计算机科学与探索
Journal of Frontiers of Computer Science and Technology
1673-9418/2024/18(02)-0538-15
doi: 10.3778/j.issn.1673-9418.2211075

zk-SNARK中数论变换的硬件加速方法研究

赵海旭[1],柴志雷[1,3+],花鹏程[1],王　锋[1],丁　冬[2]
1. 江南大学 人工智能与计算机学院,江苏 无锡 214122
2. 江南大学 物联网工程学院,江苏 无锡 214122
3. 江苏省模式识别与计算智能工程实验室,江苏 无锡 214122
+ 通信作者 E-mail: zlchai@jiangnan.edu.cn

摘　要:简洁非交互式零知识证明能够生成长度固定的证明并快速进行验证,极大地推动了零知识证明在数字签名、区块链及分布式存储等领域的应用。但其证明的生成过程极其耗时且需要被频繁调用,其中数论变换是证明生成过程的主要运算之一。然而现有的通用数论变换硬件加速方法难以满足其在简洁非交互式零知识证明中大规模、高位宽的要求。针对该问题,提出一种数论变换多级流水硬件计算架构。针对高位宽计算需求对高位模运算进行优化,设计了低时延蒙哥马利模乘单元;为了加速大规模计算,通过二维子任务划分将大规模数论变换任务划分为小规模独立子任务,并通过消除数据依赖实现了子任务间计算流水;在子任务多轮蝶形运算之间采用数据重排机制,有效缓解了访存需求并实现了不同步长蝶形运算间的计算流水。所提出的数论变换计算架构可以根据现场可编程门阵列(FPGA)片上资源灵活扩展,方便部署在不同规模的FPGA上以获得最大加速效果。所提出的硬件架构使用高层次综合(HLS)开发并基于OpenCL框架在AMD Xilinx Alveo U50实现了整套异构加速系统。实验结果表明,相比于PipeZK中的数论变换加速模块,该方法获得了1.95倍的加速比;在运行当前主流的简洁非交互式零知识证明开源项目bellman时,相比于AMD Ryzen 9 5900X单核及12核分别获得了27.98倍和1.74倍的加速比,并分别获得了6.9倍、6倍的能效提升。

关键词:现场可编程门阵列(FPGA);简洁非交互式零知识证明(zk-SNARK);模乘;数论变换;硬件加速
文献标志码:A　**中图分类号:**TP309.7;TP338

Hardware Acceleration of Number Theoretic Transform in zk-SNARK

ZHAO Haixu[1], CHAI Zhilei[1,3+], HUA Pengcheng[1], WANG Feng[1], DING Dong[2]
1. School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi, Jiangsu 214122, China
2. School of Internet of Things Engineering, Jiangnan University, Wuxi, Jiangsu 214122, China
3. Jiangsu Provincial Engineering Laboratory of Pattern Recognition and Computational Intelligence, Wuxi, Jiangsu 214122, China

Abstract: The proof in zk-SNARK has a fixed length and can be verified quickly, promoting the application of zero-knowledge proof in areas such as digital signature, blockchain, distributed storage, and outsourced computing. However, the generation of proofs is time-consuming and frequently used. As a result, NTT (number theoretic transform), one of the most time-consuming parts in proof-generation, needs to be accelerated significantly. However, the existing general NTT hardware acceleration methods cannot meet the requirements of large-bitwidth and large-scale

基金项目:国家自然科学基金(61972180);江苏省模式识别与计算智能工程实验室项目。
This work was supported by the National Natural Science Foundation of China (61972180), and the Project of Jiangsu Provincial Engineering Laboratory of Pattern Recognition and Computational Intelligence.
收稿日期:2022-11-21　**修回日期:**2023-05-09

计算机科学与探索 www.ceaj.org

in zk-SNARK. To address this issue, this paper proposes a highly pipelined architecture for NTT. First of all, large-bitwidth modular arithmetic is optimized and low-latency Montgomery modular multiplication hardware unit is designed. And then, the large-scale NTT tasks are divided into smaller sub-tasks through two-dimensional partitioning, which improves the parallelism of NTT computation and eliminates the data dependence among sub-tasks, thus realizing the pipeline among sub-tasks. Finally, the "data reordering" technique is introduced among multiple rounds of butterfly operations in a sub-task, which effectively alleviates the memory access requirements, thus realizing the bottom-level pipeline in each sub-task, among butterfly operations with different step sizes. This architecture can be flexibly scaled to different scales of FPGAs. The accelerator is prototyped on the AMD-Xilinx Alveo U50 card (UltraScale+XCU50 FPGA). To balance computing efficiency and flexibility, the OpenCL equipped with high-level synthesis (HLS) is used to implement the system. The evaluation results show that the NTT module performs 1.95 times faster than the one in PipeZK and the accelerator achieves 27.98 and 1.74 times speedup, 6.9 and 6 times energy efficiency improvement than AMD Ryzen 9 5900X respectively, when it is integrated into the well-known ZKP open-source project, bellman.

Key words: field programmable gate array (FPGA); zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK); modular multiplication; number theoretic transform; hardware acceleration

零知识证明(zero-knowledge proof, ZKP)[1]是一类密码学协议,它能够使一方向另一方证明其陈述正确性,而不泄露与该陈述相关的任何信息。简洁非交互式零知识证明(zero-knowledge succinct non-interactive arguments of knowledge, zk-SNARK)[2-5]的提出极大推动了这项技术在数字签名[6]、区块链[7-9]以及分布式存储[10]等领域的应用。zk-SNARK能够针对复杂问题生成简短、固定长度的证明并快速验证。正因为其简洁、完备、可靠的优点,zk-SNARK已经成为了众多开源项目和研究者关注的热点。

zk-SNARK虽然可以产生简短证明并且快速进行验证,却也导致证明生成过程十分复杂耗时。除此之外,证明生成过程调用频繁,也对zk-SNARK部署在真实场景下的性能提出了较高要求。在数字交易过程中,对单笔支付交易生成证明往往需要几分钟时间[9];在分布式存储[10]、外包计算[11]等应用中,生成证明的时间更长,有时甚至需要20~30分钟才能够完成[12]。而在如此耗时的证明生成过程中,仅进行多项式计算就需要花费几分钟时间[13]。尽管说后来许多学者在算法和软件实现层面上提出了改进协议[14-19],但这些协议始终都无法避免大规模多项式乘法运算。

使用数论变换(number theoretic transformation, NTT)和逆数论变换(inverse number theoretic transformation, iNTT)降低多项式乘法的计算复杂度已经成为了业界共识。在全同态加密[20-23]、后量子密码学[24-26]等应用中都涉及到了大量多项式乘法计算,研究者们都倾向于借助NTT和iNTT对多项式进行系数表示形式和点值表示形式之间的转换,以降低多项式乘法的时间复杂度。正因为NTT算法的广泛应用,目前已经涌现出了许多领域专用的NTT硬件或软硬件协同加速方案[27-30]。Mert等人[27]针对基于格的全同态加密提出了一种NTT软硬件协同加速方案,通过平衡NTT计算单元和数据通信时间,在现场可编程门阵列(field programmable gate array, FPGA)上针对多项式乘法运算实现了800 KB/s吞吐率,然而其支持的最大任务规模仅为1 024。HEAX[28]同样作为一种用于全同态加密的新型硬件架构,专门实现了用于NTT的高度可并行化架构,在可重构硬件上获得了比全同态加密纯软件实现近200倍的加速,但其所能支持的操作数位宽仅为54位。这些用于同态加密的NTT架构不足以支持zk-SNARK证明生成过程中具有百万系数、位宽高达数百位的NTT/iNTT计算任务。由于当前已有的NTT加速工作面向的实际应用场景对输入规模和操作数位宽要求不高,通常仅能支持具有几千个32~64位操作数的NTT计算任务,且不具备扩展性。如果采用这些计算架构进行规模扩展,将会带来资源占用率非线性增长,无法在真实应用场景下部署[13]。

近年来PipeZK[13]提出了专门用于zk-SNARK中大规模、高位宽多项式乘法运算的POLY子模块,加速了具有几百位操作数、百万输入规模的NTT任务。这类使用Verilog或超高速集成电路硬件描述语言(very-high-speed integrated circuit hardware de-

计算机学报 www.cai.pre

scription language，VHDL）实现、基于ASIC的零知识证明芯片虽然计算时延低，然而面向不同零知识证明协议适配性差且更新迭代困难。相比之下，采用GPU加速NTT计算任务[31]大大缩短了开发周期，但功耗高、计算延迟不稳定的缺陷严重降低了所部署zk-SNARK的服务质量，使这类加速方案难以落地实现。针对当前各种加速方案所存在的问题，本文所要解决的便是如何在保持良好扩展性和算法适配性的前提下尽可能提升zk-SNARK生成证明的速度和能效比，因此选择使用AMD-Xilinx数据中心加速卡与CPU对证明生成过程进行异构加速。

在FPGA异构计算平台上对zk-SNARK中NTT计算任务进行加速，需要解决的难题是如何通过硬件定制、软硬件协同，使得证明生成过程中NTT计算能够适配于硬件平台以实现低延迟、低功耗，进而获得最佳性能和能效。本文使用高层次综合（high-level synthesis，HLS）基于OpenCL异构计算框架设计了一种用于NTT计算的多级流水计算架构，对zk-SNARK进行了模块化加速。本文的主要工作如下：

（1）根据蝶形运算的访存特点提出了一种“数据重排”策略，并对NTT任务进行独立子任务划分，实现了不同步长蝶形运算间以及NTT子任务间的计算流水，支持最高384位数据位宽、百万输入规模的NTT任务。

（2）使用HLS实现了NTT计算核心，相比于Verilog开发有着更好的兼容性，且可以快速更新。NTT计算核心内的流水线可以根据FPGA片上资源规模灵活缩放，具有良好的扩展性。与PipeZK相比，在相同的FPGA平台上计算核心获得了1.95倍的加速比。

（3）基于bellman构建了完整的zk-SNARK系统，将bellman中NTT计算部分加载到FPGA上的NTT计算核心进行硬件加速。相比于bellman中NTT计算任务在CPU单核和12核上的纯软件实现，系统分别获得了27.98倍和1.74倍的加速比、32 MB/s的吞吐以及6.9倍和6倍的能效提升。

1 零知识证明与数论变换

1.1 零知识证明

零知识证明技术能够在证明者与验证者之间建立一种协议机制，使得证明者向验证者提供一条与其所要证明的陈述相关联的证明，在不泄露任何隐私信息的情况下，让验证者相信陈述的正确性。zk-SNARK由于生成证明数据量小、验证成本低的优势，逐渐取代了证明者与验证者交互式地生成证明、验证的机制。zk-SNARK使得证明者根据多项式只需一次性生成一条唯一对应的证明，同时提供验证参数，任何验证者即可随时自行验证该证明的正确性。

zk-SNARK在对一条陈述生成证明之前，需要通过与算法流程相互独立的高层应用程序将该陈述建模为形如式（1）的多项式方程，进而再根据该多项式方程 $f(x_1,x_2,\cdots,x_n)=0$ 生成一条零知识证明。其中 a_{ij} 为该多项式的系数，d 为该多项式的阶数。

$$f(x_1,x_2,\cdots,x_n)=\sum_{i=1}^{n}\sum_{j=0}^{d-1}a_{ij}\cdot x_i^j=0 \tag{1}$$

由此可以将证明问题描述如下：对于一个公共多项式方程 $f(x_1,x_2,\cdots,x_n)=0$，证明者与验证者共同知晓该多项式的系数 a_{ij}，证明者需要向不知晓该多项式方程一组解 x 的验证者提供一条零知识证明。对于证明者来说，这条证明可以让验证者相信证明者确实持有该多项式方程的一组解 x；然而对于验证者来说，却又无法获得与解集 x 相关的任何信息。Groth16[32]作为zk-SNARK最流行的算法之一，将协议划分为设置参数（setup）、生成证明（prove）和验证（verify）三个阶段，如图1所示。

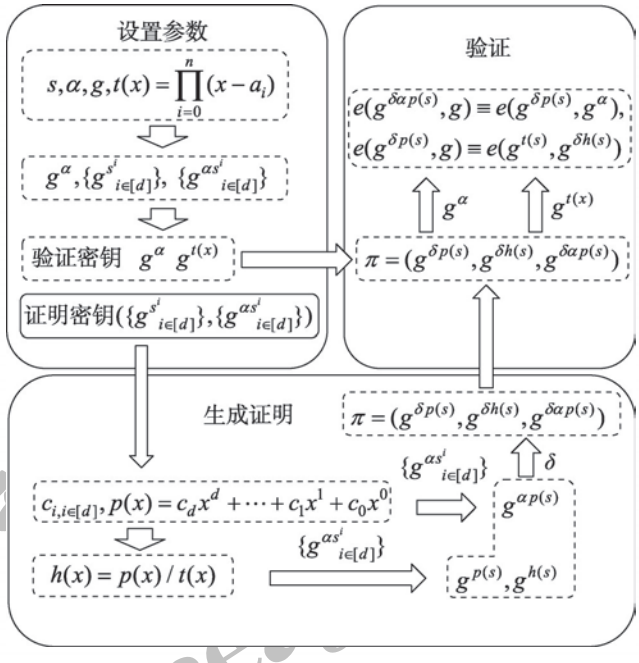


图1 Groth16计算流程

Fig.1 Groth16 algorithm flow

设置参数阶段和验证阶段的计算量并不大，并且不会随着问题规模扩大而膨胀。相比之下，生成证明阶段的计算量则要大得多。生成证明阶段最核

心的计算是根据已有多项式 $p(x)$ 和 $t(x)$ 计算 $h(x)$。考虑到计算安全与隐私需求，需对 $p(s)$、$\alpha p(s)$ 和 $t(s)$ 进行同态加密。由于一个多项式同态加密的结果与多项式各项的加密结果在加密空间中的运算结果完全一致，根据证明密钥可直接生成 $h(s)$ 的同态加密结果。进一步地，为了防止验证者通过 $p(s)$ 的同态加密结果反向推导 $p(x)$ 的相关信息，还需要选取一个随机偏移量 δ 来乘以 $p(s)$、$h(s)$ 以及 $\alpha p(s)$ 以实现零知识化。$\delta p(s)$、$\delta h(s)$ 以及 $\delta\alpha p(s)$ 同态加密的结果构成了最终证明 π。

zk-SNARK的证明生成阶段非常耗时是因为多项式乘法计算量巨大，参与计算的系数数量高达数百万且位宽通常高达几百位。而NTT、iNTT计算则是完成多项式乘法计算任务的主要计算方法。因此，加速zk-SNARK的巨大挑战之一就是在低功耗前提下减少NTT、iNTT计算耗时。以当下应用最为广泛的一个使用Rust开发的zk-SNARK库bellman[33]为例，随着多项式方程规模不断扩大，生成证明的时间越来越长，尤其是在实际应用中，输入规模达到百万级别时，多项式乘法运算中的NTT、iNTT计算总计耗费了几分钟时间，如图2所示。因此有必要对zk-SNARK中的多项式乘法设计高效的NTT计算架构。

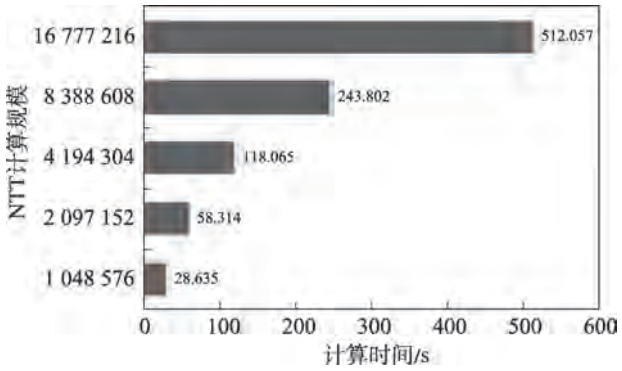


图2 不同规模NTT计算时间

Fig.2 Calculation time of NTT/iNTT at different scales

1.2 数论变换

对于系数表示形式下的多项式乘法运算，可以将多项式转换为点值表示形式进行运算，从而将计算时间复杂度从 $O(n^2)$ 降低至 $O(n\lg n)$，如图3所示。zk-SNARK中多项式乘法以及椭圆曲线加密运算都是在大整数有限域上进行的，因此采用基于素数根的数论变换以及逆数论变换进行求值和插值是最佳方案。

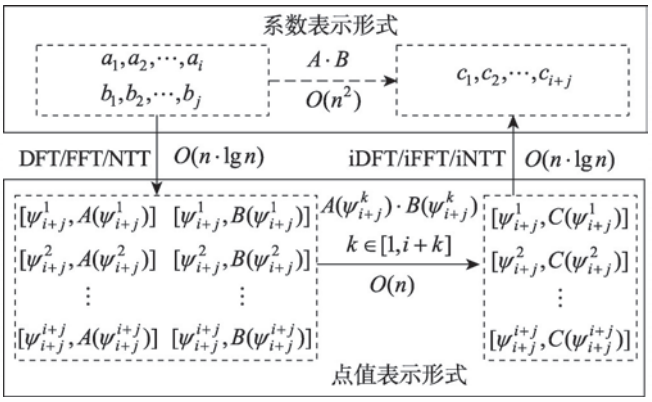


图3 多项式乘法计算流程与时间复杂度

Fig.3 Algorithm flow and time complexity of polynomial multiplication

NTT计算的输入 $\boldsymbol{a}$ 是一个由多项式系数组成的一维向量，输出 $\hat{\boldsymbol{a}}$ 同样是一维向量且长度与 $\boldsymbol{a}$ 一致。由此对NTT计算定义如下：

$$\hat{a}_j(\psi_N^j)=\sum_{i=0}^{N-1}a_i\cdot\psi_N^{ij} \tag{2}$$

其中，$\boldsymbol{a}$ 和 $\hat{\boldsymbol{a}}$ 均为长度为 N 的向量，ψ_N 为素数 p 所定义的有限域上的单位根。$\hat{a}_j$、a_i 和旋转因子 ψ_N^{ij} 均定义在位宽为 λ 的整数有限域上。相比于NTT，iNTT唯一的区别在于其运算所用到的单位根是 ψ_N 关于素数 p 的模逆。在实际应用中，可以通过补0填充将一维向量 $\boldsymbol{a}$ 的长度扩展为2的整数幂，进而利用分治策略，用 $\hat{a}_j$ 中奇数、偶数下标的系数分别定义两个规模为 $N/2$ 的快速数论变换：

$$\hat{a}_j^+(\psi_N^j)=a_0+a_2\cdot\psi_N^j+\cdots+a_{N-2}\cdot\psi_N^{\left(\frac{N}{2}-1\right)j} \tag{3}$$

$$\hat{a}_j^-(\psi_N^j)=a_1+a_3\cdot\psi_N^j+\cdots+a_{N-1}\cdot\psi_N^{\left(\frac{N}{2}-1\right)j} \tag{4}$$

$$\hat{a}_j(\psi_N^j)=\hat{a}^+_j(\psi_N^{2j})+\psi_N^j\cdot\hat{a}^-_j(\psi_N^{2j}) \tag{5}$$

由此便可以仅通过 $N/2$ 个旋转因子对 $\hat{a}_j^-$ 和 $\hat{a}_j^+$ 进行求值。这些子问题与原始问题 $\hat{a}_j$ 形式相同，但计算规模仅为原来一半。根据公式可知，每次递归调用的时间复杂度为 $O(n)$，因此通过分治递归计算NTT的时间复杂度为：

$$T(n)=2T(n/2)+O(n)=O(n\lg n) \tag{6}$$

通过计算数据的奇偶下标将NTT任务不断折半划分的确降低了算法时间复杂度，但是递归调用所导致对原始计算数据的离散访存也为算法硬件加速带来了新的挑战。采用递归分治策略对NTT任务不断进行折半划分后，最终将会产生一系列计算规模为2、需要跨步访问原始数据并计算更新的操作，这一操

计算机科学与探索 www.ceaj.org

作被称为蝶形运算。由于具有相同步长的蝶形运算间可以完全并行，一个输入规模为 n 的NTT计算任务只需依次进行 $\mathrm{lb}\,n$ 轮步长为 $2^k(k=0,1,\cdots,\mathrm{lb}\,n-1)$ 的并发蝶形运算，即可完成一次完整的NTT计算任务。以输入规模为8的NTT计算任务为例，步长为4、2、1的四轮蝶形运算间各自都可以并发执行，如图4所示。若使用GPU处理NTT计算任务，设单次蝶形运算的计算时间为 t，只要能够创建足够多的线程，那么便可以在 $\mathrm{lb}\,n\cdot t$ 的总时间内完成整个NTT计算任务。由于zk-SNARK中NTT计算任务的输入规模和数据位宽太大，难以实现NTT计算的高度并行化。以输入规模为 2^{20} 的NTT计算任务为例，想要完全实现计算并行化，需要创建 2^{19} 个线程，这对于GPU线程创建和线程调度来说极为困难，使用FPGA同样也会因片上资源受限而无法实现。

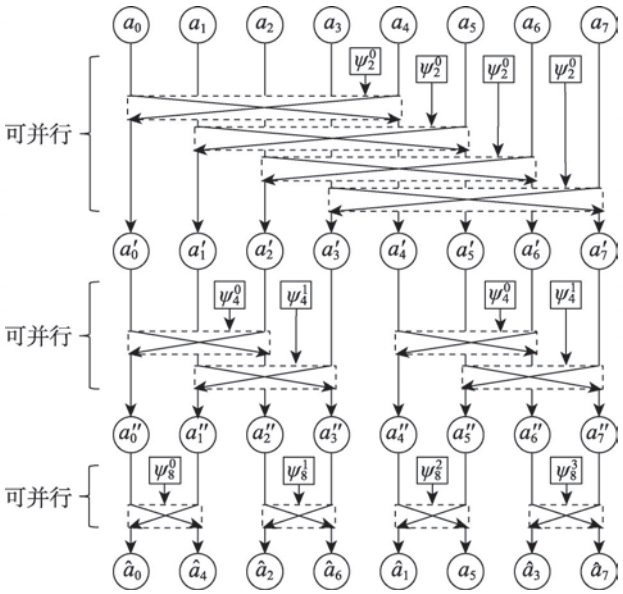


图4 NTT蝶形运算过程

Fig.4 Betterfly operation of NTT

当下用于NTT的通用硬件加速模块通常无法同时满足大规模输入、高位宽的需求[22,24,27-28]，如表1所示，难以用于zk-SNARK中以处理百万输入、位宽高达数百位的NTT计算任务。这些工作通常仅能处理输入规模较大的NTT任务而支持的操作数位宽较小，或是仅能处理操作数位宽较大的NTT任务而无法支持大规模输入。最重要的是，开发方式与硬件平台的差异性阻碍了其方案的更新迭代以及在zk-SNARK应用部署中的复用。如果强行扩展这些NTT设计方案，将会导致硬件资源占用率呈现非线性膨胀式增长，无法上板验证评估。因此，权衡计算效率与资源利用率是在zk-SNARK中对NTT进行硬件加速的关键。

表1 NTT相关硬件加速模块的处理数据规模

Table 1 NTT accelerators of different sizes in related work

相关工作	输入规模	位宽
Chen等人[24]	1 024	31
Öztürk 等人[22]	32 768	32
Mert 等人[27]	1 024	32
HEAX[28]	16 384	54
PipeZK[13]	1 048 576	768

2 多级流水NTT计算架构

2.1 高位模运算优化

zk-SNARK中的算术运算都是大整数有限域上的模运算，NTT也不例外。传统的模运算由常规算术操作和取模操作组成。取模操作通过除法运算得到小于模数的非负余数。然而，除法器的硬件实现代价高，尤其是面向zk-SNARK中的高位宽操作数，直接使用除法逻辑实现模运算器是低效的[34]。因此有必要对模加、模减、模乘运算进行高效的硬件模块化设计。

两个位宽为 n 的无符号操作数 a、b 进行加法运算的结果 c 的位宽不超过 $n+1$ 位，进行减法运算的结果 d 的位宽不超过 n 位。换句话说，c、d 的取值范围为 $[0,2^{n+1})$。由于模数 p 是 n 位有效的，即模数的取值范围为 $[2^n,2^{n+1})$，c、d 的取值必然小于模数的两倍，即 c、d 关于模数 p 的取模结果或是 c、d 本身，或是 $c-p$、$d-p$。因此对于模加、模减运算，本文通过比较器、加法器、减法器实现了取模操作，相比于使用除法逻辑既节省硬件资源也降低了计算时延。

然而对于模乘运算 $a\cdot b=c(\mathrm{mod}\,p)$，由于两个位宽为 n 的操作数 a、b 的乘积最高会有 $2n$ 位有效位，难以简单地仅使用比较器、加法器、减法器来实现 $2n$ 位数对 n 位数的取模操作。为了解决这个问题，本文采用硬件友好的蒙哥马利模乘运算[35]替代传统的模乘运算。单次蒙哥马利模乘的计算过程如算法1所示。

算法1 蒙哥马利模乘

输入：$P,A,B,R=2^n,P'=-P^{-1}\,\mathrm{mod}\,R$

输出：$C=A\cdot B\cdot R^{-1}\,\mathrm{mod}\,P$

1. begin
2. $T\leftarrow A\cdot B$

计算机科学与探索 www.ceaj.org

3. $M \leftarrow T \cdot P' \bmod R$
4. $C \leftarrow (M \cdot P + T)/R$
5. if $C \geqslant P$ then
6. $C \leftarrow C - P$
7. end if
8. end

单次蒙哥马利模乘旨在借助位移操作快速计算 $A \cdot B \cdot R^{-1} \bmod P$ 的结果。其原理为 R 的取值为2的整数幂，针对 R 的除运算及取模运算均可通过位移操作替代。不妨将单次蒙哥马利模乘运算记作 $Mon(A,B,P)$。于是，传统模乘运算 $a \cdot b = c(\bmod p)$ 可以通过顺序执行四次蒙哥马利模乘等价完成：

$$a' = Mon(a, R^2, p) \tag{7}$$

$$b' = Mon(b, R^2, p) \tag{8}$$

$$c' = Mon(a', b', p) \tag{9}$$

$$c = Mon(c', 1, p) \tag{10}$$

操作(7)、(8)将原始模乘操作数转换为蒙哥马利表示形式，操作(9)在蒙哥马利表示形式下完成了一次真正意义上的模乘运算，而操作(10)则将模乘计算结果由蒙哥马利表示形式转换回原始形式。其中(7)、(8)、(10)又被称为蒙哥马利约简。由于蒙哥马利表示形式下的操作数与原始形式遵循相同的运算法则，面对连续模乘运算，中间计算结果可在蒙哥马利表示形式下参与后续计算，而不必转换回原始形式。因此，尽管一次传统模乘运算需要四次蒙哥马利模乘才能完成，但是对于操作数稀疏、模乘运算频繁的计算任务而言，实际执行的蒙哥马利模乘的次数与传统模乘次数相当。

通过蒙哥马利约简操作，模乘操作数被转换为对特定模数 R 方便取模的蒙哥马利表示形式，进而通过三次乘法运算和多次位移操作实现了一次实质的模乘运算。与传统模乘的计算方法相比，蒙哥马利模乘虽然增加了常规乘法运算的次数，然而一次 n 位除法的计算时延代价至少是一次 n 位乘法的 n 倍，而由于用于取模的除法运算被蒙哥马利约简和位移运算所取代，计算时延整体降低，且硬件资源消耗明显减少。因此，采用蒙哥马利模乘进行高位模运算有着显著的算法优势，同样也面临着如何将其实现得更快、更节省资源的挑战。

本文实现了两种蒙哥马利模乘单元，分别适配于BN128(256位)和BLS381(384位)曲线，其中384位蒙哥马利模乘单元如图5所示。相比之下，256位模乘单元的区别仅在于高位乘法器中子乘法器的数

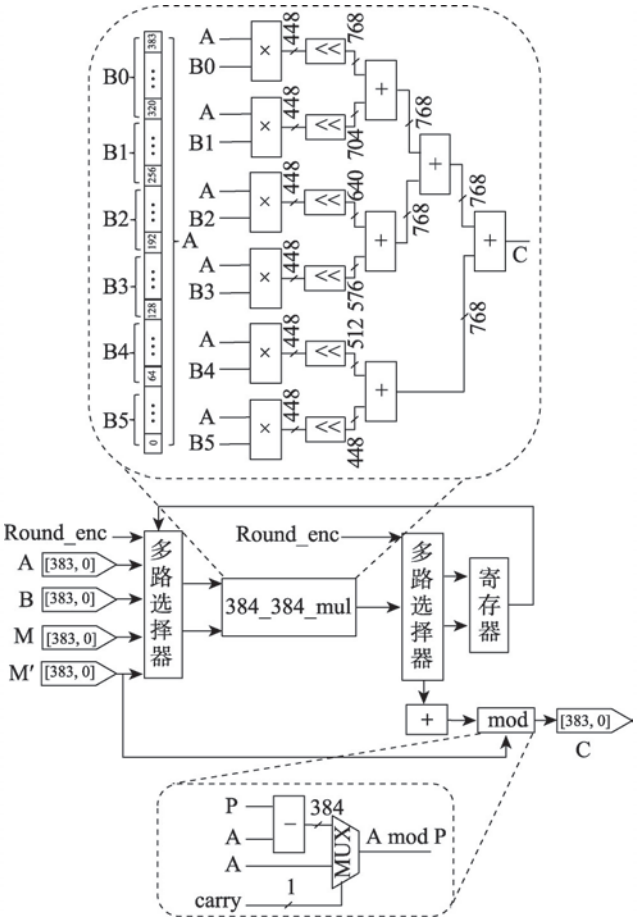


图5 384位蒙哥马利模乘单元

Fig.5 384 bit Montgomery modular multiplication unit

目及位宽。模乘单元主要由一个高位乘法器和一个由加法器、减法器、比较器组成的模运算器组成。由于直接设计一个输入为256位或384位的高位乘法器会因为复杂的布线方案导致DSP分区跨越，进而限制了计算主频并浪费了DSP资源，本文采用Karatsuba乘法计算[36-38]的思想，通过实现多个子乘法器并结合移位和加法操作，实现低时延、节省资源的高位乘法器。384位乘法器由6个64_384位乘法器组成，256位乘法器由4个64_256位乘法器组成。由于蒙哥马利模乘计算过程中的三次乘法运算有着严格计算次序约束，即使是实例化三个乘法器，也无法实现三次乘法的完全并行。因此复用一个高位乘法器三次，既能在仅增加两个时钟周期的情况下保证模乘的低时延，同时相比于实例化三个乘法器节省了近70% DSP资源，实现了资源与性能之间的平衡。

为了避免蒙哥马利模乘运算前后频繁地进行蒙哥马利约简，在整个zk-SNARK计算过程中所有的操作数均以蒙哥马利形式表示。尽管这样做会导致最终证明生成的数据与常规计算下的数值结果不一

计算机学报 www.cnki.org

致,但是并不影响验证过程中的椭圆曲线配对操作的正确性,有效避免了片上大量的蒙哥马利约简操作带来的资源和计算开销。

2.2 NTT计算任务并行化

根据奇偶下标对NTT计算任务不断进行折半划分,在降低算法时间复杂度的同时,却也导致不同规模的子任务必须严格按照递归调用顺序执行计算。这种计算方法使得NTT计算任务无法仅经过单次划分获得自定义数目的独立子任务,尤其是当NTT输入规模过大时,需要动态规划子任务的划分次数并严格控制计算次序,将会带来巨大存储开销。这无疑给NTT子任务调度以及数据流控制系统的硬件实现提出了挑战。本文摒弃传统的折半任务划分方式,采用了一种改进的二维子任务划分方式[39],一次性地将NTT任务划分为任意规模的NTT子任务,如图6所示。

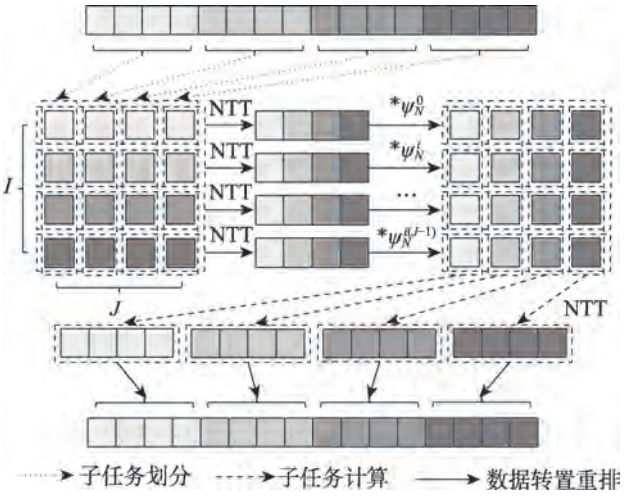


图6 二维子任务划分下的NTT计算流程

Fig.6 2D subtask partition of NTT computation

首先将一个输入规模为 N 的NTT任务的输入向量按照行优先存储的方式组织为 I 行 J 列二维矩阵,进一步按列将其划分为 J 个一维向量,作为 J 个输入规模为 I 的NTT子任务进行计算。在得到 J 个子任务的计算结果后,根据向量元素在二维矩阵中的坐标 (i,j) 分别对其乘以第 $i(j-1)$ 个旋转因子 $\psi_N^{i(j-1)}$,并将其在原二维矩阵中更新。接下来,将原始二维矩阵进行转置,重新按列划分为 I 个一维向量,作为 I 个输入规模为 J 的NTT子任务再次计算。最后,将这 I 个NTT子任务的输出重新组合成长度为 N 的一维数组,即得到了原始NTT任务的输出向量。

这种任务划分方式既保证了串行计算时 $O(n\lg n)$ 的时间复杂度,同时也保证了可以对任意规模的NTT任务一律只进行两次任务划分即可得到自定义数目、自定义规模的子任务。相比于NTT的分治-递归计算方法,这种NTT算法在时间复杂度和并行性上兼具优势,尤其适合zk-SNARK中输入规模高达数百万的NTT计算任务。受限于FPGA与主机端的传输带宽以及片上器件资源数目,通常难以在片上实现一个处理完整输入向量的NTT加速核心。因此可以采用该任务划分方式将规模为 N 的输入向量划分为 $\sqrt{N}$ 个规模为 $\sqrt{N}$ 的向量,进而设计一个计算规模较小、高效且可以被复用的NTT加速核心。进一步地,针对计算规模为 $\sqrt{N}$ 的NTT任务,核心内同样可以采用该任务划分方式生成规模更小的子任务,以实现这些子任务的并行化以及子任务内多轮蝶形运算间的并行化。

2.3 数据重排与NTT计算核心

基于蝶形运算的NTT迭代更新计算方法相比于递归计算方法有着良好的并行性,在相邻两轮步长不同的蝶形运算间,数据读写依赖有这样一种特点:每一轮蝶形运算更新不必等到上一轮更新完全结束后才能开始,且每一轮蝶形运算相对比上一轮启动间隔逐次减半。图7的下半部分通过一个需要进行5轮蝶形运算更新的计算过程展示了这种计算特点。图中数据段灰色部分表示下一轮迭代启动时当前一轮迭代已经完成更新的数据项。假设单次蝶形运算更新的时间为 $2t$ 且每轮蝶形运算循环执行,第二轮步长为8的蝶形运算的首次更新只需在第一轮蝶形

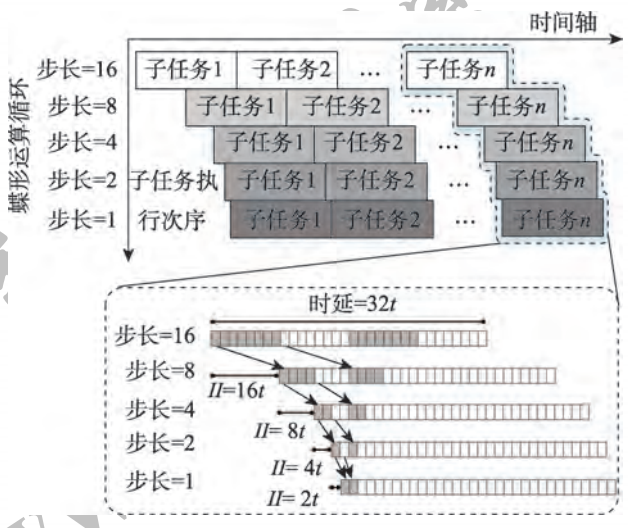


图7 不同步长蝶形运算间的数据依赖及计算流水

Fig.7 Data dependencies and pipeline exist in butterfly operations with different step sizes

运算开始16t后即可开始，第三轮步长为4的蝶形运算更新只需在第二轮更新开始8t后即可开始，以此类推。因此，对于一个输入规模为2^m的NTT子任务，假设每轮蝶形运算循环执行完毕需要的时间为T，相比于需要$m \cdot T$的计算时间才能完成的串行计算方式，如果实现多轮计算之间异步控制，那么只需要$2T$的计算时间即可完成。为了实现不同步长的多轮蝶形运算间的异步计算，本文提出了一种“数据重排”策略以设计高效的计算流水。

数据重排策略本质上是一种计算-通信掩盖策略。由于蝶形运算需要跨步访存，单次蝶形运算更新的时延一定程度上受限于访存。每一轮蝶形运算更新虽然不必等到上一轮蝶形运算完全结束才能开始，但是仍要等上一轮迭代更新足够多的数据后才可启动。因此可以在两轮蝶形运算之间设置先进先出队列（first in first out，FIFO），在更新本轮数据的同时按照下一轮数据跨步读取的顺序依次将数据填充到FIFO中，从而使得下一轮蝶形运算不必跨步访问计算数据，只需从FIFO中依次取出两个操作数进行计算即可。这种基于FIFO实现的数据重排运行机制如图8所示。

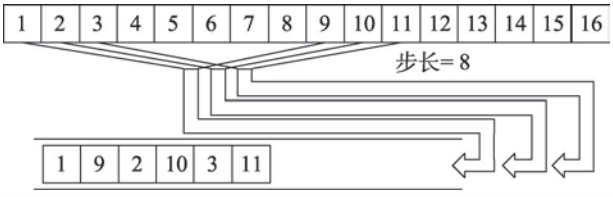


图8 数据重排策略

Fig.8 Data-reordering strategy

例如当步长为16的蝶形运算计算获得了位置为1~8、17~24的元素后，由于下一轮蝶形运算的步长为8，即元素计算顺序为1和9、2和10、3和11……当前轮蝶形运算在计算得到位置为9和25的元素后，可以在下一次计算位置为10和26的元素时，将1、9填充至FIFO。每一轮蝶形运算单元不再是以循环状态机的方式执行，而是采用触发机制进行蝶形运算。只要FIFO中始终有操作数，就会触发与之相连的蝶形运算单元取数运算，并按照特定步长暂存到局部内存中，等待数据足以触发下一轮蝶形运算时将其填充至下一级FIFO。由于多轮蝶形运算同时对一片共享内存区域中的操作数进行更新会导致竞争，这种FIFO实现的数据重排策略相比于BRAM多端口访问模式既提高了计算-访存效率，又保证了多轮蝶形运算间的并行性。本文针对不同步长的蝶形运算单元分别进行了实例化，并使用HLS stream实现了流水线的设计。

通过实现多轮不同步长蝶形运算间的流水，可以实现一个用以计算输入规模为2^n NTT任务的计算架构。考虑到经过二维划分方式得到的NTT子任务完全独立，不存在任何数据依赖，因此各个子任务完全可以复用这一硬件计算架构以节省片上资源消耗。幸运的是，多轮蝶形运算的计算时延相同，这就使得任意两个子任务的相同阶段可以实现任务间的紧密流水，如图7上半部分所示。基于FIFO实现的蝶形运算单元采用触发机制运行，这种数据流驱动模式使得子任务间的计算相比于任务驱动模式响应更快。在这种流水计算模式下，由于子任务的启动间隔相比于串行计算缩减了一半，这些子任务全部完成的总时间也缩减为原来的一半。

基于此设计的多级流水架构分别在子任务内多轮蝶形运算间以及子任务间实现了计算流水，本文将其封装成一个硬件计算核心，如图9所示。这种多级流水架构具有良好的扩展性：当硬件资源充足时，可以适当减少子任务划分数目，增加不同步长蝶形运算单元的实例化数目；当硬件资源紧张时，则可以减少蝶形运算单元的数目、划分更多的子任务。由于子任务间、相同步长的蝶形运算间是无数据依赖的，甚至可以适当增减用于不同步长蝶形运算的硬件单元以及蝶形运算流水线的数目以获得更高的计算性能。良好的扩展性使得该计算核心非常适合在FPGA上实现。

2.4 整体设计

采用上述NTT计算核心，本文设计了一种用于处理大规模、高位宽NTT计算任务的架构，如图10所示。主机端首先对输入规模为N的NTT计算任务进行二维独立子任务划分，进而将划分得到的J个规模为I的子任务的输入数据通过PCIe总线流式传输到FPGA板卡的NTT计算核心缓冲区。这些缓冲区由可多端口读写的BRAM构成。当计算缓冲区的数据足以启动NTT核心的第一轮蝶形运算，第一个NTT计算核心将被触发调用并开始逐渐充盈多级流水计算。当该NTT计算核心开始依次输出子任务的计算结果时，通过一组连接到BRAM的FIFO对输出数据进行转置重排，原本J个规模为I的子任务结果通过数据选择器分散到I个BRAM存储单元组，重新产生I个规模为J的子任务。重排后新的子任务的

计算机学报 www.cnki.cn

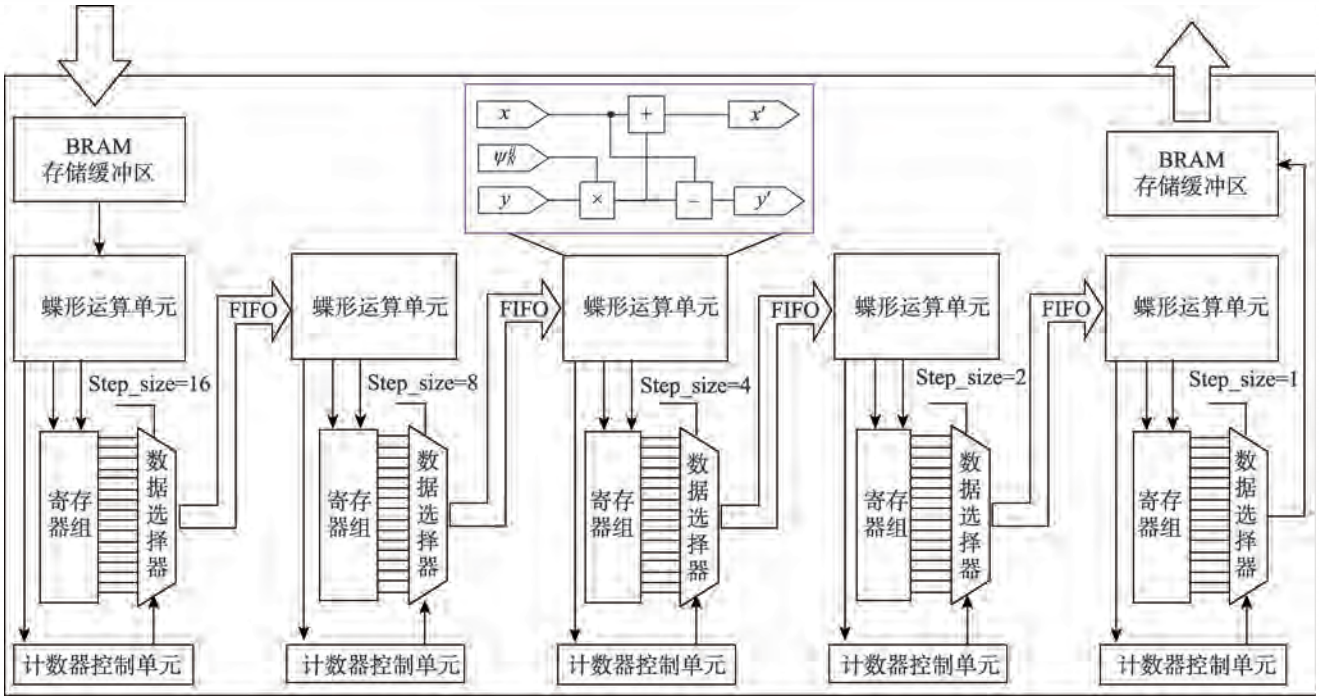


图9 进行5轮蝶形运算的NTT计算核心

Fig.9 NTT core with 5 rounds of butterfly operations

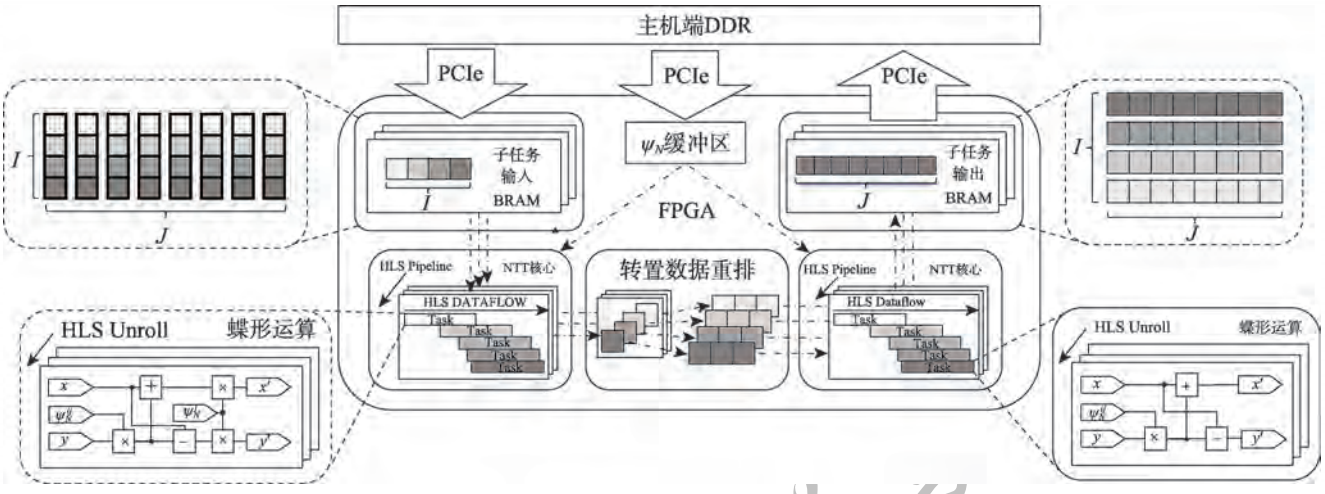


图10 多级流水NTT计算架构

Fig.10 Architecture of NTT computing based on multi-level pipeline

BRAM将依次将数据传输至另外一个NTT核心将其启动。接下来第二个NTT计算核心将逐渐充盈并依次触发不同步长的蝶形运算单元、计算产生 I 个子任务的输出结果。至此，规模为 N 的NTT计算任务的输出将流式地由第二个计算核心产生，只需将其按批次通过PCIe总线传回主机端即可。

值得一提的是，NTT任务中用到由单位根生成的旋转因子通常是在NTT计算过程中实时计算生成。然而，对于FPGA来说，由于旋转因子计算涉及到了大整数幂运算，在NTT计算核心中引入旋转因子计算过程将耗费大量DSP资源以进行模乘运算，同时还会降低计算性能。因此本文针对zk-SNARK证明生成过程中常用规模的NTT计算核心预先计算出所有的旋转因子，将其存储在片外，运算时随输入数据一同传输到片上。

另外，由于基于二维任务划分的NTT计算流程中需要对数据重排前、第一次按列划分得到的各子任务的结果向量乘以一次旋转因子，为了不引入额

外的计算模块，本文将这一步操作并入了第一个NTT核心的计算过程中。故第一次调用的NTT计算核心的输出结果是最后一轮蝶形运算结果再一次乘以旋转因子后得到的。因此，在整个架构中前后调用的两个NTT核心略有区别，但是并不会影响整个架构的计算时延。

这种多级流水架构具有良好的扩展性，可以适应不同资源规模的FPGA。通过合理调整NTT计算核心的规模以及对于原始NTT的任务划分方式，能充分利用片上资源以实现最大化加速。

3 实现与实验评估

3.1 系统实现与实验设置

对于提出的计算架构，本文使用C++编程语言和Vitis HLS实现了硬件计算架构的内核部分，包括顶层内核——NTT计算核心，以及该内核所需要用到的底层模块，如高位模运算单元。主机应用部分使用OpenCL异构编程框架来与硬件加速内核部分建立通信，管理内核调用以及主机、内核之间的数据传输。本文使用Xilinx FPGA开发套件Vitis 2021.2在Alveo U50实现了加速内核，同时基于AMD Ryzen 9 5900X实现了主机应用部分。完整的异构加速系统如图11所示。

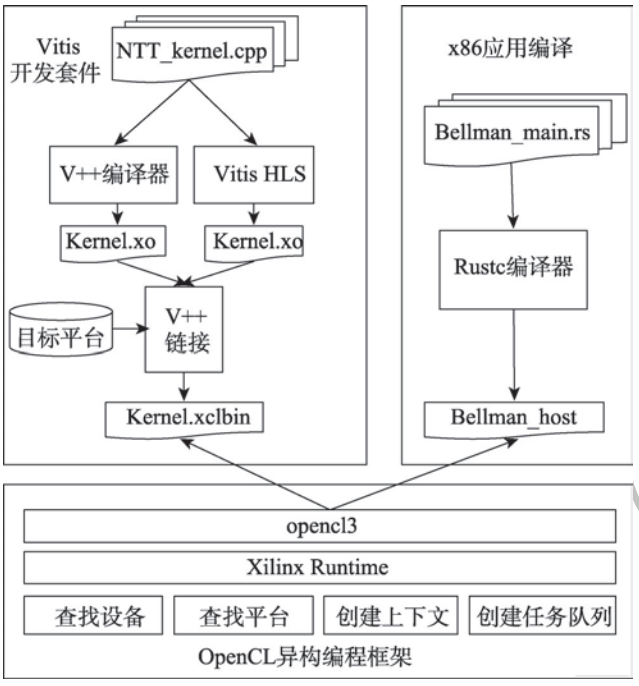


图11 基于Rust OpenCL的NTT异构加速系统

Fig.11 NTT heterogeneous acceleration system based on Rust OpenCL

主机端zk-SNARK主程序采用了bellman中的Groth16算法实现。由于bellman基于Rust语言开发，相应地在主程序中使用了Rust封装的OpenCL库——opencl3，而非原生的OpenCL API。本文重新实现了查找平台和设备的接口，从而使其能够识别AMD Xilinx XRT以及FPGA板卡。NTT计算任务通过任务队列的方式组织，使其可以依次调用NTT内核。而在设备端，两个NTT核心被封装为内核函数，并通过Vitis工具综合生成比特流，烧录在FPGA上。在NTT核心内部，子任务间的流水通过HLS PIPELINE实现，蝶形运算间的FIFO借助HLS stream类型定义，同时使用HLS DATAFLOW指令实现流水；而对于底层高位模运算单元，本文借助HLS中的任意精度无符号整数类型ap_uint定义数据类型并描述计算过程。

为了能够使得bellman中Groth16算法的NTT计算部分调用硬件计算核心，具体地，首先全局定义了一个名为Device_ctx的Rust类，其中成员指针变量在类初始化时由构造函数赋以平台、设备、创建的上下文、命令队列的句柄。在完成类初始化后，证明生成函数将其作为指针参数传递至内部调用。bellman中NTT计算函数被名为NTT_executer的函数的替代：首先输入参数不再直接参与计算，而是经由Device_ctx的自定义成员函数Create_buffer创建相同大小的设备缓冲区并进行拷贝；接下来加载xclbin格式的NTT计算内核，生成硬件可执行的program抽象对象；为该program对象设置启动参数并将其载入命令队列，片上内核异步启动；NTT计算核心完成计算后，主机端同步读取计算结果缓冲区的内容并将其作为函数返回值返回。简而言之，原本NTT计算函数中的计算流程被缓冲区创建与拷贝、内核加载、内核发起调用等OpenCL API替代，真正的计算过程则被等价卸载到了NTT计算核心上。

为了验证所提出架构的合理性，选择PipeZK中的POLY子模块进行同类工作对比，以评估面向大规模、高位宽NTT计算任务时计算核心的多级流水性能。尽管说当前用于zk-SNARK场景下的FPGA加速工作并不成熟，本文仍然选取了近年来基于FPGA的通用NTT加速工作进行对比，以证明大规模高位宽情况下本文方案的优势。而为了验证所提出架构在真实zk-SNARK应用场景下的计算时延与能耗，选择当前先进的零知识证明库bellman作为基准测试程序。此外，为了验证NTT计算架构的正确性，本文将bellman进行一次证明生成过程中的NTT、iNTT输入

www.ceaj.org 计算机学报

数据和计算结果保存到本地,作为NTT计算架构的输入以及结果正确性验证的参照。该NTT计算架构在不同规模下都获得了与bellman纯软件运行一致的计算结果。

bellman中的NTT、iNTT模块都是以2的整数次幂的规模进行计算,多项式乘法过程所需要进行的NTT、iNTT计算任务量如果不能恰好满足2的整数次幂,均通过补0进行填充。

3.2 资源消耗

由于zk-SNARK中多项式乘法运算大多定义在256、384位的素数域上,本文主要实现了256、384位的模运算单元。表2中列出了高位乘法和蒙哥马利模乘的详细资源消耗。使用DSP直接生成一个256位或384的乘法器需要消耗的DSP过多,且较难满足时序约束,因此首先使用DSP和LUT实现了多个低位子乘法器,进而级联成一个高位乘法器。由于蒙哥马利模乘单元较好地复用了单个高位乘法器,实现单个蒙哥马利模乘单元所消耗的资源数与单个高位乘法器基本一致。

表2 高位乘法、蒙哥马利模乘资源消耗

Table 2 Resource consumption of large-bitwidth multiplication and Montgomery modular multiplication

计算模块	位宽	DSP	FF	LUT
高位乘法	384	261(4.4%)	7 384(0.4%)	11 972(1.3%)
	256	224(3.7%)	1 246(0.07%)	1 842(0.2%)
蒙哥马利模乘	384	261(4.4%)	10 468(0.6%)	13 839(1.6%)
	256	224(3.7%)	1 540(0.08%)	2 017(0.2%)

表3中列出了一个输入规模为4 096的NTT核心的详细资源消耗,这对于百万级别的NTT任务划分得到的子任务来说是完全足够的。为了在完整系统中尽快提供可用的NTT加速核心,其工作频率初步设定为68 MHz,未来将设计更深的流水以提高工作频率。由于NTT核心内可以通过改变子任务的划分方式来调节核心内蝶形运算的轮次,实验中给出了单个NTT核心内进行不同轮次的蝶形运算以及实例化不同数目的蝶形运算流水线(子任务并行度)的资源消耗情况,分别为:方案1,8轮蝶形运算&16个子任务&子任务并行度为1;方案2,8轮蝶形运算&16个子任务&子任务并行度为2;方案3,4轮蝶形运算&256个子任务&子任务并行度为1;方案4,4轮蝶形运算&256个子任务&子任务并行度为2;方案5,4轮蝶形运算&256个子任务&子任务并行度为4。实验数据表明,实例化多条流水线并行处理子任务,资源占用率上升,但是计算时延同比下降。而如果片上资源极度受限,仅能实例化一条蝶形运算流水线时,通过减少蝶形运算轮次,资源占用率同样同比下降,并且计算时延基本保持不变。实验结果表明NTT计算核心在资源消耗和计算性能上有着较好的扩展性,用户可以根据片上可用资源的规模选择性地加载不同规模的NTT核心,以实现不同程度的加速。

表3 4 096-NTT计算核心资源消耗与时延(工作频率:68 MHz)

Table 3 Resource consumption and latency of 4 096-sized NTT core (frequency: 68 MHz)

方案	DSP	FF	LUT	BRAM	时延(时钟周期数)
1	1 795 (30.2%)	47 329 (2.7%)	57 058 (6.6%)	448 (16.0%)	39 094
2	3 592 (60.3%)	101 284 (5.8%)	118 680 (13.6%)	986 (35.1%)	20 361
3	899 (15.1%)	28 395 (1.6%)	33 323 (3.8%)	320 (11.4%)	39 407
4	1 798 (30.2%)	60 822 (3.5%)	67 699 (7.8%)	653 (23.2%)	20 312
5	3 597 (60.4%)	130 767 (7.5%)	137 429 (15.8%)	1 372 (48.8%)	10 524

3.3 同类工作对比

尽管Kawamura等人[40]以及Ozcan等人[41]同样采用HLS实现了用于NTT计算的硬件加速核心,但由于其分别仅能支持64位操作数以及1 024的输入规模,且拓宽其设计会因资源膨胀而无法评估,本文选择与同样用于zk-SNARK硬件加速的PipeZK[13]进行评估。PipeZK是在28 nm级ASIC(2 400 MHz)上实现的与本文工作研究方向最为接近的高水平工作,其中用于多项式乘法计算的POLY模块同样采用流水思想设计以实现NTT的计算加速。为了排除器件、频率等外部因素的差异而单纯地比较本文与PipeZK计算架构的性能,本文参考PipeZK中NTT计算模块的设计方案在U50(XCU50 FPGA,频率最高为300 MHz)上复现了其流水架构,并且保留了其计算规模可扩展的特性。

表4、表5分别展示了本文方案与PipeZK的NTT计算核心在U50上处理不同输入规模、不同数据位宽NTT计算任务的计算时延与吞吐率对比。其中计算吞吐率为NTT计算任务的输入数据量与处理时间的比值。由于本文提出的计算核心内实现了多个子任务之间的流水,相比于PipeZK,在子任务时延同为

计算机科学与探索 www.ceaj.org

$2T$（单次蝶形运算的时延为T）的情况下将核心内的子任务启动间隔缩短为T，使得具有n个子任务的整体计算时延由$2n\cdot T$缩短为$n\cdot T$，计算时间缩短了一半，计算吞吐率提升了一倍。表4、表5中的实验结果表明，随着NTT核心输入规模扩大、核心内子任务数量增多，相比于PipeZK确实获得了越来越接近于理论上2倍的加速比，同时单个NTT任务的计算吞吐率提升也逼近于2倍。

表4 NTT计算核心与PipeZK计算时延对比

Table 4 Latency comparison of NTT computing core and PipeZK 单位：ms

输入规模	256位		384位	
	PipeZK	本文方案	PipeZK	本文方案
4 096	6.772	3.984	14.424	8.327
8 192	13.554	7.737	29.819	16.866
16 384	28.446	15.205	64.572	34.363
32 768	55.469	29.346	129.797	67.789
65 536	114.455	59.722	263.696	137.597
262 144	468.860	240.973	1 064.406	553.029
1 048 576	1 892.553	968.722	4 433.208	2 294.065
2 097 152	4 039.993	2 067.869	9 106.756	4 694.532

表5 NTT计算核心与PipeZK计算吞吐率对比

Table 5 Throughput comparison of NTT computing core and PipeZK 单位：MB/s

输入规模	256位		384位	
	PipeZK	本文方案	PipeZK	本文方案
4 096	18.458	31.375	12.999	22.517
8 192	18.444	32.312	12.575	22.234
16 384	17.577	32.883	11.614	21.825
32 768	18.028	34.076	11.556	22.127
65 536	17.474	33.488	11.376	21.802
262 144	17.062	33.198	11.273	21.698
1 048 576	16.908	33.033	10.827	20.923
2 097 152	15.841	30.949	10.541	20.449

本文提出的NTT计算核心为了能够支持大规模、高位宽计算任务，在权衡资源利用率与计算时延的情况下，设计了低时延高位模运算单元，并基于二维子任务划分设计了计算流水。因此，相比于文献[41]和文献[27]中面向32位、1 024输入规模的计算架构，由于他们无需考虑硬件实现成本以及硬件平台差异性的影响，本文实验性能难以望其项背。然而，相比于文献[21]和文献[40]中规模稍大一些的NTT计算任务的性能，本文方案逐渐呈现出优势，尤其是相比于同样使用HLS实现的方案，如表6所示。在不考虑高位模运算优化的情况下，前人文献中的方法牺牲资源利用率确实换取了更好的计算性能，但是这些方法一旦扩展到zk-SNARK应用场景下便难以落地实现。本文正是在真实应用场景中计算核心可实现的前提下，尽可能地提升了NTT的计算性能。

表6 通用NTT加速方案对比

Table 6 Comparison of universal NTT acceleration

对比文献	设备	语言	位宽	输入规模	计算时延/μs
文献[27]	VIRTEX-7	Verilog	32	1 024	1.25
文献[40]	VIRTEX-7	HLS	32	32 768	1 974
文献[21]	xcvu190	Verilog	62	131 072	3 280
文献[41]	VIRTEX-7	HLS	32	1 024	0.076
本文	Alveo U50	HLS	32	1 024	2.1
			32	32 768	67.2
			64	1 024	5.7
			64	131 072	972.0

3.4 bellman工作负载下的性能对比

由于发起证明生成请求的角色表现出群体数量大、个体离散化的特征，证明生成的计算时延是评估计算性能的首要因素。除此之外，zk-SNARK在真实场景下部署时，低功耗是评估服务成本的另一关键指标。基于zk-SNARK生成证明时快速响应、节能低功耗的部署需求，本文将NTT计算架构加载到了bellman的NTT计算任务下，与其在CPU上纯软件运行时的计算时间、计算能效比、计算吞吐率方面进行了详细对比，如表7所示。其中NTT计算架构的计算能效比定义为计算吞吐率与计算功耗的比值。

bellman的证明生成过程共计进行了4次NTT和4次iNTT计算，由于其NTT与iNTT计算任务除了旋转因子取值外没有计算流程上的差别，分别测试了bellman中单次NTT计算任务在AMD Ryzen 9 5900X（3.7 GHz）单核以及12核24线程下的计算性能作为基准，以评估基于U50的NTT计算架构在进行相同的NTT计算任务时的性能。相比于CPU单核计算，NTT计算架构在百万计算规模下呈现出了28倍左右的加速比，相比于12核计算，仍然获得了1.75倍的加速效果。对于计算吞吐率，NTT计算核心在百万计算规模下达到了32 MB/s。

在能效比方面，本文使用powertop功耗分析工具获得了bellman在CPU上运行时NTT计算任务的功耗，同时使用Vitis Analyzer跟踪获取到了NTT计算核心的功耗，进而计算获得了两者工作时的能效比。相比于CPU计算，NTT计算核心获得了6~7倍

计算机学报 www.cjcp.org

表7 bellman工作负载下单次NTT计算的性能对比

Table 7 Performance evaluation of single NTT task under bellman workload

输入规模	计算设备	计算时间/ms	倍率	计算吞吐率/(MB/s)	倍率	计算能效比/(MB/(W·s))	倍率
1 000	CPU单核	103.652	51.671	0.632	51.691	0.395	12.747
	CPU12核	6.497	3.239	10.087	3.239	0.449	11.214
	U50	2.006	—	32.669	—	5.035	—
10 000	CPU单核	847.924	28.894	1.237	28.885	0.773	7.123
	CPU12核	52.964	1.805	19.799	1.805	0.881	6.250
	U50	29.346	—	35.731	—	5.506	—
100 000	CPU单核	7 295.126	30.278	1.149	30.302	0.718	7.474
	CPU12核	455.822	1.892	18.403	1.892	0.819	6.552
	U50	240.937	—	34.817	—	5.366	—
1 000 000	CPU单核	57 861.365	27.981	1.160	27.977	0.725	6.898
	CPU12核	3 616.648	1.749	18.632	1.742	0.829	6.033
	U50	2 067.869	—	32.453	—	5.001	—

的能效提升。图12更加直观地分别展示了运行在U50上的NTT计算核心与CPU单核、12核运行时在不同计算规模下的计算吞吐率以及能效比。

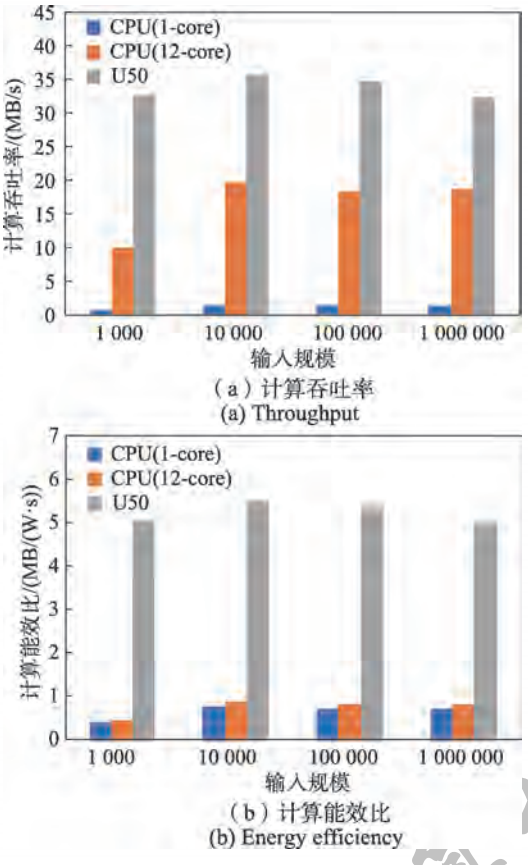


图12 不同规模下NTT计算吞吐率及计算能效比对比

Fig.12 Computation throughput and energy efficiency of NTT under U50 and CPU in different scales

此外,为了验证计算架构的稳定性,实验同时测试了NTT核心与CPU批量进行相同计算任务时的时间抖动情况,如图13所示。实验分别测试了CPU、U50进行100次百万规模的NTT计算任务的时间。相比于使用CPU计算,基于U50的NTT加速核心在进行批量计算时计算时间的方差要小得多,计算时间抖动更小,计算性能更加稳定。

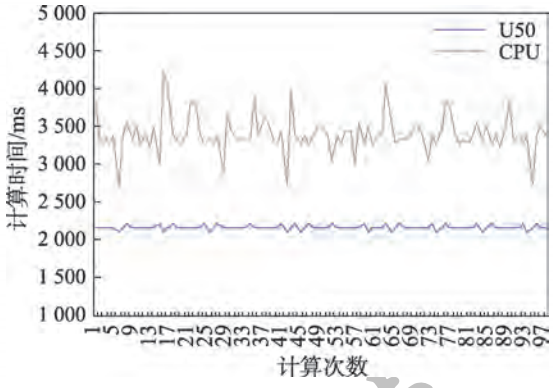


图13 NTT计算抖动对比

Fig.13 Computation stability of NTT core

4 结束语

本文提出了一种基于FPGA用于zk-SNARK中NTT计算的硬件架构。借助“数据重排”策略,实现了NTT子任务间以及子任务内不同步长蝶形运算间的计算流水。在计算核心内部,子任务划分方式、蝶形运算流水线数目以及相同步长蝶形运算单元的数目均可以动态调整,有着良好的扩展性。在与zk-SNARK硬件加速的最先进工作PipeZK以及开源库bellman的实验对比中发现,本文提出的计算架构在计算时延、能效方面有着一定的优势。研究同时表明结合OpenCL、HLS对zk-SNARK进行软硬件协同加速,是实现零知识证明高性能、低功耗部署的有效手段。在接下来的研究工作中,将探索如何在NTT

计算机科学与探索 www.ceaj.org

内核与设备I/O通信间建立更加高效的缓冲区以平衡计算、通信负载,以及如何通过任务队列调度多计算核心以实现高并发计算。

参考文献:

[1] GOLDWASSER S, MICALI S, RACKOFF C. The knowledge complexity of interactive proof-systems[M]//Providing Sound Foundations for Cryptography: on the Work of Shafi Goldwasser and Silvio Micali. New York: ACM, 2019: 203-225.

[2] BITANSKY N, CANETTI R, CHIESA A, et al. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again[C]//Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, Cambridge, Jan 8-10, 2012. New York: ACM, 2012: 326-349.

[3] BITANSKY N, CANETTI R, CHIESA A, et al. The hunting of the SNARK[J]. Journal of Cryptology, 2017, 30(4): 989-1066.

[4] BLUM M, FELDMAN P, MICALI S. Non-interactive zero-knowledge and its applications[M]//Providing Sound Foundations for Cryptography: on the Work of Shafi Goldwasser and Silvio Micali. New York: ACM, 2019: 329-349.

[5] 李威翰, 张宗洋, 周子博,等. 简洁非交互零知识证明综述[J]. 密码学报, 2022, 9(3): 379-447.
LI W H, ZHANG Z Y, ZHOU Z B, et al. An overview on succinct non-interactive zero-knowledge proofs[J]. Journal of Cryptologic Research, 2022, 9(3): 379-447.

[6] DELIGNAT-LAVAUD A, FOURNET C, KOHLWEISS M, et al. Cinderella: turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation[C]//Proceedings of the 2016 IEEE Symposium on Security and Privacy, San Jose, May 23-25, 2016. Piscataway: IEEE, 2016: 235-254.

[7] 单进勇, 高胜. 区块链理论研究进展[J]. 密码学报, 2018, 5(5):484-500.
SHAN J Y, GAO S. Reasearch progress on theory of blockchains[J]. Journal of Cryptologic Research, 2018, 5(5): 484-500.

[8] DANEZIS G, FOURNET C, KOHLWEISS M, et al. Pinocchio coin: building zerocoin from a succinct pairing-based proof system[C]//Proceedings of the 1st ACM Workshop on Language Support for Privacy- Enhancing Technologies, Berlin, Nov 4, 2013. New York: ACM, 2013: 27-30.

[9] SASSON E B, CHIESA A, GARMAN C, et al. Zerocash: decentralized anonymous payments from bitcoin[C]//Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, May 18-21. Piscataway: IEEE, 2014: 459-474.

[10] HUANG H S, CHANG T S, WU J Y. A secure file sharing system based on IPFS and blockchain[C]//Proceedings of the 2020 2nd International Electronics Communication Conference, Kuala Lumpur, Aug 12-14, 2020. New York: ACM, 2020: 96-100.

[11] ZHANG Y, GENKIN D, KATZ J, et al. vSQL: verifying arbitrary SQL queries over dynamic outsourced databases[C]//Proceedings of the 2017 IEEE Symposium on Security and Privacy, San Jose, May 22-24, 2017. Piscataway: IEEE, 2017: 863-880.

[12] BENET J. IPFS-content addressed, versioned, P2P file system[J]. arXiv:1407.3561, 2014.

[13] ZHANG Y, WANG S, ZHANG X, et al. Pipezk: accelerating zero-knowledge proof with a pipelined architecture[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, Spain, Jun 14-19, 2021. Piscataway: IEEE, 2021: 416-428.

[14] GROTH J, MALLER M. Snarky signatures: minimal signatures of knowledge from simulation-extractable SNARKs [C]//Proceedings of the 37th Annual International Cryptology Conference, Santa Barbara, Aug 20-24, 2017. Cham: Springer, 2017: 581-612.

[15] BEN-SASSON E, BENTOV I, HORESH Y, et al. Scalable, transparent, and post-quantum secure computational integrity[J]. Cryptology ePrint Archive, 2018.

[16] BOWE S, GABIZON A. Making Groth's zk-SNARK simulation extractable in the random oracle model[J]. Cryptology ePrint Archive, 2018.

[17] 黄平, 梁伟洁. 一种基于QAP问题的ZK-SNARK新协议[J]. 华南理工大学学报(自然科学版), 2021, 49(1): 1-9.
HUANG P, LIANG W J. A new ZK-SNARK protocol based on QAP[J]. Journal of South China University of Technology (Natural Science Edition), 2021, 49(1): 1-9.

[18] XIE T, ZHANG J, ZHANG Y, et al. Libra: succinct zero-knowledge proofs with optimal prover computation[C]//Proceedings of the 39th Annual International Cryptology Conference, Santa Barbara, Aug 18-22, 2019. Cham: Springer, 2019: 733-764.

[19] MALLER M, BOWE S, KOHLWEISS M, et al. Sonic: zero-knowledge SNARKs from linear-size universal and updatable structured reference strings[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, Nov 11-15, 2019. New York: ACM, 2019: 2111-2128.

[20] HALEPLIDIS E, TSAKOULIS T, EL-KADY A, et al. Studying OpenCL-based number theoretic transform for heterogeneous platforms[C]//Proceedings of the 2021 24th Euromicro Conference on Digital System Design, Palermo, Sep 1-3, 2021. Piscataway: IEEE, 2021: 339-346.

[21] KIM S, LEE K, CHO W, et al. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme[C]//Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines, Fayetteville, May 3-6, 2020. Piscataway: IEEE, 2020: 56-64.

[22] ÖZTÜRK E, DORÖZ Y, SAVAŞ E, et al. A custom accelerator for homomorphic encryption applications[J]. IEEE Transactions on Computers, 2016, 66(1): 3-16.

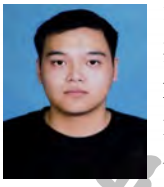
[23] 周慧凯. 同态加密的硬件卸载及其在隐私保护计算中的应用[J]. 小型微型计算机系统, 2021, 42(3): 595-600.
ZHOU H K. Homomorphic encryption offloading and its application in privacy-preserving computing[J]. Journal of Chinese Computer Systems, 2021, 42(3): 595-600.

计算机学报 www.cnki.com

[24] CHEN D D, MENTENS N, VERCAUTEREN F, et al. High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems[J]. IEEE Transactions on Circuits and Systems I: Regular Papers, 2014, 62(1): 157-166.
[25] KALES D, RAMACHER S, RECHBERGER C, et al. Efficient FPGA implementations of LowMC and picnic[C]// Proceedings of the Cryptographers' Track at the RSA Conference, San Francisco, Feb 24-28, 2020. Cham: Springer, 2020: 417-441.
[26] AGRAWAL R, BU L, EHRET A, et al. Open-source FPGA implementation of post-quantum cryptographic hardware primitives[C]//Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications Barcelona, Sep 8-12, 2019. Piscataway: IEEE, 2019: 211-217.
[27] MERT A C, ÖZTÜRK E, SAVAŞ E. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture[C]//Proceedings of the 2019 22nd Euromicro Conference on Digital System Design, Kallithea, Aug 28-30, 2019. Piscataway: IEEE, 2019: 253-260.
[28] RIAZI M S, LAINE K, PELTON B, et al. HEAX: an architecture for computing on encrypted data[C]//Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Mar 16-20, 2020. New York: ACM, 2020: 1295-1309.
[29] 沈耀坡, 梁煜, 张为. 一种高性能快速傅里叶变换的硬件设计[J]. 西安电子科技大学学报, 2018, 45(3): 63-67.
SHEN Y P, LIANG Y, ZHANG W. Hardware efficient fast Fourier transform architecture[J]. Journal of Xidian University, 2018, 45(3): 63-67.
[30] 谢星, 黄新明, 孙玲, 等. 大整数乘法器的FPGA设计与实现[J]. 电子与信息学报, 2019, 41(8): 1855-1860.
XIE X, HUANG X M, SUN L, et al. FPGA design and implementation of large integer multiplier[J]. Journal of Electronics & Information Technology, 2019, 41(8): 1855-1860.
[31] FILECOIN. Bellperson: GPU parallel acceleration for zk-SNARK[EB/OL]. (2020) [2022-10-20]. https://github.com/filecoin-project/bellperson.
[32] GROTH J. On the size of pairing-based non-interactive arguments[C]//Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, May 8-12. Cham: Springer, 2016: 305-326.
[33] FILECOIN. Bellman: zk-SNARK library[EB/OL]. (2018) [2022-10-20]. https://github.com/zkcrypto/bellman.
[34] 崔西宁, 杨经纬, 叶宏, 等. 椭圆曲线密码的优化设计方法[J]. 西安电子科技大学学报, 2015, 42(1): 69-74.
CUI X N, YANG J W, YE H, et al. Optimized design method on elliptic curve cryptography[J]. Journal of Xidian University, 2015, 42(1): 69-74.
[35] MONTGOMERY P L. Modular multiplication without trial division[J]. Mathematics of Computation, 1985, 44(170): 519-521.
[36] ÖZTÜRK E. Modular multiplication algorithm suitable for low-latency circuit implementations[J]. Cryptology ePrint Archive, 2019.
[37] KARATSUBA A. Multiplication of multidigit numbers on automata[J]. Soviet Physics Doklady, 1963, 7: 595-596.
[38] CHOW G C T, EGURO K, LUK W, et al. A Karatsuba-based Montgomery multiplier[C]//Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, Milano, Aug 31-Sep 2, 2010. Piscataway: IEEE, 2010: 434-437.
[39] SZE T W. Schönhage-Strassen algorithm with Mapreduce for multiplying terabit integers[C]//Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation, San Jose, Jun 7-9, 2012. New York: ACM, 2012: 54-62.
[40] KAWAMURA K, YANAGISAWA M, TOGAWA N. A loop structure optimization targeting high-level synthesis of fast number theoretic transform[C]//Proceedings of the 2018 19th International Symposium on Quality Electronic Design, Santa Clara, Mar 13-14, 2018. Piscataway: IEEE, 2018: 106-111.
[41] OZCAN E, AYSU A. High-level synthesis of number-theoretic transform: a case study for future Cryptosystems[J]. IEEE Embedded Systems Letters, 2019, 12(4): 133-136.

赵海旭(1997—),男,山东人,硕士研究生,CCF学生会员,主要研究方向为计算机体系结构、FPGA。
ZHAO Haixu, born in 1997, M.S. candidate, CCF student member. His research interests include computer architecture and FPGA.

柴志雷(1975—),男,山西人,博士,教授,CCF高级会员,主要研究方向为计算机系统结构、智能计算系统。
CHAI Zhilei, born in 1975, Ph.D., professor, CCF senior member. His research interests include computer architecture and intelligent computing system.



花鹏程(1999—),男,江苏人,硕士研究生,主要研究方向为计算机体系结构、并行计算。
HUA Pengcheng, born in 1999, M.S. candidate. His research interests include computer architecture and parallel computing.

王锋(1997—),男,山东人,硕士研究生,主要研究方向为计算机体系结构、并行计算。
WANG Feng, born in 1997, M.S. candidate. His research interests include computer architecture and parallel computing.

丁冬(1998—),男,江苏人,硕士研究生,主要研究方向为计算机体系结构、FPGA。
DING Dong, born in 1998, M.S. candidate. His research interests include computer architecture and FPGA.

计算机科学与探索 www.ceaj.org