Journal of Parallel and Distributed Computing 173 (2023) 20–31

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU

Ning Ni [a], Yongxin Zhu [b,*]

[a] Shanghai Harbor e-Logistics software Co., Ltd., Shanghai 200080, PR China
[b] Shanghai Advanced Research Institute, Chinese Academy of Sciences, Shanghai 201210, PR China

ARTICLE INFO

Article history:
Received 28 November 2021
Received in revised form 29 June 2022
Accepted 27 October 2022
Available online 11 November 2022

Keywords:
zk-SNARK
ZKP
NTT/INTT
Montgomery multiplication
GPU acceleration

ABSTRACT

As a recent cryptography protocol, Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) allows one party to prove that it possesses certain information without revealing information to these untrusted proof provers. This mechanism has the ability to provide the function for constructing and verifying information integrity without privacy leaking. However, the computation kernels of zk-SNARK consume too much computing power and produce a significant performance bottleneck with the growing data volume and security requirement. In this paper, we take advantage of Graphic Processing Unit (GPU) to enhance zk-SNARK efficiency by accelerating the most time-consuming computation kernels: modular multiplication and Number-Theoretic Transform (NTT)/Inverse Number-Theoretic Transform (INTT) in Elliptic Curve Cryptography (ECC) pairing with two major improvements: (1) Adopting interval limbs multiply-add quaternary operation to directly accelerate ECC pairing by making full advantage of information entropy within the limited hardware bit width; (2) Data layout and shuffle methods in GPU global memory and shared memory for data space consistency maintenance accelerating NTT/INTT which indirectly works on ECC pairing. To the best of our knowledge, our work would be the first exploration to accelerate these improvements on GPU. The measured results show that our methods are able to accelerate modular multiplication and NTT/INNT by 1.22× and 4.67× times respectively compared with the previous GPU implementation. With these accelerated kernels, we are able to achieve 3.14× speedup for Groth16, which is the most efficient zk-SNARK implementation working on BLS12-381 ECC field. With the bottleneck tackled, our work will expand the deployment scenarios of zk-SNARK in Zero Knowledge Proof (ZKP).

1. Introduction

The cryptography protocol zk-SNARKs were proposed as relatively efficient proof systems by Goldwasser Micali and Rackoff [21], which enables a prover to convince a verifier that a statement is true or false without revealing anything else. They pursue the most efficient mathematical constructions in terms of proof and verification, which are two critical criteria for large scale data applications.

In practice, efficiency is the key indicator of zk-SNARKs because the complicated and huge calculations consume most of time on creating proofs, which hinders the implementation of some applications [36]. After years of research, various implementations of zk-SNARKs have been proposed and Groth's construction from EUROCRYPT 2016: Groth16 [23] is the most efficient and widely deployed implementation [5] with excellent empirical performance [1]. Furthermore, to accelerate zk-SNARKs, the polynomial modular multiplication based on NTT/INTT and modular exponentiation based on modular multiplication play the core role by taking up 30% and 70% of the proving time respectively [48] in ECC pairing.

Currently, cryptography algorithms are increasingly concerned with performance. A variety of acceleration solutions [42][4][47][29] were published with different hardware architectures such as CPU, Field Programmable Gate Array (FPGA), GPU. Among these research, modular multiplication and NTT/INTT are the focal points and GPU has received a lot of attention [45][30] because it provides much higher instruction throughput and bandwidth of global memory composed of off-chip Double Data Rate (DDR) than the CPU within a similar price and power envelope and offers much more programming flexibility than FPGA [33].

However, these studies are theoretical research only and not efficient enough to adapt the scalability of zk-SNARKs for many scenarios which require flexible computing, real-time response, low power consumption etc. In detail, device utilization is not high

* Corresponding author.
E-mail address: zhuyongxin@sari.ac.cn (Y. Zhu).

https://doi.org/10.1016/j.jpdc.2022.10.009
0743-7315/

enough because of lacking operators which are efficient enough in the cryptography field. In addition, the data parallelism performance degrades considerably for large cores counts in GPU [10] caused by the block access method with DDR memory outside of GPU chip.

Having identified the inefficiencies of current implementation of zk-SNARKs kernels, we reconstruct modular multiplication and NTT/INTT in algorithm complexity and data flow parallelism fields. We reduce the mathematical complexity by more meticulous calculation behaviors which are implemented by Parallel Thread Execution (PTX) instructions in multiplication reductions. Then we adapt them to multi-precision multiplications as a novel implementation technique of Coarsely Integrated Operand Scanning (CIOS) Montgomery algorithm to improve ECC efficiency. And targeting the Single Instruction Multiple Threads (SIMT) specification of GPU, we upgrade the data parallelism by splitting storage into levels and handling different kinds of data transpositions at different levels. We handle fine-grained data transposition in shared memory based on GPU cache and coarse-grained data transposition in global memory based on DDR to take advantage of GPU memory bandwidth. In the experiment and evaluation section, we profile the reconstructed modular exponentiation and NTT/INTT performance independently as well as their improvement for the zk-SNARKs.

The contributions of our work can be summarized as follows:

1. To accelerate zk-SNARK, we propose one mathematical scheme and corresponding implementation technique to replace the multiplication reduction in Montgomery algorithm for modular exponentiation and NTT/INTT respectively, which are kernels of the most efficient implementation: Groth16.
2. We verify that our mathematical scheme sufficiently exploits the utilization of bit width in hardware, thereby reducing the complexity of zk-SNARK.
3. We also propose a Dislocation Columns Transpose (DCTrans) implementation to parallelly handle large data transposition method for NTT/INTT in the GPU global memory by reconstructing data shuffle layout.
4. We conduct extensive experiments on improved zk-SNARK and other multiplication reduction based kernels, whose results indicate the effectiveness of our scheme and implementation.

The rest of our paper will be organized as follows. In Section 2, the development and current situation of GPU and zk-SNARKs are briefly reviewed. In Section 3, overview and analysis of a typical zk-SNARK algorithm is presented. In Section 4, we present modular exponentiation based on Montgomery multiple-precision multiplication reduction behavior and analysis. In Section 5, we describe and analyze the behavior of parallel NTT/INTT algorithm. In Section 6, experiment environment platform and results are described. In Section 7, result evaluation is presented with discussion. In Section 8, we wrap up the paper with concluding remarks. In Section 9, we append the brief relationship among zk-SNARK Groth16 ECC modular exponentiation and NTT.

2. Background

2.1. Overview of the zk-SNARK algorithm

Recently, a lot of zero knowledge based cryptographic primitives which work as powerful protocols appear in real-world applications [13][18][46] for privacy protection, and among these protocols zk-SNARKs are widely deployed by their outstanding performance in working size as well as efficiency [7][1].

In the production environment, the zk-SNARKs always involve solving handle quadratic arithmetic programs (QAPs) which consist

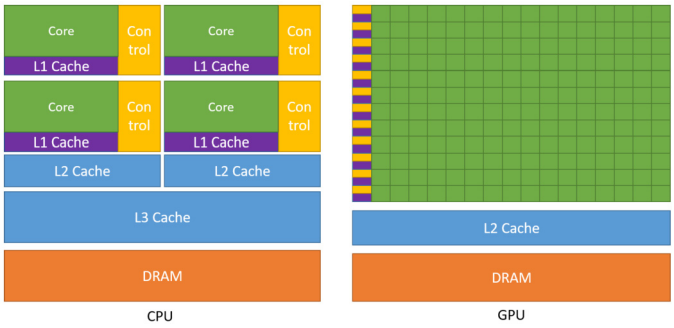

Fig. 1. GPU and CPU Architecture Comparison.

most of multiple large-scale tasks related to polynomial multiplication and multi-exponentiations on a large prime field. QAPs are designed for turning an NP-statements problem into equivalent arithmetic circuits by group elements on ECC pairing by considering an arithmetic circuit consisting of addition and multiplication gates over a finite field $\mathbb{F}$. We may designate some of the input/output wires as specifying a statement and use the rest of the wires in the circuit to define a witness. This gives us a binary relation $\mathscr{R}$ consisting of statement wires and witness wires that satisfy the arithmetic circuit, i.e., make it consistent with the designated input/output wires [23]. These algorithms are so heavy that they consume more than 95% running time [48]. Therefore, many scholars and experts proposed various optimization methods to improve the calculation time the size of the proof and the size of circuit [38][14][31]. In these researches, hardware acceleration schemes including CPU FPGA as well as GPU have different features following the regulation that the closer the solutions implemented by hardware level, the more performance can be tapped. CPU has the best Transactions Per Second (TPS) to get rid of hardware level implementations but lacks computation power. FPGA has the best Energy Efficiency Ratio (EER) brought by hardware but it is difficult to develop and hard to expand the scale of chip due to customized production. And GPU has the best performance with medium power consumed and development difficulty based on General Parallel Computing Architecture (GPCA) which specializes in hardware controlling and widely used in practice such as filecoin Zcash etc. Thanks to GPCA, the bottleneck of GPU zk-SNARKs implementation lies in the underlying hardware scheduling as well as mathematical methods.

2.2. Overview of the GPU developments and application

Parallel computing has become the only venue in sight for continued growth in application performance. Consequently, a significant portion of computation must be done with scalable parallel algorithms in future generations of hardware [24]. GPU is specialized for highly parallel computations and designed with more transistors devoted to data processing rather than data caching and pipeline control compared with CPU. Therefore, the distribution of chip resources is totally different between CPU and GPU architecture in Fig. 1 [33].

With additional logic units named Streaming Multiprocessor (SM), the GPU can accelerate computations on the CPU by executing parts of the code with high compute-loading. In order to match the enormous computation power, GPU is designed with much more Memory Management Unit (MMU) and has a much higher memory bandwidth than the CPU. But GPU reaches excellent performance by stacking hardware and short of automatic management strategies for threads arrangement in SMs/MMUs. The parallelism of program is guaranteed by the programmer causing threads to end at the same time.

During the execution on GPUs, most scientific computing applications are memory limited rather than processor logical resources limited [11]. The ratio of memory speed to logical units speed is an important factor that limits throughput. Inappropriate cache operations may cause the data serial access for GPU.

GPU parallel operates code by dangling all returned threads until the final thread return which is caused by the features of SIMT. Compared with the mutex locks of the thread switching mechanism to enhance the Instruction Per Clock (IPC) with data consistency in CPU, GPU performance tuning is totally different with its synchronization mechanism without guaranteeing to finish all threads at the same time other than dangling. And the data parallelism is as important as instruction parallelism for GPU performance improvement. Many applications leverage these parallel specifications of SIMT to run faster on the GPU than on the CPU [33] by conquering not only the processor logical power limited but also the memory limited, especially scientific computing such as cuFFT [34] cuBLAS [35].

2.3. Overview of the zk-SNARKs systems with GPU

The practical zk-SNARKs systems are always designed as integrity proof generations and verifiers referring to large files, which involves 2^{20} multi-exponentiations as well as 2^{20} points NTT/INTT as actual input data sizes in [48]. In calculation, we depict an element in Galois Field as a point in NTT/INTT. The large-scale modular exponentiations and polynomial multiplications which consume 95% time [48] are always assigned to the GPUs to break through the bottleneck of computing power because multiple-precision integer operations are computationally more expensive on contemporary CPUs than GPUs.

Thanks to the SIMT architecture, GPU has advantages over other architectures such as CPU and FPGA by off-chip memory access bandwidth within a similar price and power envelope by coalesced data access [33]. Although the huge off-chip bandwidth is provided by GPUs, challenges focus on improving data parallelism which greatly affects off-chip bandwidth utilization and always appears in out-of-order memory access applications such as NTT/INTT in the zk-SNARKs.

In summary, the computing power bottleneck in zk-SNARK is divided into data shuffle problem as well as mathematical problem and result in the $\mathcal{O}(\mathcal{N} \cdot \log_{\mathcal{R}} \mathcal{N})$ [50] paralleled calculation complexity and $\mathcal{O}(\mathcal{N})$ [20] butterfly data shuffle complexity, meanwhile research of modular exponentiation on GPU also made considerable progress [39].

2.4. Notations

As some general conclusions are derived with groups map generators and propositional relations, we summarize the involved notations in Table 1.

Table 1
Notations.

Notations	Paraphrase
$\mathcal{R}$	a binary relation
p	a prime order
$\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$	groups of prime order p
e	a bilinear map from $\mathbb{G}_1, \mathbb{G}_2$ to $\mathbb{G}_T$
g	a generator for $\mathbb{G}_1$
h	a generator for $\mathbb{G}_2$
$u_i(X), v_i(X), w_i(X), t(X)$	polynomial
$\mathbb{z}_p^\ell$	a field $\mathbb{z}$ with p as the prime order and ℓ as the umber of elements
$(\sigma, r) \leftarrow \text{Setup}(\mathcal{R})$	the setup produces a common reference string and a simulation trapdoor for the relation
$\pi \leftarrow \text{Prove}(\mathcal{R}, \sigma, a_1, \ldots, a_m)$	the prover algorithm takes input as a common reference string and returns an argument
$0/1 \leftarrow \text{Vfy}(\mathcal{R}, \sigma, a_1, \ldots, a_\ell, \pi)$	the verification algorithm takes a common reference string a statement and an argument as input then returns 0 (reject) or 1 (accept)
$\mathcal{A}$	input set for the quaternary multiply-add operation with the fixed bit width
$\mathcal{B}$	a set with twice bit width as A proven to be the output of the quaternary multiply-add operation
$\Leftrightarrow$	connector between two equivalence propositions

3. Analysis of zk-SNARK implementation: Groth16

3.1. Overview of the Groth16 algorithm

Groth16 as a typical zk-SNARK algorithm implementation which was created by Groth in 2016 [23], achieved a much more complex proof generation as well as a relatively simpler prover algorithm [43]. The Groth16 constructs a Non-Interactive Zero-Knowledge (NIZK) argument for arithmetic circuit satisfiability where a proof consists of only three group elements in practice. The minimum number of group elements to implement zk-SNARK with Groth16 are two theoretically which leads to a rapid expansion of calculations compared with the actual number three.

3.2. Implementation of the Groth16

Groth16 contains a proof generation and a prover, among which the proof is easy to verify by computing modular exponentiations in a small size proportional to the statement size and checking one single pairing product equation. Furthermore, Groth16 has the construction to be instantiated with any type of ECC pairings including the most efficient type III pairings.

The definitions of Groth16 elements are listed in Equ. (1) (2) (3):

Define relation generators $\mathcal{R}$:

$$\mathcal{R} = (\mathrm{p}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h, \ell, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^{m}, t(X)) \tag{1}$$

With $|p| = \lambda$ the relation defines a field $\mathbb{z}_p$ with:

$$\text{Statements: } (a_1, \ldots, a_i) \in \mathbb{z}_p^\ell \tag{2}$$

$$\text{Witnesses: } (a_{\ell+1}, \ldots, a_m) \in \mathbb{z}_p^{m-\ell}$$

Exist quotient polynomial $h(X)$:

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X) \tag{3}$$

There are two types of pairing-friendly elliptic curves ($\mathbb{G}_1$ and $\mathbb{G}_2$) [19] in Groth16 as well as official zk-SNARK [43] with typedef struct { FIELD c0; FIELD c1; } FIELD2 that the FIELD variable represents $\mathbb{G}_1$ and FIELD2 variable represents $\mathbb{G}_2$. Since the group element representations are smaller in $\mathbb{G}_1$ than in $\mathbb{G}_2$, the following NIZK arguments of proving processes are presented in Equ. (4) (5) as well as verifying processes in Equ. (6) with three elements A($\mathbb{G}_1$) B($\mathbb{G}_1$) and C($\mathbb{G}_2$):

Pick: $\alpha, \beta, \gamma, \delta, x \leftarrow \mathbb{z}_p^*$

Define: $r = (\alpha, \beta, \gamma, \delta, x)$

Let:

$$\delta_1 = \left(\begin{array}{c} \alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \dfrac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^{\ell} \\ \left\{ \dfrac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \right\}_{i=\ell+1}^{m}, \left\{ \dfrac{X^i t(x)}{\delta} \right\}_{i=0}^{n-2} \end{array} \right) \quad (4)$$

$$\delta_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1})$$

$(\sigma, r) \leftarrow \text{Setup}(\mathcal{R})$: Compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$

Pick: $r, s \leftarrow \mathbb{Z}_p$

$$\mathcal{A} = \alpha + \sum_{i=0}^{m} a_i u_i(x) + r\delta$$

$$\mathcal{B} = \beta + \sum_{i=0}^{m} a_i v_i(x) + s\delta$$

Let:

$$\mathcal{C} = \frac{a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + \mathcal{A}s + \mathcal{B}r - rs\delta \quad (5)$$

$\pi \leftarrow \text{Prove}(\mathcal{R}, \sigma, a_1, \ldots, a_m)$:

Computer $\pi = ([\mathcal{A}]_1, [\mathcal{C}]_1, [\mathcal{B}]_2)$

$0/1 \leftarrow \text{Vfy}(\mathcal{R}, \sigma, a_1, \ldots, a_\ell, \pi)$:

Parse $\pi = ([\mathcal{A}]_1, [\mathcal{C}]_1, [\mathcal{B}]_2) \in \mathbb{G}_1^2 \times \mathbb{G}_2$

Accept the proof if and only if:

$$[\mathcal{A}]_1 \cdot [\mathcal{B}]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^{\ell} a_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [r]_2 + [\mathcal{C}]_1 \cdot [\delta]_2 \quad (6)$$

3.3. Efficiency of Groth16

Although the authors of Groth16 proposed mathematical schemes such as NTT/INTT and Montgomery algorithms to reduce the complexities of polynomial modular multiplication and modular exponentiation, computations in the proving phase of zk-SNARK are particularly complicated and require significant computing time [48]. With Amdahl's law, the most effective way to enhance the efficiency of Groth16 is based on the NTT/INTT and Montgomery algorithms.

The traditional NTT/INTT and Montgomery implementations were proposed decades ago and have been thoroughly studied in almost all fields such as high-precision calculation as well as cryptography in the mathematical theory field [4]. To further efficiency improvement, parallel computing is introduced into these algorithms [48][30][12].

From the implementation perspective, as the $\mathbb{G}_2$ calculations are composed of $\mathbb{G}_1$ calculations, improvements of polynomial modular multiplication and modular exponentiation have full effect on both $\mathbb{G}_1$ and $\mathbb{G}_2$ except the less than 5% time consumed for data pre-processing [48] because they work on ECC pairing which is insensitive enough to ignore working memory space expansion.

4. Improving modular exponentiation by reducing complexity of Montgomery reduction

4.1. Modular exponentiation and Montgomery multiplication brief review

Modular multiplication operates the core of modular exponentiation, then they are involved in many common cryptographic algorithms with the decisive factors of performance. Modular multiplication consumes most of the time in ECC, which is similar to modular exponentiation in RSA. And it also affects the critical path of the composite field arithmetic for multiplicative inverters in Galois Field (2^8) in high throughput Advanced Encryption Standard (AES) [41].

The most popularly studied modular multiplication implementations for efficient are Barrett reduction [6] and Montgomery reduction [32]. Latest research proposed that the precomputation can be omitted by setting the special parameters in modular multiplication [27]. And the Barrett reduction has slightly larger complexity than Montgomery reduction in SIMT platform by the speedup ratios comparison in [49]. The Montgomery modular multiplication algorithm provides certain advantages in assembly instruction implementation over large integer numbers because it replaces division operations with shift operations in binary situations.

In recent years, many studies have been proposed to accelerate Montgomery multiplication such as variant: Number Theory Research Unit (NTRU) but they are not perfect with some defects which are unacceptable in zk-SNARK such as Learning with Errors over Rings (R-LWE). Note that, compared with the NTT algorithm, Montgomery multiplication reduction has the smaller complexity but no limit of the modulo length to implement modular multiplication. Therefore, the NTT algorithm is suitable for polynomial modular multiplication but not modular multiplication in the Galois Field.

4.2. Efficiency analysis of Montgomery multiplication

Up until now, SOS/CIOS/FIOS/FIPS methods are the most widely used Montgomery multiplication method and the CIOS has the most excellent overall performance [28]. The CIOS method is friendly to bitwise operations and provides the ability to solve the modular exponentiation calculation in certain instruction cycles in Algorithm 1:

Algorithm 1 CIOS Montgomery Multiplication [28].

Input: $p < 2^K, p' = -p^{-1} \bmod 2^w, w, s$

bit length: $K = s \cdot w, R = 2^K, a, b < p$

Output: $a \cdot b \cdot R^{-1} \bmod p$

```
1  T ← 0
2  for i ← 0 to s − 1 do
3  |  C ← 0
4  |  for j ← 0 to s − 1 do
5  |  |  (C, S) ← T[j] + a[i] · b[j] + C
6  |  |  T[j] ← S
7  |  (C, S) ← T[s] + C
8  |  T[s] ← S
9  |  T[s + 1] ← C
10 |  m ← T[0] · p' mod 2^w
11 |  (C, S) ← T[0] + m · p[0]
12 |  for i ← 1 to s − 1 do
13 |  |  (C, S) ← T[j] + m · p[j] + C
14 |  |  T[j − 1] ← S
15 |  (C, S) ← T[s] + C
16 |  T[s − 1] ← S
17 |  T[s] ← T[s − 1] + C
18 |  Return T
```

Obviously, the multiplication reduction in step 5 and 13 of Algorithm 1 loops the most frequently and has the potential to be optimized for efficiency.

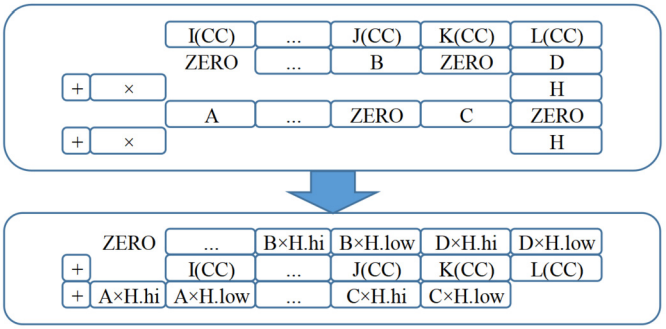

Fig. 2. Interval Limbs Multiplication in Montgomery Multiplication.

4.3. A novel implementation of CIOS Montgomery multiplication in GPU

The carry variable defined by step 3 is accessed by step 5 as well as 13 in Algorithm 1 and spends a lot of instructions splitting the terms and adding them to the correct result register [16]. We propose to separate the loops of step 5 and 13 as the interval limb multiplication in Fig. 2, which improves the loops efficiency by accessing and caching the carry in/out implicatively. In our implementation, there is no overlap between two multiplication results, so the overflow variable always records pure carry out other than the high bits of the multiplication result. And in the next section, the significant bits with overflow variable as carry in are proven to be non-overflowing.

In hardware, the efficient implied cache of GPU is designed with some special register and memory hierarchy such as condition code register (CC) as well as carry flag bit (CC.CF) which are integrated by integer operations instructions to hold carry in/out or borrow in/out. In Algorithm 2, we present the novel implementation of step 5 and 13 in Algorithm 1 with only one independent carry out:

Algorithm 2 Interval Limbs Multiplication.

Input: $p < 2^K, b < 2^{K+w}, q < 2^w, w, s, bit\ length: K = s \cdot w$
Output: $b = p \cdot q + b$

```
1  for i ← 0 to s − 2 do
2  |  b[i] ← madc.lo.cc(p[i], q, b[i])
3  |  b[i + 1] ← madc.hi.cc(p[i], q, b[i + 1])
4  |  i ← i + 2
5  b[s] ← addc()
6  for i ← 1 to s − 1 do
7  |  b[i] ← madc.lo.cc(p[i], q, b[i])
8  |  b[i + 1] ← madc.hi.cc(p[i], q, b[i + 1])
9  |  i ← i + 2
10 Return b
```

Compared with the Column Oriented Approach and Two-Pass algorithm in [16], our interval limb multiplication avoids shift operations and almost all independent additions with the same advantages that most of the carries are consumed by the next multiply instruction.

4.4. A mathematical scheme of interval limbs multiplication with proof of efficiency

To further improve the performance of our interval limbs multiplication for multi-precision multiplication by GPU hardware platform, we would like to introduce the quaternary multiply-add operation in style $\alpha_1 \cdot \alpha_2 + \beta_1 + \beta_2$ with the superiority of low computational consumption. The input/output of the quaternary multiply-add operation makes full utilization of information entropy and guarantees its uniform bit width, which was proved as follows in Equ. (7)–(12):

Definitions.

$$\begin{gathered}
a_0, \dots a_4 \text{ are natural number with the bit width } \mathcal{N} \\
\text{in range } i = 2^{\mathcal{N}} - 1 \\
\mathcal{A} = \{a_0, \dots a_i\} \\
b_0, \dots b_f \text{ are natural number with the bit width } 2 \cdot \mathcal{N} \\
\text{in range } j = 2^{2\cdot\mathcal{N}} - 1 \\
\mathcal{B} = \{b_0, \dots b_j\} \\
\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathcal{A} \\
f(\alpha_1, \alpha_2, \beta_1, \beta_2) = \alpha_1 \cdot \alpha_2 + \beta_1 + \beta_2 \\
\mathcal{C} = \{\dots\} \\
\mathcal{M}ax(\mathcal{C}) \in \mathcal{C} \text{ and } \mathcal{M}ax(\mathcal{C}) \geq \forall \in \mathcal{C}
\end{gathered} \tag{7}$$

Proof. The $\mathcal{M}ax(\mathcal{A})$ and $\mathcal{M}ax(\mathcal{B})$ are determined by their bit width $\mathcal{N}$ and $2 \cdot \mathcal{N}$ in (7):

$$\begin{aligned}
\mathcal{M}ax(\mathcal{A}) &= 2^{\mathcal{N}} - 1 \\
\mathcal{M}ax(\mathcal{B}) &= 2^{2\cdot\mathcal{N}} - 1
\end{aligned} \tag{8}$$

Any variable decreases of α_1, α_2, β_1, β_1 leads to decrease of $\alpha_1 \cdot \alpha_2 + \beta_1 + \beta_2$, which is proven by recursion in (9) and (10):

$$\begin{aligned}
f(\alpha_1, \alpha_2, \beta_1, \beta_2) \geq & f(\alpha_1 - 1, \alpha_2, \beta_1, \beta_2) \Leftrightarrow 0 \geq -\alpha_2 \\
f(\alpha_1, \alpha_2, \beta_1, \beta_2) \geq & f(\alpha_1 - 1, \alpha_2, \beta_1, \beta_2) \\
\geq & f(\alpha_1 - 2, \alpha_2, \beta_1, \beta_2) \\
\geq & \dots \\
f(\alpha_1, \alpha_2, \beta_1, \beta_2) \geq & f(\alpha_1, \alpha_2 - 1, \beta_1, \beta_2) \\
\geq & f(\alpha_1, \alpha_2 - 2, \beta_1, \beta_2) \\
\geq & \dots
\end{aligned} \tag{9}$$

$$\begin{aligned}
f(\alpha_1, \alpha_2, \beta_1, \beta_2) > & f(\alpha_1, \alpha_2, \beta_1 - 1, \beta_2) \Leftrightarrow 0 > -1 \\
f(\alpha_1, \alpha_2, \beta_1, \beta_2) > & f(\alpha_1, \alpha_2, \beta_1 - 1, \beta_2) \\
> & f(\alpha_1, \alpha_2, \beta_1 - 2, \beta_2) \\
> & \dots \\
f(\alpha_1, \alpha_2, \beta_1, \beta_2) > & f(\alpha_1, \alpha_2, \beta_1, \beta_2 - 1) \\
> & f(\alpha_1, \alpha_2, \beta_1, \beta_2 - 2) \\
> & \dots
\end{aligned} \tag{10}$$

Then we calculate the maximum value of $\alpha_1 \cdot \alpha_2 + \beta_1 + \beta_2$ by introducing (8) in the conclusion of (9) and (10):

$$\begin{aligned}
& \mathcal{M}ax\left(f(\alpha_1, \alpha_2, \beta_1, \beta_2)\right) \\
& = f(\mathcal{M}ax(\mathcal{A}), \mathcal{M}ax(\mathcal{A}), \mathcal{M}ax(\mathcal{A}), \mathcal{M}ax(\mathcal{A})) \\
& = f\left(2^{\mathcal{N}} - 1, 2^{\mathcal{N}} - 1, 2^{\mathcal{N}} - 1, 2^{\mathcal{N}} - 1\right) \\
& = \left(2^{\mathcal{N}} - 1\right) \cdot \left(2^{N} - 1\right) + \left(2^{N} - 1\right) + \left(2^{\mathcal{N}} - 1\right) \\
& = 2^{2\cdot\mathcal{N}} - 1
\end{aligned} \tag{11}$$

$$\mathcal{M}ax(\mathcal{B}) = \mathcal{M}ax\left(f(\alpha_1, \alpha_2, \beta_1, \beta_2)\right) \geq f(\alpha_1, \alpha_2, \beta_1, \beta_2) \tag{12}$$

With Equ. (12) we reveal that the output of the quaternary operation $(\alpha_1 \cdot \alpha_2 + \beta_1 + \beta_2)$ has the same maximum value with the variable twice bit width of the input. This feature improves the efficiency of multi-precision multiplication by avoiding the carrying in/out of uniform bit width types such as int (32 bit) long

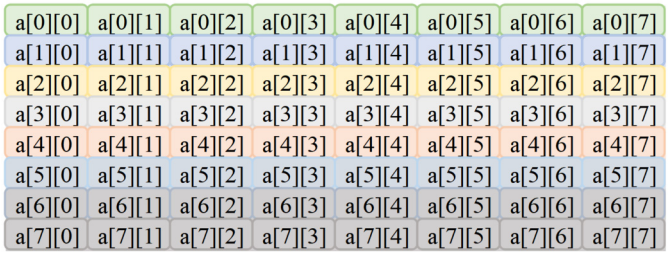

Fig. 3. Bank conflicts Caused by Row Data Structure.

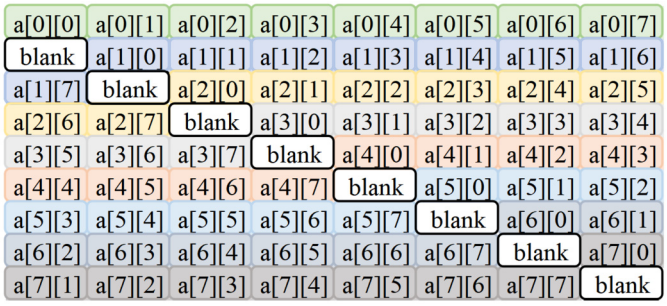

Fig. 4. Data Structure Avoiding Bank conflicts.

(64 bit) between limbs. Then compared with other operations with less complexity of computation which shall be no more than one multiplication and three additions such as $(\alpha_1 \cdot \alpha_2 + \beta_1)$ and $(\alpha_1 + \alpha_2 + \beta_1)$, they obvious dissatisfy Equ. (12) with the under-utilized uniform output bitwidth.

From the program side of our quaternary multiply-add operation, PTX code of NVIDIA GPU provides assembly instructions with similar functions. For the efficient implementation of linear algebra on the GPU platform, introduced in Equ. (13):

$$\begin{aligned} & madc\{.hi, .lo\}\{.cc\}.type \quad d, a, b, c; \\ & t = a \cdot b; \\ d = t < 63..32 > + c + CC.CF; \quad & //\text{for . hi variant} \\ d = t < 31..0 > + c + CC.CF; \quad & //\text{for . lo variant} \end{aligned} \tag{13}$$

Step 5 and 13 in Algorithm 1 of the CIOS Montgomery algorithm are also the critical path in multi-precision multiplication and benefits from the interval limbs multiplication.

4.5. The Montgomery multiplication data structure improvement for GPU cache architecture

As the regulation of GPU on-chip cache, data congestion named bank conflict would be caused by the big integer data access with the row structure which contains limbs of one big integer in every row. We illustrate this structure in Fig. 3 and a[i] variable stores a whole big integer with a[i][j] stores every limb of the big integer.

The bank conflicts occur in situations where parallel threads access data in the same bank. For example, parallel threads accessing a[parallel_thread_id][0] located in the same column in Fig. 3 causes bank conflicts. To eliminate the bank conflicts we propose to upset the index of the 2D array by inserting blanks in Fig. 4.

The blanks in Fig. 4 distribute parallel threads accessing a[parallel_thread_id][0] to different banks (columns) by the multi threads access.

5. Parallel NTT/INTT algorithm for zk-SNARK

5.1. NTT/INTT algorithm brief review

NTT/INTT [37] is based on Fast Fourier Transform/Inverse Fast Fourier Transform (FFT/IFFT) which is efficient and stable to establish complex functions in many fields such as image registration [17]. These FFT/IFFT based functions are appropriate to execute in parallel and make full use of SIMT architecture including GPU. But the topological structures reconstruction for data flow in storage space is necessary caused by the capabilities difference between GPU and CPU. Compared with the Single Instruction Single Datastream (SISD) which is designed to pursue the ultimate performances by improving the flexibility of threads shifting mechanism in hardware to guarantee logical unit utilization, the SIMT is rich in threads resources but short of self-management and rely on programmers to synchronize tasks in software. Therefore, the improvements of the NNT/INNT in GPU are always related to algorithm complexity based on mathematical principles [9].

The FFT/IFFT evolves from Discrete Fourier Transform (DFT) algorithms to transform a set of numbers from one basis to another. Generally, the FFT/IFFT is used to accelerated convolution operations in cryptography but with the accuracy problem. The NTT/INTT is proposed on the base of FFT/IFFT to solve the accuracy and performance problem of polynomial modular multiplication by replacing complexity $\mathcal{O}(\mathcal{N}^2)$ with $\mathcal{O}(\mathcal{N} \cdot \log_{\mathcal{R}} \mathcal{N})$ [22] of the $\hbar(X)$ coefficients in Equ. (14):

$$\hbar(X) = \frac{\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) - \sum_{i=0}^{m} a_i w_i(X)}{t(X)} \tag{14}$$

The whole step for a big integer polynomial modular multiplication requires twice NTT and once INTT for domain conversion with once parallel scalar multiplications in Equ. (15):

$$\begin{aligned} \hat{u} &= NTT\left(\sum_{i=0}^{m} a_i u_i(X)\right) \\ \hat{v} &= NTT\left(\sum_{i=0}^{m} a_i v_i(X)\right) \\ \hat{w} &= \hat{u} \times \hat{v} \\ w = INTT(\hat{w}) &= \sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) \end{aligned} \tag{15}$$

The larger the size of the data provided to the NTT, the more efficiency is presented. In practical, the comparatively small NTT/INTT has 256 points with 16 bit width in [30].

5.2. The efficiency and data shuffle analysis of NTT/INTT

The most popular NTT/INTT algorithm method named butterfly in Fig. 5 has the theoretical lowest complexity except for the data shuffle at every layer including transposition and bit reserve operation implementation.

It is evident that every unit continuously changes the data interaction object in every layer and achieves direct communication at all times [44] in the butterfly structure. This is a typical memory limited situation in GPU because global memory access is inevitable and frequent. In practice, data order and shape impact efficiency greatly in many computing fields, including linear algebra image processing as well as the FFT/IFFT/NTT/INTT with the SIMT processors implementation. Therefore, the computational efficiency of FFT/IFFT/NTT/INTT is still not high, due to their low utilization of computational units [10].

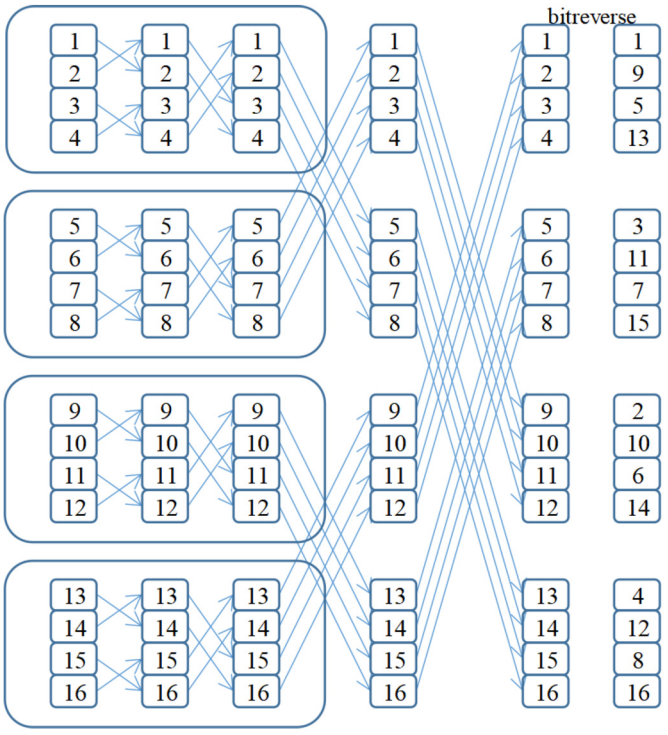

Fig. 5. NTT/INTT Butterfly Algorithm Data Flow.

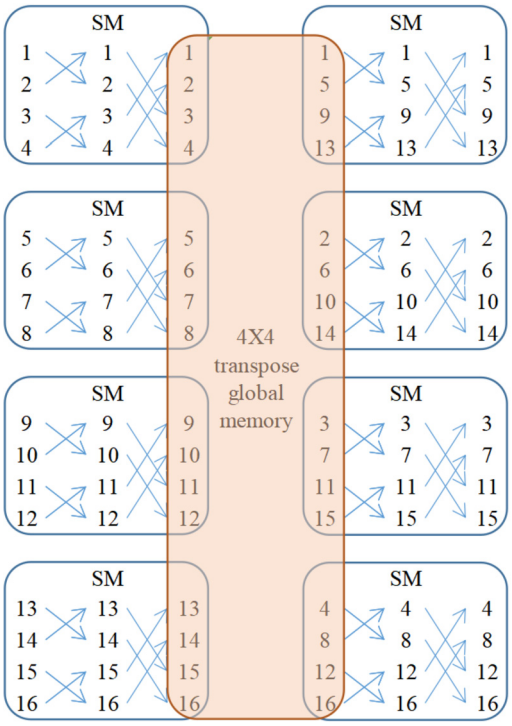

Fig. 6. Transposition Improve Butterfly Algorithm Within SMs.

5.3. The improvements of NTT/INTT data shuffle

5.3.1. The butterfly algorithm improvement

Our NTT/INTT improvement of GPU data parallelism is based on the data layout and shuffle operations rearrangement that transfer disorder memory access from global memory to GPU shared memory based on cache. With the memory hierarchy of GPU, cache has the highest speed and flexible access but it is only available for their located SM [8]. Therefore, the data parallelism is improved for the data size exceeding shared memory storage and always focuses on data exchange among SMs other than coarse-grained optimizations in [10]. Considering the butterfly algorithm operating the pairing elements by bitonicsort which are too far to store in the same SM, we take advantage of transposition to reconstruct the butterfly algorithm in Fig. 6 to solve the data fragmentation problem.

As the global memory is slower and more inflexible than cache in GPU, we split the whole butterfly algorithm into sub butterfly algorithms with small enough size to be stored in cache for the bitonicsort operations in our improvement. Due to the nature of matrix transposition, bit-reserve algorithm is also available to be assimilated into the sub butterfly algorithms by spreading bit-reserve on different bit width in Fig. 7.

With the above operations, the NTT/INTT data shuffle problem is turned to matrix transposition problem.

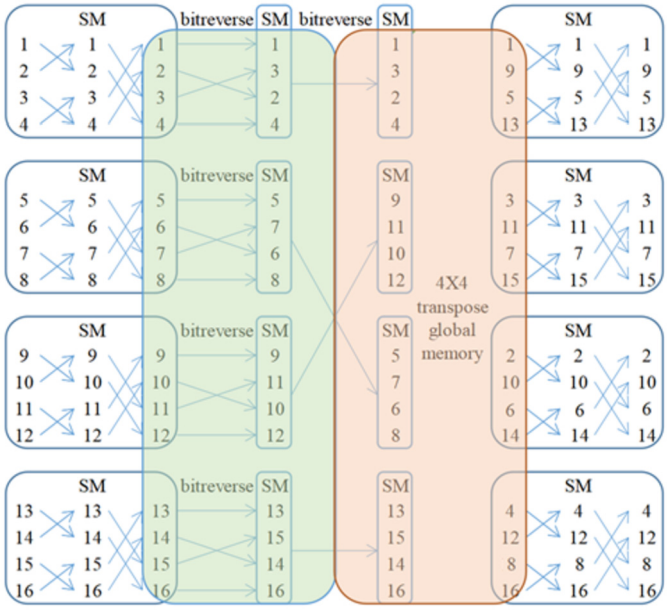

Fig. 7. Assimilate Bit-reserve into Butterfly Algorithm Within SMs.

5.3.2. The transposition improvement in global memory of GPU

It is available for high efficiency transposition implementation within the data which is small enough to be stored in one SM cache. In detail, we separate transposition into tiny sub transpositions to execute in cache and handle overall transposition among these tiny sub transpositions in global memory. In Fig. 8, we present our transposition improvement with tiny sub transpositions in block and overall transposition marked by colors.

The global memory out of the GPU is designed to block access and is restricted by the access width and data fragmentation. The data is prescribed as 32/64/128 bytes bit width [33] and stored in

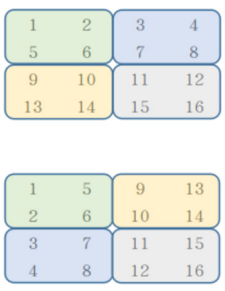

Fig. 8. Tiny Sub Transpositions and Overall Transposition.

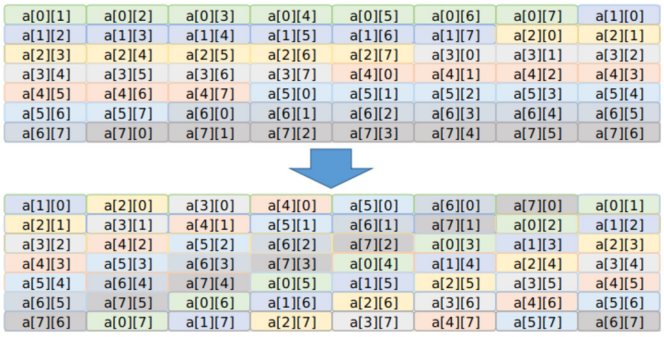

Fig. 9. GPU On-chip Shared Memory Tiny Transposition Without Bank conflicts.

a continuous address for SMs to trigger coalesced memory access which is to approach the bandwidth limit of global memory. We propose Algorithm 3 as the tiniest layer of any size of the transposition, which has the ability to stack layers or replace the array elements with structs to adapt overall transposition size as well as hardware architectures of SIMT.

Algorithm 3 Tiniest Layer of Overall Transposition.

Precondition: tiniest bitreverse of bit width 2

[00] [01] [10] [11] ⇒ [00] [10] [01] [11]

Pre Step1: inner bitreverse:

a[1]	a[2]	a[3]	a[4]	⇒	a[1]	a[3]	a[2]	a[4]
a[5]	a[6]	a[7]	a[8]	⇒	a[5]	a[7]	a[6]	a[8]
a[9]	a[10]	a[11]	a[12]	⇒	a[9]	a[11]	a[10]	a[12]
a[13]	a[14]	a[15]	a[16]	⇒	a[13]	a[15]	a[14]	a[16]

//on-chip shared memory without bank conflicts

//data shuffle among columns

Pre Step2: outer bitreverse:

a[1]	a[3]	a[2]	a[4]		a[1]	a[3]	a[2]	a[4]
a[5]	a[7]	a[6]	a[8]	×	a[9]	a[11]	a[10]	a[12]
a[9]	a[11]	a[10]	a[12]		a[5]	a[7]	a[6]	a[8]
a[13]	a[15]	a[14]	a[16]		a[13]	a[15]	a[14]	a[16]

//coalesced global memory access by row elements

//data shuffle among rows

Step1: tiny transposition:

a[1]	a[3]	⇒	a[1]	a[9]	a[2]	a[4]	⇒	a[2]	a[10]
a[9]	a[11]		a[3]	a[11]	a[10]	a[12]		a[4]	a[12]
a[5]	a[7]	⇒	a[5]	a[13]	a[6]	a[8]	⇒	a[6]	a[14]
a[13]	a[15]		a[7]	a[15]	a[14]	a[16]		a[8]	a[16]

//on-chip shared memory without bank conflicts in bottom 2×2 elements

Step2: outer transposition:

a[1]	a[9]	a[2]	a[10]		a[1]	a[9]	a[5]	a[13]
a[3]	a[11]	a[4]	a[12]	⇒	a[3]	a[11]	a[7]	a[15]
a[5]	a[13]	a[6]	a[14]		a[2]	a[10]	a[6]	a[14]
a[7]	a[15]	a[8]	a[16]		a[4]	a[12]	a[8]	a[16]

//coalesced global memory access

//the upper layer of transpose the bottom layer

//by 2×2 structs as one element

Furthermore, we provide the data structure with dislocation columns to implement tiny transposition (DCTrans) in shared memory to avoid bank conflicts in Fig. 9.

Compared with Fig. 4, the data structure in Fig. 9 is specified for transpose without blanks as well as diagonal elements, which reduces shared memory consumption by two rows. And the whole shared memory works for single transpose other than multi-transpose in [26], which makes the large-scale transpose more efficient. With our data layout and shuffle operations, NTT/INTT is totally parallel in both cache and global memory without bank conflicts and unconsolidated data access. This improvement significantly improves the overall performance of NTT/INTT by feeding data more efficiently to logic units in GPU. Meanwhile, the multiplication reduction is also involved in NTT/INTT as the multiplication in FFT/IFFT to handle the twiddle factors and our DCTrans can be superimposed with the interval limbs multiplication for further improvement.

Generally, our DCTrans method achieves the same improvement for NTT/INTT as well as float point types FFT/IFFT by exploiting similar data shuffle operations and be applicable to many signal-based algorithms. Compared with the generic and popular parallel FFT library cuFFT applied in almost all CUDA based algorithms, which handles input in global memory [3], DCTrans also reduces the global memory access a lot by the radix FFT totally executed in shared memory.

Table 2
Hardware Platform for Experiment.

	Hardware	Architecture	Cores	GFLOPS/TOPS	Language
CPU	Intel(R) Core(TM) CPU i5-9400F @ 2.90GHz	X86	6	344.7 GFLOPS	Rust
	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz		28	800.5 GFLOPS	
GPU	2080 Ti	Turing	4352	13.45 TOPS	OpenCL
	1080 Ti	Pascal	3584	11.34 TOPS	

6. Experiment environment platform and result

6.1. Experiment software platform

In coding, bellman [15] is an open source Groth16 implementation within a Rust create branch named bellperson [12], which works as the zk-SNARKs in the filecoin project. The bellperson provides circuit traits and primitive structures, as well as basic gadget implementations such as boolean and number abstractions. Most calculations of bellperson are implemented by modular exponentiation and NTT/INTT through the Rust GPU create ocl [2] as the interface of Open Computing Language (OpenCL) [25]. Our improvements are implemented by code reconstruction involving the above two programming languages:

The Rust is a programming language that focuses on concurrency safety, and supports multi-paradigm languages such as functional, imperative, and generic programming paradigms. It is designed for performance as well as superior memory security and suitable for cryptography.

The OpenCL is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. The OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories, including cryptography.

6.2. Experiment hardware platform

The experimental hardware platforms are designed close to the production environment for authenticity. In SIMT architecture, the high-level hardware platform consists of a 2080 Ti recommended by the official filecoin team and the low-level hardware platform consists of a 1080 Ti as the previous generation flagship GPU. In SISD architecture, Intel(R) Core(TM) i5-9400F and Intel(R) Xeon(R) CPU E5-2697 v3 are provided as desktop and server CPU hardware platforms in our experiment. All the detailed computing power information is concentrated in Table 2.

6.3. Experiment profile platform

The NVIDIA® Nsight™ Visual Studio Edition is a modified popular Integrated Development Environment (IDE), which profiles OpenCL performance analysis by the Graphical User Interface (GUI) friendly. In addition to performance analysis, we also profile the parallelism of modular exponentiation and NTT/INTT on GPUs compared with CPUs with the homology of CPU implementations in bellperson.

6.4. Experiment parallel parameters and result

As the applications of our modular exponentiation and NTT/INTT always involve large files in GB size, the parallel parameters are designed by the consideration of computing power as well as memory bottleneck.

The GPU modular exponentiation divides the exponent values into smaller windows and the bases into several groups, which highly increases the number of threads running in parallel. As the parallelism of GPU modular exponentiation, two engineering restrictions are presented about the parallel parameters as follows:

1. The final value is accumulated from the returns of all threads. The more threads allocated, the heavier the SISD computation would be produced.
2. The workspace grows exponentially with window size on GPU global memory as the product of windows quantity groups quantity as well as 1 << window size.

The GPU NTT/INTT accesses every point in each calculation layer in Fig. 5, and any data transmission between host and global memory is harmful to efficiency. With our works theoretically improved GPU global memory access, threads with half of the NTT/INTT points number are the most efficient. The corresponding block size depends on the data shuffle size in the shared memory of located block.

In the production environment of [12], modular exponentiations have variable parallelism with the typical 7488999 elements data size as the most of the computation consumed, which leads to the engineering restriction 2 above. Considering the global memory size and engineering restriction 1, the threads arranged as twice the number of GPU cores and the block size of 64 with 2^{20} elements sliced data size are the most effective configuration for 1080 Ti and 2080 Ti. The NTT/INTT on GPU is only restricted by the global memory size. Thereby, the 4GB with 2^{27} points is the only size of actual NTT/INTT and the most appropriate data size with the other twiddle factors storage, and the corresponding parallel parameters are 2^{26} threads with block size of 256.

Within the X86 SISD platforms, we profile two CPUs of typical server and desktop model in Table 3 by varying the number of cores from one to all. The analysis of modular exponentiation and NTT/INTT parallelism on CPUs is listed in Table 3/4 and Fig. 10/11.

The analysis above reveals that SISD architecture reaches the bottleneck of parallel computing at a relatively low cores number, which causes significant performance degradation in applications such as modular exponentiation and NTT/INTT. This issue is caused by restriction of memory bandwidth and cache size but not the effect of nonlinear part in Amdahl's law. The more data pieces are separated for parallelization, the more MMU behaviors are executed, which prevents performance improvement in SISD architecture.

To evaluate our efficiency and parallelism improvements of the SIMT platforms (GPUs), we profile the original and reconstructed GPU modular exponentiation and NTT/INTT within 1080 Ti and 2080 Ti with the concentrated profile results in Table 5.

Table 3
Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz Duration.

CPU Speedup Parallel Number	1core	2core	3core	4core	5core	6core	7core
Modular exponentiation(ms)	263493	146285	97424	79603	63298	61938	60130
Polynomial multiplication NTT(ms)	175677	93364	93715	55545	55218	54928	55301
CPU Speedup Parallel Number	8core	9core	10core	11core	12core	13core	14core
Modular exponentiation(ms)	43195	42793	42603	42042	41503	39755	39195
Polynomial multiplication NTT(ms)	35328	35275	35215	35292	35234	35228	35222
CPU Speedup Parallel Number	15core	16core	17core	18core	19core	20core	21core
Modular exponentiation(ms)	30701	31114	30416	30476	31167	30497	30618
Polynomial multiplication FFT(ms)	35271	37145	36933	37108	37180	36898	36878
CPU Speedup Parallel Number	22core	23core	24core	25core	26core	27core	28core
Modular exponentiation(ms)	30489	30856	31085	30673	30611	31147	31251
Polynomial multiplication NTT(ms)	36962	37076	37056	36960	37045	36884	36883

Table 4
Core(TM) CPU i5-9400F@ 2.90GHz Duration.

CPU Speedup Parallel Number	1core	2core	3core	4core	5core	6core
Modular exponentiation(ms)	190088	105079	66679	53732	41204	40927
Polynomial multiplication NTT(ms)	138262	73649	73543	41054	41149	41226

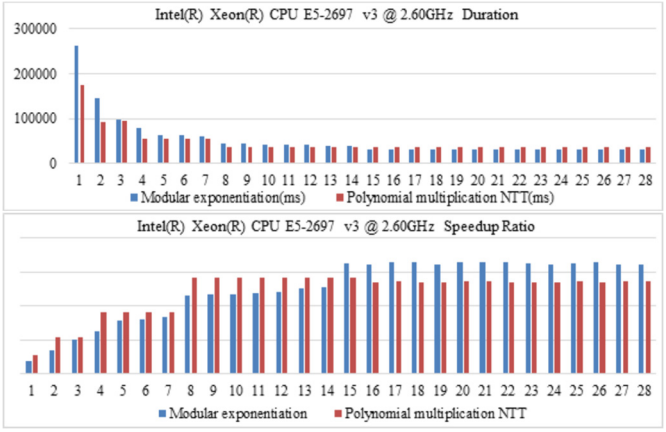

Fig. 10. Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz Duration and Speedup Ratio.

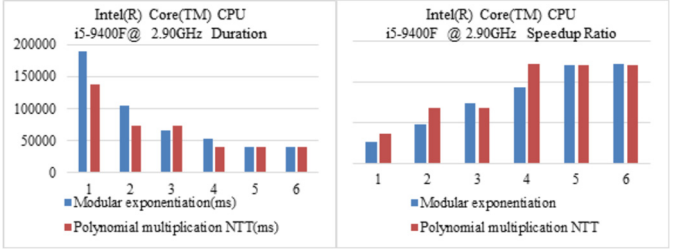

Fig. 11. Intel(R) Core(TM) CPU i5-9400F@ 2.90GHz Duration and Speedup Ratio.

Table 5
Original and Improved Profile of Multi-exponentiations and NTT on GPU.

	Multi-exponentiations(ms)		NTT/INTT(ms)		
	Original	Improved	Original	Improved	First Radix Improved
NVIDIA® GeForce® RTX 1080 Ti	7984.60	6223.52	5367.45	1685.73	673.31
NVIDIA® GeForce® RTX 2080 Ti	2706.20[31]	2220.18	1602.36[31]	343.88	91.3

Table 6
Montgomery Multiplication Reduction Throughput and Speedup Ratio Comparison.

Works	Overall Time(ms)	Throughput(MBps)	Throughput Speedup Ratio
filecoin[31]	2706.20	17.74	21.87%
our work	2220.18	21.62	
Works	**Addition Operations and Data Flow Time(ms)**	**Montgomery Kernel(ms)**	**Montgomery Kernel Speedup Ratio**
filecoin[31]	1175.3	$2706.20 - 1175.3 = 1530.9$	1.47×
our work		$2220.18 - 1175.3 = 1044.88$	

As the two model GPUs perform similar in behavior and the 2080 Ti has the better efficiency in experiment as well as production environment, we choose the result of 2080 Ti to participate in comparison with other works.

7. Result evaluation

7.1. Modular exponentiation evaluation

We propose to evaluate works of modular exponentiation by application throughput and the kernel speedup ratio which is defined as Equ. (16), for the reason that modular exponentiation data scales vary in a large range in the cryptography field with the uncertain data slicing overhead.

$$\text{Speedup Ratio} = \frac{\frac{1}{\text{Improved Time Consumed}}}{\frac{1}{\text{Original Time Consumed}}} \tag{16}$$

To approximate the production environment, we set up the data size as $2^{20} \cdot 384$ bit as the bases and $2^{20} \cdot 256$ bit as the exponentiations, which is large enough to produce typical data slices in [12]. Furthermore, we remove the Montgomery function to get the addition operations and data flow cost 1175.3 ms on average. With these precondition and intermediate results, we get the typical multiplication reduction speedup ratio in Table 6.

We achieve 1.47× speedup ratio of Montgomery kernel compared with 1.25× in [45]. And in the modular exponentiation application our work achieves 21.87% throughput enhancement compared with 20% in [45].

7.2. NTT/INTT evaluation

Since the NTT/INTT is divided and conquered, ignoring the huge size difference in practice, the tiny 'butterfly's are the same as the Fig. 7 shows. And the varied size NTT/INTTs are comparable among works such as Kyber [30] zk-SNARK [48][12]. In order to get an impartial evaluation result, we define the other two evaluation standards:

1. Relative efficiency is defined as the ratio of complexity divided by consumed time.
2. Improvement ratio is defined in Amdahl's law from the original performance to improved performance in Equ. (17).

$$\text{Improvement Ratio} = \frac{\text{Improved Performance} - \text{Original Performance}}{\text{Original Performance}} \times 100\% \tag{17}$$

The NTT/INTT algorithm generally contains two kinds of complexities in Table 7.

As the complexities definitions above and $O()/\dot{O}()$ are all linear, we abstract the complex of NTT/INTT in Equ. (18) and unify the unit dimension as $\frac{O(1)\cdot\dot{O}(1)}{millisecond}$.

Table 7
Complexity Definition and Calculation.

Complexity Definitions	Complexity
The number of modular multiplications involved in NTT/INTTs	$\mathcal{O}(\mathcal{N} \times \log_{\mathcal{R}} \mathcal{N})$[38] with the $\mathcal{N}$ of points length and $\mathcal{R}$ as radix
The number of modular multiplication reduction loops involved in Montgomery algorithm	$\mathcal{O}(\mathcal{N}^2)$ with the $\mathcal{N}$ of bit width

Table 8
NTT/INTT Relative Efficiency Comparison.

Similar Works	Platform	Data size & Point of NTT	Complexity	Time(ms)	Relative Efficiency
filecoin[31]	RTX 2080 Ti	2^{27}×256bit full point	$\mathcal{O}(2^{27} \cdot log_2 2^{27})$ $\dot{\mathcal{O}}(256^2)$	1602.36[31]	$2.375\times10^{14}/1602.36 \approx 1.48\times10^{11}$
	Intel(R) Xeon(R) E5-2697 v3			36883[29]	$2.375\times10^{14}/36883 \approx 6.44\times10^{9}$
Kyber[12]	RTX 2060 SUPER	92524×16bit 256 point*1	92524 $\mathcal{O}(256 \cdot log_2 256)$ $\dot{\mathcal{O}}(16^2)$	3[12]*1	$4.851\times10^{11}/3 \approx 1.62\times10^{11}$
	Intel CPU i9-9700F			391.2[12]*2	$4.851\times10^{11}/391.2 \approx 1.24\times10^{9}$
Zcash and filecoin[6]	ASIC	2^{20}×256bit full point	$\mathcal{O}(2^{20} \cdot log_2 2^{20})$ $\dot{\mathcal{O}}(256^2)$	11[6]	$1.374\times10^{12}/11 \approx 1.25\times10^{11}$
	Intel(R) Xeon(R) Gold 6145			333[6]	$1.374\times10^{12}/333 \approx 4.13\times10^{9}$

Our Works	Platform	Data size & Point of NTT	Complexity	Time(ms)	Relative Efficiency
multiplication reduction improvement	RTX 2080 Ti	2^{27}×256bit full point	$\mathcal{O}(2^{27} \cdot log_2 2^{27})$ $\dot{\mathcal{O}}(256^2)$	1002.38	$2.375\times10^{14}/1002.38 \approx 2.37\times10^{11}$
data shuffle improvement of butterfly				704.31	$2.375\times10^{14}/704.31 \approx 3.37\times10^{11}$
total improvement				343.88	$2.375\times10^{14}/343.88 \approx 6.91\times10^{11}$
total improvement without global memory access*3			$\mathcal{O}\left(\frac{2^{27} \cdot log_2 2^{27}}{3}\right)$ $\dot{\mathcal{O}}(256^2)$	91.3	$7.916\times10^{13}/91.3 \approx 8.67\times10^{11}$

*1 GPU part of this work reached 92524 per second of 256 bit Kyber encapsulation (encap/s) and NTT consumed 0.3%.
*2 In this work, the Hash and KDF time consumed by CPU increases in proportion from 10.1% to 87.8% with absolute time unchanged. Meanwhile NTT time consumed from 4.5% to 0.3% by GPU acceleration. Considering these two effects, the total speedup ratio is $\frac{4.5\%}{0.3\%} \cdot \frac{87.8\%}{10.1\%} \approx 130.4$. Then the absolute CPU NTT time = GPU NTT time × speedup ratio.
*3 Refer to the butterfly algorithm in Fig. 5 the first sub-NTTs/INTTs are completed in their located SMs without data interaction.

$$\mathcal{O}\left(\mathcal{N} \cdot \ell o g_{\mathcal{R}} \mathcal{N}\right) \cdot \dot{\mathcal{O}}\left(\dot{\mathcal{N}}^2\right) = \mathcal{N} \cdot \ell o g_{\mathcal{R}} \mathcal{N} \cdot \dot{\mathcal{N}}^2 \cdot \mathcal{O}(1) \cdot \dot{\mathcal{O}}(1) \tag{18}$$

With the unified complexities for NTT/INTT, we evaluate our improvements for NTT/INTT compared with the latest software and hardware works comprehensively including CPU [43] GPU [30][12] and ASIC [48] implementations in both academic and industry by relative efficiency and improvement ratio.

7.2.1. NTT/INTT relative efficiency evaluation

Our total improvement NNT/INTT rank first in Table 8 by relative efficiency based on complexity and data feeding improvement.

As the GPU based Kyber applications experiments [30] platform is an RTX 2060 SUPER which has a lower computing power than RTX 2080 Ti we introduce Tera Operations Per Second (TOPS) to evaluate the relative efficiency of computing power in Table 9.

Table 9 reveals that our works are also ahead of similar works of NTT/INTT obviously with the total improvements of the relative efficiency per TOPS.

7.2.2. NTT/INTT improvement and speedup ratio evaluation

The improvement ratio is another method to evaluate our NTT/INTT in Table 10.

In Table 10, our improvement ratios are the highest. Then we list the speedup ratios of all works in Table 11.

In Table 11, our works reach 150.5× as the highest speedup ratio. Meanwhile, compared with the previous work [12] our works achieve 4.67× in practice as well as 5.86× in theory speedup ra-

Table 9
Efficiency of Computing Power.

Works	Relative Efficiency	GPU Model	TOPS(int32)	Relative Efficiency/TOPS
Kyber[12]	1.617×10^{11}	RTX 2060 SUPER	7.181	2.25×10^{10}
our multiplication reduction improvement	2.37×10^{11}	RTX 2080 Ti	13.45	1.76×10^{10}
our data shuffle improvement of butterfly	3.37×10^{11}			1.51×10^{10}
our total improvement	6.91×10^{11}			5.14×10^{10}
total improvement without global memory access*4	8.67×10^{11}			6.45×10^{10}

*4 Note that to be impartial we extract the first sub-radix of our NTT/INTTs which get rid of the data shuffle to approximate the tiny data size situation in [30].

Table 10
NTT/INTT Improvement Ratio.

Improvement Ratio Compared with Our Works	Similar Works	GPU		ASIC
		bellperson[31]	Kyber[12]	bellman[6]
	Relative Efficiency	1.48×10^{11}	1.62×10^{11}	1.25×10^{11}
our multiplication reduction improvement of Montgomery	2.37×10^{11}	60%	46%	90%
our data shuffle improvement of NTT	3.37×10^{11}	128%	108%	170%
our total improvement	6.91×10^{11}	367%	327%	453%

Table 11
NTT/INTT Speedup Ratio Comparison.

Speedup Ratio Compared with Our Works	Similar Works	GPU		ASIC
		bellperson[31]	Kyber[12]	bellman[6]
	Speedup Ratio	25.7×[31]	130.4×[12]*2	29.3×~105.6×[6]
our multiplication reduction improvement	41.1×	√	×	○
our data shuffle improvement of butterfly	58.5×	√	×	○
our total improvement	119.9×	√	×	√
our total improvement without global memory access *4	150.5×	√	√	√

tios in the same hardware environment, which exceeds the 2.65× in [30].

7.3. Total evaluation in production environment

We test the bellperson by merging our improved reconstructions, which is integrated in filecoin application which produces within the hardware environment of 2080 Ti × 3, and get an acceleration of 3.14× speedup ratio compared with the original GPU implementation.

8. Conclusion

To handle the performance bottleneck of zk-SNARKs, we accelerated the most efficient implementation: Groth16 by proposing a GPU based modular reduction for Montgomery kernel and a GPU based polynomial multiplication for NTT/INTT kernel. With these two proposed techniques, we took advantage of the numerous logic units in GPU by reducing the number of assembly instructions and feeding data more efficiently. Our extensive experiments show that our improvements produce significant speedup or better efficiency for these kernels as well as Groth16 over existing GPU and CPU based implementations.

As our improved modular multiplication and polynomial multiplication are designed to replace dot product in many cryptography algorithms, our improvements would be applied in other contemporary cryptography algorithms such as the Kyber, a Post-Quantum Cryptography(PQC) algorithm.

9. Appendix

zk-SNARKs are the most efficient proof systems in terms of proof size and verification [5]. Recently there has been a lot of progress both in theory and practice on constructing highly efficient non-interactive arguments with small size and low verification complexity [23].

Many constructions of zk-SNARKs rely on pairing-based cryptography. In these constructions a proof consists of a number of group elements and the verification consists of checking a number of pairing product equations [23]. The Groth16 is the most efficient zk-SNARK implementation working on BLS12-381 ECC pairing-based field [5].

The BLS12-381 pairing-friendly elliptic curve, based on a 381b prime field, has been recently proposed and is part of ongoing standardization led by the Internet Engineering Task Force (IETF) [40].

The constant size of zk-SNARK argument in Groth16 is based on constructing a set of polynomial equations and using pairings to efficiently verify these equations [23].

Within Groth16, pairing-based field polynomial multiplications are the basic operation for QAPs, and modular exponentiations are the basic operation for pairing-based ECC pairing. In addition, NIZK arguments for QAPs contain polynomial operations on two pairing-based fields. But there is the third field as the bilinear map of the above two fields, which seriously consumes computing power.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported in part by National Natural Science Foundation of China (grant No. U2032125), National Key Research and Development Project (grant No. 2020SKA0120202), Independent Deployment Project of Shanghai Advanced Research Institute (grant No. E052891ZZ1, E0560W1ZZ0).

References

[1] 3for, Empirical performance, https://github.com/3for/libsnark/tree/master/libsnark/zk_proof_systems/ppzksnark. (Accessed 4 April 2019).
[2] Patryk27, ocl, https://github.com/cogciprocate/ocl. (Accessed 3 July 2019).
[3] K. Adámek, S. Dimoudi, M. Giles, W. Armour, GPU fast convolution via the overlap-and-save method in shared memory, ACM Trans. Archit. Code Optim. 17 (3) (2020) 1–20.
[4] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, T. Lepoint, NFLlib: NTT-based fast lattice library, in: Cryptographers' Track at the RSA Conference, Springer, 2016, pp. 341–356.
[5] K. Baghery, Z. Pindado, C. Ràfols, Simulation extractable versions of Groth's zk-SNARK revisited, in: International Conference on Cryptology and Network Security, Springer, 2020, pp. 453–461.
[6] P. Barrett, Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, in: Conference on the Theory and Application of Cryptographic Techniques, Springer, 1986, pp. 311–323.
[7] S. Bowe, A. Gabizon, M.D. Green, A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK, in: International Conference on Financial Cryptography and Data Security, Springer, 2018, pp. 64–77.
[8] F. Candel, S. Petit, J. Sahuquillo, J. Duato, Accurately modeling the on-chip and off-chip GPU memory subsystem, Future Gener. Comput. Syst. 82 (2018) 510–519, https://doi.org/10.1016/j.future.2017.02.012.
[9] B. Catanzaro, A. Keller, M. Garland, A decomposition for in-place matrix transposition, ACM SIGPLAN Not. 49 (8) (2014) 193–206, https://doi.org/10.1145/2692916.2555253.
[10] X. Chen, Y. Lei, Z. Lu, S. Chen, A variable-size FFT hardware accelerator based on matrix transposition, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 26 (10) (2018) 1953–1966, https://doi.org/10.1109/TVLSI.2018.2846688.
[11] S. Cook, et al., A developer's guide to parallel computing with GPUs, 2013.

[12] Cryptonemo, Bellperson, https://github.com/filecoin-project/bellperson. (Accessed 13 October 2021).
[13] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, B. Parno Cinderella, Cinderella: turning shabby X. 509 certificates into elegant anonymous credentials with the magic of verifiable computation, in: 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 235–254.
[14] D. Demirel, L. Schabhüser, J. Buchmann, Proof and argument based verifiable computing, in: Privately and Publicly Verifiable Computing Techniques, Springer, 2017, pp. 13–22.
[15] Ebfull, Bellman, https://github.com/zkcrypto/bellman. (Accessed 10 September 2021).
[16] N. Emmart, J. Luitjens, C. Weems, C. Woolley, Optimizing modular multiplication for nvidia's Maxwell gpus, in: 2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH), IEEE, 2016, pp. 47–54.
[17] J. Fernández-Fabeiro, A. Gonzalez-Escribano, D.R. Llanos, Distributed programming of a hyperspectral image registration algorithm for heterogeneous GPU clusters, J. Parallel Distrib. Comput. 151 (2021) 86–93, https://doi.org/10.1016/j.jpdc.2021.02.014.
[18] H.S. Galal, A.M. Youssef, Verifiable sealed-bid auction on the Ethereum blockchain, in: International Conference on Financial Cryptography and Data Security, Springer, 2018, pp. 265–278.
[19] S.D. Galbraith, K.G. Paterson, N.P. Smart, Pairings for cryptographers, Discrete Appl. Math. 156 (16) (2008) 3113–3121, https://doi.org/10.1016/j.dam.2007.12.010.
[20] A. Gholami, J. Hill, D. Malhotra, G. Biros, AccFFT: a library for distributed-memory FFT on CPU and GPU architectures, preprint, arXiv:1506.07933.
[21] S. Goldwasser, S. Micali, C. Rackoff, The knowledge complexity of interactive proof systems, SIAM J. Comput. 18 (1) (1989) 186–208, https://doi.org/10.1137/0218012.
[22] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, High performance discrete Fourier transforms on graphics processors, in: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Ieee, 2008, pp. 1–12.
[23] J. Groth, On the size of pairing-based non-interactive arguments, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2016, pp. 305–326.
[24] W.-m. Hwu, What is ahead for parallel computing, J. Parallel Distrib. Comput. 74 (7) (2014) 2574–2581, https://doi.org/10.1016/j.jpdc.2014.02.005.
[25] Khronos, Opencl, https://www.khronos.org/opencl/. (Accessed 19 November 2021).
[26] S. Kim, W. Jung, J. Park, J.H. Ahn, Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus, in: 2020 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2020, pp. 264–275.
[27] M. Knezevic, F. Vercauteren, I. Verbauwhede, Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods, IEEE Trans. Comput. 59 (12) (2010) 1715–1721.
[28] C.K. Koc, T. Acar, B.S. Kaliski, Analyzing and comparing Montgomery multiplication algorithms, IEEE MICRO 16 (3) (1996) 26–33, https://doi.org/10.1109/40.502403.
[29] W.-K. Lee, S. Akleylek, W.-S. Yap, B.-M. Goi, Accelerating number theoretic transform in GPU platform for qTESLA scheme, in: International Conference on Information Security Practice and Experience, Springer, 2019, pp. 41–55.
[30] W.K. Lee, S.O. Hwang, High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of things applications, IEEE Transactions on Services Computing, https://doi.org/10.1109/TSC.2021.3103956.
[31] H. Lipmaa, Prover-efficient commit-and-prove zero-knowledge SNARKs, in: International Conference on Cryptology in Africa, Springer, 2016, pp. 185–206.
[32] P.L. Montgomery, Modular multiplication without trial division, Math. Comput. 44 (170) (1985) 519–521, https://doi.org/10.1090/S0025-5718-1985-0777282-X.
[33] N. Corporation, Cuda C++ programming guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (Accessed 20 October 2021).
[34] N. Corporation, Cufft, https://docs.nvidia.com/cuda/cufft/. (Accessed 2 August 2021).
[35] N. Corporation, Cublas, https://docs.nvidia.com/cuda/cublas/. (Accessed 2 August 2021).
[36] B. Peng, Y. Zhu, N. Jing, X. Zheng, Y. Zhou, Design of a hardware accelerator for zero-knowledge proof in blockchains, in: International Conference on Smart Computing and Communication, Springer, 2020, pp. 136–145.
[37] J.M. Pollard, The fast Fourier transform in a finite field, Math. Comput. 25 (114) (1971) 365–374.
[38] A. Rahimi, M.A. Maddah-Ali, Multi-party proof generation in QAP-based zk-SNARKs, preprint, arXiv:2103.01344.
[39] A. Razaque, W. Jinrui, W. Zancheng, Q.B. Hani, M.A. Khaskheli, W.A. Bhutto, Integration of CPU and GPU to accelerate RSA modular exponentiation operation, in: 2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT), IEEE, 2018, pp. 1–6.
[40] Y. Sakemi, T. Kobayashi, T. Saito, R.S. Wahby, Pairing-friendly curves, Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-05.
[41] K. Shahbazi, S.-B. Ko, High throughput and area-efficient FPGA implementation of AES for high-traffic applications, IET Comput. Digit. Tech. 14 (6) (2020) 344–352.
[42] V. Soni, A. Hadjadj, O. Roussel, G. Moebs, Parallel multi-core and multi-processor methods on point-value multiresolution algorithms for hyperbolic conservation laws, J. Parallel Distrib. Comput. 123 (2019) 192–203, https://doi.org/10.13154/tches.v2019.i3.180-201.
[43] Str4d, Zero-knowledge cryptography in rust, https://github.com/zkcrypto. (Accessed 2 August 2021).
[44] S. Voigt, M. Baesler, T. Teufel, Dynamically reconfigurable dataflow architecture for high-performance digital signal processing, J. Syst. Archit. 56 (11) (2010) 561–576, https://doi.org/10.1016/j.sysarc.2010.07.010.
[45] J. You, Q. Zhang, C. D'Alves, B. O'Farrell, C.K. Anand, Using z14 fused-multiply-add instructions to accelerate elliptic curve cryptography, Cryptology ePrint Archive, https://doi.org/10.5555/3370272.3370302.
[46] J. Zhang, Z. Fang, Y. Zhang, D. Song, Zero knowledge proofs for decision tree predictions and accuracy, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 2039–2053.
[47] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, L. Liu, Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT, IACR Trans. Cryptogr. Hardw. Embed. Syst. (2020) 49–72, https://doi.org/10.13154/tches.v2020.i2.49-72.
[48] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, G. Sun, PipeZK: accelerating zero-knowledge proof with a pipelined architecture, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2021, pp. 416–428.
[49] L. Zhao, J. Zhang, J. Huang, Z. Liu, G. Hancke, Efficient implementation of kyber on mobile devices, in: 2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2021, pp. 506–513.
[50] Y. Zhu, Z. Liu, Y. Pan, When NTT meets Karatsuba: preprocess-then-NTT technique revisited, in: International Conference on Information and Communications Security, Springer, 2021, pp. 249–264.

Ning Ni received his Master degree in Integrated Circuit Design from Shanghai Jiao Tong University, China in 2018. He is currently Chief Data Science and Technology Officer with the 7Road Holdings Limited. His research interests include homomorphic encryption, heterogeneous computing, signal processing. He has rich experience in blockchain cryptography architecture design and optimization.

Yongxin Zhu received his Ph.D. in Computer Science from National University of Singapore in 2001. He is currently a full Professor with Shanghai Advanced Research Institute, Chinese Academy of Sciences (CAS). He is also an Adjunct Professor with the School of Microelectronics at the Shanghai Jiao Tong University (SJTU). He has published over 170 English journal and conference papers, 50 Chinese journal papers and 20 China patent approvals in the areas of computer architecture, embedded systems and big data processing. Prior to his tenure with CAS and SJTU, he worked as a research fellow with National University of Singapore in 2002-2005, a senior consultant with S1 Incorporation in 1999-2002 and a teaching assistant with the Department of Computer Science and Engineering, SJTU in 1994-1995. He was also a Visiting Professor with National University of Singapore in 2013-2017. He has served as a guest editor of Journal of Systems Architecture, program chairs of more than 10 conferences, a distinguished member of China Computer Federation (CCF), a senior member of IEEE and a professional member of ACM.