南京郵電大學
Nanjing University of Posts and Telecommunications

多标量乘法算法的GPU负载均衡并行实现

汇报人　：邓同贵
指导老师：董建阔

2024年5月18日

英文文献

Load-Balanced Parallel Implementation on GPUs for Multi-Scalar Multiplication Algorithm

Yutian Chen, Cong Peng✉, Yu Dai, Min Luo✉ and Debiao He

School of Cyber Science and Engineering, Wuhan University, Wuhan, China.
wind.0xdktb@gmail.com,{cpeng,mluo}@whu.edu.cn

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2024, No. 2, pp. 522–544.

目录

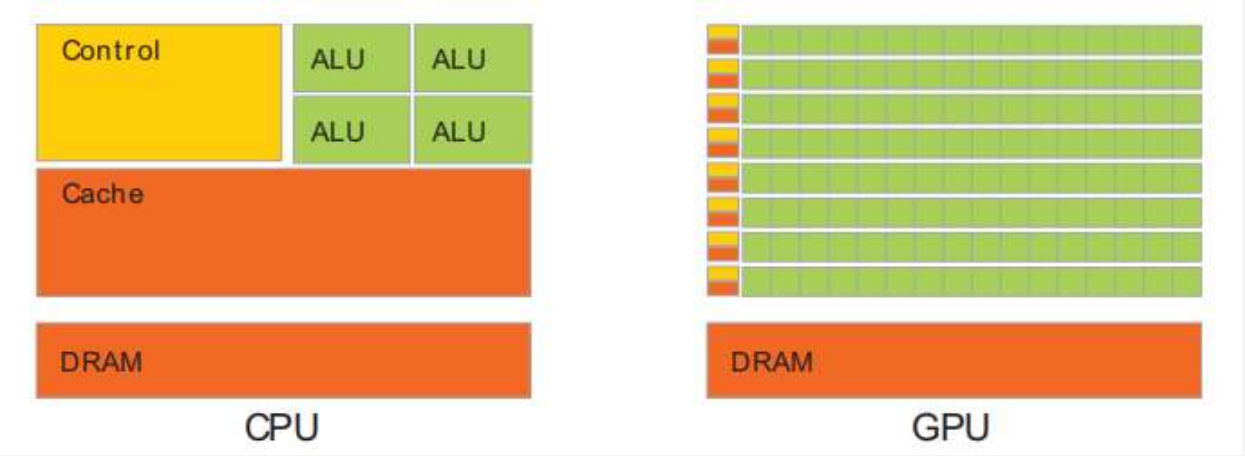
01

相关知识背景

1.1 GPU相关知识介绍

GPU 是指图形处理器单元（Graphics Processing Unit）的缩写。它最初是为图形渲染和处理而设计。然而，随着时间的推移，GPU 的用途逐渐扩展到了其他领域，尤其是科学计算和深度学习等需要大规模并行处理的任务。

ALU：算术逻辑单元



与传统的中央处理器（CPU）不同，GPU 具有大量的小型处理核心，可以同时处理多个任务。这使得 GPU 成为处理大规模数据并执行复杂计算任务的强大工具

简单来说就是：gpu可以起不同的线程，并行完成不同的任务（SIMT）。

1.2

多标量乘法概念

多标量乘法（MSM）是大多数基于椭圆曲线的零知识证明系统的重要组成部分。MSM（Multiple Scalar Multiplication）指的是给定一系列的椭圆曲线上的点（P_i）和标量（K_i），计算：$Q = \sum_{i=1}^{n} k_i P_i$

Algorithm 1 The double-and-add algorithm for computing MSM

Input: The scalars $\{k_i\}_{i\in[n]}$ and the points $\{P_i\}_{i\in[n]} \in \mathbb{G}$
Output: The output point $Q = \sum_{i=1}^{n} k_i P_i$

```
Q = O
for j = λ − 1 to 0 by 1 do
    Q = PDBL(Q)
    for i = 1 to n by 1 do
        if k_i[j] = 1 then
            Q = PADD(Q, P_i)
        end if
    end for
end for
return Q
```

ki is λ-bit

PADD refers to the point addition for unequal points
PDBL refers to the point doubling for equal points
>>没有出现 ki * Pi ?

1.3 负载均衡/平衡

负载平衡（英语：load balancing）是一种电子计算机技术，用来在多个计算机（计算机集群）、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到优化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。

负载平衡服务通常是由专用软件和硬件来完成。 主要作用是将大量作业合理地分摊到多个操作单元上进行执行，用于解决互联网架构中的高并发和高可用的问题。

文献中涉及的负载平衡是：让GPU中起的所有线程，均匀的处理任务，不出现线程闲置的情况，避免资源浪费。

Keywords: Multi-scalar Multiplication · Zero-knowledge Proof · Parallel Implementation

1.4 文章动机

2020年代：零知识证明的研究和应用进入了新的阶段，人们开始关注如何提高零知识证明的效率，以满足大规模应用的需求。

通常来讲，零知识证明加速一般指的对部分运算加速。其中，MSM的计算量相对来说最大，NTT次之。

2023年，Lu等人提出了一种新的并行MSM算法cuZK。在本文中，研究者重新审视这一算法，以该算法为标靶，并提出了一个新的基于GPU的实现，以进一步提高MSM算法的性能，从提高零知识证明的效率。

02

算法加速优化实现

2.1

The Pippenger 算法（MSM的优化实现）

文章是基于 Pippenger 算法 进一步优化 MSM算法。

先给出一些符号:

$Q = \sum_{i=1}^{n} k_i P_i$ --------*MSM 的输出*

Window size： c , k_i *is* λ *bit*

$k_{i,j} = k_i[j * c : j * c + c - 1]$ ---c bits

$k_i = \sum_{j=0}^{\lambda_c} k_{ij} 2^{jc}$ --- 2^c 进制

2.1

The Pippenger 算法（MSM的优化实现）

Step-1: 将主任务分解成多个子任务

Pippenger算法首先选择整数c ∈ [λ]作为窗口大小，并且将每个λ位标量ki分解为多个c位标量片 $k_{i,j} = k_i[\,j * c : j * c + c - 1]$，所以 $k_i = \sum_{j=0}^{\lambda_c} k_{ij} 2^{jc}$，因此，主任务可以被认为是计算 λ/c 向上取整的子任务，被称为小标量 MSM Q_j，记作：$Q_j = \sum_{i=1}^{n} k_{ij} P_i$

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

2.1

The Pippenger 算法（MSM的优化实现）

Step-2 : 在每一个子任务中计算 bucket points

For Q_j

Pippenger 算法 将创建的缓冲点 (buffer point) 称为 **“bucket”**

随后再次将 每个子任务 分成两个 阶段：桶堆积（bucket accumulation）和桶聚合（bucket aggregation）

我们将每一个 bucket 存储的点记作 B_t， 因此：

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} tB_t = \sum_{t=1}^{2^c-1} G_t.$$

接下来我会详细介绍该阶段

2.1

The Pippenger 算法（MSM的优化实现）

Eg. $Q_j = 3P_0 + 5P_1 + 3P_2$

➔ $Q_j = 3(P_0 + P_2) + 5P_1$

➔ $B_3 = (P_0 + P_2), B_5 = P_1$

Step-2.1 : 桶堆积（bucket accumulation）

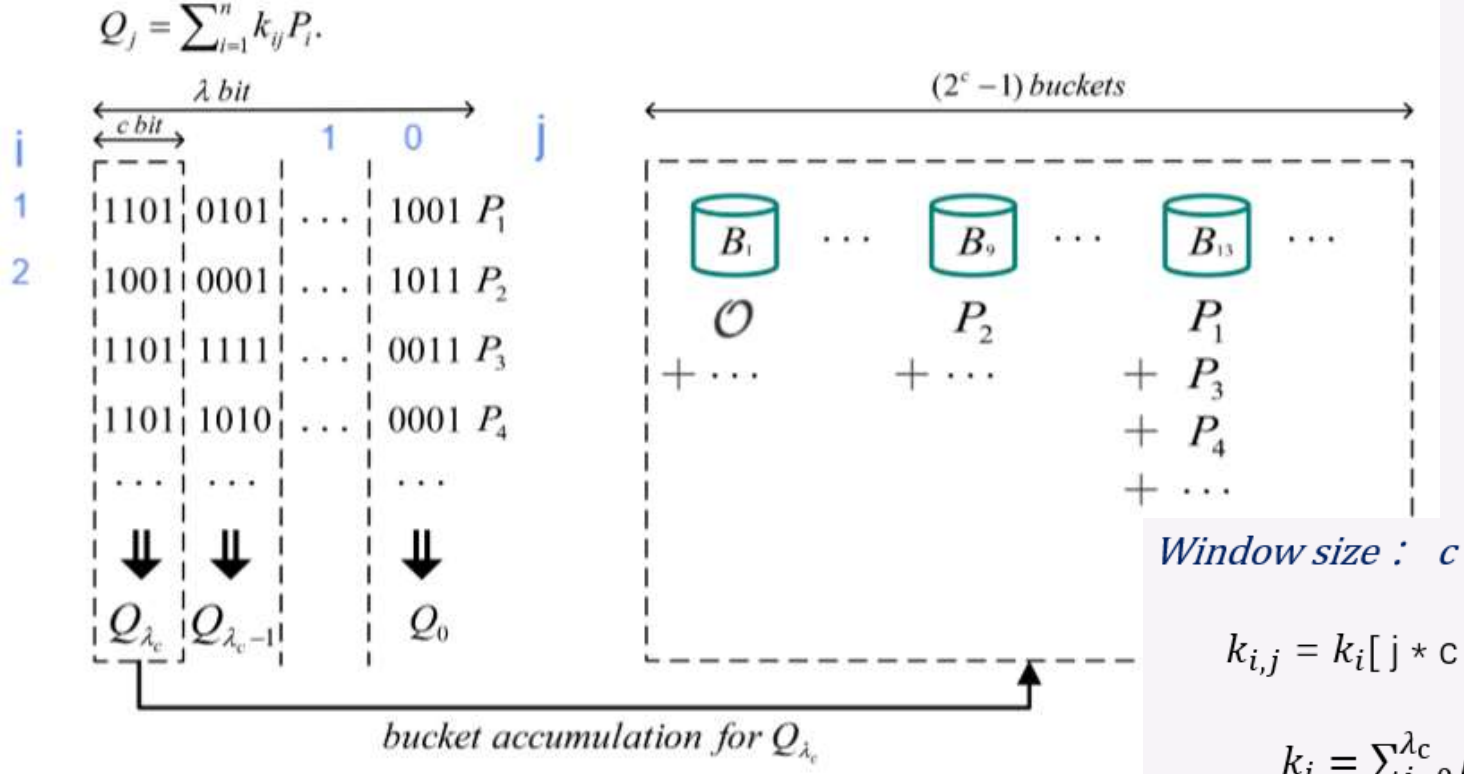


Figure 1: An example of the bucket accumulation phase.

bucket 的数量：
k_{ij} 可能的个数

Window size： c , k_i is λ bit

$k_{i,j} = k_i[\,j * c : j * c + c - 1]$ ---c bits

$k_i = \sum_{j=0}^{\lambda_c} k_{ij} 2^{jc}$ --- 2^c 进制

2.1

The Pippenger 算法（MSM的优化实现）

Step-2.2 : 桶聚合（bucket aggregation）

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} tB_t = \sum_{t=1}^{2^c-1} G_t.$$

已有：B_t

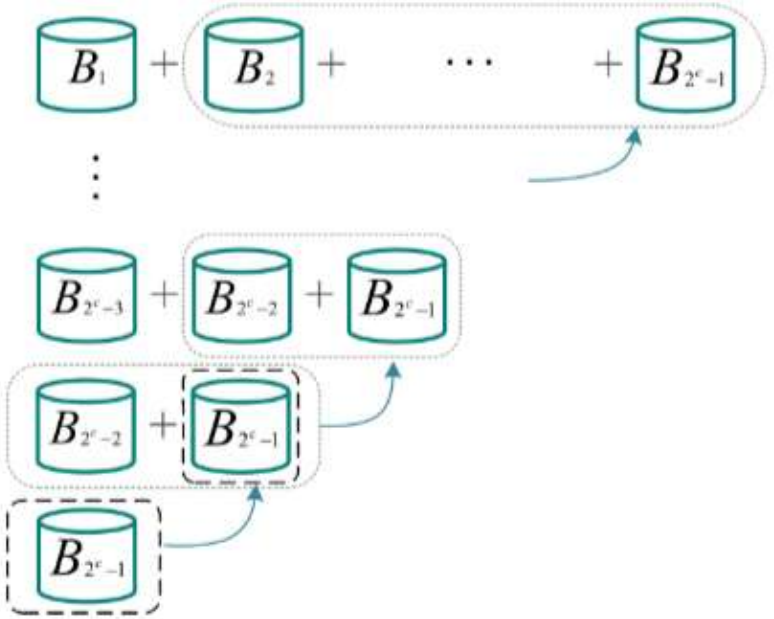


Figure 2: The original bucket aggregation phase (excluding the final summation).

类似于前缀和

2.1

The Pippenger 算法（MSM的优化实现）

Step-3 : 将子任务聚合成主任务

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

$\text{sets } \mathrm{Q} = \mathrm{Q}\ \lambda_c$

$\text{comp}utes\ Q = 2^c Q + Q_i$

$\text{from } i = \lambda_c\ to\ 0$

$finally\ we\ get\ the\ result\ Q.$

2.2

Pippenger 算法的GPU并行实现及挑战

桶堆积（bucket accumulation）阶段

>>对于子任务 Q_j 输出：$B_t, t\ is\ 1\ to\ 2^c - 1$

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i$$

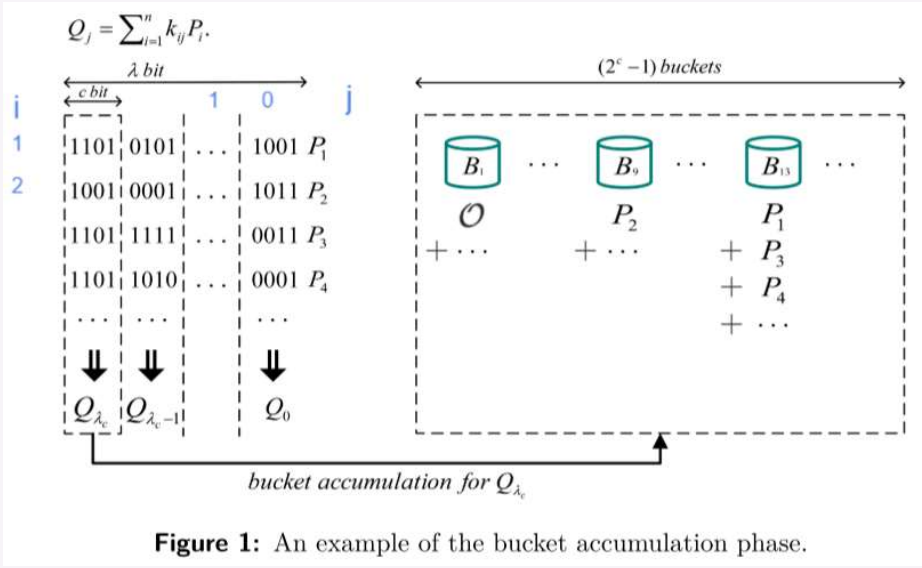


Figure 1: An example of the bucket accumulation phase.

bucket 的数量：
k_{ij} 可能的个数

Eg. $Q_j = 3P_0 + 5P_1 + 3P_2 + P_3$

➔ $(1, P_3), (3, P_0), (3, P_2), (5, P_1)$

➔ $B_1 = P_3, B_3 = (P_0 + P_2), B_5 = P_1$

$thread\ 0$ $thread\ 1$ >>load imbalance

2.2

Pippenger 算法的GPU并行实现及挑战

桶堆积（bucket accumulation）阶段　>>对于子任务 Q_j　输出：$B_t, t\ is\ 1\ to\ 2^c - 1$

Algorithm 2 Previous parallel bucket accumualtion algorithm

Input: Sorted array (ascending) of tuple pairs $\{(a_i, p_i)\}_{i\in[n]}$, points $\{P_i\}_{i\in[n]}$　i : 1-n
Output: Buckets $\{B_t\}$ $(0 \le t < 2^c)$, which is initialized as $\{\mathcal{O}\}$ beforehand

```
1: s = ⌈n/N⌉ · tid
2: e = ⌈n/N⌉ · (tid + 1)        ▷ tid ∈ [0, N) is the index of thread
3: if s ≥ n then return
4: if e ≥ n then e = n
5: while s ≠ 0 and s < n and a_s = a_{s−1} do
6:     s = s + 1
7: end while
8: while e < n and a_e = a_{e−1} do
9:     e = e + 1
10: end while
11: for i = s to e − 1 by 1 do
12:     B_{a_i} = PADD(B_{a_i}, P_{p_i})
13: end for
```

Then we could get the Qj

➔ $(1, P_3), (3, P_0), (3, P_2), (5, P_1)$
➔ $B_1 = P_3, B_3 = (P_0 + P_2), B_5 = P_1$

$thread\ 0$　$thread\ 1$　>>load imbalance

2.2

Pippenger 算法的GPU并行实现及挑战

桶聚合（bucket aggregation）阶段

>>对于子任务 Q_j 输入：$B_t, t\ is\ 1\ to\ 2^c-1$
输出：Q_j

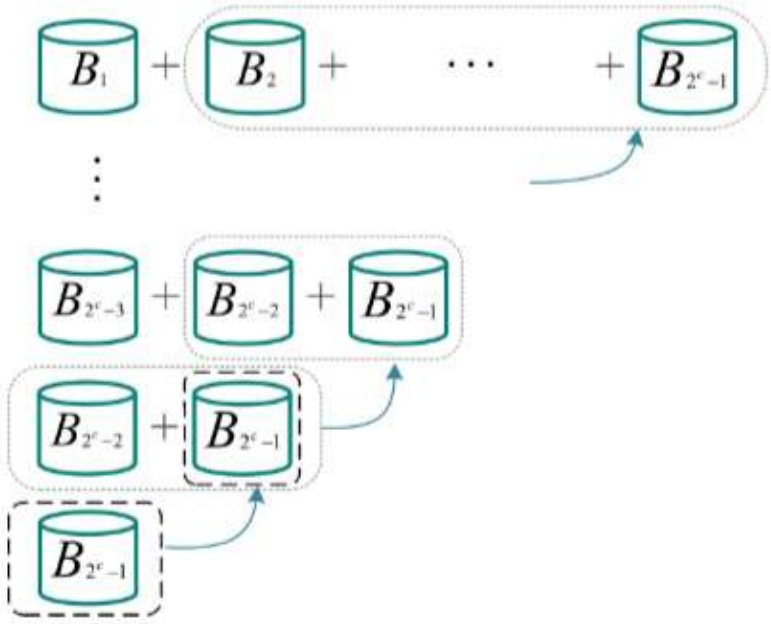


Figure 2: The original bucket aggregation phase (excluding the final summation).

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} tB_t$$

类似于前缀和

最终加法(final summation)：并行规约算法(parallel reduction algorithms)

2.3 文章提出的 MSM GPU加速优化算法

文章基于此前介绍的Pippenger 算法进行了一系列的优化:

1.标量处理和点预计算(Scalar processing and point precomputation)

2.单点和双点加法优化(Single-point and double-point addition optimization)

3.从原始点到桶的负载平衡累积(Load-balanced accumulation from original points to buckets)

4.桶聚集的分层并行约简算法(A layered parallel reduction algorithm for the bucket aggregation)

2.3 文章提出的 MSM GPU加速优化算法

文章基于此前介绍的Pippenger 算法进行了一系列的优化：

1.标量处理和点预计算(Scalar processing and point precomputation)　　对于子任务Q_j

标量$(k_{i,j})$处理 能够将桶(buckets)的数量从2^c-1减少到$2^{c-2}-1$，相比于原始的Pippenger算法，减少了约1/4，**点预计算**则是进一步加快了子任务的处理效率。

$$Q_j=\sum_{i=1}^{n}k_{i,j}P_i$$

宏观过程：

$$k_{i,j}P_i \rightarrow \tilde{k}_{i,j}\cdot(-1)^{s_{i,j}}P_i \rightarrow \bar{k}_{i,j}\cdot(-1)^{s_{i,j}}2^{h_{i,j}}P_i \rightarrow \bar{k}_{i,j}\cdot(-1)^{s_{i,j}}P'_{i,h_{i,j}}$$

where $\bar{k}_{i,j}$ is an odd number of at most $c-1$ bits and $P'_{i,h_{i,j}}$ is a precomputed point.

2.3 文章提出的 MSM GPU加速优化算法

标量($k_{i,j}$)处理阶段(Scalar processing)

对于子任务Q_j

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} t B_t$$

1.将无符号子标量(k_{ij})转换为有符号数($(-1)^{s_{ij}}\tilde{k}_{ij}$)(line 1-6)

$$k_{i,j} P_i \rightarrow \tilde{k}_{i,j} \cdot (-1)^{s_{i,j}} P_i$$

Algorithm 3 Scalar conversion to the float representation

Input: The bit-length λ, n integers $\{k_i\}_{i\in[n]}$ and the window size $c \in [\lambda]$

Output: The integer tuples $\{\bar{k}_{i,j}, h_{i,j}, s_{i,j}\}$

kij is c bits

1: $s_{i,-1} = 0$, $t[0] = 0$, $t[2] = 2^c$
2: **for** $j = 0$ **to** λ_c **by** 1 **do**
3: $k_{i,j} = k_i[jc : jc + c - 1]$
4: $t[1] = k_{i,j} + s_{i,j-1}$ ▷ $k'_{i,j} = t[1]$
5: $s_{i,j} = (t[1] >> c) | (t[1] >> (c-1))$
6: $\tilde{k}_{i,j} = t[s_{i,j} + 1] - t[s_{i,j}]$ ▷ $\tilde{k}_{i,j} = (s_{i,j})\ ?\ (2^c - k'_{i,j}) : k'_{i,j}$
7: $h_{i,j} = max\{\eta : 2^\eta \mid \tilde{k}_{i,j}\}$, $\bar{k}_{i,j} = \tilde{k}_{i,j} >> \eta$ ▷ factor $\tilde{k}_{i,j}$
8: **end for**

注意： $\tilde{k}_{ij}$ 的数据类型是无符号类型， $k_{ij} != (-1)^{s_{ij}}\tilde{k}_{ij}$

但是可以保证： $k_i = \sum_{j=0}^{\lambda_c} k_{ij} 2^{jc} = \sum_{j=0}^{\lambda_c} (-1)^{s_{ij}} \tilde{k}_{ij} 2^{jc}$

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

k_{ij} : 0110 1011 0101
$\tilde{k}_{ij}$: 0111 0101 0101
s_{ij} : 0 1 0

此外添加了限制，保证了不会溢出

效果：最终处理后的标量的数值范围减少了一半

2.3

文章提出的 MSM GPU加速优化算法

标量($k_{i,j}$)处理阶段(Scalar processing)

对于子任务Q_j

1.将有符号数($(-1)^{s_{ij}}\tilde{k}_{ij}$)转换为浮点数表示($(-1)^{s_{ij}}2^{h_{ij}}\bar{k}_{ij}$)(line 7)

$$Q_j = \sum_{i=1}^{n} k_{i,j}P_i = \sum_{t=1}^{2^c-1} tB_t$$

Algorithm 3 Scalar conversion to the float representation

Input: The bit-length λ, n integers $\{k_i\}_{i\in[n]}$ and the window size $c \in [\lambda]$

Output: The integer tuples $\{\bar{k}_{i,j}, h_{i,j}, s_{i,j}\}$

1: $s_{i,-1} = 0$, $t[0] = 0$, $t[2] = 2^c$

2: **for** $j = 0$ **to** λ_c **by** 1 **do**

3: $\quad k_{i,j} = k_i[jc : jc + c - 1]$

4: $\quad t[1] = k_{i,j} + s_{i,j-1}$ ▷ $k'_{i,j} = t[1]$

5: $\quad s_{i,j} = (t[1] >> c)|(t[1] >> (c-1))$

6: $\quad \tilde{k}_{i,j} = t[s_{i,j} + 1] - t[s_{i,j}]$ ▷ $\tilde{k}_{i,j} = (s_{i,j})\ ?\ (2^c - k'_{i,j}) : k'_{i,j}$

7: $\quad h_{i,j} = max\{\eta : 2^\eta \mid \tilde{k}_{i,j}\}$, $\bar{k}_{i,j} = \tilde{k}_{i,j} >> \eta$ ▷ factor $\tilde{k}_{i,j}$

8: **end for**

kij is c bits

$$k_{i,j}P_i \to \tilde{k}_{i,j} \cdot (-1)^{s_{i,j}}P_i \to \bar{k}_{i,j} \cdot (-1)^{s_{i,j}}2^{h_{i,j}}I$$

注意： $\bar{k}_{ij}$ 会被构造成奇数，以此来减少bucket的数量

0110 -> 0011 $h_{\mathrm{ij}} = 1$

$\tilde{k}_{ij}$ $\bar{k}_{ij}$

效果：最终处理后的标量对应的bucket减少了一半

2.3

文章提出的 MSM GPU加速优化算法

标量($k_{i,j}$)处理阶段(Scalar processing)小结：

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

$$k_{i,j}P_i \to \tilde{k}_{i,j} \cdot (-1)^{s_{i,j}} P_i \to \bar{k}_{i,j} \cdot (-1)^{s_{i,j}} 2^{h_{i,j}} P_i \to \bar{k}_{i,j} \cdot (-1)^{s_{i,j}} P'_{i,h_{i,j}}$$

原始　　有符号表示　　浮点型表示　　点预计算

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i$$

最终效果就是：将bucket的数量（标量的取值范围/个数） 从 $2^c - 1$（c bits） 减少到了 $2^{c-2} - 1$ （c-1 bits 且 是奇数），因此算法的效率可以得到提升。

2.3

文章提出的 MSM GPU加速优化算法

2.单点和双点加法优化(Single-point and double-point addition optimization)

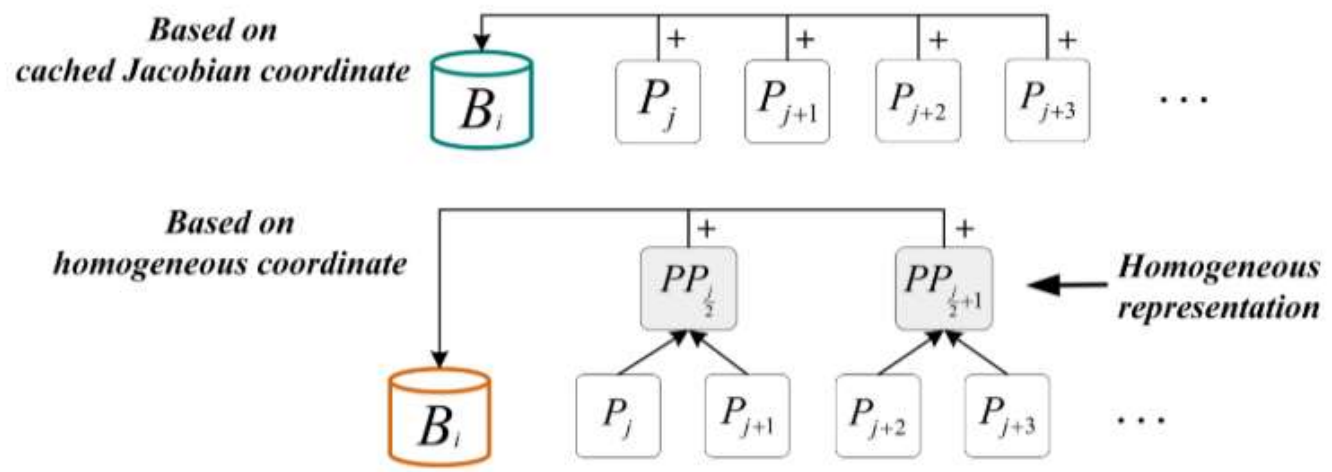


Figure 4: Two methods of adding affine points to the bucket.

2.3 文章提出的 MSM GPU加速优化算法

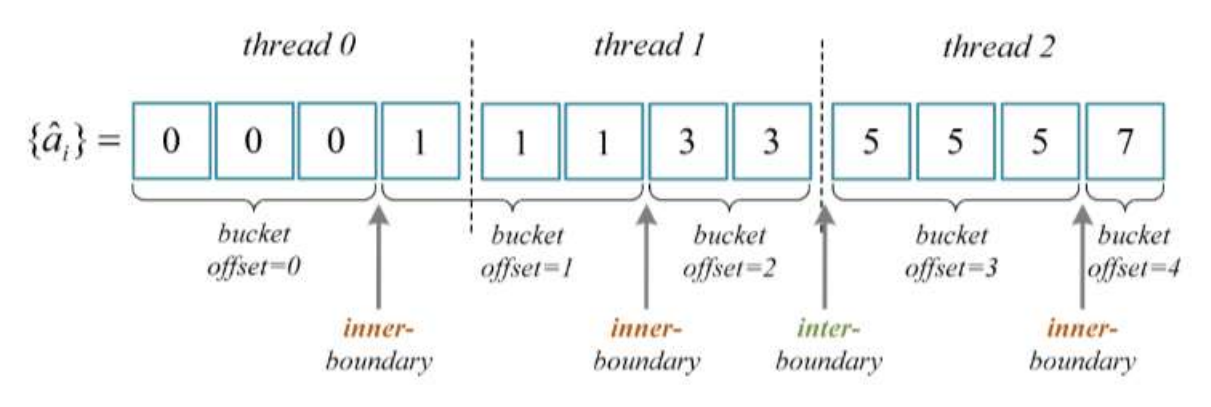
3.从原始点到桶的负载平衡累积(Load-balanced accumulation from original points to buckets)

Eg. $Q_j = 3P_0 + 5P_1 + 3P_2 + P_3$

➔ $(1, P_3), (3, P_0), (3, P_2), (5, P_1)$

➔ $B_1 = P_3, B_3 = (P_0 + P_2), B_5 = P_1$

$thread\ 0$ $thread\ 1$ >>load imbalance



文章给出的解决办法是：
创建一个缓冲区(buffer)，来解决thread的读写冲突问题。
即：原本 thread 0 和thread 1 会同时向 bucket 1 中写入数据，这样引发了冲突。解决办法就是，分两步：① 暂时安排 thread 0 和thread 1 不同时向 bucket 1 写入数据，而是向 buffer 中的不同位置写入数据。② 最后再整合buffer中的数据，完成整个 bucket accumulation。

2.3

文章提出的 MSM GPU加速优化算法

3.从原始点到桶的负载平衡累积(Load-balanced accumulation from original points to buckets) ①

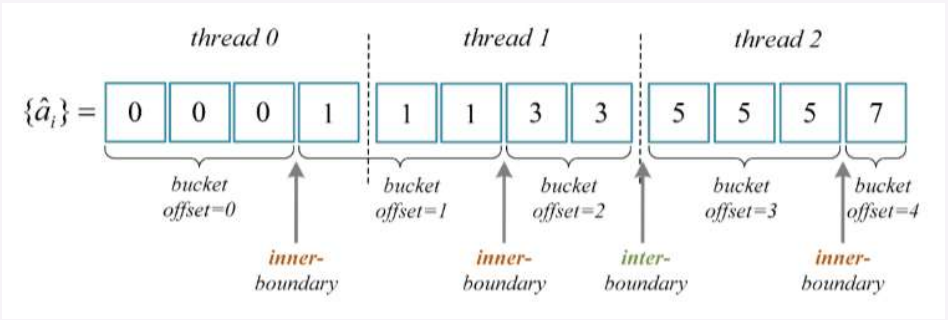
Algorithm 4 Accumulation of bucket parts into buffers using the shared memory

Input: Sorted array (ascending) of tuple pairs $\{(\hat{a}_i, p_i)\}_{i\in[n]}$, points $\{P_i\}_{i\in[n]}$, thread index tid, intra-block thread index tid_{inner}

Output: $buffer$,$buffer\_offset$,$buffer\_index$,$buffer\_used$

1: Obtain the boundaries (s, e)
2: $pre\_bucket\_idx = 0x8000$ ▷ 0x8000: non-existent bucket index
3: $buffer\_offset[tid] = offset = tid + \lfloor \frac{\hat{a}_s.ODD+1}{2} \rfloor$
4: $num = 0$
◇ Cached Jacobian-based:
5: $smem[2 \cdot tid_{inner} + 1] = \mathcal{O}$
6: **for** $i = s$ **to** $e - 1$ **by** 1 **do**
7: **if** $\hat{a}_i.ODD \neq pre\_bucket\_idx \wedge i \neq s$ **then**
8: $buffer[offset + num] = smem[2 \cdot tid_{inner} + 1]$
9: $buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
10: $smem[2 \cdot tid_{inner} + 1] = \mathcal{O}$, $num = num + 1$
11: **end if**
12: $pre\_bucket\_idx = \hat{a}_i.ODD$
13: $smem[2 \cdot tid] = 2^{MIN(\tau, \hat{a}_i.EXP)} P_{p_i}$ ▷ precomputed
14: **for** $j = 1$ **to** $\hat{a}_i.EXP - \tau$ **by** 1 **do**
15: $smem[2 \cdot tid_{inner}]$ =**PDBL**$(smem[2 \cdot tid_{inner}])$
16: **end for**
17: **if** $\hat{a}_i.SIGN = 1$ **then**
18: $smem[2 \cdot tid_{inner}] = -smem[2 \cdot tid_{inner}]$
19: **end if**
20: $smem[2 \cdot tid_{inner} + 1]$ =**PADD**$(smem[2 \cdot tid_{inner} + 1], smem[2 \cdot tid_{inner}])$
21: **end for**
22: $buffer[offset + num] = smem[2 \cdot tid_{inner} + 1])$
23: $buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
24: $buffer\_used[tid] = num + 1$

$$offset_{tid} = tid + min\left\{ \left\lfloor \frac{\hat{a}_i.ODD + 1}{2} \right\rfloor \right\}_{i\in[s,e)}$$



0 1 3 5 7 ---> 0 1 2 3 4
thread 0 : 1 → buffer[1]
thread 1 : 1 → buffer[3]
无读写冲突

2.3

文章提出的 MSM GPU加速优化算法

3.从原始点到桶的负载平衡累积(Load-balanced accumulation from original points to buckets) ②

将buffer缓冲区中的点，聚合到 bucket 中

文章分配2^{c-2}个线程，其中索引为 tid 的线程用于搜索（二分搜索）桶（bucket）$B_{2*tid+1}$对应的所有缓冲点，然后写入到$B_{2*tid+1}$ 中。

这样我们就能得到所有的 桶（bucket）B_t，随后设法求出 Q_j

先前：

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} tB_t$$

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

2.3

文章提出的 MSM GPU加速优化算法

4.桶聚集的分层并行约简算法(A layered parallel reduction algorithm for the bucket aggregation)

$$Q_j = \sum_{i=0}^{2^{c-2}-1} (2 \cdot i + 1) B_{2 \cdot i+1} \quad (j = 0, 1, \cdots, c-1)$$

→ 现在

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} t B_t$$

→ 原来

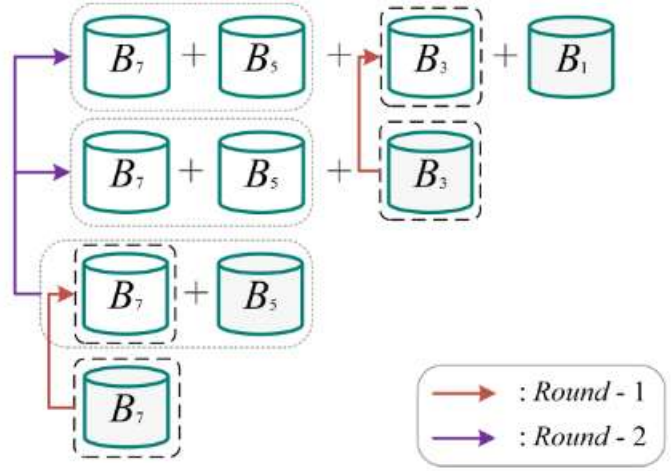


Figure 6: Example of the layered reduction algorithm (scale=4).

Eg. $Q_j = 3P_0 + 5P_1 + 3P_2$

➔ $Q_j = 3(P_0 + P_2) + 5P_1$

➔ $B_3 = (P_0 + P_2), B_5 = P_1$

得到了Q_j 后

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

最终能够算出 结果Q

03 性能测评与总结

3.1

性能测评与总结

Table 3: Execution times (millisecond) of BLS12-381 MSM on different GPUs (V100/RTX3090/RTX4090) and speedup ratios compared to the recent implementation.

Size	V100		RTX3090		RTX4090	
	cuZK	ours	cuZK	ours	cuZK	ours
2^{19}	115.39	44.97 (2.566×)	69.17	29.89 (2.314×)	51.18	17.95 (2.85×)
2^{20}	195.94	84.28 (2.325×)	112.37	56.91 (1.974×)	77.43	32.86 (2.36×)
2^{21}	321.92	161.08 (1.998×)	183.02	110.82 (1.652×)	113.94	62.92 (1.81×)
2^{22}	574.47	315.51 (1.821×)	326.13	214.94 (1.517×)	185.33	124.21 (1.49×)
2^{23}	1128.36	620.74 (1.818×)	645.15	425.78 (1.515×)	355.22	250.68 (1.42×)
2^{24}	2022.47	1233.87 (1.639×)	1181.98	843.18 (1.402×)	1385.76	500.07 (2.77×)

3.1

性能测评与总结

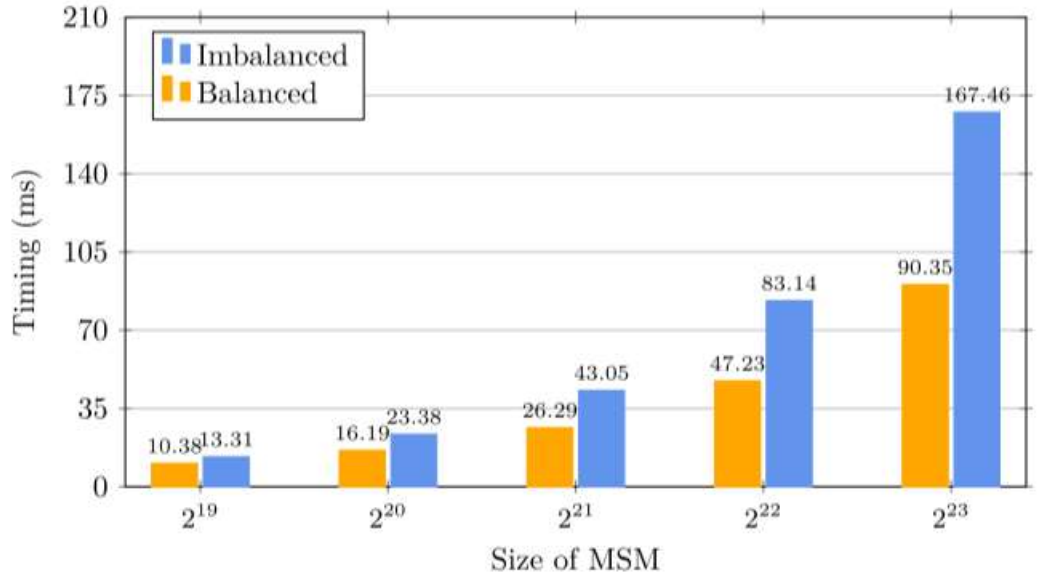


Figure 7: Comparison of the load-balanced and imbalanced versions of our method based on the homogeneous coordinate system (BLS24-315 MSM on RTX4090).

3.1

性能测评与总结

1. 文章由于涉及到多个子任务 并且 launch 了多个核函数（kernel function），核融合（kernel fusion）可以是往后的优化方向。
2. 一些基本运算，比如点加，倍加等，可以通过内联汇编指令集（CUDA PTX）来进行优化，预计可以进一步提升性能。

南京郵電大學
Nanjing University of Posts and Telecommunications

感谢各位老师
请各位老师批评指正！

汇报人：邓同贵　　指导老师：董建阔