Univ. Bordeaux, 2019, available from http://pari.math.u-bordeaux.fr/.

[54] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 76–86, 2019.

[55] Y. Wang, J. Chen, Y. Pu, and Y. Ha, "Energy-efficient arbitrary precision multi-bit multiplication with bi-serial in/near memory computing," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[56] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang, "Accelerating generalized linear models with ml-weaving: A one-size-fits-all system for any-precision learning," vol. 12, no. 7, pp. 807–821, 2019.

[58] Winograd and S., "On the time required to perform multiplication," *Journal of the Acm*, vol. 14, no. 4, pp. 793–802, 1967.

[59] T. J. Ypma, "Historical development of the newton–raphson method," *SIAM review*, vol. 37, no. 4, pp. 531–551, 1995.

[60] Y. Zhao, Z. Du, Q. Guo, S. Liu, L. Li, Z. Xu, T. Chen, and Y. Chen, "Cambricon-F : Machine Learning Computers with Fractal von Neumann Architecture," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 787–800.

[61] P. Zimmermann, "Karatsuba Square Root," INRIA, Research Report RR-3805, 1999. [Online]. Available: https://hal.inria.fr/inria-00072854

[57] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785