Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems

Zhuoran Ji
School of Cyber Science and Technology
Shandong University, China
zrji@sdu.edu.cn

Zhiyuan Zhang[†]
School of Cyber Science and Technology
Shandong University, China
zhangzhiyuan@mail.sdu.edu.cn

Jiming Xu
Ant Group
Shenzhen, China
jiming.xjm@antgroup.com

Lei Ju[*][†]
School of Cyber Science and Technology
Shandong University, China
julei@sdu.edu.cn

Abstract

Zero-knowledge proof is a cryptographic primitive that allows for the validation of statements without disclosing any sensitive information, foundational in applications like verifiable outsourcing and digital currency. However, the extensive proof generation time limits its widespread adoption. Even with GPU acceleration, proof generation can still take minutes, with Multi-Scalar Multiplication (MSM) accounting for about 78.2% of the workload. To address this, we present *DistMSM*, a novel MSM algorithm tailored for distributed multi-GPU systems. At the algorithmic level, *DistMSM* adapts Pippenger's algorithm for multi-GPU setups, effectively identifying and addressing bottlenecks that emerge during scaling. At the GPU kernel level, *DistMSM* introduces an elliptic curve arithmetic kernel tailored for contemporary GPU architectures. It optimizes register pressure with two innovative techniques and leverages tensor cores for specific big integer multiplications. Compared to state-of-the-art MSM implementations, *DistMSM* offers an average 6.39× speedup across various elliptic curves and GPU counts. An MSM task that previously took seconds on a single GPU can now be completed in mere tens of milliseconds. It showcases the substantial potential and efficiency of distributed multi-GPU systems in ZKP acceleration.

[*]Corresponding author.
[†]Also with Quan Cheng Laboratory, Jinan, China.

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651364

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**; • **Security and privacy**;

Keywords: Zero knowledge proof; multi-scalar multiplication; multi-GPU systems; Pippenger's algorithm

ACM Reference Format:
Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. 2024. Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3620666.3651364

1 Introduction

Zero-knowledge proof (ZKP) is a cryptographic primitive that allows one party to prove a statement without revealing any confidential information [16, 17]. For instance, one can authenticate access to a website's restricted area without disclosing their identity (e.g., username). This technique is essential in diverse applications, including verifiable outsourcing [41], electronic voting [19], verifiable machine learning [2, 10], and online auctions [13, 25]. Recent significant advancement in this field is *zkSNARK* (Zero-Knowledge Succinct Non-interactive ARgument of Knowledge) [15, 16, 18, 20]. It offers succinct proofs for any program, with proof sizes under 1*KB* and rapid verification times of less than 10 *ms*. As such, *zkSNARK* has been widely adopted in blockchain [22, 28] and digital currency platforms [6, 8, 11, 12].

While *zkSNARK* offers rapid verification, its widespread deployment is still limited by the compute-intensive proof generation phase [29, 30, 33, 38, 40]. Central to this challenge are operations including multi-scalar multiplication (MSM) and number-theoretic transform (NTT). In real-world ZKP applications [2, 10], these operations are often conducted on high-degree polynomials (e.g., 2^{27}), necessitating billions of big integer modular multiplications. Thus, generating one proof on current mainstream CPU servers can take minutes to hours [2, 6, 10]. Notably, MSM accounts for about 78.2% of this time. Given the inherent parallel nature of MSM and

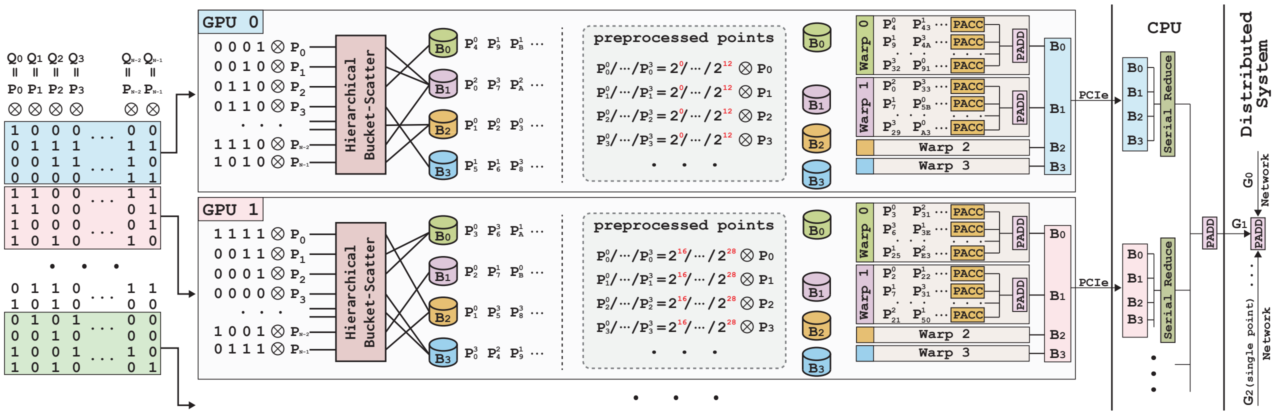

Figure 1. Overall workflow of *DistMSM*

NTT, GPUs have emerged as essential accelerators for *zk-SNARK*. Several industrial implementations, such as Bellperson [11] and Mina [12], have incorporated GPU acceleration.

However, proof generation can still take several minutes on a single GPU [11, 12]. The extensive duration primarily arises from the vast number of operations involved in MSM. Each point-scalar multiplication requires thousands of big integer modular multiplications. An MSM comprises millions to billions of point-scalar multiplications, and generating a proof demands several MSMs. In many ZKP applications, the fastest participant reaps the rewards [12, 13, 22, 25, 28]. It motivates to employ multiple GPUs to further accelerate proof generation, particularly for MSM. Many implementations, which were initially designed for single GPU setups, have been adapted for multi-GPUs. However, they often yield suboptimal performance due to their initial design focus.

To this end, this paper introduces *DistMSM*, an **MSM** algorithm tailored for **Dist**ributed multi-GPU systems. Our approach innovates on two fronts:

At the algorithmic level, *DistMSM* adapts Pippenger's algorithm for multi-GPU systems, as shown in Figure 1. When analyzing the workload on a per-thread basis, we notice a shift in performance bottlenecks when scaling to multiple GPUs. While parallelism eases compute-intensive steps, previously minor non-parallel steps now emerge as bottlenecks. A prime example is the *bucket-scatter* step, which categorizes points based on their coefficients. Paradoxically, as more GPUs are added, its per-thread workload increases, attributed to the rising costs of atomic operations caused by increased concurrent writes. To mitigate this, *DistMSM* introduces a local bucket scatter phase within each thread block, effectively reducing global atomic operations. However, a direct implementation may introduce bucket skewness and complex control logic. *DistMSM* addresses these challenges by capitalizing on the relationship among point IDs, bucket IDs, and workload assignments. *DistMSM* also refines other steps of Pippenger's algorithm for multi-GPU setups. For instance, in the *bucket-sum* step, the buckets of each window can be distributed across multiple GPUs to enhance parallelism. Moreover, recognizing the inefficiencies of executing the *bucket-reduce* step in parallel, *DistMSM* offloads this step to CPUs and allows it to run concurrently with GPU tasks.

At the GPU kernel level, *DistMSM* introduces an elliptic curve arithmetic kernel tailored for modern GPU architectures. A primary challenge identified is register pressure. To illustrate, the point addition operation involves multiple big integer arithmetic operations with complex dependencies. At its peak, it demands 9 concurrent live big integers, using up to 216 registers per thread. To mitigate this, *DistMSM* formulates an optimal execution sequence for these arithmetic operations and employs explicit register spilling routines. These techniques reduce the concurrent live big integers from 9 to 5, improving GPU occupancy and, consequently, performance. Furthermore, contemporary GPUs' tensor cores provide a 8× (i.e., Nvidia A100 [34]) integer arithmetic throughput compared to traditional CUDA cores, but exclusively for matrix multiplication. To exploit this, *DistMSM* proposes a novel representation for constant big integers, allowing big integer multiplication in Montgomery multiplication to be transformed into matrix multiplication. An inherent inefficiency caused by zero bits in the tensor cores' outputs is further addressed using a distinct on-the-fly compaction technique.

This work makes the following major contributions:

- We present *DistMSM*, an MSM algorithm accelerated with multi-GPUs. Experiments show that *DistMSM* achieves a 6.39× speedup for multi-GPU setups compared with the state-of-the-art approaches.
- We adapt Pippenger's algorithm tailored for multi-GPU systems, addressing inherent bottlenecks that arise during scaling and enhancing several steps to improve parallelism and performance.
- We design an EC arithmetic kernel tailored for contemporary GPU architectures. It optimizes register pressure using two innovative techniques and leverages tensor cores for selected big integer multiplication.

2 Background

2.1 The *zkSNARK* Protocol

ZKP stands as a powerful cryptographic primitive [16, 17]. At its core, a prover can assure a verifier that, "for a given function F and input x, I know a secret witness w such that $F(x, w) = 0$", without revealing the exact value of w. Among the array of ZKP protocols, *zkSNARK* [15, 16, 20] is particularly noteworthy because of three key features: (1) succinctness: compact proof sizes ($< 1KB$) and fast verification ($< 10ms$); (2) non-interactive: a single message from the prover to the verifier; (3) zero-knowledge: proof verified without disclosing w.

With these properties, *zkSNARK* has been implemented in various domains, such as verifiable outsourcing, blockchain, and digital currency. For example, traditional blockchain applications require every node to execute identical on-chain computations, leading to significant overhead and latency; in contrast, *zkSNARK* enables off-chain computation, allowing nodes to merely verify lightweight proofs to detect illegal state transitions. The zero-knowledge property also supports confidential transactions. This is well-illustrated by platforms such as Zcash [6] and Pinocchio Coin [8], where transaction amounts and user addresses are hidden, but the validity of each transaction remains verifiable.

Table 1. Number of bits for some elliptic curves

EC	BN254	BLS12-377	BLS12-381	MNT4753
k_i	254 bits	253 bits	255 bits	753 bits
P_i	254 bits	377 bits	381 bits	753 bits

2.2 Multi-Scalar Multiplication

MSM plays a pivotal role in polynomial commitments for *zkSNARK*. Formally, MSM is represented as $\sum k_i P_i$, where each P_i denotes a point on a predetermined elliptic curve and k_i are λ-bit scalars. Table 1 lists the number of bits for points and scalars associated with several elliptic curves commonly employed in ZKP. Notably, while the point vectors P_i remain constant, the scalar vectors k_i vary based on witnesses in different contexts. Each product $k_i P_i$ requires a point scalar multiplication (PMUL). Similar to the fast exponentiation technique, PMUL can be efficiently achieved using point addition (PADD) and point doubling (PDBL) operations.

$$n = \frac{y_2 - y_1}{x_2 - x_1}, \quad x_3 = n^2 - x_1 - x_2, \quad y_3 = n * (x_1 - x_3) - y_1 \quad (1)$$

Many short Weierstrass curves, recognized for their pairing-friendly properties, are widely used in ZKP. Formula 1 provides the PADD formula for them. Direct application of this formula can be compute-intensive, mainly due to the modular inverse operation. Thus, many implementations employ the *XYZZ* coordinate system (Algorithm 1) [5], demanding only 14 modular multiplications regardless of the bit length.

Algorithm 1 PADD in the *XYZZ* coordinate system

```
U1 = X1 * ZZ2, U2 = X2 * ZZ1
S1 = Y1 * ZZZ2, S2 = Y2 * ZZZ1
P = U2 - U1, R = S2 - S1
PP = P * P, PPP = PP * P, Q = U1 * PP
V = R*R, V = V-PPP, V = V-Q, X3 = V-Q
T = Q-X3, Y = R*T, T = S1*PPP, Y3 = Y-T
ZZ = ZZ1 * ZZ2, ZZ3 = ZZ * PP
ZZZ = ZZZ1 * ZZZ2, ZZZ3 = ZZZ * PPP
```

Modular multiplication is particularly challenging due to the costly division operation. A widely adopted approach to address this challenge is Montgomery modular multiplication [31]. Various optimized implementations, such as SOS, CIOS, and FIOS, are thoroughly discussed in [23]. Taking SOS (Algorithm 2) as an example, a primary optimization employed by these methods is to substitute n' with $n'_0 = n' \bmod 2^{32}$ when computing $C * n'$, which significantly reduces the computation. Some elliptic curves exhibit unique properties for n'_0, allowing further optimization.

Algorithm 2 Montgomery multiplication with SOS method

```
1: $C[0{:}2N] = A[0{:}N] \times B[0{:}N]$ ▷ 32N-bit integer multiply
2: for $i$ from $0$ to $N-1$ do
3:     $m[i] = (C[i] \times n'_0)$ & 0xffffffff
4: $C[0{:}2N]$ += $m[0{:}N] \times n[0{:}N]$
5: ret $C[N{:}2N] > n[0{:}N]\ ?\ (C[N{:}2N] - n[0{:}N]) : C[N{:}2N]$
```

2.3 Pippenger's Algorithm

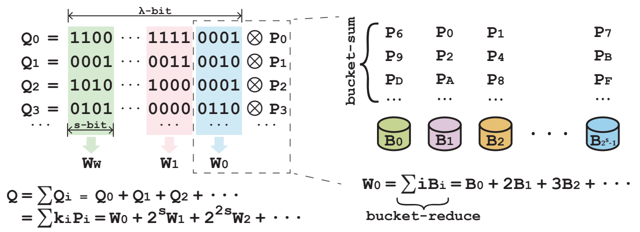
λ-bit
Q0 = 1100 ··· 1111 0001 ⊗ P0
Q1 = 0001 ··· 0011 0010 ⊗ P1
Q2 = 1010 ··· 1000 0001 ⊗ P2
Q3 = 0101 ··· 0000 0110 ⊗ P3
s-bit
Ww
W1
W0
bucket-sum
P6 P0 P1 P7
P9 P2 P4 PB
PD PA P8 PF
B0
B1
B2
B2^s-1
Q = ∑Qi = Q0 + Q1 + Q2 + ···
= ∑kiPi = W0 + 2^s W1 + 2^2s W2 + ···
W0 = ∑iBi = B0 + 2B1 + 3B2 + ···
bucket-reduce

Figure 2. Illustration of Pippenger's algorithm

Many prior optimizations, such as Pippenger's algorithm, use the distributive law to reduce the number of PMUL operations. As illustrated in Figure 2, it partitions the MSM task into $\lceil \lambda/s \rceil$ windows by breaking the λ-bit scalars down into s-bit chunks. Each window corresponds to a subtask that computes $\sum m_i P_i$, where m_i is the s-bit scalar component of k_i relevant to the current window. Within each window, PMUL operations are categorized into buckets based on their scalar component (i.e., m_i), a process referred to as *bucket scatter*. As an example, both P_0 and P_2 would be multiplied by the scalar `0001` and would therefore be placed in bucket 1. Then, the distributive law is applied to first sum the points within each bucket to obtain B_i (*bucket-sum*), followed by a summation across buckets to yield the subtask result W_j (*bucket-reduce*). Finally, results from each window are combined (*window-reduce*) to produce the final result. This approach reduces the computational costs of MSM to $\lceil \lambda/s \rceil (N + 2^{s+1})$ PADDs and λ PDBLs [4, 29].

2.3.1 Parallel Pippenger's Algorithm . Many studies have explored the potential for parallelizing Pippenger's algorithm [4, 11, 12, 29, 30, 38]. As revealed in Figure 2, the algorithm has three dimensions of parallelization: B-dim, W-dim, and Q-dim. Among these, the W-dim offers minimal parallelism, with only tens of independent tasks. Moreover, partitioning tasks along the Q-dim may decrease the algorithm's efficiency, since fewer points are involved in the distributive law. Recent GPU implementations have chiefly focused on parallelization across the B-dim [29, 30]. For each window: (1) all threads of a GPU collaboratively distribute all points into 2^s buckets, (2) each bucket is then allocated to individual or multiple threads, and (3) all threads collaboratively perform a parallel reduction to aggregate the results of each bucket into a single elliptic curve point. This approach ensures adequate parallelism without diminishing the number of points involved in the distributive law.

A hybrid approach that combines different parallelism dimensions is also feasible. For example, windows could be distributed across multiple GPUs (W-dim), and the buckets within a window are assigned to different threads within a single GPU (B-dim). Additionally, many GPU implementations employ precomputation[30, 39]. For a given point P_i, values such as $2^s P_i$, $2^{2s} P_i$, and so on, are precomputed. As a result, in window j, the point P_i is replaced with $2^{js} P_i$. It allows elliptic curve points from two different windows to be directly summed using a single PADD operation.

3 Pippenger's Algorithm for Multi-GPUs

DistMSM follows the parallelization strategy outlined above, where each bucket is mapped to one or several threads within a GPU, and each GPU manages one or several windows. While this sets a general framework for parallelization, the specifics of algorithmic implementation, such as addressing data race issues in *bucket-scatter*, have a vast design space. Although many implementations of Pippenger's algorithm support multi-GPU configurations, most of them are initially designed for single GPU setups and later expanded to multi-GPU systems. In many real-world ZKP applications, the fastest participant reaps the rewards [12, 13, 22, 25, 28]. To illustrative, Cysic [7] employs about 40 high-end FPGAs in their ZKP system. In light of this trend, *DistMSM* designs algorithms tailored for distributed multi-GPU systems.

3.1 Workload Analysis of Individual Threads

In analyzing the parallel Pippenger's algorithm, we emphasize that the execution time is determined by the workload assigned to each thread, not the total workload. Consider a multi-GPU system that comprises N_{gpu} GPUs. Each GPU can accommodate N_T concurrent threads. Given an MSM task with N points and λ-bit scalars, along with a window size s. The number of windows is then derived as $N_{win} = \lceil \lambda/s \rceil$, and each GPU manages $\lceil N_{win}/N_{gpu} \rceil$ windows.

In the *bucket-scatter* step, all threads within a GPU collaborate to categorize the N points into 2^s distinct buckets. Each thread manages $\lceil N/N_T \rceil$ points per window, accumulating in $\lceil N_{win}/N_{gpu} \rceil \times \lceil N/N_T \rceil$ bucket insertions in total. To prevent data races, atomic operations are employed to allocate empty slots, whose cost generally scales with the number of simultaneous writes to a memory address [9].

The subsequent *bucket-sum* step is an embarrassingly parallel task. Specifically, the points within each individual bucket can be summed independently. On average, each thread executes N/N_T PADD operations. In cases where $2^s < N_T$, it becomes essential to allocate at least $t = N_T/2^s$ threads to each bucket to ensure maximum GPU utilization. Adhering to the SIMD (Single Instruction Multiple Data) execution paradigm, each thread performs $\log_2(t)$ PADD operations during the intra-bucket reduction process. As a result, the total number of PADD operations executed by each thread is $\lceil N_{win}/N_{gpu} \rceil (N/N_T + \log_2(t))$.

The *bucket-reduce* step distinguishes itself from conventional reduce operations, primarily due to the computation of $2^i B_i$ prior to parallel reduction. Calculating each $2^i B_i$ requires s PADD and s PDBL operations in the SIMD execution model. Thus, the parallel reduction phase costs $2s \times \lceil \frac{2^s}{N_T} \rceil + \lceil \frac{2^s}{N_T} \rceil + \log_2(N_T)$ EC arithmetic operations. The last term, $\log_2(N_T)$, requires global synchronization. In cases where $2^s < N_T$, only s PADD operations are essential for the parallel reduction. Thus, for every window, a thread executes $2s \times \lceil \frac{2^s}{N_T} \rceil + \min(\lceil \frac{2^s}{N_T} \rceil + \log_2(N_T), s)$ EC arithmetic operations. As *bucket-reduce* and *window-reduce* are commutative with precomputation, the workload for each individual thread is $\lceil \frac{2^s}{N_T} \rceil \times (\lceil N_{win}/N_{gpu} \rceil + 2s) + \min(\lceil \frac{2^s}{N_T} \rceil + \log_2(N_T), s)$.

In the *window-reduce* step, only N_{gpu} points are left for the CPU to sum, with its workload being negligible.

In summary, the per-thread workload is: $\lceil \frac{N_{win}}{N_{gpu}} \rceil \times \lceil \frac{N+2^s}{N_T} \rceil + \lceil \frac{2^s}{N_T} \rceil \times 2s + \min(\lceil \frac{2^s}{N_T} \rceil + \log_2(N_T), s)$. This formula assumes that each GPU manages at least one complete window. In real-world ZKP applications, N can be exceedingly large (e.g., 2^{27}). It strongly incentivizes to distribute a window's buckets across multiple GPUs to enhance parallelism. In such scenarios, the cost is modified as: $(N + 2^s \times 2s)/(\lfloor N_{gpu}/N_{win} \rfloor \times N_T) + \log_2(2^s/\lfloor N_{gpu}/N_{win} \rfloor)$.

Figure 3 illustrates the per-thread costs derived from the above formulas, with varying window sizes and platforms.

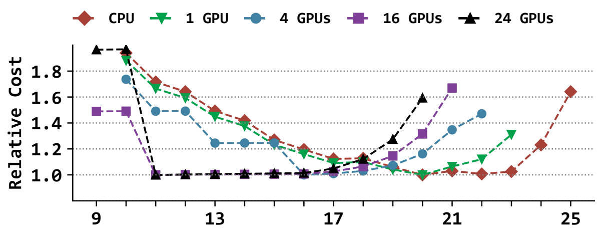

Figure 3. Per-thread workload estimation

The other parameters are set to widely accepted values: $N = 2^{26}$, $N_T = 2^{16}$, and $\lambda = 253$. To highlight the trends, the estimated costs have been normalized based on the smallest value. A notable insight is that the optimal s is platform-dependent. For a 16-GPU system, the optimal s is 11, while for a single GPU, s is best set at 20. This difference arises because, while *bucket-sum* scales effectively across multiple GPUs, the per-thread cost of *bucket-reduce* increases linearly with s and does not decrease substantially with more GPUs.

3.2 Algorithm Design for Multi-GPU Systems

A smaller window size shifts the performance bottleneck, necessitating a significant algorithm redesign. First, the *bucket-scatter* step becomes a bottleneck. With fewer buckets, more threads access the same memory address concurrently, increasing the overhead of global atomic operations. While a basic bucket scatter implementation is sufficient for single GPU setups, it falls short in a multi-GPU system. Second, in the *bucket-sum* step, the reduced number of buckets requires allocating multiple threads to a single bucket to optimize GPU utilization. Third, as the *bucket-reduce* step becomes negligible with a small s, executing this step on a CPU might be more efficient. Figure 1 shows an overview of *DistMSM*, with subsequent subsections delving into the design details.

3.2.1 Hierarchical Bucket Scatter. To reduce the number of global atomic operations, *DistMSM* introduces a novel hierarchical bucket scatter algorithm. At its core, each thread block undertakes a local bucket scatter for its assigned points with shared memory. Each bucket's points are then committed to global memory with a single atomic operation. A practical challenge arises from the presence of numerous (i.e., N_{bucket}) buckets. An intuitive approach might be to partition shared memory into N_{bucket} equal segments and treat each as an individual bucket, as shown in Figure 4a. However, this approach presents two main drawbacks. First, threads within a thread block increment each bucket's counter concurrently. It poses a potential conflict: a thread block may need to append n elements to a bucket that has less than n available slots. Addressing this issue requires complex control logic and thread-block-level synchronization upon every insertion. Moreover, it also introduces skewness: one bucket may be full while others remain nearly empty. Flushing all buckets increases the number of global atomic operations, whereas selectively flushing the full buckets can cause inefficient global memory utilization.

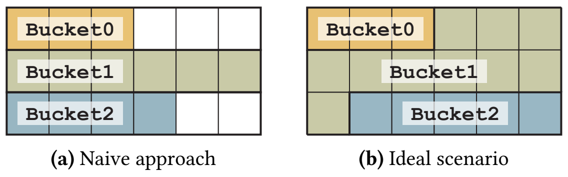

Figure 4. Distribute shared memory to N_{bucket} buckets

The ideal scenario is depicted in Figure 4b, where each bucket is allocated shared memory precisely equivalent to its element count. It ensures that shared memory is filled exactly after each thread has processed K elements, where K is derived from the size of shared memory divided by the thread count. This arrangement effectively addresses the aforementioned challenges, and the problem then shifts to determining bucket sizes. To address this, *DistMSM* proposes a three-level hierarchical bucket scatter algorithm, as shown in Algorithm 3. Each thread fetches K coefficients (scalars) from device memory, computes its *bucket_id*, stores the *bucket_id* in a register, and then increases the associated bucket counter. Upon processing K elements, the offset for each bucket is calculated by accumulating the counts of its preceding buckets. After that, each thread stores the *point_id* of its K points into the buckets in shared memory.

Algorithm 3 Three-level hierarchical bucket scatter

```
1:  $tid$, $gid$, $bid$ = local/global thread, thread block index
2:  for $reg\_idx$ from $0$ to $K-1$ do
3:      $addr = reg\_idx \times grid\_dim \times block\_dim + gid$
4:      $bucket\_id = coeffs[addr]\ \&\ mask \gg win\_off$
5:      $reg\_cache[reg\_idx] = bucket\_id$
6:      atomic_inc($shm\_count[bucket\_id]$)
7:  $shm\_off$ = parallel_prefix_sum($shm\_count$)
8:  for $reg\_idx$ from $0$ to $K-1$ do
9:      $bucket\_id = reg\_cache[reg\_idx]$
10:     $pos$ = atomic_inc($shm\_off[bucket\_id]$)
11:     $shm\_cache[pos] = reg\_idx \parallel tid$          ▷ $point\_id$
12: for $i$ from $0$ to $2^s-1$ do
13:     $reg\_idx, tid = shm\_cache[shm\_off[i] - tid]$
14:     $devm\_off$ = atomic_add($devm\_off[i]$, $shm\_count[i]$)
15:     $devm\_off[i][devm\_off - tid] = reg\_idx \parallel bid \parallel tid$
```

The *point_id* of the points stored in registers can be expressed as $reg\_idx \parallel bid \parallel tid$. Since the *point_id* can be directly inferred, it is only necessary to store the *bucket_id* for each point (Line 5). In contrast, points in shared memory inherently reveal the *bucket_id* through their addresses. It only requires to record the *point_id*, and the *bid* segment can be omitted (Line 11). Given a thread block with 1024 threads and 128*KB* shared memory for *point_id* storage, 64*K* points can be scattered locally, reducing global atomic operations by a factor of $\frac{1}{64}$ for $N_{bucket} = 1024$. The corresponding register usage per thread is 32, regardless of the bucket count.

3.2.2 Highly Parallel Bucket-Sum.

The *bucket-sum* step dominates the computations, but its design remains straightforward. Essentially, it comprises N_{bucket} independent accumulations of N_{bucket} sets of elliptic curve points. Traditional methods often allocate a single bucket to each thread. However, as mainstream GPUs can support approximately 2^{16} concurrent threads, this leads to poor GPU utilization for a small s. Addressing this, *DistMSM* allocates multiple threads to each bucket, denoted as N_{thread}. After the independent summations, the N_{thread} threads collaboratively perform a reduction to combine their partial sums. In the SIMD execution model, each thread executes $\log_2(N_{thread})$ PADD operations for the parallel reduction. To illustrate, with $N_{thread} = 32$ and $N = 2^{26}$, it incurs 0.49% extra PADD operation. Thus, the overhead from the intra-bucket reduction is negligible. Additionally, N_{thread} is typically set as a multiple of 32 (e.g., a warp). It allows *DistMSM* to utilize the hardware scheduler to address workload imbalances among buckets.

In practical ZKP applications, the value of N can be substantial. When processing all buckets of a window on a GPU, each thread executes thousands of PADD operations, causing significant latency. To address this, *DistMSM* proposes distributing a window's workload across multiple GPUs. Two potential approaches emerge: the first divides N into *GPUs-per-window* segments, not modifying the GPU code but increasing the CPU's workload; the second splits the buckets into *GPUs-per-window* parts, which keeps the CPU workload consistent but requires allocating more threads per bucket for optimal GPU utilization. *DistMSM* adopts the second approach because it prevents excessive CPU burden, promoting better scalability. In an extreme scenario with 1024 threads per bucket, assigning 128 buckets per GPU ensures full GPU utilization. The corresponding overhead from intra-bucket reduction for each thread is only 128 PADD operations, resulting in a mere 4% overhead with $N = 2^{28}$. Additionally, *DistMSM* supports flexible bucket distribution across multiple GPUs. Consider a scenario where three GPUs process two windows: two GPUs could handle $\frac{2}{3}$ of each window, and the third GPU manages the remaining $\frac{1}{3}$ from both windows. In *DistMSM*, this can be achieved simply by launching a different number of thread blocks.

3.2.3 CPU Deployment of Bucket-Reduce.

In multi-GPU systems, the parallel *bucket-reduce* step is notably inefficient. However, when executed serially, it requires only a few thousand PADD operations. Thus, *bucket-reduce* is better suited for CPU execution. Using an aggressive extrapolation based on existing implementations [11, 29, 30], a GPU could be up to 128× faster than a high-end CPU. In configurations where one CPU manages 8 GPUs, the *bucket-reduce* step on the CPU spends less time than a GPU as long as N_{bucket} is less than $\frac{N}{8\times128}$. For example, when $N = 2^{28}$, the CPU is not the bottleneck provided $N_{bucket} < 2^{15}$. Proof generation involves several MSM calculations and other GPU tasks, which means that *bucket-reduce* can be efficiently pipelined.

4 Efficient EC Arithmetic on GPUs

Two primary operations, PADD and PDBL, are involved in MSM. Of these, PADD requires more big integer arithmetic operations and is executed more frequently in Pippenger's Algorithm. Although many GPU implementations exactly follow the pseudocode in Algorithm 1, there remains significant potential for improvement. This section delves into optimizing PADD operations on GPUs. Notably, the optimizations discussed also apply to PDBL operations.

4.1 Dedicated Point Accumulation Kernel

In Pippenger's Algorithm, there are two distinct types of PADD operations. The first type merges two partial results into one, while the second type accumulates an EC point to a partial result (i.e., $acc' = acc + P_i$), denoted as point accumulation (PACC). When the window size s is small, the majority of PADD operations belong to the PACC type. For any P_i, the values of its *ZZ* and *ZZZ* are always 1. Conversely, when s is large, many PADD operations are of the first type, where neither *ZZ* nor *ZZZ* are necessarily 1. In multi-GPU setups, *DistMSM* prefers a smaller s. Thus, it becomes advantageous for *DistMSM* to adopt a specialized PACC implementation. With prior knowledge that *ZZ* and *ZZZ* equal 1, the corresponding multiplications can be bypassed. The PACC kernel is depicted in Algorithm 4, where the partial result [X_{acc}, Y_{acc}, ZZ_{acc}, ZZZ_{acc}] is updated to [$X_{acc'}$, $Y_{acc'}$, $ZZ_{acc'}$, $ZZZ_{acc'}$] by accumulating the point [X_P, Y_P, 1, 1].

Algorithm 4 PACC in the XYZZ coordinate system

```
U2 = $X_P$ * $ZZ_{acc}$, S2 = $Y_P$ * $ZZZ_{acc}$
P = U2 - $X_{acc}$, R = S2 - $Y_{acc}$
PP = P * P, PPP = PP * P, Q = $X_{acc}$ * PP
V = R * R, V = V - PPP, V = V - Q
$\mathbf{X_{acc'}}$ = V-Q
T = Q - $X_{acc'}$, Y = R * T, T = $Y_{acc}$ * PPP
$\mathbf{Y_{acc'}}$ = Y - T
$\mathbf{ZZ_{acc'}}$ = $ZZ_{acc}$ * PP, $\mathbf{ZZZ_{acc'}}$ = $ZZZ_{acc}$ * PPP
```

4.2 Reducing Register Pressure

When optimizing the PADD operations that handle big integers, a critical factor to consider is register pressure. Register pressure refers to the demand placed on available registers by concurrently live variables, which determines the kernel's occupancy. A widely used strategy for accumulation is to keep the accumulator in registers, thus enhancing memory access efficiency. For the PACC operation, the accumulator, *acc* (consisting of *X*, *Y*, *ZZ*, and *ZZZ*), is live at the beginning. By the end, *acc′* should remain in registers.

Additionally, Montgomery multiplication necessitates a temporary big integer for holding intermediate results. Given these constraints, the peak register pressures for straightforward PADD (i.e., Algorithm 1) and PACC (i.e., Algorithm 4) implementations are 11 and 9 big integers, respectively. Using real-world elliptic curves as benchmarks, the straightforward PADD implementation requires 132 registers per thread for BLS12-377 and 264 for MNT4753.

4.2.1 Optimal Execution Sequence.

To optimize occupancy, it is desirable to allocate no more than 64 registers per thread. These registers are multifunctional, serving not only to store big integers but also for auxiliary purposes, such as calculating memory addresses and storing thread indices. Given that a single big integer can consume 8 to 24 registers, reducing the number of concurrently live big integers is critical to achieving higher GPU occupancy. Although GPU compilers inherently possess mechanisms for instruction scheduling to enhance performance [32], their focus is predominantly at the machine instruction level. Within the context of PADD, each multiplication operation is a sophisticated big integer modular multiplication function. It encompasses a myriad of arithmetic computations, memory accesses, and even synchronization. Such complexity might hinder the compiler's capability to schedule instructions efficiently. To this end, *DistMSM* explicitly reschedules operations within Algorithm 1 and Algorithm 4 to minimize the peak number of concurrently live big integers.

While instruction scheduling algorithms have been extensively studied in the compiler literature, most existing approaches employ heuristic or approximate strategies. Recognizing the importance of the PADD operation, there is a clear incentive to seek an optimal execution sequence. To achieve this, *DistMSM* evaluates all potential topological orders of the operations in PADD and PACC, aiming to identify the sequence with the fewest concurrently live variables. Notably, sequencing specific pairs of operations consecutively can guarantee optimal results. For instance, executing `P = U2 - X1` right after `U2 = X2 * ZZ1` removes `U2` from the set of live variables, adding only `P`, and thus maintaining the number of live variables. Such insights merge the instructions of both PADD and PACC to just 12 scheduling units. Given that the maximum possible topological orders are capped at 12!, and the actual number is even much less due to data dependencies, brute-force methods are feasible.

Figure 5 presents an optimal sequence for the PACC operation, highlighting the live variables associated with each instruction. Compared to the straightforward implementation, the peak number of concurrently live big integers is reduced from 9 to 7, which translates to a saving of 16 to 48 registers. Applying the same methodology to the PADD operation yields a comparable improvement, reducing the peak count of concurrently live variables from 11 to 9.

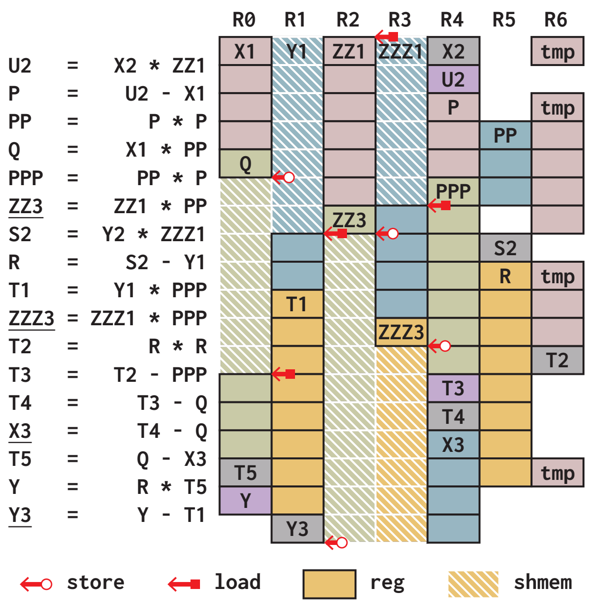

Figure 5. An optimal execution sequence of PACC operation

4.2.2 Explicit Register Spilling to Shared Memory.

Register pressure in the PADD operation remains a significant issue, even in highly optimized implementations such as the one shown in Figure 5. During peak demand, compilers might resort to register spilling [32], where some registers are temporarily stored in device memory and retrieved subsequently. This process is particularly inefficient on GPUs, as each spilling instance requires all threads to transfer a 4-byte variable between the register and device memory.

To further reduce register pressure, *DistMSM* proposes using shared memory as a cache for selected big integers, aiming to reduce the peak register pressure. This approach adapts the concept of register spilling but takes advantage of the considerably higher bandwidth of shared memory compared to device memory. The movement of big integers between registers and shared memory is managed by explicitly integrated code within the GPU kernel. Decisions about which big integers to cache can be guided by traditional register spilling algorithms. In Figure 5, a slash pattern indicates variables stored in shared memory. It reduces the peak number of big integers in registers from 7 to 5 with the cost of transferring 4 big integers between registers and shared memory. Meanwhile, at any given point, only a maximum of 3 big integers are stored in shared memory.

4.3 Montgomery Multiplication with Tensor Core

Tensor Cores (TC) represent a significant advancement in GPU architecture by enabling matrix-matrix computations

[26], in contrast to the scalar-scalar operations of CUDA cores. TC boasts a much higher integer arithmetic throughput than CUDA cores. For instance, on the Nvidia A100 [34], TC delivers an *int8* throughput of 624 TOPS, equivalent to 156 *int32* TOPS. It is 8× the throughput of CUDA cores, which only achieve 19.5 TOPS.

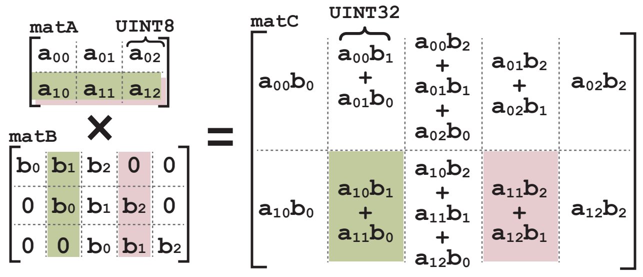

Figure 6. Big integer multiply via matrix multiply

The key to effectively using TC for big integer multiplication is to represent integers as matrices, where each element is of type *uint8*, a format that TC supports. Figure 6 provides an example of such a data layout. It is practical only when *matB* is constant due to the significant overhead of transferring an integer from conventional memory layouts (e.g., row-major) to the specific format required by *matB*. In Montgomery multiplication, n is a constant. Thus, the second wide-multiplication operation, $m \times n$ in Algorithm 2, can be accelerated with TC.

For N-bit integers, using TC for multiplication produces an output of $\frac{N}{4}$ *uint32* elements, denoted as an array $C_{0:\frac{N}{4}}$. This is because the inputs are divided into *uint8* for multiplication, with the accumulation conducted in *uint32*. For widely used elliptic curves, each element C_i has at most 23 significant bits. To elucidate, multiplying two *uint8* values yields an *uint16* result, and up to $\lceil \frac{753}{8} \rceil = 95$ such *uint16* values are accumulated, giving a result with no more than 23 significant bits. Additionally, the bases of C_i and C_{i+1} are offset by 8-bit, allowing for compaction. In fact, a $2N$-bit integer can be represented by just $\frac{N}{16}$ *uint32* elements. A conventional method suggests transferring them to memory using official APIs before compacting. However, it incurs a memory transfer overhead that is 4× the optimal.

To this end, *DistMSM* introduces a technique to compact outputs of TC on-the-fly within registers. Consider a 256-bit product for illustration, though the technique can be generalized for other bit lengths. Figure 7a shows a logical view of eight 256-bit products computed by TC, each consisting of 32 *uint32* elements. However, the actual data layout within the registers is as Figure 7b, where each thread contains two consecutive *uint32* elements. For instance, *thread0* holds C_{i0} and C_{i1}, while *thread1* holds C_{i2} and C_{i3}. Moreover, each 8 consecutive *uint32* elements are spread across 4 threads, so *thread0* can be managed to hold the 8th and 9th element (i.e.,

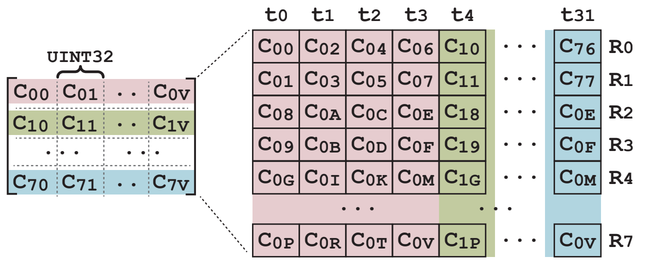

Figure 7. Data layout for tensor core's outputs

C_{i8} and C_{i9}). For efficient compaction, *DistMSM* shuffles the columns of *matB* to make each thread manage 4 consecutive elements. To illustrate, by swapping columns {2, 3, 18, 19} with {8, 9, 24, 25}, $C_{i0} \sim C_{i3}$ and $C_{iG} \sim C_{iJ}$ are all allocated to *thread0*. Then, $C_{i0} \sim C_{i3}$ can be compacted to a single integer using the formula: $\Sigma_{j=0}^{4} C_{ij} * 2^{8j}$, resulting in a 45-bit integer. After compaction, each thread holds two 45-bit integers: one from the lower segment and another from the upper segment. The reason for dividing the result into two parts is that, $m \times n$ in Algorithm 2 is a multiply-high operation. During addition, only the overflowed value is needed for the lower part, while the higher part requires a complete look-ahead addition.

The advantages of using TC for big integer multiplication extend beyond just its arithmetic throughput. Most GPUs support inter-warp context switching, allowing one warp to execute an integer multiply instruction using CUDA cores, while another warp concurrently executes a matrix-matrix multiple operation using TC. By deploying multiplication operations to different units, the total arithmetic throughput is essentially the sum of their throughput. Furthermore, although the outputs of the TC are in *uint32* format, the higher bits are consistently zero, as discussed above. It allows the *uint32* to be interpreted as the mantissa of floating-point numbers, and the additions can be executed by *float32* units. As most GPUs support concurrent execution of integer and floating-point instructions [34], the overall throughput is further amplified by incorporating the *float32* units.

Table 2. Baseline GPU implementations used for evaluation

#	Baseline	Supported Elliptic Curves
1	Bellperson[11]	BLS12-381
2	cuZK [29]	BLS12-377, BLS12-381, MNT4753
3	Icicle [21]	BN254, BLS12-377, BLS12-381
4	Mina [12]	MNT4753
5	Sppark [27]	BN254, BLS12-377, BLS12-381
6	Yrrid [39]	BLS12-377

Table 3. Execution time (in milliseconds) of *DistMSM* and the most efficient baseline

Elliptic Curve	Size	$1 \times$ A100		$8 \times$ A100		$16 \times$ A100		$32 \times$ A100	
		BG	*DistMSM*	BG	*DistMSM*	BG	*DistMSM*	BG	*DistMSM*
BN254	22	63.58^5	29.04 (2.2×)	22.91^5	4.78 (4.8×)	20.35^5	2.88 (7.0×)	9.51^5	2.04 (4.7×)
	24	218.6^5	115.1 (1.9×)	37.08^5	16.54 (2.2×)	37.17^5	8.960 (4.1×)	25.72^5	5.430 (4.7×)
	26	825.1^5	414.8 (1.9×)	113.9^5	56.15 (2.0×)	60.17^5	30.36 (1.9×)	35.51^5	17.46 (2.0×)
	28	2898^5	1578 (1.8×)	420.6^5	202.7 (2.0×)	218.2^5	103.8 (2.1×)	107.6^5	54.43 (1.9×)
BLS12 -377	22	30.07^6	52.24 (0.5×)	9.530^6	7.790 (1.2×)	7.710^6	4.480 (1.7×)	6.870^2	3.010 (2.2×)
	24	126.3^6	213.6 (0.5×)	29.84^6	30.35 (0.9×)	21.50^6	15.86 (1.3×)	17.29^2	8.750 (1.9×)
	26	517.4^6	728.8 (0.7×)	105.7^6	97.93 (1.0×)	74.55^6	51.46 (1.4×)	63.38^2	28.14 (2.2×)
	28	4165^5	2624 (1.5×)	392.2^6	334.9 (1.1×)	276.2^6	169.9 (1.6×)	174.1^5	87.47 (1.9×)
BLS12 -381	22	132.3^5	58.01 (2.2×)	76.82^5	8.520 (9.0×)	61.04^5	4.890 (12×)	33.98^5	2.950 (11×)
	24	448.6^5	234.4 (1.9×)	79.99^5	33.30 (2.4×)	97.87^5	17.43 (5.6×)	75.94^5	9.400 (8.0×)
	26	1288^5	855.2 (1.5×)	289.5^2	113.7 (2.5×)	129.1^5	59.36 (2.1×)	76.22^5	32.17 (2.3×)
	28	5038^5	3137 (1.6×)	907.1^2	399.0 (2.2×)	434.4^5	202.0 (2.1×)	281.7^2	103.4 (2.7×)
MNT 4753	22	11.7K^4	863.8 (13×)	1750^4	116.8 (14×)	970.2^4	75.62 (12×)	665.0^4	45.60 (14×)
	24	47.9K^4	4061 (11×)	5713^4	531.2 (10×)	2987^4	270.3 (11×)	1756^4	146.9 (11×)
	26	194K^4	10.8K (17×)	23.8K^4	1382 (17×)	11.3K^4	696.2 (16×)	5763^4	353.1 (16×)
	28	786K^4	38.4K (20×)	104K^4	4944 (20×)	46.0K^4	2477 (18×)	23.7K^4	1243 (19×)

5 Evaluation

The evaluation is structured into three parts. First, we assess the overall performance regarding MSM operation and end-to-end zkSNARK workloads. Second, the performance is evaluated across different GPU models. Third, we delve into a breakdown analysis of the proposed optimizations.

We compare *DistMSM* with several leading GPU MSM implementations (Table 2). The baselines are selected based on their performance and the ECs they support. For example, *Yrrid* is the champion in a renowned MSM competition [43], while Icicle supports a vast array of elliptic curves.

5.1 Overall Performance

This section presents a thorough evaluation of *DistMSM*'s performance in comparison to the baseline methods. The evaluation encompasses a variety of elliptic curves, input sizes, and GPU counts. For each test case, all compatible baseline implementations for the specific elliptic curve are benchmarked. The execution time of the most efficient baseline is represented as *BG* (*Best-GPU*), with the superscript indicating the implementation's identifier as per Table 2. For baselines without inherent multi-GPU support, we augmented them by parallelizing along the N-dim. The experiments are conducted on an Nvidia DGX system, equipped with 8 Nvidia A100 80GB GPUs and dual AMD Rome 7742 CPUs. When more than eight GPUs are needed, tasks intended for $N_{gpu}/8$ Nvidia DGX systems are executed sequentially on just one Nvidia DGX system. The longest runtime from these $N_{gpu}/8$ runs is reported as the overall execution time.

As shown in Table 3, *DistMSM* outperforms *BG* by achieving an average speedup of 6.39× for multi-GPU setups, with peaks of up to 20×. The speedup of *DistMSM* comes from two primary factors: the adaptation of Pippenger's algorithm tailored for multi-GPU systems, and the optimizations of PADD kernel to align with modern GPU microarchitectures.

Designed for multi-GPU systems, *DistMSM* shows a modest speedup on a single GPU. As the number of GPUs increases, this speedup becomes more significant. Specifically, for the first three elliptic curves, *DistMSM* achieves a 1.56× speedup on a single GPU, which rises to 3.88× with a setup of 32 GPUs. While *DistMSM*'s optimizations benefit multi-GPU setups, they may hinder performance on a single GPU. As a result, while *DistMSM* outperforms *BG* in most scenarios, it lags behind *Yrrid* for BLS12-377 when using only one GPU. When tested on an 8-GPU system, *DistMSM*'s throughput matches *Yrrid*'s. With more GPUs, *DistMSM*'s superiority becomes evident. With 16 GPUs, *DistMSM* achieves 1.54× the throughput of *Yrrid*. With 32 GPUs, not only does *Yrrid* lag behind *DistMSM*, but it also gets outpaced by *cuZK*, and *DistMSM*'s throughput is 2.17× that of *cuZK*.

The efficient PADD kernel also contributes to the speedup. This is evident when comparing *DistMSM*'s performance to *Mina* on MNT4753, which achieves an average speedup of 15.5×. However, it does not imply that *DistMSM* is uniquely efficient for MNT4753. With the same PADD algorithm, *DistMSM* takes 15.4× execution time on MNT4753 compared to BLS12-377, even though the arithmetic operations of MNT4753 are only about 12× those of BLS12-377. A key factor is the register pressure associated with MNT4753, which demands 24 registers per big integer.

Figure 8 shows the average scalability of various methods across different GPU counts. At 4 GPUs, most methods demonstrate effective scalability, averaging a 3.54× speedup. Increasing to 8 GPUs, the leading baseline method achieves

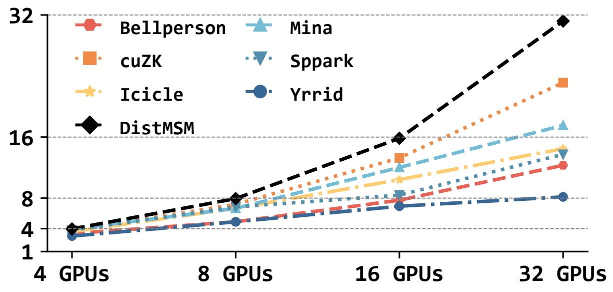

Figure 8. Speedup of multi-GPUs over single-GPU

a 7.18× speedup, while *DistMSM* slightly surpasses it with a 7.94× speedup. The scalability gap becomes more pronounced with a higher number of GPUs. Notably, *Yrrid*, despite its superior single-GPU performance, scales the least effectively. In contrast, *DistMSM* maintains near-linear scalability, especially when the input size ensures at least one PADD operation per thread. For instance, at the data point where $N = 2^{28}$, the performance on 32 GPUs is 31× that of a single GPU. These results affirm the effectiveness of optimizations tailored for multi-GPU systems.

5.1.1 End-to-End Performance of zkSNARK Workloads.

Table 4 shows the end-to-end proof generation times for various zkSNARK workloads using the BN254 curve on an Nvidia DGX system. The constraints are generated with the R1CS protocol, and the workloads span digital currency (i.e., Zcash-Sprout [6]) and verifiable machine learning (i.e., Otti-SGD [2] and Zen$_{acc}$-LeNet [10]). We incorporate the NTT implementation of Sppark [27]. It is notable that while the MSM is parallelized across 8 GPUs, the NTT is a single-GPU implementation. *DistMSM* generates proofs in the same format as those produced on CPUs, allowing for verification by *libsnark*. Due to zkSNARKs' succinct nature, both proof size and verification exhibit have $O(1)$ complexity, at 127 bytes and 1.2 *ms* on CPUs, regardless of the proving statements' complexity.

Table 4. Performance of end-to-end workloads (in seconds)

Application	Size	libsnark	DistMSM
Zcash-Sprout	2,585,747	145.8	5.8 (25.0 ×)
Otti-SGD	6,968,254	291.0	11.7 (26.7 ×)
Zen$_{acc}$-LeNet	77,689,757	5036.7	188.7 (24.9 ×)

DistMSM yields an average speedup of 25.5× over the CPU version for end-to-end proof generation, with peak performances reaching up to 26.7×. The end-to-end proof generation comprises three primary stages: MSM, NTT, and *others*. These stages account for 78.2%, 17.9%, and 3.9% of the total proof generation time, respectively. Given that *others* is not optimized via GPU acceleration, Amdahl's law reveals a theoretical average speedup of 25.6× for the overall proof generation process.

The operations categorized under *others*, such as element-wise multiplication and polynomial evaluation, are inherently suited for parallelization. These operations remain on CPUs, not due to technical limitations, but because of their minor contribution to the total computation time and the significant engineering effort needed for GPU implementation. In single-GPU setups, GPU acceleration results in substantial speedups: 871× for MSM and 898× for NTT. Given the simpler parallelization of *others* compared to MSM, similar speedups are expected for these operations. Thus, the expected time distribution would be 78.9% for MSM, 17.1% for NTT, and 3.92% for *others*. Employing 8 GPUs for MSM substantially reduces its portion of the total computation time, adjusting the distribution to: 38.1% for MSM, 50.4% for NTT, and 11.5% for *others*. In such a setup, *DistMSM* achieves an average 1.39× end-to-end speedup over *Best-GPU*, with both methods using 8 GPUs. Nonetheless, this analysis still underestimates the potential speedup, as it does not account for the possibility that NTT and *others* could also benefit from multi-GPU acceleration.

5.2 Performance on Various GPUs

This section compares the performance of *DistMSM* and *Bellperson* on three different GPUs: Nvidia A100, Nvidia RTX4090, and AMD 6900XT. *Bellperson* has an OpenCL implementation, whereas *DistMSM* is based on the HIP platform [1]. For Nvidia GPUs, the operating system is Ubuntu 20.04, equipped with CUDA Driver 530.30.02 and CUDA Runtime 12.1; for AMD 6900XT, the operating system is Ubuntu 20.04 with ROCm Toolkit [1] 5.0 installed. Figure 9 shows the execution time of both algorithms, also highlighting the hardware specifications that potentially impact the results.

As shown in Figure 9, *DistMSM* outperforms *Bellperson* by an average speedup of 16.5× on both Nvidia A100 and RTX4090 GPUs. It proves that the optimizations in *DistMSM* are versatile, effectively improving performance across different GPU models. However, the speedup is notably lower on the AMD 6900XT, with a rate of 9.4×. It can be traced back to two main reasons. First, although AMD 6900XT has similar register capabilities and memory bandwidth as its Nvidia counterparts, its integer arithmetic throughput is notably lower. As *DistMSM* prioritizes memory access optimization, it becomes more sensitive to the capabilities of integer arithmetic throughput. Secondly, the HIP platform might be less efficient than OpenCL, further contributing to this performance gap.

Additionally, both *DistMSM* and *Bellperson* show superior performance on the Nvidia RTX4090 compared to the Nvidia A100. Specifically, *DistMSM* achieves a speedup of 1.89× on the Nvidia RTX4090 in comparison to the Nvidia A100, while *Bellperson* shows a 1.61× speedup. Notably, although the Nvidia A100 outperforms the Nvidia RTX4090 in

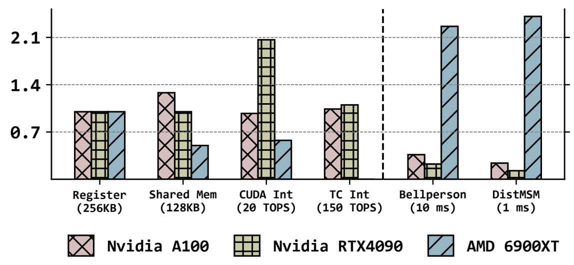

Figure 9. Execution time of *Bellperson* and *DistMSM*. The left part compares the hardware resources of the three GPUs, corresponding to the left y-axis. The right part compares the execution, corresponding to the right y-axis. The unit of the y-axis is specified in parentheses under each group.

most metrics, the Nvidia RTX4090 has a distinct advantage in its CUDA cores' integer arithmetic throughput, delivering 2.12× the capability of the Nvidia A100. Two primary insights can be drawn from these results. First, considering the rapid growth of arithmetic throughput relative to other metrics in emerging GPU models, we can reasonably anticipate further improvements in *DistMSM*'s speedup in the future. Second, although *DistMSM* has advanced by deploying some integer multiplications to tensor cores, it still requires further optimization, particularly for big integer modular multiplication.

5.3 Breakdown Analysis

This section offers an analytical breakdown of the multi-GPU Pippenger's algorithm and PADD optimizations, then presents detailed breakdowns of these two optimizations.

5.3.1 Overall Breakdown. Figure 10 shows the effectiveness of the multi-GPU Pippenger's algorithm and PADD optimizations individually. The baseline utilizes the single-GPU Pippenger algorithm without PADD optimizations, referred to as *NO-OPT*. Figure 10 also presents the calculated speedup (i.e., the product of the two optimizations' speedups) and the actual observed overall speedup. Adopting the multi-GPU Pippenger's algorithm yields significant speedups, with performance gains increasing alongside GPU counts. It highlights that employing the single-GPU Pippenger's algorithm in multi-GPU setups leads to poor performance. Moreover, *NO-OPT* shows even worse scalability compared to the baseline presented in Table 2, due to its rigid adherence to the single-GPU design.

Interestingly, the benefits of PADD optimizations diminish as the GPU count increases, and the observed overall speedup significantly exceeds the product of the two optimizations' individual speedups. This phenomenon occurs because, without the multi-GPU Pippenger's algorithm, adding more GPUs reduces the workload for *bucket-sum* but not for *bucket-reduce*. As a result, for *NO-OPT*, the proportion of PACC operations decreases with increasing GPU count, leading to a reduced impact from PACC kernel optimization. In contrast, the multi-GPU Pippenger's algorithm addresses this issue by minimizing the workload for the less parallelizable *bucket-reduce* step, ensuring most elliptic curve arithmetic operations are PACC. Thus, the two optimizations result in a synergistic effect, where the multi-GPU Pippenger's algorithm enhances the benefits of PACC kernel optimization, thereby achieving a higher overall speedup.

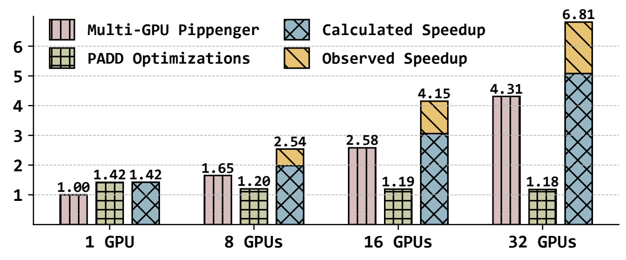

Figure 10. Breakdown of *DistMSM*'s optimizations

5.3.2 Effects of Hierarchical Bucket Scatter. Figure 11 presents a comparison between the hierarchical bucket scatter algorithm and the naive implementation. This comparison is conducted for window sizes ranging from 6 to 24. The naive approach utilizes a global atomic operation for every coefficient. In single GPU setups, where larger window sizes are often favored, the naive method is more efficient. This can be attributed to the fact that, as s increases, there is a decrease in the expected number of points in each local bucket. As a result, the overhead from the local bucket scatter often outweighs the advantages of reducing global atomic operations. Notably, when $s > 14$, shared memory is insufficient to hold the size of each bucket, leading to execution failures.

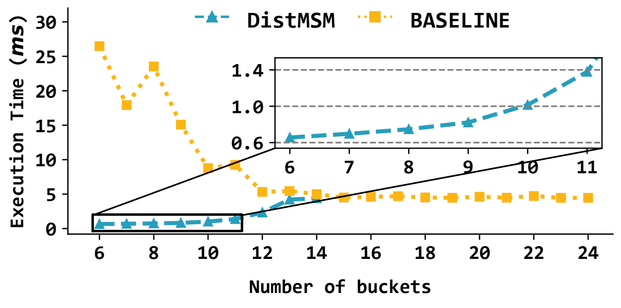

Figure 11. Execution time of the *bucket-scatter* step

On the other hand, in multi-GPU setups where the window size s is significantly smaller than 14, the hierarchical bucket scatter algorithm shows a significant improvement over the naive implementation. For example, with 16 GPUs, the optimal window size s is 11. In this scenario, the hierarchical algorithm delivers a speedup of 6.71× compared

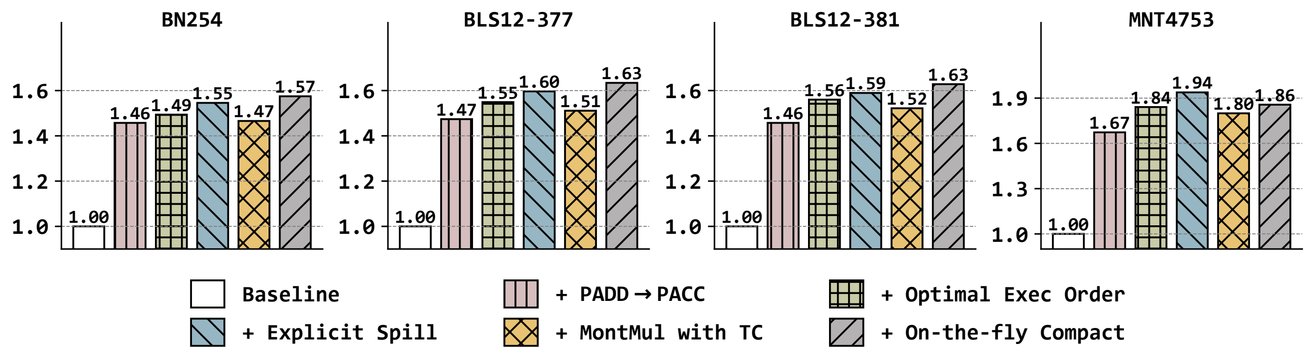

Figure 12. Performance breakdown of PADD optimizations: **"PADD→PACC"** represents switching to dedicated PACC kernels, **"Optimal Exec Order"** signifies adopting an execution sequence with the least register pressure, **"Explicit Spill"** indicates spilling registers to shared memory through explicitly integrated routines, **"MontMul with TC"** stands for deploying selected big integer multiplication to TC, and **"On-the-fly Compact"** designates compacting TC's outputs directly within registers.

to the naive method. This advantage becomes even more pronounced when the window size is decreased to 9, resulting in a remarkable 18.3× speedup. The optimization of the *bucket-scatter* step is critical in multi-GPU settings. For context, in the case mentioned earlier, the naive bucket scatter approach accounts for 16.5% of the entire MSM computation time, whereas the hierarchical bucket scatter only takes 3.6%.

5.3.3 Breakdowns of PADD optimizations.

Figure 12 provides a comprehensive evaluation of the proposed optimizations for the PADD kernel across various elliptic curves on the Nvidia A100. The experiments introduce each optimization incrementally and measure the accumulated speedup over the baseline. For example, the reported speedup for **Optimal Exec Order** not only reflects its own improvements but also includes those introduced by **PADD→PACC**.

In summary, the collective impact of the optimizations results in a speedup of 1.94× for MNT4753 and 1.61× for the other three elliptic curves. However, as previously mentioned, it does not imply that *DistMSM* is more efficient on MNT4753 than on other elliptic curves. In fact, the execution time of the PADD kernel on MNT4753 is roughly 5.2× of that on BLS12-377, even though PADD on MNT4753 requires only 4× the arithmetic operations of BLS12-377. This discrepancy can be attributed to the significantly increased register pressure when dealing with 753-bit big integers.

The dedicated PACC kernel offers the most significant speedup primarily due to two main factors. First, it reduces the number of modular multiplication operations from 14 to 10, translating to an approximate 40% acceleration. Second, the kernel decreases the peak register pressure by two big integers. It results in a 27.3% throughput improvement for MNT4753. In contrast, the other three elliptic curves experience a more modest 6.27% improvement. The significant performance boost for MNT4753 can be attributed to its handling of 753-bit big integers, which inherently incurs higher register pressure. Therefore, a reduction in register pressure is particularly advantageous for MNT4753.

Both instruction scheduling and shared memory spilling optimizations aim to address the issue of register pressure. The impact of these optimizations is also more notable for MNT4753 compared to others elliptic curves. Although each technique reduces the peak register pressure by two big integers, the mechanisms by which they achieve this reduction are distinct, resulting in varying degrees of speedup. Instruction scheduling directly reduces the number of peak concurrently live variables, offering a more significant impact on performance. In contrast, shared memory spilling temporarily saves 4 big integers to shared memory, yielding a more modest speedup. Additionally, using tensor cores for big integer multiplications offers great potential because of their impressive arithmetic throughput. However, a direct implementation reduces the performance by 6.8%, primarily due to the overhead of transferring the expanded intermediate results from tensor cores to memory. The optimization, which compacts these intermediate results within registers, yields an average improvement of 5.2% for the first three elliptic curves. However, for MNT4753, there remains a 8.2% slowdown. It is caused by the zero values introduced when representing big integers as matrices, which worsens the register pressure issue.

The speedup achieved, aside from **"PADD→PACC"**, may seem modest but is indeed satisfactory and offers practical benefits for two main reasons. Firstly, the baseline sets a high benchmark in implementation expertise by integrating most best practices and existing optimizations, thoroughly exploring conventional design spaces, and utilizing assembly-level coding for key operations. Achieving further speedups presents a significant challenge, requiring innovative strategies for even modest gains. Nevertheless, *DistMSM*'s optimizations in PADD arithmetic, even excluding **"PADD→PACC"**, achieve an average speedup of 17.8%, peaking at 27.4%. Secondly, given MSM's extensive use in proof generation and the significant computational load of each MSM, the average speedup of 17.8% leads to a reduction in GPU time by tens of seconds per proof, marking a notable practical improvement.

6 Related Work

Prior studies have proposed many novel optimizations to enhance MSM performance on GPUs. The winners of the ZPrize competition [29, 39, 43] are the most efficient single GPU implementations for BLS12-377. These implementations employ techniques such as precomputation, signed digits, pipelining, and lazy Montgomery reduction, many of which are also adopted by *DistMSM*. GZKP [30] introduces an innovative point-merging algorithm to reduce redundant operations and uses floating-point units for modulus estimation. cuZK [29] implements an efficient parallel Pippenger's algorithm through sparse matrix multiplication, and it offers an efficient distribution of subtasks across multiple GPUs, demonstrating near-linear scalability for up to 8 GPUs. However, as the number of GPUs increases, bottlenecks shift from compute-intensive parts to non-parallel steps. To this end, *DistMSM* redesigns Pippenger's algorithm for scalability.

Optimizing elliptic curve cryptography with GPUs is well-studied [3, 14, 35]. *DistMSM* focuses on pairing-friendly elliptic curves and optimizes the PADD operation in the *XYZZ* coordinate system by reducing register pressure. Another body of previous works optimizes Montgomery modular multiplication on GPUs [24, 37, 42]. *DistMSM* augments them by introducing a set of techniques for efficiently deploying big integer multiplication to tensor cores.

Many studies have improved ZKP performance through hardware acceleration, primarily focusing on two key operations: MSM and NTT. Leading commercial ZKP systems, including Bellperson [11] and Mina [12], utilize GPUs to parallelize these operations. Recent studies have introduced many novel optimizations. DIZK [38] is a distributed ZKP system designed for CPU clusters. Ray et al. use FPGAs to accelerate ZKP tasks [36], specifically with BLS12-377, by adopting an optimized memory structure and a specialized pipeline to achieve top-tier performance. PipeZK [40], a dedicated *zkSNARK* accelerator, presents an innovative data flow and dynamic task dispatcher for MSM and NTT. Cysic [7] employs around 40 high-end FPGAs to accelerate proof generation, setting the benchmark for multi-FPGA setups. Notably, scaling strategies differ between multi-FPGA and multi-GPU systems: the former prioritizes pipelining, while the latter emphasizes parallelism.

7 Conclusion

In this study, we presented *DistMSM*, a tailored MSM algorithm designed for distributed multi-GPU systems. *DistMSM* identifies and addresses the performance bottlenecks that arise when scaling MSM to multi-GPU systems. Additionally, it features an elliptic curve arithmetic kernel optimized for modern GPU architectures. Comprehensive evaluations demonstrate its superior efficiency, with an average 6.39× throughput compared to the *Best-GPU*. Notably, on a 32-GPU system, *DistMSM* reduces the execution time for a task that required 2.6 *s* on a single GPU to just 87.5 *ms*. These results highlight multi-GPU systems' significant potential and efficiency for ZKP acceleration.

Acknowledgments

We appreciate our shepherd and the anonymous reviewers for their constructive comments and suggestions. This work is supported by the Natural Science Foundation of China grant No. 62372272, Department of Science & Technology of Shandong Province grant No. SYS202201, Quan Cheng Laboratory grant No. QCLZD202302, Taishan Scholars Program No. tsqn202211281.

References

[1] Inc Advanced Micro Devices. Amd rocm open software platform. https://rocm.docs.amd.com, 2023.

[2] Sebastian Angel, Andrew J Blumberg, Eleftherios Ioannidis, and Jess Woods. Efficient representation of numerical optimization problems for snarks. In *31st USENIX Security Symposium*, 2022.

[3] Samuel Antao, Jean-Claude Bajard, and Leonel Sousa. Elliptic curve point multiplication on gpus. In *ASAP 2010-21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 192–199. IEEE, 2010.

[4] Gautam Botrel and Youssef El Housni. Faster montgomery multiplication and multi-scalar-multiplication for snarks. 2023.

[5] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology—ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings*, pages 51–65. Springer, 1998.

[6] ZCash Crop. Zcash is cash for the new age. https://z.cash, 2023.

[7] Cysic. Hardware accelerating zero-knowledge proofs. http://cysic.xyz, 2023.

[8] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In *First ACM workshop on Language support for privacy-enhancing technologies*, pages 27–30, 2013.

[9] Marwa Elteir, Heshan Lin, and Wu-chun Feng. Performance characterization and optimization of atomic operations on amd gpus. In *2011 IEEE International Conference on Cluster Computing*, 2011.

[10] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive*, 2021.

[11] Filecoin. bellperson: Gpu parallel acceleration for zk-snark. https://github.com/filecoin-project/bellperson, 2023.

[12] Mina Foundation. Gpu groth16 prover (3x faster than cpu). https://github.com/MinaProtocol/gpu-groth16-prover-3x, 2023.

[13] Hisham S Galal and Amr M Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *Financial Cryptography and Data Security: FC 2018 International Workshops*, pages 265–278. Springer, 2019.

[14] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. Dpf-ecc: accelerating elliptic curve cryptography with floating-point computing power of gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium*, 2020.

[15] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *43th annual ACM symposium on Theory of computing*, pages 99–108, 2011.

[16] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.

[17] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. COMPUT*, 18(1):186–208, 1989.

[18] Yinjie Gong, Yifei Jin, Yuchan Li, Ziyi Liu, and Zhiyi Zhu. Analysis and comparison of the main zero-knowledge proof scheme. In *2022 International Conference on Big Data, Information and Computer Network*, pages 366–372. IEEE, 2022.

[19] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Applied Cryptography and Network Security: Third International Conference*, pages 467–482. Springer, 2005.

[20] Jens Groth. On the size of pairing-based non-interactive arguments. In *35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[21] Icicle. a gpu library for zero-knowledge acceleration. https://github.com/ingonyama-zk/icicle, 2023.

[22] Immutable. Immutable x: powering the next generation of web3 games. https://www.immutable.com/products/immutable-x, 2023.

[23] C Kaya Koc, Tolga Acar, and Burton S Kaliski. Analyzing and comparing montgomery multiplication algorithms. IEEE, 1996.

[24] Karl Leboeuf, Roberto Muscedere, and Majid Ahmadi. A gpu implementation of the montgomery multiplication algorithm for elliptic curve cryptography. In *2013 IEEE International Symposium on Circuits and Systems*, pages 2593–2596. IEEE, 2013.

[25] Honglei Li and Weilian Xue. A blockchain-based sealed-bid e-auction scheme with smart contract and zero-knowledge proof. *Security and Communication Networks*, 2021:1–10, 2021.

[26] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[27] Supranational LLC. Zero-knowledge template library. https://github.com/supranational/sppark, 2023.

[28] Loopring. zkrollup layer 2 for trading and payment. https://loopring.org, 2023.

[29] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *Cryptology ePrint Archive*, 2022.

[30] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. Gzkp: A gpu accelerated zero-knowledge proof system. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023.

[31] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[32] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[33] Ning Ni and Yongxin Zhu. Enabling zero knowledge proof by accelerating zk-snark kernels on gpu. *Journal of Parallel and Distributed Computing*, 173:20–31, 2023.

[34] NVIDIA. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[35] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. An efficient elliptic curve cryptography signature server with gpu acceleration. *IEEE Transactions on Information Forensics and Security*, 12(1):111–122, 2016.

[36] Andy Ray, Ben Devlin, Fu Yong Quah, and Rahul Yesantharao. High performance, open source cryptographic solutions for large scale number theoretic transforms and multi-scalar multiplications in hardcaml. https://github.com/fyquah/hardcaml_zprize, 2023.

[37] Nicolae Roşia, Virgil Cervicescu, and Mihai Togan. Efficient montgomery multiplication on gpus. In *International Conference for Information Technology and Communications*. Springer, 2015.

[38] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium*, pages 675–692, 2018.

[39] Yrrid. https://www.yrrid.com, 2023.

[40] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, pages 416–428. IEEE, 2021.

[41] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy*, pages 863–880. IEEE, 2017.

[42] Kaiyong Zhao. Implementation of multiple-precision modular multiplication on gpu. In *GPU Technology Conference*, 2009.

[43] Zprize. Accelerating the future of zero knowledge cryptography. https://www.zprize.io, 2023.