# 基于 GPU 的椭圆曲线运算库及相关算法优化

高钰洋[1], 张健宁[1], 王 刚[1,3], 苏 明[1,2,3], 刘晓光[1,3]

[1] 南开大学计算机学院 网络空间安全学院 天津 中国 300350
[2] 贵州大学公共大数据国家重点实验室 贵阳 中国 550025
[3] 数据与智能系统安全教育部重点实验室 天津 中国 300350

**摘要** 在区块链场景下, 往往需要引入数字签名、零知识证明等密码学算法以保护数据安全性与用户隐私。但由于这些算法依赖于大量的大数与椭圆曲线运算, 包括范围证明在内的许多密码学算法已经成为了区块链系统的性能瓶颈。而密码学算法的 GPU 优化也在近几年获得了广泛的关注与研究。本文充分利用 GPU 作为众核处理器的优势, 设计了基于 GPU 的椭圆曲线运算库。在运算库中, 本文在 GPU 上实现并优化了常用的椭圆曲线运算与大数运算, 同时针对不同的需求设计了不同的实现与接口。本文对寄存器与常量内存等存储空间进行了合理分配, 并通过利用预计算等优化手段减少了计算量, 从而最大化了运算库的吞吐与性能。为了验证运算库的实用性与有效性, 本文利用该运算库实现了代理重加密与 Bulletproofs 范围证明的验证算法, 同时充分利用了算法的内部并行性进行优化。实验表明, 本文实现的运算库在各个运算中都取得了远超于 OpenSSL 等常用 CPU 端运算库的性能。基于该运算库实现的代理重加密算法相比 CPU 实现能达到最高 145 倍左右的加速比, Bulletproofs 范围证明验证算法相比于 CPU 端实现也能达到 5.57 倍左右的加速效果, 平均证明验证时间在 1 ms 内,可以满足数字货币隐私保护场景下超过每秒 2000 笔交易的性能需求。可见该运算库能为区块链系统隐私保护等对密码学计算具有高吞吐需求的场景提供坚实支持。

**关键词** 椭圆曲线; 图形处理单元; 统一计算架构; 范围证明; 代理重加密
**中图法分类号** TP309.2 DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.11.01

# Optimization for GPU-based Elliptic Curve Library and Related Algorithms

GAO Yuyang[1], ZHANG Jianning[1], WANG Gang[1,3], SU Ming[1,2,3], LIU Xiaoguang[1,3]

[1] College of Computer Science, College of Cyber Science, Nankai University, Tianjin 300350, China
[2] State Key Laboratory of Public Big Data, Guizhou University, Guiyang 550025, China
[3] Key Laboratory of Data and Intelligent System Security, Ministry of Education, Tianjin 300350, China

**Abstract** In blockchain systems with privacy protection, cryptographic algorithms such as digital signatures and zero-knowledge proofs are often used to protect data security and user privacy. However, many cryptographic algorithms, including range proofs, have become performance bottlenecks of blockchain systems due to their heavy reliance on operations of large numbers and elliptic curves. Moreover, the GPU optimization of cryptographic algorithms has gained extensive attention and research in recent years. We make full use of the advantages of GPU as many-core processors and design a GPU-based elliptic curve operation library. In the library, we implement and optimize the common operations of elliptic curves and large numbers on GPU, and design different implementations and interfaces to accommodate various requirements. To maximize the throughput and performance of the library, we carefully allocate storage such as registers and constant memory, and use optimization methods such as precomputation to reduce the amount of calculation. For testing the usability and effectiveness of the library, we use it to implement the proxy re-encryption algorithm and the verification algorithm of Bulletproofs range proofs. We further optimize them by making full use of the intrinsic parallelism of the algorithms. Experiments show that the operation library achieves a performance that far exceeds that of commonly used CPU-side libraries such as OpenSSL in each operation. Compared with the CPU-side implementation, the proxy re-encryption algorithm implemented with the library achieves up to 145 times speedup. The Bulletproofs range proof verification algorithm implemented with the library achieves a speedup of about 5.57 times as well. The average verification time of GPU-based Bulletproofs is within 1 millisecond, which meets the performance requirement of privacy protection for over 2000 digital currency transactions per second. Therefore, the operation library provides a solid foundation for applications requiring high throughput of cryptographic calculations, such as the privacy protection of blockchain systems.

**通讯作者:** 王刚, 博士, 教授, Email: wgzwp@163.com。
本课题得到国家自然科学基金资助项目(No. 62272253, No. 62272252, No. 62141412), 公共大数据国家重点实验室开放课题(No. PBD2022-12), 天津市科技计划重点研发计划(No. 19YFZCSF00900, No. 20JCZDJC00610), 中央高校基本科研业务费的资助。
收稿日期: 2023-05-22; 修改日期: 2023-07-15; 定稿日期: 2024-09-05

**Key words** elliptic curve; graphics processing unit; compute unified device architecture; range proof; proxy re-encryption

## 1 引言

随着区块链技术的广泛应用, 链上数据的规模也随之越发庞大。而区块链系统数据的公开性在保证了系统的透明性的同时, 也会带来数据泄露的风险。如何保证用户数据的安全性, 使得在存储与计算中都能保证用户数据与用户隐私的安全, 成为了众多研究人员日益关注的课题。

密码学是保护网络安全与数据安全的重要工具, 有着几千年的发展历史[1]。在近几十年获得了飞速的发展, 尤其是以 AES 为代表的对称密码与以椭圆曲线密码学为代表的非对称公钥密码体制问世之后, 密码学在数据共享与传输的过程中扮演着越来越重要的角色。然而随着数据量以及密钥长度的增加, 椭圆曲线密码学所依赖的椭圆曲线操作与大数运算的计算速度渐渐无法满足日益增长的数据量的计算需求, 开始逐渐成为计算瓶颈, 这一点在区块链系统中尤为显著。为了实现对区块链系统中用户数据与隐私的保护, 系统中需要引入环签名、范围证明等高度依赖于椭圆曲线群运算的密码学算法。而相应算法的大量计算开销不仅成为了区块链系统的性能瓶颈, 也阻碍了区块链系统中隐私保护技术的应用。

随着以图形处理单元 (Graphics processing unit, GPU) 为代表的众核平台的发展, 越来越多的研究人员开始尝试使用 GPU 加速各种不同的应用, 尤其在 NVIDIA 公司推出统一计算架构 (Compute unified device architecture, CUDA) 之后, 利用 CUDA 进行的并行优化被应用于包括密码学在内的各种科学计算的加速中。CUDA 是一个适用于 NVDIA GPU 的并行计算开发环境, 允许用户通过一个 C 语言的超集便利地编写利用 GPU 的并行计算程序。由于 GPU 具有高带宽高吞吐的特点, CUDA 在高性能计算中的应用有着较好的效果。

随着开源社区的不断发展, 目前在 CPU 端已有广泛应用且优化细致的密码学与大数运算库可供用户调用, 如 OpenSSL、GMP、GmSSL 等。但在 GPU 端上, 相应的运算库则比较缺乏, 给 GPU 加速在密码学上的应用带来了困难。

目前已有众多研究者针对密码学算法日益突出的性能问题尝试采用 GPU 等众核平台进行优化, 并取得了一定的效果。在大整数运算上, Szerwinski 等人[2]利用余数系统实现了大数运算的优化, 并通过 CUDA 进行了 GPU 实现, 采用细粒度的并行方式实现了模乘与模幂等大数运算与椭圆曲线密码学算法, 取得了较好的成果。Harrison 等人[3]同样采用余数系统实现了蒙哥马利模乘的 CIOS 实现。通过细粒度的并行方案在小规模数据下实现了更高的吞吐。随着 GPU 的发展, 新一代 GPU 的浮点计算能力获得了飞速的提升, 也有研究者尝试采用浮点数实现大数计算。Zheng 等人[4]利用浮点数类型对大整数进行存储与运算, 采用细粒度的任务分配策略实现蒙哥马利模乘算法, 利用 GPU 的浮点计算能力达到了较高的大整数模乘运算吞吐。Dong 等人[5]提出了 sDPF 方法, 使用双精度浮点数加速蒙哥马利模乘, 利用浮点数的符号位增加每个浮点值所处理的信息量, 采用进位保留与进位预测技术, 取得了很好的加速结果。Ochoa 等人[6]在 GPU 与 CPU 上对蒙哥马利模乘均进行了实现与优化, 在 CPU 端使用 AVX2 技术进行并行化计算, 在 GPU 端利用 CUDA 采用细粒度并行方式, 均取得了较好的吞吐。

此外, 也有许多研究者采用 GPU 加速椭圆曲线相关的运算。Pan 等人[7]针对椭圆曲线运算利用 GPU 实现了专门的椭圆曲线算法签名与验证服务器, 针对三条不同的曲线在 GPU 端为椭圆曲线签名与验证进行了细致的优化。通过将部分操作在 CPU 端预先计算, 提高了 GPU 端的吞吐量。Dong 等人[8]利用嵌入式 GPU NVIDIA TX2 实现了椭圆曲线算法, 在较低的功率下便可以达到较高的吞吐。Gao 等人[9]将 sDPF 方法应用在椭圆曲线运算中, 充分利用 GPU 浮点运算能力提高 ECC 吞吐量。Huang 等人[10]针对 Bulletproofs 算法的内积证明利用 CPU-GPU 异构计算进行加速, 采用了细粒度的任务分配策略, 最高能达到 3.7 倍左右的加速效果。

但是这些工作只是聚焦于某一特定运算或平台的优化, 各功能间深度耦合, 缺乏可复用性。其他研究者进行更高层次算法的并行优化时无法便捷地使用相应的 GPU 实现以达到简化底层运算的目的, 反而会在实现中引入新的复杂性, 这一现状不利于密码学上 GPU 加速的应用。为解决这一问题, 本文设计并实现了基于 GPU 的椭圆曲线与大数运算库, 对其进行了合理的设计与优化, 在保证算法库具有良好性能的情况下仍然便于使用者调用, 从而达到易用性与性能的合理平衡。同时我们通过运算库实现了代理重加密与 Bulletproofs 算法验证操作 (Verification) 等在区块链场景下具有广泛应用场景的顶层密码学算法, 并利用算法内部的并行性进行优化,

对计算任务进行了合理的划分与分配, 取得了良好的加速效果。这同时也证明了运算库的实用性与有效性。

代理重加密[11]是一种密文转换算法, 可以实现在不向代理公开明文或私钥的情况下方便地共享存储在代理中的数据。这可以应用于区块链数字资产的授权与转移[12], 允许用户便捷地在不同的服务提供商间共享与移动数字资产, 同时保障用户的数据安全与隐私安全。我们利用本文中的运算库针对代理重加密中的 ECC 加密, 重加密, ECC 解密模块进行了并行优化, 有效提升了算法的计算吞吐。

Bulletproofs[13]是一种非交互式的零知识范围证明算法。证明者可以通过 Bulletproofs 算法证明某个承诺 (Commitment) 对应的秘密值在一个给定的范围内, 但不泄露秘密值的具体大小。Bulletproofs 不需要可信设置 (Trusted setup), 且支持证明聚合与批量验证, 同时生成的证明大小随着证明范围的增大仅仅以对数级别增长。这些优势使得其在众多场景中获得了广泛的应用。如包括门罗币[14]、Zcash[15]在内的许多数字货币就利用了范围证明来隐藏交易中的交易货币量, 以达到隐私保护的目的。我们基于本文中的运算库, 对 Bulletproofs 范围证明验证算法中的计算任务进行了合理的分划与分配, 提升了算法的并行性, 取得了良好的加速效果。

本文主要的研究工作如下。

(1) 分析、设计并实现了大数运算库, 对包含模加、模乘等大数运算操作进行了 GPU 实现, 同时进一步设计了二次剩余求解、Fiat-Shamir 启发式[16]等其他常见密码学操作的 GPU 实现, 并针对不同的应用情况进行了不同的设计和优化, 以保证运算库能在达到高吞吐性能的前提下保持良好的可重用性与泛用性。

(2) 基于大数运算库设计并实现了椭圆曲线运算库, 对包含点加、倍点等操作在内的椭圆曲线群运算进行了 GPU 实现, 并针对不同的情况进行了相应的设计与优化。

(3) 利用本文实现的运算库实现了代理重加密算法, 同时充分利用了算法内部的并行性, 针对 GPU 并行架构进行了优化, 达到了良好的计算吞吐。

(4) 利用本文实现的运算库实现了 Bulletproofs 范围证明的验证算法, 并针对算法内部的并行性合理划分并分配了计算任务, 提高了计算流程的并发度, 实现了良好的并行加速。

本文组织结构如下:

第一节介绍了本文工作的背景与动机, 并对相关的工作与本文的贡献进行了介绍; 第二节介绍了本文实现的 GPU 运算库的实现与优化; 第三节介绍了基于本文运算库实现的代理重加密和 Bulletproofs 范围证明验证算法的实现与优化; 第四节对本文实现的运算库、代理重加密与 Bulletproofs 范围证明的验证算法进行了性能的测试与分析; 第五节对本文的工作进行了总结。

## 2 运算库实现与优化

密码学算法中大数运算与椭圆曲线运算的应用场景可以抽象地分为服务器端场景与用户端场景。在服务器端场景下由于需要同时计算多组数据并且具有强大的算力, 通常更侧重于吞吐; 用户端场景相比而言计算量小, 更侧重于延迟。以 GPU 为代表的众核平台的优势之一就在于高吞吐, 能使得算力强的服务器端处理多批次大数据量的计算, 因此本文 GPU 运算库的实现也侧重于吞吐的提升。

运算库的架构如图 1 所示。由于椭圆曲线运算依赖于大数运算, 运算库首先实现了基本的大整数模运算, 在此基础上实现了椭圆曲线公钥密码体制的常用运算, 包括点加、任意点标量乘、基点标量乘以及字节串与椭圆曲线群之间的映射函数, 并进一步利用椭圆曲线与大整数运算实现了包括代理重加密与 Bulletproofs 在内的上层密码学算法。

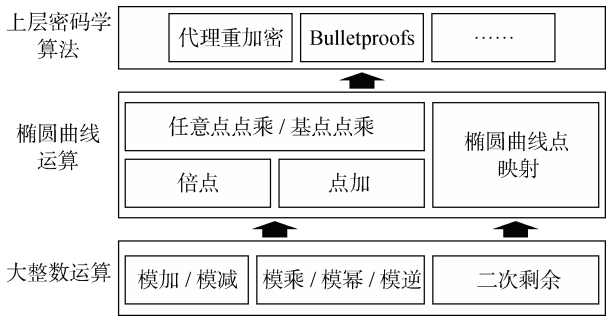



**图 1 运算库整体架构**

**Figure 1 Overall architecture of the operation library**

由于基础的大数与椭圆曲线运算内部计算量不大, 运算步骤间存在较强的依赖关系, 且在上层密码学算法计算中可能需要执行成千上万次这样的底层运算。为取得易用性与性能的平衡, 运算库中的单个大数与椭圆曲线群运算并未采用多线程任务划分的并行策略, 而是采用单一线程负责单个运算的执行策略, 并在上层算法中通过“运算间并行”来充分利用 GPU 运算能力, 以减少线程间的同步开销, 最大化系统吞吐。

本文实现的 GPU 库的部分接口如表 1 所示, 考

虑到篇幅限制这里只列出部分主要接口。由于 GPU 端的计算需要合理安排线程间的任务分配, 为了方便上层算法的调用, 运算库中将各个接口均定义为能在 GPU 端方便调用的 device 函数。对于涉及到大数运算的接口, 本文实现了操作数分别为大整数与 32 位整数的不同版本, 对于点乘等算法则实现了 in-place 与 out-of-place 的不同版本以适应不同的运算需求。

**表 1 GPU 运算库函数接口定义**

**Table 1 Defination of the interface of GPU library**

| 函数 | 功能 |
|---|---|
| dh_mod_add | 大数模加 |
| dh_mod_mon_mul | 大数蒙哥马利模乘 |
| dh_mod_inv | 大数模逆 |
| dh_point_add | 椭圆曲线 Jacobian 坐标点加 |
| dh_point_double | 椭圆曲线 Jacobian 坐标倍点 |
| dh_point_mul | 椭圆曲线 Jacobian 坐标点乘 |
| d_base_point_mul | 椭圆曲线基点点乘 |
| check_quadratic_residue | 判断二次剩余 |
| cal_quadratic_residue | 求解二次剩余 |
| get_challenge | 计算 Fiat-Shamir 启发式 |

## 2.1 大数运算实现与优化

选择合适的表示与存储方式是进行大数运算实现与优化的关键。本文实现聚焦于椭圆曲线 secp256k1, 其依赖的大整数长度均为 256 位, 而目前计算机中无法直接存储 256 位整数, 所以实现中采用了数组形式存储, 将多个机器支持的原生整数类型(下文称为节)组合在一起表示整个大整数。本文中的实现将 4 个 64 位整型用作 256 位大整数的存储。相比于采用 8 个 32 位整型的存储方式, 本文所采用的的存储方式在实验中的吞吐更高。

常见的大数运算操作包含模加、模减、模乘、模幂、模逆、二次剩余求解等操作。在本文运算库中对这些常见的运算给出了相应的 GPU 实现。为了便于上层算法实现, 我们没有采用类似于文献[3]的多线程协同分节计算的方案, 而是使用单线程完成单个大数运算, 并针对单个运算进行线程内优化, 从而使得运算库能在达到良好性能的同时仍然保持易用性。

模加与模减的操作类似, 以模加操作为例, 模加操作需要计算

$$X = A + B (\mathrm{mod}\, N),$$

由于 $0 \leqslant A, B < N$, 所以 $0 \leqslant A + B < 2N$, 因此可以通过最多一次减 $N$ 操作将结果限制在 $[0, N)$ 之内, 从而避免常规的取模运算带来的额外开销。模加运算采用常见的从低位到高位对应节相加, 同时向高位传递进位的加法思路进行实现。

合理利用 GPU 指令集能够简化程序的计算流程, 同时充分利用硬件能力, 从而提升性能。针对大整数运算场景, CUDA PTX 提供了方便的带进位的加减与乘加指令。本文的实现就采用了 CUDA PTX 指令中的 add.cc.u64 提高计算性能。这些相应的 PTX 指令, 会将进位与借位自动存储在条件码寄存器 (Condition code register, CC) 的进位标志位 (CC.CF) 中, 同时 CC.CF 也会参与下一次运算。这可以减少对全局内存不必要的访问, 进而提高吞吐。模减运算的计算流程是类似的, 也可以采用相应的优化手段。

大数运算中的模乘运算需要计算

$$X = A \times B (\mathrm{mod}\, N).$$

不同于模加模减运算, 模乘运算无法通过单次加减法完成取模操作。为了避免取模操作中吞吐较低的模逆运算, 本文采用蒙哥马利模乘算法[17]来设计和实现运算库中的 GPU 模乘运算。蒙哥马利模乘算法具有多种不同的实现方法, 根据运算的扫描模式以及乘法和约减是聚合还是分开进行可以分为五种不同的方式[18]。经过分析与测试, CIOS 方法较之其他方法具有较少的访存次数与计算次数[18], 更适合在 GPU 中进行实现, 因此本文中蒙哥马利模乘的实现方式即为 CIOS 方法。

蒙哥马利模乘算法计算的是下式的结果:

$$mon(A, B) = ABR^{-1} (\mathrm{mod}\, N),$$

其中 $R$ 为一个比 $N$ 大且与 $N$ 互素的数, 为计算方便一般取恰好大于 $N$ 的 2 的幂次, 在本运算库中即取 $R = 2^{256}$。计算过程中需要先将大整数 $X$ 与 $R^2 \bmod N$ 进行蒙哥马利模乘, 转换到蒙哥马利表示 $XR \bmod N$, 全部计算完成后再将计算结果转换回一般表示。

蒙哥马利模乘的实际计算中, 还需要一些常数的值, 如 $n' = N^{-1} (\mathrm{mod}\, R)$。因为在运算中 $N$ 与 $R$ 均为固定值, 所以如 $1, N, R, R^2$ 这样的大整数常数可以事先计算完成并存储在 GPU 的常量内存中, 并通过常量内存的广播与缓存机制方便多线程并行读取, 提高性能。同时实现中还采用了 CUDA 内置函数来加速计算, 如使用__umul64hi()函数获取 64 位无符号整数相乘的高位。实现中也进一步考虑了寄存器的复用以减少寄存器的使用, 提高程序的占用率。

此外, 本运算库还实现了模幂、模逆、二次剩余判断和求解算法。其中蒙哥马利模幂基于蒙哥马利

模乘, 通过快速幂的计算方式, 采用从高位到低位遍历乘数的方式进行幂运算。蒙哥马利模逆算法采用了文献[19]的方法进行实现, 并在实现时通过寄存器复用减少了寄存器的使用, 从而提升了程序的 GPU 占用率, 达到提升吞吐的目的。二次剩余算法则需分为两个部分来考虑: 判断与求解。判断部分采用欧拉准则进行实现, 求解部分则采用了 Tonelli-Shanks 算法[20]。由于本文关注的 secp256k1 曲线的基域的阶 $p$ 满足 $p=3(\mathrm{mod}\,4)$, 因此在该基域上可以通过下式计算得到二次剩余的解:

$$R=\pm A^{(p+1)/4}.$$

## 2.2 椭圆曲线运算实现与优化

如上文所述, 为减少并行化中的同步开销, 椭圆曲线群运算的实现采用了单线程的方式。因此本节中的优化主要关注减少寄存器使用、访存优化等线程内优化。对于多个互不依赖的椭圆曲线运算, 可以将任务划分成不相交的任务集, 再通过任务间并行支持并行计算, 提高运行效率。

椭圆曲线运算接口采用 Jacobian 射影坐标系实现, 这可以避免仿射坐标系下计算时的模逆运算, 提高计算的吞吐, 其实现如算法 1 所示。

减少寄存器的使用可以提高程序占用率, 使得更多的线程得以同时运行在 GPU 上以提高计算吞吐。本文修改了教科书算法[21]中原本的计算顺序, 通过中间变量(算法 1 与算法 2 中变量$t_1$到$t_4$)的重用来减少寄存器的使用, 从而提高计算核心的占用率。

全局内存的访存相较于寄存器与常量内存的访问时间开销极大, 若能减少全局访存或能将其替换为更快的存储方式则有利于运算的性能提升。本文合理规划了全局内存的访存, 只在最开始对全局内存中的数据进行一次访问, 减少了全局内存的访存次数。同时计算中所需要的常量(如 2 的逆元)均存储在常量内存中以加速访存。

**算法 1**. Jacobian 射影坐标系点加 CUDA 伪代码

**输入:** 椭圆曲线 Jacobian 坐标点

$P=(X_1,Y_1,Z_1),\quad Q=(X_2,Y_2,Z_2)$

**输出:** 椭圆曲线 Jacobian 坐标点 $R=(X_3,Y_3,Z_3)$

```
1)  __host__ __device__ JPOINT dh_point_add(
    JPOINT P, JPOINT Q) {
2)      BIGINT t1,t2,t3,t4; JPOINT R
3)      t1 ← Z2 × Z2
4)      t2 ← Z1 × Z1
5)      t3 ← t1 × X1
6)      t4 ← t2 × X2
7)      t1 ← t1 × Z2
8)      t2 ← t2 × Z1
9)      t1 ← t1 × Y1
10)     t2 ← t2 × Y2
11)     Y3 ← t3 − t4
12)     t3 ← t3 + t4
13)     t4 ← t1 − t2
14)     t1 ← t1 + t2
15)     Z3 ← Z1 × Z2
16)     Z3 ← Z3 × Y3
17)     X3 ← t4 × t4
18)     t2 ← Y3 × Y3
19)     Y3 ← Y3 × t2
20)     t2 ← t2 × t3
21)     X3 ← X3 − t2
22)     t3 ← X3 + X3
23)     t2 ← t2 − t3
24)     t1 ← t1 × Y3
25)     Y3 ← t2 × t4
26)     Y3 ← Y3 − t1
27)     Y3 ← Y3 × 2^-1
28)     R ← (X3,Y3,Z3)
29)     RETURN R
30) }
```

Jacobian 射影坐标系倍点采用了与点加类似的优化思路, 本文更改了计算顺序以减少寄存器的使用并实现全局内存访存的合理规划。此外倍点运算需要椭圆曲线参数 $a$ 参与运算, 而本文针对的 secp256k1 曲线中参数 $a=0$。以此带入计算可以化简计算流程, 提高计算的吞吐, 如算法 2 所示。

**算法 2.** Jacobian 射影坐标系倍点 CUDA 伪代码

**输入:** 椭圆曲线 Jacobian 坐标点 $P=(X_1,Y_1,Z_1)$

**输出:** 椭圆曲线 Jacobian 坐标点 $R=(X_3,Y_3,Z_3)$

```
1)  __host__ __device__ JPOINT dh_point_double(
    JPOINT P) {
2)      BIGINT t1,t2,t3; JPOINT R
3)      t1 ← Y1 + Y1
4)      Z2 ← t1 × Z1
```

5) $t_2 \leftarrow X_1 \times X_1$
6) $t_3 \leftarrow t_2 + t_2$
7) $t_3 \leftarrow t_3 + t_2$
8) $t_1 \leftarrow t_1 \times Y_1$
9) $t_1 \leftarrow t_1 \times X_1$
10) $t_2 \leftarrow t_2 + t_2$
11) $X_2 \leftarrow t_3 \times t_3$
12) $Y_2 \leftarrow t_2 + t_2$
13) $X_2 \leftarrow X_2 - Y_2$
14) $t_1 \leftarrow t_1 \times t_1$
15) $t_1 \leftarrow t_1 + t_1$
16) $t_2 \leftarrow t_2 - X_2$
17) $Y_2 \leftarrow t_3 - t_2$
18) $Y_2 \leftarrow Y_2 - t_1$
19) $R \leftarrow (X_3, Y_3, Z_3)$
20) RETURN $R$
21) }

椭圆曲线点乘分为基点点乘与任意点点乘, 其区别主要在于基点坐标已知而任意点坐标未知。由于基点坐标已知, 本文在实现中采用了预计算的方式对基点点乘进行优化。通过将预先计算的结果存入常量内存, 在计算基点点乘时从常量内存中读取相应的预计算值完成计算, 可以减少计算中的运算量, 从而提高点乘的吞吐。考虑如下的基点点乘

$$P = kG,$$

其中 $k$ 为长度为 256 位的大整数。为了便于预计算, 我们可以把 $k$ 分解为 $l$ 个长度为 $w$ 的二进制串 $k_i \in [0, 2^w)$, 其中 $i$ 为 $[0, l-1]$ 间的整数, $l$ 和 $w$ 满足 $l \times w = 256$。那么基点点乘就可以分解为

$$P = kG = \sum_{i=0}^{l-1} 2^{iw} k_i G.$$

在预计算中会针对所有的 $i \in [0, l-1]$, $j \in [1, 2^w - 1]$ 计算 $2^{iw} jG$, 并存储到常量内存中。计算基点点乘时则会根据 $k_i$ 的取值取出相应的预计算值, 完成基点点乘的运算。

实现中 $l$ 不同的取值不仅会对计算的迭代次数有影响, 也会影响预计算的点的个数, 进而影响对常量内存的占用。常见 GPU 的常量内存大小为 64KB, 满足 $l = 256, 128, 64$ 时的内存占用, 但是由于本文目的在于实现大数与椭圆曲线基础运算的运算库, 所以考虑到使用者的其他算法优化中也会存在对常量内存的需求, 在实现运算库时我们尽可能少地减少了对常量内存的占用。所以在实现中我们选取了 $l = 256$, $w = 1$, 预计算中共需计算 256 个预计算点, 基点点乘中共需迭代 256 次, 共需占用 16KB 常量内存。$l = 256$, $w = 1$ 对应的实现如算法 3 所示。

**算法 3.** Jacobian 射影坐标系基点点乘 CUDA 伪代码

**输入:** 椭圆曲线基点 $G$, 大整数 $k = \sum_{i=0}^{l-1} 2^i k_i$
**输出:** 椭圆曲线 Jacobian 坐标点 $R = kG$
1) 预先计算 $2^i G$ $(0 \leqslant i < l)$ 并存储在常量内存中
2) __host__ __device__ JPOINT d_base_point_mul( BIGINT $k$ ) {
3) JPOINT $R$
4) $R \leftarrow \infty$
5) FOR ( $j \leftarrow l-1$ TO 0 ) {
6) IF ( $k_j = 1$ ) {
7) $R \leftarrow$ dh_point_add($R, 2^j G$)
8) }
9) }
10) RETURN $R$
11) }

任意点点乘中的椭圆曲线点是未知的, 因此无法通过预计算的方式进行优化, 本文采用快速幂的方式进行实现, 相比于算法 3 在每一轮迭代中需要多进行一次倍点操作。具体算法如算法 4 所示。

**算法 4.** Jacobian 射影坐标系任意点点乘 CUDA 伪代码

**输入:** 椭圆曲线点 $P$, 大整数 $k = \sum_{i=0}^{l-1} 2^i k_i$
**输出:** 椭圆曲线 Jacobian 坐标点 $R = kP$
1) __device__ JPOINT dh_point_mul( JPOINT $P$, BIGINT $k$ ) {
2) JPOINT $R$
3) $R \leftarrow \infty$
4) FOR ( $j \leftarrow l-1$ TO 0 ) {
5) $R \leftarrow$ dh_point_double($R$)
6) IF ( $k_j = 1$ ) {
7) $R \leftarrow$ dh_point_add($R, P$)
8) }
9) }
10) RETURN $R$
11) }

由于任意点点乘使用频繁, 本文针对输入数据与输出数据的位置不同实现了 in-place 与 out-of-place 版本的点乘接口, 用户可以根据不同应用场景选择对应接口, 也可调用二合一接口函数通过判断输入输出数据位置自动选择合适的实现。

总的来说, 由于需要为 secp256k1 基点乘法运算存储预计算表, 同时还需要对 secp256k1 的基域和标量域以及椭圆曲线运算存储大整数常量, 运算库总共需要占用 16.25KB 的常量内存。这一空间占用是恒定的, 共占据了 GPU 常量内存的 25.4%, 仍留有 47.75KB 的常量内存, 以便于上层算法实现时优化使用。

## 3 上层相关密码学算法实现与优化

本文实现的运算库提供了方便的接口供使用者调用, 本节利用运算库实现并优化了包括代理重加密以及 Bulletproofs 范围证明验证算法①在内的上层密码学算法, 以检验运算库在实际应用场景下的实用性与性能表现。

### 3.1 代理重加密实现与优化

随着对用户隐私的重视不断加强, 用户存放在区块链上的数字资产等数据往往是以密文的形式存储的, 除了被授权服务提供商, 第三方无法获取数据的真实信息。而当用户在提供商间进行数据移动和共享数字资产时则需要在多个提供商间进行数据共享, 但存储密文的方式却给数字资产的共享带来了障碍。由于数据以密文存储, 在没有密钥的情况下, 数据接收方和中介第三方均无法对密文进行解密和分享。而如果数据分享方单独用接收方的公钥进行加密则会失去通用性, 即若想对另一个接收方分享数据则需要重新加密与上传, 带来了不必要的开销, 这对数据分享方造成了极大的不便, 而代理重加密技术[11]则通过允许第三方在不知道数据明文和密钥的情况下, 将原有密文转换为接收方可以解密的密文, 从而解决了这一难题。

由于许多代理重加密方案[22-23]使用的双线性对 (bilinear map) 与格密码学 (lattice) 计算超出了本文运算库的范围, 本文实现的代理重加密主要基于文献[12]中的方案。代理重加密中主要有三个参与方: 数据分享方、数据接收方与代理第三方。其中数据分享方掌握数据的所有权, 数据接收方被授权获取分享方的数据, 代理第三方则提供数据存储与计算的服务, 但不被授权获知具体的数据内容, 只会存储和处理数据的密文。在区块链场景中, 分享方是用户当前的数字资产服务提供商, 而需要共享的数据即为用户希望转移的数字资产, 接受方受到用户的授权获取转移的数字资产, 而代理第三方则可以由任意一个第三方服务提供商扮演, 负责转换存储在区块链上的密态数字资产。

为了实现将密文转换为接收方可以处理的密文, 重加密过程中涉及了 ECC 加密、重加密和 ECC 解密三个操作。下面假设 $G$ 为已知的椭圆曲线基点, 分享方与接收方各自拥有公私钥对 $(a,aG)$ 和 $(b,bG)$, 且 $b$ 对分享方已知。图 2 展示了代理重加密各个操作中各方之间的交互流程, 各操作具体细节如下:

(1) ECC 加密: 分享方将明文数据 $msg$ 映射为 $I$ 个椭圆曲线点 $C_i,(i\in[0,I-1])$, 并随机选取临时密钥 $k$, 同时利用自己的公钥计算得到 $C_k=kaG$, 利用私钥计算得到 $C_{Ai}=aC_i+C_k,(i\in[0,I-1])$, 并将 $C_k$ 与 $C_{Ai}$, $(i\in[0,I-1])$ 发送给第三方。

(2) 重加密: 分享方计算 $r_{AB}=a^{-1}b$, 并将 $r_{AB}$ 发送给第三方。第三方通过计算 $C_{Bi}=r_{AB}C_{Ai}$, $(i\in[0,I-1])$ 与 $C_k'=r_{AB}C_k$, 将密文重加密为接收方可以解密的密文 $C_{Bi},(i\in[0,I-1])$ 与 $C_k'$。

(3) ECC 解密: 接收方从第三方获取 $C_k'$ 和 $C_{Bi},(i\in[0,I-1])$, 计算 $kG=b^{-1}C_k'$, 进而计算得到 $C_i=C_{Bi}b^{-1}-kG,(i\in[0,I-1])$。通过将这 $I$ 个椭圆曲线点映射回明文得到最终解密的结果 $msg$。

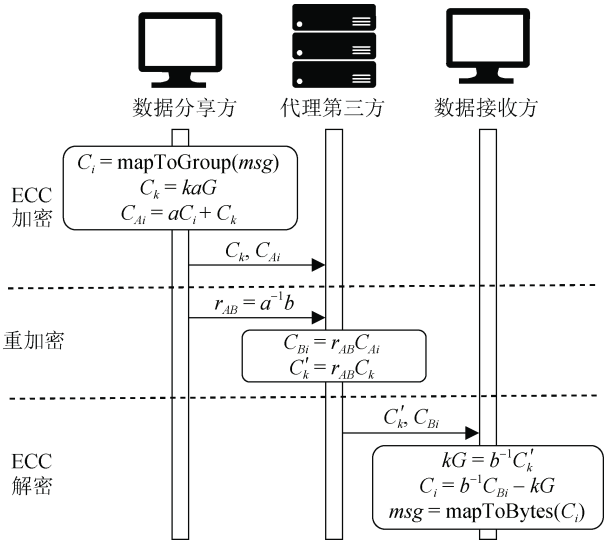



**图 2 代理重加密操作流程**

**Figure 2 Processes of operations in proxy re-encryption**

① 具体的实现代码参见 https://gitee.com/z_jn/gpuec。

仔细考察代理重加密中的三个操作, 可以看到在各操作中对每个分组的处理基本上是相互独立的, 这意味着算法具有非常良好的并行性。因此在实现中我们主要采用了每个分组由一个线程单独负责的优化思路, 这样的任务分配策略可以使得不同线程之间的通信和同步开销最小, 让每个线程的空闲时间尽量少, 从而实现最大程度的“满负荷工作”, 以达到最高的吞吐。为了充分利用 GPU 的计算能力, 每个block的线程数目都根据占用率最大原则进行计算, 通过每个线程的共享内存以及寄存器使用情况计算得到能使得占用率指标最大的线程数目作为 block 的大小。

注意到在 ECC 加密操作中, 涉及 $C_k$ 的运算对于每个分组都是相同的, 因此 $C_k$ 只需计算一次。在实现中, 我们将其放在 CPU 端进行计算, 与 GPU 端异步计算, 从而掩盖计算的时间, 提高吞吐。ECC 解密操作也是类似的, 可以将 $kG$ 的计算分配到CPU端进行异步计算, 以提升算法吞吐。

重加密操作主要由代理第三方负责计算。代理第三方除了要面临单个重加密庞大的分组数带来的计算规模, 也常常会同时接受多个重加密请求。这使得重加密运算的吞吐成为了整个代理重加密流程中的瓶颈。为了解决这一问题, 可以进一步采用单个线程负责多个明文分组的任务分配策略, 以避免线程上下文切换带来的开销, 从而最大化地提升算法的吞吐。此外, 在 GPU 的实现过程中, 我们使用寄存器与共享内存来保存点乘与大数运算后的中间变量, 以减少全局内存的访存次数。同时由于不同线程间互不依赖, 读写共享内存时无需进行额外的线程间同步, 也免去了相应的同步开销。

### 3.2 Bulletproofs 范围证明验证实现与优化

Bulletproofs 可以以对数级别的证明大小实现范围关系的零知识证明, 因此在机密交易和算数电路的隐私计算等领域取得了广泛的应用。虽然 Bulletproofs 在证明大小上有着明显的优势, 但是其引入的内积证明 (Inner product proof) 具有较为耗时的多轮迭代, 复杂度较高, 成为了计算中的瓶颈。随着数据规模的日益增大, 已经无法满足日益增长的计算需求。因此 Bulletproofs 算法的并行加速实现具有重要的应用价值。我们在实现中使用了本文实现的运算库达到了便捷实现的效果, 同时利用了算法内在并行性以最大限度地提升计算性能。

Bulletproofs 范围证明算法具体指的是如下的零知识证明算法: 对于秘密值 $v$, 给定基点 $g$, $h$, $u$ 与 $g_i$ 和 $h_i$, $(i\in[0,n-1])$, 以及秘密值的 Pederson 承诺 $V$, Bulletproofs 范围证明算法可以证明秘密值落在 $[0,2^n-1]$ 的范围内, 同时不泄露 $v$ 具体的值。在 Bulletproofs 范围证明中有证明者和验证者两方。证明者通过证明算法生成一个证明 $\pi$, 并将 $\pi$ 发送给验证者。验证者再通过验证算法验证 $\pi$, 若验证通过, 则说明 $v$ 确实位于 $[0,2^n-1]$ 的范围中, 否则认为 $v\in[0,2^n-1]$ 的关系不成立。

由于本文主要关注验证算法的实现与优化, 我们不深入证明算法的细节, 具体可以参考文献[13]。重要的是, 证明算法会生成形如下式的证明 $\pi$,

$$\pi=(A,S,T_1,T_2,\tau_x,\mu,\hat{t},a,b,L_0,\cdots,L_{\log n-1},R_0,\cdots,R_{\log n-1}),$$

其中 $\tau_x,\mu,\hat{t}$ 均为大整数, 其余元素均为椭圆曲线上的点。验证者收到证明 $\pi$ 后, 就可以利用证明以及给定的基点与承诺进行验证了。由于 Bulletproofs 范围证明验证算法只基于基本的椭圆曲线和大整数运算, 本文实现与 Bünz 等人[13]的协议基本一致, 具体的验证算法如算法 5 所示。为便于表述, 下面记 $\langle \boldsymbol{a},\boldsymbol{b}\rangle$ 为向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 的内积, $\boldsymbol{x}^n$ 为向量 $(1,x,x^2,\ldots,x^{n-1})$。

**算法 5.** Bulletproofs 范围证明验证算法

**输入:** Bulletproofs 范围证明 $\pi$

**输出:** 接受或拒绝证明

1) $x,y,z\leftarrow \text{get\_challenge}(A,S,T_1,T_2)$
2) IF $\hat{t}g+\tau_x h\neq z^2V+\delta(y,z)g+xT_1+x^2T_2$ THEN
3) RETURN 拒绝
4) END IF
5) $g_{(0),i}\leftarrow g_i,\ h_{(0),i}\leftarrow h_i,\ h_i'\leftarrow y^{-i+1}h_i,\ \forall i\in[0,n-1]$
6) $P_0\leftarrow A+xS-z\sum_{i=0}^{n-1}g_i+\sum_{i=0}^{n-1}(zy^i+2^iz^2)h_i'$
7) $j\leftarrow 0, n_j\leftarrow n$
8) WHILE $n_j>1$ DO
9) $n_{j+1}\leftarrow n_j/2$
10) $x_j\leftarrow \text{get\_challenge}(L_j,R_j)$
11) $g_{(j+1),i}\leftarrow x^{-1}g_{(j),i}+xg_{(j),i+n_{j+1}},\forall i\in[0,n_{j+1}-1]$
12) $h_{(j+1),i}\leftarrow xh_{(j),i}+x^{-1}h_{(j),i+n_{j+1}},\forall i\in[0,n_{j+1}-1]$
13) $P_{j+1}\leftarrow x^2L_j+P_j+x^{-2}R_j$
14) $j\leftarrow j+1$
15) END WHILE
16) IF $P_{\log n-1}\neq ag_{(\log n-1),0}+bh_{(\log n-1),0}+abu$ THEN
17) RETURN 拒绝

18) END IF
19) RETURN 接受

算法 5 中，$\delta(y,z)=(z-z^2)\left\langle \mathbf{1}^n,\boldsymbol{y}^n\right\rangle - z^3\left\langle \mathbf{1}^n,\mathbf{2}^n\right\rangle$，而 get_challenge 函数计算 Fiat-Shamir 启发式[16]中的挑战值。可以看到，算法 5 主要验证了两个关系，其中行 1～4 和行 5～18 验证的条件分别保证了证明 $\pi$ 满足 Bulletproofs 范围证明的承诺关系和内积关系。我们下面分别考虑这两部分的实现以及基于 GPU 的优化方式。

### 3.2.1 承诺关系验证优化

Bulletproofs 验证算法内部有众多向量操作，有着较好的并行潜力。如承诺关系验证中的式(1)，

$$\hat{t}g+\tau_x h=z^2V+\delta(y,z)g+xT_1+x^2T_2, \qquad (1)$$

等式右侧的众多点乘点加操作看似互不相同，实际上其计算模式是类似的。我们在实现中将各个数据存储在相邻的共享内存中，再用相应的线程对其进行点乘操作，最后进行树状归约。计算过程如图 3 所示，其中灰色方块代表椭圆曲线点，白色方块代表大整数，箭头表示计算过程中的数据流动方向。整个过程均在共享内存中完成，以减少对全局内存的访问，提升算法吞吐。

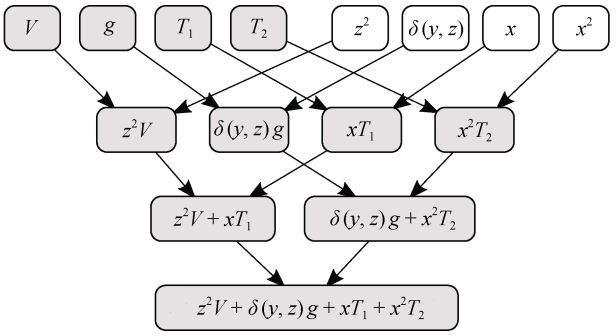



**图 3 Bulletproofs 承诺关系验证优化**

**Figure 3 Optimization of the verification of commitment relation in Bulletproofs**

不过实际上并不是所有的向量操作都必须使用多线程并行计算，如 $\delta(y,z)$ 中的 $\left\langle \mathbf{1}^n,\boldsymbol{y}^n\right\rangle$ 和 $\left\langle \mathbf{1}^n,\mathbf{2}^n\right\rangle$ 等向量内积就可以通过等比数列求和公式直接计算得出，以减少计算开销。

### 3.2.2 内积关系验证优化

内积关系的验证操作是整个验证算法中最耗时的部分，也是整个计算的瓶颈。本节深入分析内积证明验证的数据依赖性，并对内积证明验证部分进行深入优化。

内积关系验证过程中会进行“折半”操作，也即每一轮循环中将内积向量长度 $n$ 减半。经过 $\log n$ 次循环后，最后计算得到点 $g_{(\log n-1),0}, h_{(\log n-1),0}$。这一过程中，每轮循环间的数据依赖性较强，并行难度较大。

以 $n$=8, $\log n=3$ 时为例，经过细致的并行性与数据依赖性的分析，可以发现在 3 轮循环后有

$$\begin{aligned} g_{(2),0}=&(x_0^{-1}x_1^{-1}x_2^{-1})g_0+(x_0^{-1}x_1^{-1}x_2)g_1\\ &+(x_0^{-1}x_1x_2^{-1})g_2+(x_0^{-1}x_1x_2)g_3+(x_0x_1^{-1}x_2^{-1})g_4\\ &+(x_0x_1^{-1}x_2)g_5+(x_0x_1x_2^{-1})g_6+(x_0x_1x_2)g_7. \end{aligned} \qquad (2)$$

可以看到，$g_0$ 的索引值为 0，其二进制表示为 000，相应的其指数部分为 $x_0^{-1}x_1^{-1}x_2^{-1}$，$g_1$ 的索引值为 1，其二进制表示为 001，相应的其指数部分为 $x_0^{-1}x_1^{-1}x_2$，所以令二进制 0 代表 $x_i^{-1}$，1 代表 $x_i$，便可以通过判断其索引的二进制表示位是 0 或 1 取相应的值进行计算，如图 4 所示。

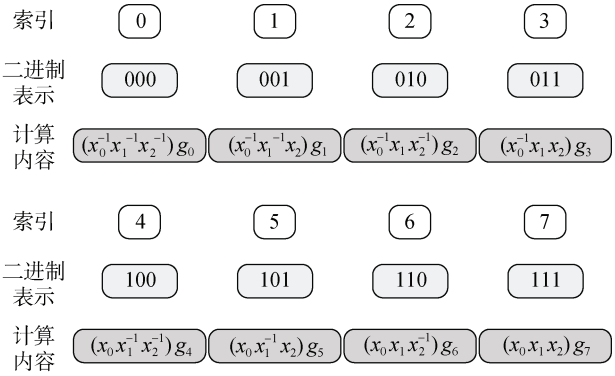



**图 4 内积关系验证计算内容示意图**

**Figure 4 An illustration of computations in a verification of inner product relation**

利用这一规律，在 $n=8$，$\log n=3$ 的情况中，可以通过 8 个线程分别进行 $g_0$ 至 $g_7$ 的相关运算，再进行树状归约，如图 5 所示。由于 $x_0$ 至 $x_2$ 的相关量计算互不依赖，因此也可以通过多线程计算得出，并将结果存储在共享内存中。最后，通过多线程计算并归约，就得到最终的归约值结果 $g_{(2),0}$。

$h_{(\log n-1),0}$ 的结构与 $g_{(\log n-1),0}$ 相似，类似于式(2)，$n=8$ 时具体的计算公式为

$$\begin{aligned} h_{(2),0}=&(x_0x_1x_2)h_0+(x_0x_1x_2^{-1})h_1+(x_0x_1^{-1}x_2)h_2\\ &+(x_0x_1^{-1}x_2^{-1})h_3+(x_0^{-1}x_1x_2)h_4+(x_0^{-1}x_1x_2^{-1})h_5\\ &+(x_0^{-1}x_1^{-1}x_2)g_6+(x_0^{-1}x_1^{-1}x_2^{-1})g_7, \end{aligned}$$

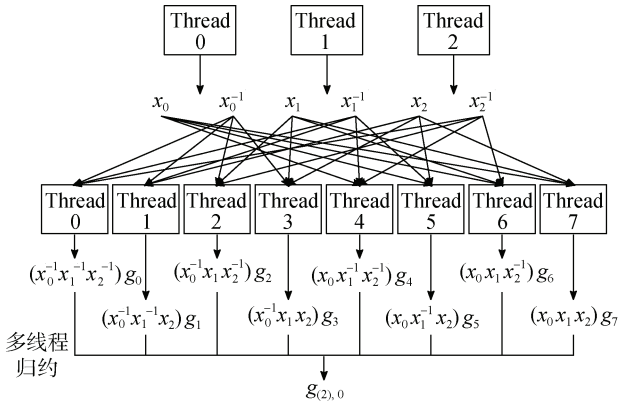



**图 5　内积验证任务分配示意图**

**Figure 5　An illustration of the task assignment in an inner product verification**

因此可以使用类似于 $g_{(2),0}$ 的计算方式分配 8 个线程进行计算。只需令二进制位 1 代表 $x_i^{-1}$, 0 代表 $x_i$ 即可。

由此计算得到 $g_{(\log n-1),0}$ 与 $h_{(\log n-1),0}$ 之后, 再验证 $P_{\log n-1}$ 与 $ag_{(\log n-1),0}+bh_{(\log n-1),0}+abu$ 是否相等就完成了内积关系的验证。注意这里的计算模式实际上与承诺关系验证非常相似, 因此也可以利用 3.2.1 一节中的方法进行优化。

## 4　实验结果与分析

本节将通过实验验证 GPU 运算库的性能表现。实验环境的各个参数如表 2 所示。

**表 2　实验环境配置**

**Table 2　Configuration of the experiment environment**

| 项目 | 值 |
|---|---|
| CPU | Intel Core i7-3770 |
| 主存 | 4×8 GB DDR3 |
| 操作系统 | CentOS release 6.4 |
| GPU | NVIDIA TITAN V |
| GMP | V6.2.0 |
| GmSSL | V2.5.4 |
| OpenSSL | V1.1.0d |
| CUDA | V10.0.130 |
| GPU 代码编译参数 | -arch=sm_70 -std=c++11 -rdc=true |

在实验中, 我们使用了 C++ 标准库中的 independent_bits_engine 随机数生成器生成实验中所需的随机数, 通过随机生成 $x$ 坐标并进行二次剩余求解来选取实验中所需的椭圆曲线点。

利用表 2 所述的实验环境, 我们对基于运算库实现的大数运算操作, 椭圆曲线操作、代理重加密与范围证明验证算法的性能进行了测试与分析。

实验中我们主要关注以下几个指标:

(1) **吞吐**(Throughput): 吞吐指的是单位时间内能完成的计算量, 反映了系统处理数据的能力。

(2) **延迟**(Latency): 延迟为完成工作所需的耗时, 即从开始计算到返回结果这一过程中, 所有阶段处理时间的总和。

(3) **加速比**(Speedup): 加速比为针对同一任务的另一算法的耗时与本算法耗时之比, 主要反映了不同算法间相对速度的定量关系。

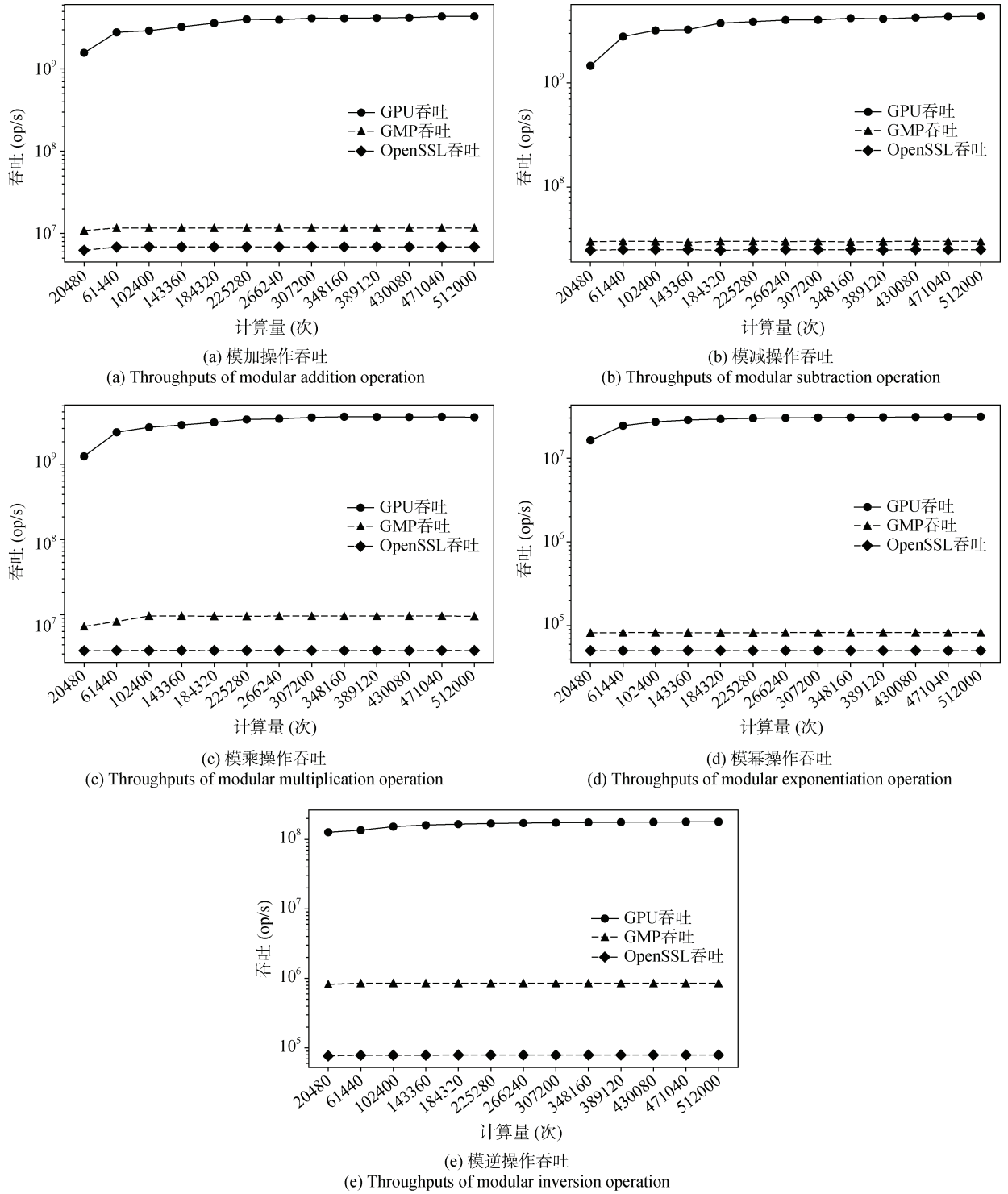
本节主要以当前取得广泛应用的高性能 CPU 端计算库作为性能基准。通过实验测试本文运算库与相关密码学算法的吞吐与延迟等指标, 再将其性能表现与 CPU 库以及一些代表性的 GPU 实现进行比较与分析。由于目前各类 GPU 加速工作缺乏相关的公开开源实现, 下文中的 GPU 相关工作的比较数据均引用自各工作的相应论文。

### 4.1　大数运算性能测试

对于大数运算实现, 我们通过实验比较了本文实现与 CPU 的 GMP 和 OpenSSL 库性能表现。由于 GPU 上的寄存器数量等资源是有限的, 要使得运算中尽量多的线程处于活跃状态, 就需要在实验中保证寄存器的占有率最大化。这意味着我们需要合理选取各个实验中 GPU 核函数的块内线程数量, 即 BlockSize 参数。通过 cuobjdump 命令分析编译生成的 CUDA 核函数, 可以统计得到在运算库中, 单次大数模加、模减、模乘、模幂和模逆运算分别需要 44、38、78、126、94 个寄存器。我们经过 CUDA 占用率的计算[24], 得到 5 组实验中分别选取 640、768、768、512 和 640 作为 BlockSize 时能达到最高的占用率。以此作为参数实验最终得到的延迟与吞吐表现如图 6 所示。

随着计算量的增大, GPU 运算库的并行优势逐渐显现, 吞吐逐渐增大并最终趋于平稳, 最高能在模加、模减、模乘、模幂和模逆操作中分别达到 $4.38\times10^9$、$4.376\times10^9$、$4.30\times10^9$、$3.12\times10^7$ 与 $1.79\times10^8$ 次/秒的吞吐。相比于 GMP 库与 OpenSSL 库加速效果显著, 例如对模加运算, 最高能达到 374 与 638 倍左右的加速比。

由于 GPU 端吞吐最后趋于稳定, 在实际调用中可以减少单次计算量来达到吞吐与延迟的平衡。以模减为例, 当计算量为 266240 时, 吞吐有着较高的 $4.03\times10^9$ 次/s, 而通过 延时=计算量/吞吐 的关系




(a) 模加操作吞吐
(a) Throughputs of modular addition operation

(b) 模减操作吞吐
(b) Throughputs of modular subtraction operation

(c) 模乘操作吞吐
(c) Throughputs of modular multiplication operation

(d) 模幂操作吞吐
(d) Throughputs of modular exponentiation operation

(e) 模逆操作吞吐
(e) Throughputs of modular inversion operation

**图 6　GPU 与 CPU 大整数运算实现在不同计算量下的吞吐**

**Figure 6　Throughputs of big integer operations on GPU and CPU under different amounts of computation**

可以计算得到, 此时延迟只有 66 μs, 只有最高吞吐($4.38\times10^9$ 次/s)时延迟 117 μs 的一半左右。

表 3 展示了运算库大数运算与已有的 GPU 实现间的吞吐差异。由于不同工作的实验平台与优化场景不同, 比较时需要考虑具体的大整数位数以及 GPU 运算能力等因素。为此表 3 在展示了各工作实测性能数据的同时, 还列举了各工作 GPU 理论浮点计算能力以及缩放后的吞吐数据, 以公正比较相互之间的性能差异。具体的缩放方法为各工作吞吐数据除以各自使用的 GPU 的每秒浮点运算次数(FLoating-point Operations Per Second, FLOPS), 再乘以本文实验使用的 NVIDIA TITAN V 的 FLOPS, 以

**表 3 运算库与 GPU 大整数运算实现吞吐比较 (Kop/s)**

**Table 3 Comparison of big integer operations throughput between the library and GPU implementations (Kop/s)**

| 实现 | GPU | 理论计算性能 | 大整数比特数 | 模乘 | 模乘 (缩放后) | 模幂 | 模幂 (缩放后) |
|---|---|---|---|---|---|---|---|
| **本文** | TITAN V | 14.90 TFLOPS | 256 | **4298273** | **4298273** | **31159.71** | **31159.71** |
| Szerwinski[2] | 8800 GTS | 228.1 GFLOPS | 1024 | — | — | 0.813 | 849.71 |
| Harrison[3] | 8800 GTX | 345.6 GFLOPS | 1024 | — | — | 5.537 | 3681.54 |
| Zheng[4] | GTX TITAN | 4.709 TFLOPS | 1024 | 110500 | 5594224 | — | — |

估算相关工作若使用本文中的 TITAN V 后的理论性能开销。其中, FLOPS 数据由 TechPowerUp GPU 规格数据库[25]中的数据计算得到, 计算公式为

$$\begin{aligned}&\text{FLOPS}\\&=\text{GPU动态超频频率}\times\text{GPU流处理器总数}\\&\quad\times\text{GPU单个流处理器核心数量}\\&\quad\times\text{每周期单个GPU核心32位浮点操作计算数量.}\end{aligned}$$

这一数据体现了 GPU 的硬件计算能力, 衡量了 GPU 的计算吞吐上限, 以此来估算其他工作在更换 本文使用的 GPU 后的计算性能是比较合理的。此外, 对于不同工作的不同位数大数模乘性能的比较, 考虑乘法操作的复杂度, 我们会额外乘以位数比的平方来进行缩放, 以保证比较的公正。对于 Szerwinski 等人[2]和 Pan 等人[7]的工作而言, 这意味着我们需要在 FLOPS 缩放后再额外乘以 $(1024/256)^2=16$ 的缩放系数以排除大整数位数对性能的影响。

可以看到, 本文运算库相比文献[2]与[3]在考虑了硬件与运算位数后仍有一个数量级以上的吞吐优势。而与文献[4]相比则略显不足, 吞吐有 23.2% 的下降。但文献[4]中的方案实际上引入了浮点操作以及相当细致的跨线程并行化计算来完成运算。这在带来性能提升的同时也使得实现的复用性不佳。因此相比之下, 本文的运算库较为恰当地平衡了易用性与计算吞吐。

### 4.2 椭圆曲线运算性能测试

对于椭圆曲线运算实现, 我们通过实验比较了本文实现与 OpenSSL 库在点加、倍点、基点点乘与任意点点乘上的性能表现。类似于大数运算实验, 我们通过分析编译得到的 CUDA 核函数得知, 点加、倍点、基点与任意点点乘在单次运算中, 各需 214、148、188、254 个寄存器。因此我们通过占用率计算得到 4 组实验各自选取 BlockSize 为 256、384、256、256 时, 能使得占用率最大。最后的实验结果如图 7 所示。

从图 7 (a) 与 (b) 可以看到, 本文的 GPU 点加与倍点实现吞吐可以分别最高达到 $2.51\times10^8$ 与 $1.18\times10^9$ 次/s。从吞吐与延迟上看, CPU 上的 OpenSSL 库与 GPU 实现相比在点加操作上存在着两个数量级的差距。而由于 GPU 实现对 secp256k1 曲线在倍点上进行了针对性优化, 相对 OpenSSL 可以达到三个数量级左右的差距。

基点与任意点点乘的性能如图 7 (c) 与 (d) 所示。由于通过预计算方法优化, 基点点乘的吞吐相比任意点点乘有 3~4 倍左右的性能提升。基点与任意点点乘最高可分别达到 6.99×106 与 1.70×106 次/秒的吞吐, 相比 OpenSSL 各达到了 784 与 522 倍的加速比。实际应用中可以进一步地根据实际需求调整预计算中的参数 $w,l$, 以获得常量内存占用与计算吞吐的平衡。

表 4 展示了运算库椭圆曲线运算与已有的 GPU 实现间的吞吐差异。同样地, 表 4 也列出了各实现 GPU 的理论性能以及缩放后的吞吐。其中缩放后的吞吐数据使用与 4.2 节中相同的方法计算得到。由于表中各工作使用的椭圆曲线的基域元素大小均为 256 位, 因此无需针对元素位数进行缩放计算。为了更确切地比较运算库的性能表现, 表 4 中的实现均为对 Weierstrass 格式曲线的 GPU 实现。可以看到, 本运算库的吞吐是 Szerwinski 实现的近 5 倍。而与 Pan 等人[7]的工作相比, 由于 Pan 等人针对签名场景进行了十分深入的优化加速, 因此在考虑硬件差异后, 本文实现在基点点乘上的吞吐仅为 Pan 等人实现的 28.5%, 而在任意点点乘上的吞吐也仅为 Pan 等人的实现的 65.7%。但 Pan 等人的实现难以直接应用在椭圆曲线签名与验证以外的密码学算法加速上。相比之下, 本文算法库可复用性更高, 可以达到更强的泛用性。

### 4.3 代理重加密性能测试

对于代理重加密算法, 我们测试并比较了 GPU 与 CPU 实现在 ECC 加密、重加密与 ECC 解密运算上的性能差异。实验中, 我们取 GPU 的 BlockSize 为 256, 采用了 128KB、1MB 与 10MB 三组具有代表性的金融支付数据规模进行了实验。结果如表 5 所示。

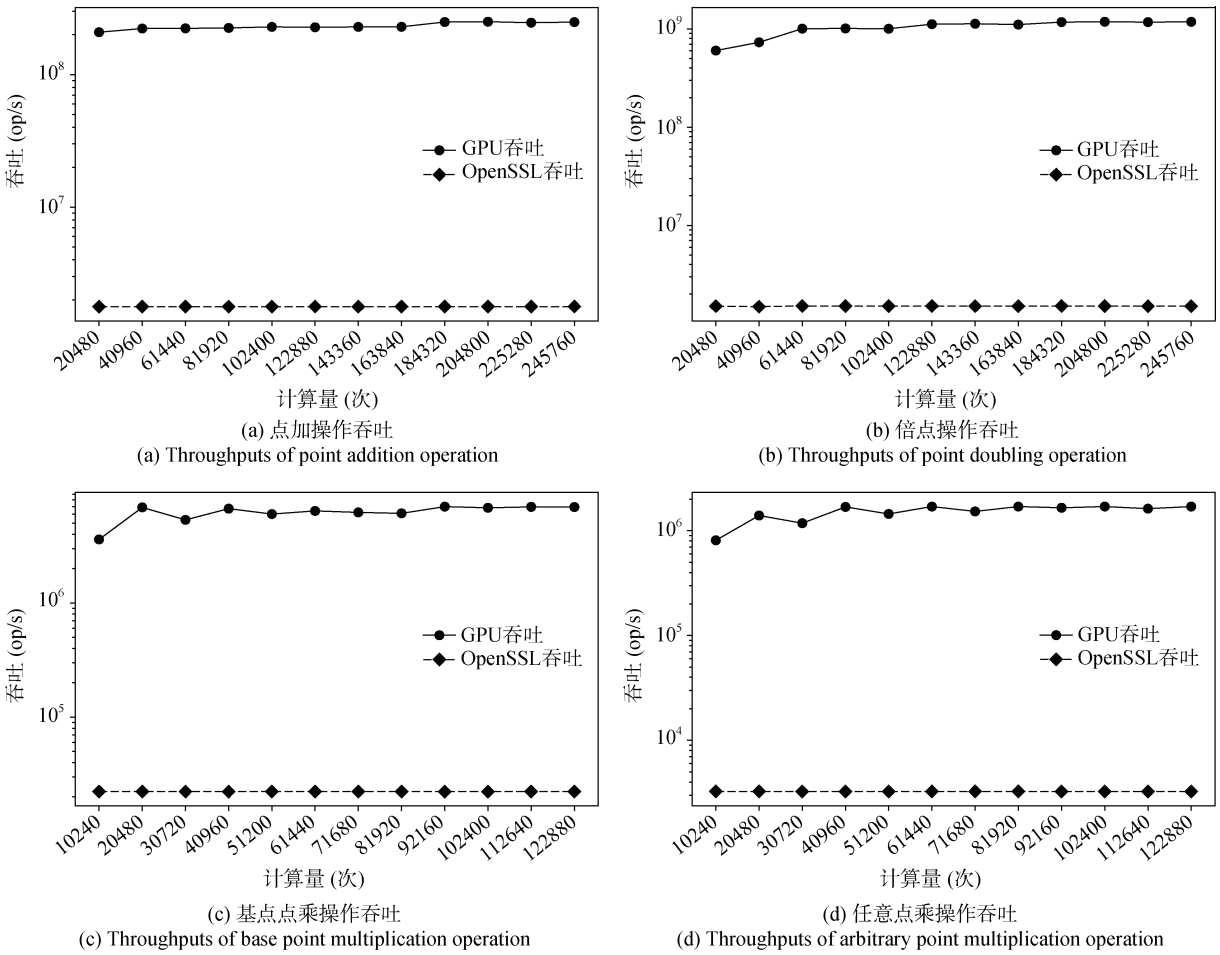



**图 7　GPU 与 CPU 椭圆曲线群运算实现在不同计算量下的吞吐**

**Figure 7　Throughputs of elliptic curve operations on GPU and CPU under different amounts of computation**

**表 4　运算库与 GPU 椭圆曲线运算实现吞吐比较 (Kop/s)**

**Table 4　Comparison of elliptic curve operations throughput between the library and GPU implementations (Kop/s)**

| 实现 | GPU | 理论计算性能 | 曲线 | 任意点点乘 | 任意点点乘(缩放后) | 基点点乘 | 基点点乘(缩放后) |
|---|---|---|---|---|---|---|---|
| 本文 | TITAN V | 14.90 TFLOPS | secp256k1 | **1702.01** | **1702.01** | **6991.88** | **6991.88** |
| Szerwinski[2] | 8800GTS | 228.1 GFLOPS | NIST P-224 | 1.412 | 348.62 | — | — |
| Pan[7] | GTX 780 Ti | 5.345 TFLOPS | NIST P-256 | 929 | 2589.73 | 8710 | 24280.44 |

**表 5　CPU 与 GPU 代理重加密实现不同模块延迟**

**Table 5　Latencies of different modules in CPU-based and GPU-based proxy re-encryption**

| 数据量 | | 128KB | 1MB | 10MB |
|---|---|---|---|---|
| ECC 加密 | GPU | 37.538ms | 77.827ms | 566.385ms |
| | CPU | 685.172ms | 4488.13ms | 43038.2ms |
| | 加速比 | 18.2528 | 57.668 | 75.9876 |
| 重加密 | GPU | 18.136ms | 33.387ms | 257.353ms |
| | CPU | 444.378ms | 3391.7ms | 33980.5ms |
| | 加速比 | 24.5025 | 101.587 | 132.039 |
| ECC 解密 | GPU | 18.851ms | 31.686ms | 287.619ms |
| | CPU | 545.68ms | 4119.43ms | 41643.7ms |
| | 加速比 | 28.947 | 130.008 | 144.788 |

由表 5 可以看到, 在数据规模较大时(10MB), GPU 实现的性能优势开始显现。此时 ECC 加密、重加密与 ECC 解密三个模块分别相比于 CPU 算法能达到 76、132、145 倍左右的加速比。

为了进一步验证 GPU 实现的吞吐表现, 我们在多个数据规模下测量了 GPU 代理重加密实现的吞吐, 结果如图 8 所示。

可以看到, 随着数据规模的增大, 代理重加密各个模块的吞吐先增加, 之后达到饱和。重加密与 ECC 解密操作最高能分别达到 38.9 GB/s 与 37.8 GB/s 左右的吞吐。而 ECC 加密操作由于需要首先将明文映射到椭圆曲线群上, 因此最高能仅能

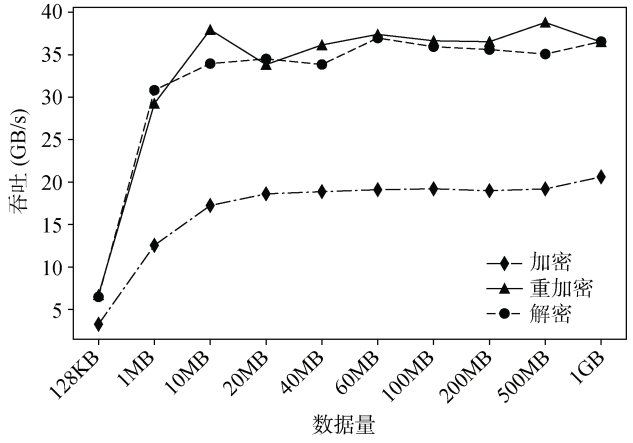



**图 8　基于 GPU 的代理重加密实现不同模块吞吐**

**Figure 8　Throughputs of different modules in GPU-based proxy re-encryption**

达到 20.6 GB/s 的数据吞吐。若略去 ECC 加密模块中 mapToGroup 的映射步骤, 只测量其余计算的耗时, ECC 加密操作的极限吞吐实际上也能达到与重加密与 ECC 解密相当的吞吐。

### 4.4　Bulletproofs 范围证明验证性能测试

对于 Bulletproofs 范围证明验证, 我们测试并比较了 GPU 实现与 CPU 实现[26]的聚合证明验证与非聚合证明验证的性能表现, 结果如图 9 所示。

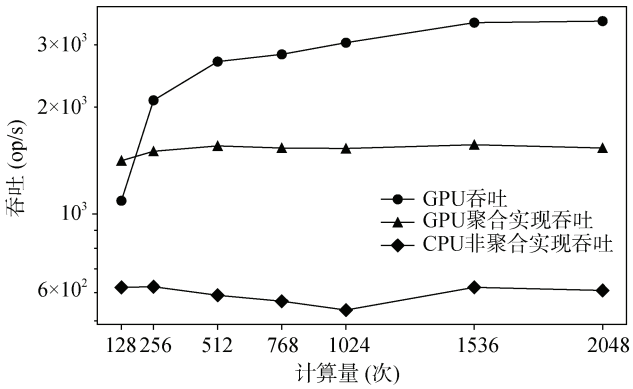



**图 9　基于 GPU 与 CPU 的 Bulletproofs 范围证明验证算法吞吐**

**Figure 9　Throughputs of GPU-based and CPU-based Bulletproofs range proof verification**

通过图 9 可以发现, 在计算量较小时, 由于 GPU 没有被充分利用, GPU 的运行时间大于 CPU 版本, 但是随着数据量的增加, GPU 版本的性能逐渐高于 CPU 版本, 随着计算量的增加优势逐渐明显。这表明利用本文 GPU 库实现的算法有着更好的可扩展性。相比非聚合与聚合版本的 CPU 实现最高可以达到 5.74 与 2.27 倍左右的加速比。实验中平均证明验证时间最高为 917.789 μs, 始终在 1 ms 内。当证明数量为 256 及以上时, 我们基于 GPU 的 Bulletproofs 实现已明显优于 CPU 实现, 能达到 0.5 ms 的平均验证时间, 因此可以支持数字货币隐私保护场景下超过每秒 2000 笔交易的性能需求。

## 5　总结

本文针对区块链场景对密码学算法的需求, 分析、设计并实现了针对大数与椭圆曲线运算的 GPU 运算库。实现中充分利用了 GPU 的优势, 采用了预计算等一系列方式优化算法, 并通过合理分配寄存器、常量内存等快速存储最大化了访存性能。为验证运算库的实用性, 我们还进一步基于 GPU 运算库实现了代理重加密与 Bulletproofs 范围证明验证算法, 并利用算法内部的并行性进行了优化。

为验证本文实现的 GPU 库及相关算法的加速性能, 我们进一步测试了上述功能的延迟与吞吐表现, 并与常见的 CPU 与 GPU 实现进行性能比较。实验表明, 本文实现的 GPU 运算库在大数运算以及椭圆曲线的相关操作上的吞吐与延迟均优于常见的 CPU 库, 相比其他 GPU 实现也具有一定的性能优势, 以此构建的基于 GPU 的代理重加密算法与 Bulletproofs 范围证明验证算法均取得了良好的加速效果。其中 Bulletproofs 验证算法可以满足数字货币场景下超过每秒 2000 笔交易的性能需求。可见运算库能为区块链隐私保护等对密码学计算具有高吞吐需求的场景提供坚实的支持。

目前运算库主要针对 secp256k1 曲线进行优化, 这未必适用于所有场景, 因此有必要考虑对不同元素大小与参数的椭圆曲线提供支持。此外本文运算库主要为最优化吞吐而进行设计, 对延迟敏感的场景的优化也是值得考虑的后续研究方向之一。目前包括代理重加密[11]在内的大量密码学协议在基本的椭圆曲线群运算之外, 还需要进行椭圆曲线双线性对的计算。如何通过 GPU 进一步加速双线性对运算也有待未来工作的进一步探索。

## 参考文献

[1] Yang B. Modern cryptography[M]. 2nd ed. Beijing: Tsinghua University Press, 2007.
(杨波. 现代密码学[M]. 2 版. 北京: 清华大学出版社, 2007.)

[2] Szerwinski R, Güneysu T. Exploiting the power of GPUs for asymmetric cryptography[C]. *Cryptographic Hardware and Embedded Systems–CHES 2008: 10th International Workshop*, 2008: 79-99.

[3] Harruson O, Waldron J. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware[C]. *AFRICACRYPT 2009;*

*International conference on cryptology in Africa*, 2009: 350-367.

[4] Zheng F Y, Pan W Q, Lin J Q, et al. Exploiting the Floating-Point Computing Power of GPUs for RSA[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014: 198-215.

[5] Dong J K, Zheng F Y, Emmart N, et al. SDPF-RSA: Utilizing Floating-Point Computing Power of GPUs for Massive Digital Signature Computations[C]. *2018 IEEE International Parallel and Distributed Processing Symposium*, 2018: 599-609.

[6] Ochoa-Jiménez E, Rivera-Zamarripa L, Cruz-Cortés N, et al. Implementation of RSA Signatures on GPU and CPU Architectures[J]. *IEEE Access*, 2020, 8: 9928-9941.

[7] Pan W Q, Zheng F Y, Zhao Y, et al. An Efficient Elliptic Curve Cryptography Signature Server with GPU Acceleration[J]. *IEEE Transactions on Information Forensics and Security*, 2017, 12(1): 111-122.

[8] Dong J K, Zheng F Y, Lin J Q, et al. EC-ECC: Accelerating Elliptic Curve Cryptography for Edge Computing on Embedded GPU TX2[J]. *ACM Transactions on Embedded Computing Systems*, 2022, 21(2): 1-25.

[9] Gao L L, Zheng F Y, Wei R, et al. DPF-ECC: A Framework for Efficient ECC with Double Precision Floating-Point Computing Power[J]. *IEEE Transactions on Information Forensics and Security*, 2021, 16: 3988-4002.

[10] Huang Y, Zheng X Y, Zhu Y X, et al. CPU-GPU Collaborative Acceleration of Bulletproofs - a Zero-Knowledge Proof Algorithm[C]. *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications*, *Big Data & Cloud Computing*, *Sustainable Computing & Communications*, *Social Computing & Networking*, 2021: 674-680.

[11] Qin Z G, Xiong H, Wu S K, et al. A Survey of Proxy re-Encryption for Secure Data Sharing in Cloud Computing[J]. *IEEE Transactions on Services Computing*, 2016, PP(99): 1.

[12] Wen Z, Su M, Gao Y, et al. A Digital Asset Authorization Method Based on Proxy Re-encryption: 202310159404.4[P]. 2023-02-24.
(文周之, 苏明, 高钰洋, et al. 一种代理重加密数字资产授权方法: 202310159404.4 [P]. 2023-02-24.)

[13] Bünz B, Bootle J, Boneh D, et al. Bulletproofs: Short Proofs for Confidential Transactions and More[C]. *2018 IEEE Symposium on Security and Privacy*, 2018: 315-334.

[14] Noether S, MacKenzie A, Research Lab T M. Ring Confidential Transactions[J]. *Ledger*, 2016, 1: 1-18.

[15] Ben Sasson E, Chiesa A, Garman C, et al. Zerocash: Decentralized Anonymous Payments from Bitcoin[C]. *2014 IEEE Symposium on Security and Privacy*, 2014: 459-474.

[16] Fiat A, Shamir A. How to Prove Yourself: Practical Solutions to Identification and Signature Problems[M]. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007: 186-194.

[17] Montgomery P L. Modular Multiplication without Trial Division[J]. *Mathematics of Computation*, 1985, 44(170): 519-521.

[18] Kaya Koc C, Acar T, Kaliski B S. Analyzing and Comparing Montgomery Multiplication Algorithms[J]. *IEEE Micro*, 1996, 16(3): 26-33.

[19] Savas E, Koc C K. The Montgomery Modular Inverse-Revisited[J]. *IEEE Transactions on Computers*, 2000, 49(7): 763-766.

[20] Shanks D. Five number-theoretic algorithms[C]. *Proceedings of the Second Manitoba Conference on Numerical Mathematics*, 1973.

[21] General Administration of Quality Supervision, Inspection and Quarantine of the People's Republic of China, Standardization Administration of China. Information Security Technology - Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves - Part 1. General: GB/T 32918.1-2016[S]. BEIJING: Standards Press of China, 2016: 11-13
(中华人民共和国国家质量监督检验检疫总局, 中国国家标准化管理委员会. 信息安全技术 SM2 椭圆曲线公钥密码算法 第 1 部分: 总则: GB/T 32918.1-2016[S]. 北京: 中国标准出版社, 2016: 11-13)

[22] Chow S S M, Weng J, Yang Y, et al. Efficient unidirectional proxy re-encryption[C]. *Progress in Cryptology–AFRICACRYPT 2010: Third International Conference on Cryptology in Africa*, 2010: 316-332.

[23] Kirshanova E. Proxy re-encryption from lattices[C]. *Public-Key Cryptography–PKC 2014: 17th International Conference on Practice and Theory in Public-Key Cryptography*, 2014: 77-94.

[24] NVIDIA, CUDA C Best Practices Guide [M/OL]. (2018-09-01) [2023-06-05]. https://docs.nvidia.com/cuda/archive/10.0/cuda-c-best-practices-guide/index.html#calculating-occupancy.

[25] TechPowerUp GPU Specs Database [DB/OL]. (2023-06-30) [2023-06-30]. https://www.techpowerup.com/gpu-specs/.

[26] Optimized C library for EC operations on curve secp256k1 [M/OL]. (2019-10-01)[2021-04-01].https://github.com/debitCrossBlockchain/secp256k1-mw.

**高钰洋** 于 2022 年在南开大学计算机科学与技术专业获得硕士学位。研究领域为高性能计算。研究兴趣包括: 并行计算、异构计算。Email: gaoyy@nbjl.nankai.edu.cn

**张健宁** 于 2021 年在中山大学信息与计算科学专业获得学士学位。现在南开大学计算机科学与技术专业攻读硕士学位。研究领域为高性能计算、密码学。研究兴趣包括: 异构计算、零知识证明。Email: zhangjn@nbjl.nankai.edu.cn

**王刚** 于 2002 年在南开大学控制理论与控制工程专业获得博士学位。现任南开大学计算机学院、网络空间安全学院教授。研究领域为海量信息存储、并行与分布式计算。研究兴趣包括: 新型存储设备性能优化、信息检索系统性能优化、联邦学习系统性能优化等。Email: wgzwp@163.com

**刘晓光** 于 2002 年在南开大学控制理论与控制工程专业获得博士学位。现任南开大学计算机学院教授。研究领域包括分布式系统、区块链系统等。Email: liuxguang@nankai.edu.cn

**苏明** 于 2004 在南开大学信息科学专业获得博士学位。现任南开大学计算机学院、网络空间安全学院副研究员。研究领域为序列复杂度和相关算法, 数字水印、区块链等。Email: nksuker@gmail.com