HOFSTRA UNIVERSITY

MASTERS THESIS

Scaling Zero-Knowledge: Comparative MSM Acceleration Across CPU, GPU, and HPC Platforms

---

Author:
Raham Butt

Supervisor:
Dr. Xiang Fu

A thesis submitted in fulfillment of the requirements
for the degree of Masters of Science

in the

Department of Computer Science

May 13, 2026

Declaration of Authorship

I, Raham Butt, declare that this thesis titled, "Scaling Zero-Knowledge: Comparative MSM Acceleration Across CPU, GPU, and HPC Platforms" and the work presented in it are my own. I confirm that: [1][2]

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

[1] Template taken from: https://www.overleaf.com/latex/templates/template-for-a-masters-slash-doctoral-thesis/mkzrzktcbzfl

[2] The committee page was adapted from: https://www.overleaf.com/latex/templates/mit-thesis- template/sgzcswxftpwx

HOFSTRA UNIVERSITY

Abstract

Department of Computer Science

Masters of Science

Scaling Zero-Knowledge: Comparative MSM Acceleration Across CPU, GPU, and HPC Platforms

by Raham Butt

Zero-knowledge proofs (ZKPs) have moved from theoretical cryptography into production systems for blockchain scalability, privacy-preserving identity, and verifiable computation, but their practical deployment is constrained by the cost of proof generation. Within modern proof systems, multi-scalar multiplication (MSM) on elliptic curve groups consumes between 70 and 90 percent of total prover runtime, making it the dominant target for hardware acceleration.

This thesis presents a comparative performance analysis of MSM acceleration across three hardware tiers: a CPU baseline, a single consumer-grade GPU (NVIDIA RTX 4080), and a multi-GPU HPC cluster (Hofstra's StarHPC, with eight-device A100 and H100 nodes). The implementation targets the Pallas elliptic curve and integrates with the Nova folding-scheme prover through a feature-gated substitution at Nova's commitment trait, leaving the upstream codebase otherwise unmodified. The GPU implementation evolves through four design points: a shared-memory spinlock baseline whose performance is structurally bounded by an SM occupancy ceiling of 16.7%; a sort-based Pippenger redesign using CUB radix sort with 12-bit signed-digit windows that lifts buckets out of shared memory entirely; partition-based concurrent streams that overlap memory transfers with kernel execution; and a batch API exposing proof-system-level concurrency to the device. The multi-GPU extension partitions input across all visible devices, runs an independent sort-based pipeline per device, and merges partial results on the host without inter-device communication.

Empirical results across input sizes from 100,000 to 200 Million scalar–point pairs yield platform selection guidelines mapping input size to optimal hardware tier, demonstrating that memory-architecture awareness, rather than raw parallelism, governs efficient ZKP acceleration on GPUs.

Acknowledgements

I would like to begin by expressing my sincere gratitude to my advisor, Dr.Xiang Fu, for his guidance and support throughout this project. His insight, patience, and thoughtful suggestions shaped the direction of the work at every stage, and this thesis would not have reached completion without him.

I am also grateful to Dr.Krishnan Pillaipakkamnatt and Dr.Jianchan Shan for serving on my thesis committee, and for the time and attention they have given to reviewing this work. My thanks also go to the Department of Computer Science at Hofstra University for providing the resources that made this project possible, in particular access to the StarHPC cluster on which the multi-GPU experiments of this thesis were conducted.

To my friends, thank you for the encouragement and companionship that have sustained me through the long stretches of this degree. Finally, I owe my deepest gratitude to my family. Without their love and unwavering support, I would not have come this far.

THESIS COMMITTEE

THESIS SUPERVISOR

Xiang Fu
Professor of Computer Science
Department of Computer Science

THESIS READERS

Krishnan Pillaipakkamnatt
Professor of Computer Science
Department of Computer Science

Jianchan Shan
Associate Professor of Computer Science
Department of Computer Science

Contents

Declaration of Authorship **i**

Abstract **ii**

Acknowledgements **iii**

Thesis Committee **iv**

Contents **v**

List of Figures **ix**

List of Tables **x**

List of Abbreviations **xi**

1 Introduction **1**
1.1 What are Zero-Knowledge Proofs? 1
1.2 Motivation for Zero-Knowledge Proofs 2
1.3 Applications of Zero-Knowledge Proofs 3
1.4 Research Objectives 3

2 Background and Related Work **5**
2.1 The Nova Folding Scheme 5
2.2 Multi-Scalar Multiplication 6
2.2.1 Multi-Scalar Multiplication as the Dominant Bottleneck 7
2.3 The Pallas/Vesta Cycle of Curves 7
2.4 Prior Hardware Acceleration Approaches 8
2.4.1 FPGA-Based Approaches 8
2.4.2 GPU-Based Approaches 9

3 The Nova Framework and Target Platforms **11**
3.1 The Nova Codebase 11
3.2 The Provider Abstraction 12
3.3 The Pallas Provider and Its Commitment Code Path 13

3.4 The Upstream CPU MSM Implementation . . . 14
3.5 Extension Points for Hardware Acceleration . . . 15
3.6 Target Hardware Platforms . . . 15
3.6.1 CPU Baselines . . . 15
3.6.2 Consumer GPU: NVIDIA GeForce RTX 4080 . . . 16
3.6.3 HPC GPUs: NVIDIA A100 and H100 . . . 16

4 GPU Acceleration of MSM – Design and Implementation 18
4.1 GPU Architecture and the CUDA Programming Model . . . 18
4.1.1 Execution Hierarchy: SMs, Warps, and SIMT . . . 19
4.1.2 Memory Hierarchy . . . 19
4.1.3 Streams, Atomics, and Concurrency . . . 20
4.1.4 Occupancy and the Role of Resource Pressure . . . 20
4.2 Arithmetic Foundations for Pallas MSM . . . 21
4.2.1 Montgomery Form and Where It Is Used . . . 21
4.2.2 Pallas Point Addition in Projective Coordinates . . . 21
4.2.3 Signed-Digit Recoding and the 12-Bit Window . . . 22
4.2.4 The Four-Phase Pippenger Pipeline . . . 23
4.3 Rust–CUDA Integration . . . 23
4.3.1 Three-Layer Design . . . 24
4.3.2 The FFI Boundary and Data Conversion . . . 24
4.3.3 Device Memory Management . . . 25
4.3.4 Concurrency Model at the FFI Layer . . . 26
4.4 V1: Shared-Memory Spinlock Kernel . . . 27
4.4.1 Design and Working Set . . . 27
4.4.2 Why V1 Was Retired . . . 28
4.5 V2: Sort-Based Pippenger . . . 29
4.5.1 Design Overview . . . 29
4.5.2 Scalar Recoding and Pair Generation . . . 31
4.5.3 The CUB Radix Sort . . . 32
4.5.4 Boundary Finding . . . 33
4.5.5 Warp-Per-Bucket Accumulation . . . 33
4.5.6 Parallel Bucket Reduction . . . 35
4.5.7 Internal Evolution of V2 . . . 36
4.6 V3: Partition-Based Concurrent Streams . . . 37
4.6.1 Design Overview . . . 38
4.6.2 Per-Slot Resource Layout . . . 39
4.6.3 Per-Partition Dispatch and Stream Concurrency . . . 40
4.6.4 Partial Result Merge . . . 41

4.6.5 Partition Count Selection . . . 41
4.7 V4: Batch API and Multi-GPU Dispatch . . . 42
4.7.1 Motivation: Inter-MSM and Inter-Device Concurrency . . . 43
4.7.2 Batch API for Nova-Level Concurrency . . . 43
4.7.3 Multi-GPU Dispatch . . . 44
4.7.4 Host-Side Projective Merge and Parallel Conversion . . . 46
4.7.5 Summary . . . 47
4.8 Architectural Comparison . . . 47

5 Experimental Setup **50**
5.1 Hardware Platforms . . . 50
5.1.1 Local Workstation . . . 50
5.1.2 StarHPC Cluster . . . 51
5.2 Software Stack and Build Configuration . . . 51
5.2.1 Toolchains and Versions . . . 51
5.2.2 Build Targets and Feature Flags . . . 52
5.2.3 SLURM Job Configuration . . . 53
5.3 Benchmark Methodology . . . 54
5.3.1 Input Generation and Reproducibility . . . 54
5.3.2 Warmup, Averaging, and Correctness Verification . . . 54
5.3.3 Per-Phase Timing Instrumentation . . . 56
5.3.4 What Is and Is Not Measured . . . 56

6 Results and Analysis **58**
6.1 Local GPU Results (RTX 4080) . . . 58
6.1.1 CPU vs. GPU Speedup Across Input Sizes . . . 58
6.1.2 Per-Phase Bottleneck Analysis . . . 59
6.2 Optimisation Impact Breakdown . . . 61
6.2.1 Contribution of Each Design Stage . . . 61
6.2.2 Partition Count Sensitivity . . . 62
6.3 Cluster Results (A100 & H100) . . . 62
6.3.1 Single-Device Cluster Timings . . . 62
6.3.2 Cluster Partition-Count Sweep . . . 63
6.4 Multi-GPU Scaling . . . 64
6.4.1 Headline V4 Performance Across the Full Input Range . . . 64
6.4.2 Strong Scaling at Fixed Input Size . . . 66
6.4.3 Weak Scaling and the Multi-GPU Threshold . . . 67
6.5 Cross-Platform Comparison and Platform Selection Guidelines . . . 68

7 Conclusion **70**
7.1 Summary of Contributions 70
7.2 Limitations and Future Work 71

Bibliography **74**

List of Figures

3.1 Pallas commitment code path in upstream Nova 14

4.1 V2 sort-based Pippenger dataflow 30

6.1 RTX 4080 wall-clock and speedup curves 60
6.2 V4 8-GPU speedup curve 65
6.3 CPU vs. 8× A100 wall-clock 66
6.4 V4 multi-GPU strong scaling 67

List of Tables

3.1 Target GPU platforms 17

5.1 Software toolchain versions 52

6.1 RTX 4080 CPU vs. V2 vs. V3 59
6.2 Per-phase breakdown at $n = 1\text{M}$ 60
6.3 Cumulative V2→V3→V4 speedup at $n = 10\text{M}$ 61
6.4 RTX 4080 K-sweep 62
6.5 Single-device cluster timings 63
6.6 Cluster K-sweep, single A100 64
6.7 V4 headline: $1\times$ vs. $8\times$ A100 65
6.8 V4 strong scaling on A100 66
6.9 V4 on $8\times$ A100 at largest sizes 68
6.10 Platform selection guidelines 68

List of Abbreviations

ZKP	**Z**ero **K**nowledge **P**roof
zk-SNARK	**Z**ero-**K**nowledge **S**uccinct **N**on-Interactive **A**rgument of **K**nowledge
IVC	**I**ncrementally **V**erifiable **C**omputation
R1CS	**R**ank-**1** **C**onstraint **S**ystem
MSM	**M**ulti-**S**calar **M**ultiplication
IPA	**I**nner **P**roduct **A**rgument
ECC	**E**lliptic **C**urve **C**ryptography
CIOS	**C**oarsely **I**ntegrated **O**perand **S**canning
GPU	**G**raphics **P**rocessing **U**nit
CPU	**C**entral **P**rocessing **U**nit
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HPC	**H**igh **P**erformance **C**omputing
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
SM	**S**treaming **M**ultiprocessor
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hread
HBM	**H**igh **B**andwidth **M**emory
SXM	NVIDIA's **S**erver e**X**tension **M**odule (mezzanine GPU form factor)
CUB	**C**UDA **U**nbound (library)
D2H	**D**evice-**to**-**H**ost (memory transfer)
H2D	**H**ost-**to**-**D**evice (memory transfer)
OOM	**O**ut **O**f **M**emory (CUDA `cudaErrorMemoryAllocation`)
API	**A**pplication **P**rogramming **I**nterface
ABI	**A**pplication **B**inary **I**nterface
FFI	**F**oreign **F**unction **I**nterface
MPI	**M**essage **P**assing **I**nterface
NCCL	**N**VIDIA **C**ollective **C**ommunications **L**ibrary
SLURM	**S**imple **L**inux **U**tility for **R**esource **M**anagement
PCIe	**P**eripheral **C**omponent **I**nterconnect **E**xpress
BRAM	**B**lock **R**andom **A**ccess **M**emory

To my family...

Chapter 1

Introduction

This chapter establishes the foundational context for understanding zero-knowledge proof systems and the need for their acceleration. Section 1.1 introduces zero-knowledge proofs and their formal properties. Section 1.2 discusses the motivation for hardware-accelerated ZKP systems. Section 1.3 surveys the primary applications driving ZKP adoption. Finally, Section 1.4 presents the research objectives and outlines the structure of this thesis.

1.1 What are Zero-Knowledge Proofs?

A Zero-Knowledge Proof (ZKP) is a cryptographic protocol that enables a prover to convince a verifier of the truthfulness of a mathematical statement while revealing no information beyond the validity of the statement itself. This concept can be illustrated through a common analogy: authenticating a password on a website without ever transmitting the password to the server. ZKPs constitute an integral component of what the National Institute of Standards and Technology (NIST) designates as Privacy-Enhancing Cryptography (PEC) [1].

ZKPs are characterized by three formal properties, which serve as foundational pillars for any valid zero-knowledge protocol [2]. The first property is *completeness*: if a statement is true, an honest prover can convince an honest verifier of its validity. The second property is *soundness*: a cheating prover cannot convince a verifier of a false statement, except with negligible probability. The third and most critical property is *zero-knowledge*: if the statement is true, the verifier learns nothing beyond the fact that the statement is true. This property implies that any computation the verifier might perform to validate the statement could be executed without any interaction with the prover.

Zero-knowledge proofs are not a recent development; they were first introduced by Goldwasser, Micali, and Rackoff in 1985 [2]. Goldreich, Micali, and Wigderson subsequently demonstrated that zero-knowledge proofs exist for all languages in NP, establishing the broad theoretical applicability of the paradigm [3]. However, they remained

largely theoretical for decades due to computational constraints. In recent years, the technology has experienced a significant surge in popularity, interest, and adoption, driven primarily by blockchain scalability requirements and growing privacy demands. The global zero-knowledge proof market was valued at approximately USD 1.28 billion in 2024 and is projected to reach USD 7.59 billion by 2033, representing a compound annual growth rate of 22.1% [4].

1.2 Motivation for Zero-Knowledge Proofs

While zero-knowledge proof systems have become essential building blocks for modern cryptographic applications, the computational complexity of proof generation remains a significant barrier to widespread adoption. ZKP operations involve extensive large-integer arithmetic, with elliptic-curve operations employing integers of 254–256 bits, as seen in curves such as BN254 [5] and the Pasta cycle [6]. These operations are fundamentally mismatched with general-purpose CPUs optimized for 64-bit word sizes, creating a computational bottleneck that dominates proof generation time. In particular, Multi-Scalar Multiplication (MSM) operations typically consume 70–90% of total proof generation runtime [7], [8].

This performance gap between theoretical capabilities and practical deployment has motivated the development of hardware acceleration solutions. Among the available platforms, Graphics Processing Units (GPUs) have emerged as a particularly attractive target: their massive thread-level parallelism is well matched to the structurally regular bulk-arithmetic profile of MSM, and their broad availability, from consumer cards to datacentre accelerators, makes them accessible across a wide range of deployment scenarios. The same characteristics that make GPUs effective on a single device also extend to multi-device clustering, where commitments over very large input sets can be partitioned across multiple accelerators within a high-performance computing (HPC) node.

Despite the availability of these acceleration paths, a gap exists in systematic empirical comparison across the spectrum of GPU-class hardware. Prior work has evaluated individual platforms in isolation [7], [9]–[11], but comprehensive benchmarking with consistent workloads spanning a single CPU baseline, a consumer GPU, and an HPC-class multi-GPU cluster, remains lacking. This gap motivates the comparative analysis presented in this thesis, which aims to provide empirical guidance for hardware platform selection in ZKP applications.

1.3 Applications of Zero-Knowledge Proofs

Although ZKPs have existed for several decades, the most substantial development in the field has occurred within the last decade, largely attributable to the rise of cryptocurrencies. Primarily, ZKPs are employed to facilitate secure and privacy-preserving transactions through technologies such as zk-SNARKs [12], which generate proofs of transaction validity while preserving the integrity and privacy of both transacting parties. A prominent application is Zcash, which employs zk-SNARKs to enable fully private cryptocurrency transactions [13]. The use of zk-SNARKs also enables scalability solutions such as zk-rollups, which aggregate multiple transactions into a single proof for on-chain verification [14].

Beyond cryptocurrency applications, ZKPs have found utility in several other domains. In identity verification, ZKPs enable users to prove attributes about themselves, such as age or citizenship, without revealing the underlying personal data, a capability increasingly formalized through frameworks such as Verifiable Credentials [15]. In supply chain management, organizations can prove compliance with regulations without exposing proprietary business information. In voting systems, ZKPs can ensure ballot integrity while preserving voter privacy.

Zero-knowledge virtual machines (zkVMs) represent an emerging application area that enables general-purpose verifiable computation [16]. Systems such as Nexus zkVM have demonstrated significant performance improvements through algorithmic innovations, underscoring the potential for optimization through both software and hardware acceleration. These diverse applications highlight the growing importance of efficient ZKP implementations and the need for hardware acceleration to make such systems practical at scale.

1.4 Research Objectives

This thesis presents a comparative performance analysis of hardware acceleration platforms for zero-knowledge proof systems. The primary objective is to provide empirical insights into the trade-offs between latency, throughput, and scalability for Multi-Scalar Multiplication (MSM) acceleration across CPU, single-GPU, and multi-GPU HPC platforms.

The specific research objectives are as follows:

1. Implement an MSM kernel for the Pallas elliptic curve [6] that integrates cleanly with the Nova folding-scheme prover [17], using Montgomery modular multiplication [18] as the underlying field-arithmetic primitive.

2. Benchmark this implementation against Nova's upstream CPU MSM across input sizes ranging from 100 thousand to 200 Million scalar–point pairs, on both a local workstation and an HPC compute node.

3. Extend the single-GPU implementation to a multi-device configuration on Hofstra's StarHPC cluster, and evaluate it on eight-GPU nodes equipped with NVIDIA A100 and H100 accelerators.

4. Characterize platform performance under systematically varied input sizes and partition parameters, identifying the architectural and algorithmic factors that govern speedup at each tier.

5. Develop platform selection guidelines that map application requirements — input size, batch structure, and deployment context — to the appropriate hardware choice.

The remainder of this thesis is organized as follows. Chapter 2 provides background on the Nova folding scheme, the Pasta cycle of curves, the Pippenger MSM algorithm, and prior GPU acceleration work for ZKP systems. Chapter 3 describes the Nova framework as a software artifact: the provider abstraction at which hardware acceleration is inserted, the upstream CPU MSM that serves as the performance baseline, and the target hardware platforms used throughout the evaluation. Chapter 4 presents the GPU implementation, including the CUDA programming model, the underlying field and curve arithmetic, and the architectural progression from a shared-memory baseline kernel through sort-based Pippenger and multi-GPU scaling. Chapter 5 describes the experimental setup, including the SLURM job configuration, benchmark methodology, and verification harness. Chapter 6 presents results and analysis across all platforms. Finally, Chapter 7 concludes the thesis and discusses future work.

Chapter 2

Background and Related Work

This chapter establishes the technical foundations on which the remainder of the thesis builds. Section 2.1 introduces the Nova folding scheme and identifies the recursive argument structure that motivates the curves and operations targeted by this work. Section 2.2 formalises multi-scalar multiplication (MSM) and explains, at the algorithmic level, why it dominates proof generation time in folding-based systems. Section 2.3 describes the Pallas/Vesta cycle of curves, which Nova uses to instantiate its recursion efficiently and which defines the arithmetic substrate of the GPU implementation presented in later chapters. Section 2.4 surveys prior hardware acceleration efforts on FPGAs and GPUs, positioning the contribution of this thesis within the existing literature.

2.1 The Nova Folding Scheme

Nova, introduced by Kothapalli, Setty, and Tzialla [17], is a recursive argument system for incrementally verifiable computation (IVC), a primitive first formalised by Valiant [19]. Unlike SNARK-based recursion, which requires a full proof to be generated and verified at every step of the computation, Nova defers succinct proving to the end and instead *folds* successive computation steps into a single accumulating instance. This substitution of folding for in-circuit verification is the central source of Nova's efficiency: each recursive step costs roughly two group scalar multiplications rather than a full SNARK verification circuit.

At the core of the construction is a generalisation of the Rank-1 Constraint System (R1CS) known as *relaxed R1CS*. A standard R1CS instance over a prime field $\mathbb{F}_p$ is specified by matrices $A, B, C \in \mathbb{F}_p^{m \times n}$ and is satisfied by a witness $z \in \mathbb{F}_p^n$ when

$$(Az) \circ (Bz) \;=\; Cz,$$

where $\circ$ denotes the Hadamard (entry-wise) product. Relaxed R1CS introduces a scalar $u \in \mathbb{F}_p$ and an error vector $E \in \mathbb{F}_p^m$, yielding the relation

$$(Az) \circ (Bz) \;=\; u \cdot (Cz) \;+\; E.$$

The additional slack allows two satisfying instances to be linearly combined into a third satisfying instance with a new error term, which is precisely what folding exploits.

A Nova folding step operates on two *committed* relaxed R1CS instances, each consisting of a Pedersen commitment to the witness z and to the error vector E, together with public inputs. Given instances (U_1, W_1) and (U_2, W_2) and a Fiat–Shamir challenge r, the prover computes a cross-term commitment and outputs a folded instance (U, W) that is satisfying if and only if both inputs were satisfying, except with negligible probability [17]. Because each folded instance retains the same size as a single input instance, the prover's per-step state remains constant regardless of the number of computation steps already folded.

The folding operation is cheap in absolute terms, but its dominant cost, and the cost that scales with circuit size, is the computation of the Pedersen commitments required to represent witnesses and cross terms as group elements. Each commitment of the form $\mathrm{Com}(z) = \sum_{i=0}^{n-1} z_i \cdot G_i$, where $\{G_i\}$ is a public set of generators, is a multi-scalar multiplication of length equal to the witness vector. For non-trivial circuits, n ranges from 2^{20} to 2^{27} or more, placing MSM at the centre of Nova's performance profile.

2.2 Multi-Scalar Multiplication

Given a vector of scalars $\mathbf{s} = (s_0, s_1, \ldots, s_{n-1}) \in \mathbb{F}_p^n$ and a vector of elliptic curve points $\mathbf{P} = (P_0, P_1, \ldots, P_{n-1}) \in \mathbb{G}^n$, the multi-scalar multiplication problem asks for

$$R \;=\; \sum_{i=0}^{n-1} s_i \cdot P_i.$$

A naive evaluation performs n independent scalar multiplications followed by $n-1$ point additions, giving complexity $\mathcal{O}(n\lambda)$ point operations for λ-bit scalars. This is prohibitively expensive for the vector lengths required by Nova.

The standard optimisation is Pippenger's bucket method [20]. Each λ-bit scalar is decomposed into $\lceil \lambda / c \rceil$ windows of c bits. For each window, the algorithm maintains 2^c buckets indexed by the window value; each input point is accumulated into the bucket whose index matches its scalar digit. A summation-by-parts pass then reduces each window's buckets into a single window sum, and the window sums are finally combined using Horner's method. The total cost is dominated by bucket accumulation and scales as $\mathcal{O}\left(\frac{n\lambda}{\log n}\right)$ point additions, an asymptotic improvement that becomes decisive for n in the millions.

For 256-bit scalars, a window size of $c = 12$ yields $\lceil 256/12 \rceil = 22$ windows with $2^{12} = 4096$ buckets per window; applying signed-digit recoding halves the effective bucket count to 2048, which matches the configuration used in the implementation described in Chapter 4. The bucket method exposes two forms of parallelism, across

windows and across buckets within a window, that are essential for efficient hardware implementations [21].

2.2.1 Multi-Scalar Multiplication as the Dominant Bottleneck

As noted in Section 1.2, empirical profiling of modern proof systems attributes 70–90% of end-to-end proving time to MSM [7], [9]. This chapter refines that observation by identifying the underlying causes, which in turn motivate the design choices made in the GPU implementation.

First, MSM is *arithmetically heavy but structurally regular*. Each bucket addition is an elliptic curve point addition over a 256-bit prime field, and each such addition decomposes into on the order of a dozen modular multiplications in the base field. A single modular multiplication on a 256-bit prime requires a full Montgomery reduction [18], which in turn expands to approximately $2k^2 + k$ word-level multiply-accumulate operations for k-word operands. On a 64-bit CPU with $k = 4$, this amounts to tens of word operations per modular multiplication and hundreds per point addition, and MSM performs billions of them. The workload is thus a long sequence of narrow-integer operations with highly regular data flow, a profile that general-purpose CPUs handle poorly but that massively parallel accelerators are well suited to exploit [22].

Second, the workload exhibits *limited temporal locality but abundant task parallelism*. Successive points in the input stream touch unrelated buckets, defeating conventional caches; however, partitioning the input into independent subsets enables straightforward data-parallel execution across cores, streams, or devices. This property is what makes both multi-stream single-GPU execution and multi-GPU scaling tractable for MSM, as explored in Chapters 3 and 4.

Third, for folding schemes specifically, MSM length grows linearly with the size of the folded computation, whereas the rest of the Nova prover remains effectively constant per step. As circuit sizes scale beyond 2^{20} constraints, MSM's share of total runtime approaches unity, and any improvement to proving latency or throughput is necessarily an improvement to MSM.

2.3 The Pallas/Vesta Cycle of Curves

Nova's recursion relies on being able to verify, inside an arithmetic circuit, the elliptic curve operations used to form commitments. A standard construction instantiates the prover's commitments on a curve E_1 defined over a base field $\mathbb{F}_p$ whose scalar field is $\mathbb{F}_q$. Verifying an E_1 commitment inside a circuit requires that circuit's native field to be $\mathbb{F}_p$, that is, the circuit must itself be arithmetised over the field in which E_1's coordinates

live. Recursing back then requires a second curve E_2 whose base field is $\mathbb{F}_q$ and whose scalar field is $\mathbb{F}_p$, so that E_2-commitments can in turn be verified inside an $\mathbb{F}_q$-circuit.

A pair of curves (E_1, E_2) with this property is known as a *cycle of curves*. The construction originates with the Halo proof system of Bowe, Grigg, and Hopwood [23], which first exploited a cycle to eliminate the trusted setup required by earlier recursive SNARKs. The Pallas and Vesta curves, specified as part of the Halo 2 toolchain [6], form such a cycle: both are Weierstrass curves of the form $y^2 = x^3 + 5$ with prime order, and both expose efficient field arithmetic near the 256-bit word boundary. Concretely, Pallas is defined over the field

$$p \;=\; 2^{254} + 45560315531419706090280762371685220353,$$

and Vesta is defined over the field

$$q \;=\; 2^{254} + 45560315531506369815346746415080538113,$$

with the scalar field of Pallas equal to $\mathbb{F}_q$ and the scalar field of Vesta equal to $\mathbb{F}_p$ [6].

Two properties make the Pasta cycle attractive for Nova [24]. First, neither curve is pairing-friendly, so there is no low-embedding-degree security concern, and both achieve approximately 128-bit classical security. Second, both fields admit efficient Montgomery reduction with small, structured moduli, which simplifies modular arithmetic in both software and hardware implementations. The Nova reference implementation uses the Pasta cycle for precisely these reasons, and the GPU acceleration presented in this thesis targets Pallas as the commitment curve on which MSM is performed. Chapter 3 details the Montgomery arithmetic layer specific to Pallas and shows how the underlying field structure influences kernel design.

2.4 Prior Hardware Acceleration Approaches

The performance pressure created by MSM has driven a sustained line of research on hardware acceleration for zero-knowledge proof systems. This section surveys the two platforms that have received the most attention, FPGAs and GPUs, and identifies where the present work extends the state of the art.

2.4.1 FPGA-Based Approaches

Field Programmable Gate Array implementations have targeted MSM acceleration by exploiting fine-grained control over datapaths and memory hierarchy. PIPEMSM [8] is a representative example: it implements a fully pipelined bucket-method MSM on a Xilinx UltraScale+ device, overlapping point addition with bucket lookup through custom

on-chip BRAM structures. The authors report throughput improvements of roughly 3× over contemporary CPU baselines for BN254 MSM, at the cost of significant development effort and a fixed window-size configuration baked into the bitstream.

MSMAC [25] takes a complementary approach, proposing a hardware/software co-designed accelerator in which the FPGA handles the inner bucket-accumulation loop while a host CPU manages scalar recoding and final window reduction. This partition reduces on-chip resource pressure and enables support for multiple curves from the same bitstream, but introduces PCIe transfer overheads that limit speedup on smaller problem sizes. Related work on FPGA Montgomery multipliers for BN254 on Xilinx Alveo platforms [26] has further demonstrated the viability of FPGA-based modular arithmetic at the word widths required by ZKP workloads.

More recent efforts have explored cloud FPGA deployment, notably on AWS F1 [27] and F2 [28] instances, in an attempt to reduce the capital barrier to entry. These systems achieve deterministic latency and favourable energy efficiency, but they share three structural limitations relevant to the present work. First, reconfiguration cost is non-trivial, so parameter changes (such as window size or curve choice) typically require bitstream rebuilds. Second, the highest-throughput designs rely on operand widths and bucket counts that are tuned for a specific curve, limiting portability across the Pasta cycle. Third, peak throughput is bounded by on-chip BRAM capacity, which constrains the achievable degree of parallelism for the 2^{12}-bucket configurations favoured by modern MSM implementations.

2.4.2 GPU-Based Approaches

GPU implementations trade the fine-grained control of FPGAs for massive thread-level parallelism and a mature software toolchain. cuZK [7] was among the first systems to demonstrate that a carefully engineered CUDA implementation of MSM could outperform optimised CPU baselines by an order of magnitude on BLS12-381. The design uses sparse-matrix representations of the bucket accumulation phase to expose regular parallelism and relies on warp-level primitives for reduction. GZKP [11] extended this line of work with a parallel bucket-method variant that decouples window processing from bucket reduction, reporting up to 8× speedup over prior GPU baselines for large instances. Production-grade sort-based MSM kernels developed for the ZPrize competition [29] demonstrated that CUB radix sort combined with stream-level pipelining can sustain high throughput on commodity NVIDIA hardware, a design pattern adopted and extended in Chapter 4.

Multi-GPU scaling has been investigated by DistMSM [10] and, more recently, by the multi-GPU MSM framework of Ji et al. [9], both of which partition the input across devices using a static point decomposition and reduce partial results on the host. These works report near-linear scaling up to four GPUs for BN254 MSM, with efficiency falling

off at larger device counts due to reduction and host-side synchronisation costs. A separate line of work, exemplified by the ICICLE library [30], provides production-grade CUDA primitives for several curves and proof systems, but emphasises ease of integration over aggressive per-curve tuning. Recent performance characterisation studies [22] have further quantified the occupancy and memory-bandwidth constraints that shape GPU ZKP performance.

Two observations emerge from this body of work and motivate the present thesis. First, the vast majority of published GPU MSM results target BN254 or BLS12-381; Pallas has received comparatively little attention despite its centrality to Nova and to Halo 2-derived systems. Differences in field modulus structure and curve form mean that results from BN254 implementations do not transfer directly, and a Pallas-specific implementation is required to characterise folding-scheme performance. Second, most prior multi-GPU evaluations stop at four devices and at problem sizes below 100M points; the behaviour of sort-based Pippenger MSM at the scale enabled by eight A100 or H100 devices, and across the K-partition design space, has not been systematically reported. Chapters 4 and 5 address both gaps directly.

Chapter 3

The Nova Framework and Target Platforms

The two chapters that follow describe the GPU acceleration developed for this thesis and the experimental campaign that evaluates it. Before presenting modifications, however, it is necessary to establish what the modifications are *to*: the Nova prover is not a monolith but a Rust library with an internal structure that dictates where hardware acceleration can be inserted, and the machines on which the acceleration has been evaluated span two distinct classes of hardware, a consumer workstation and an institutional HPC cluster, each with its own architectural character.

This chapter is therefore descriptive rather than constructive. Section 3.1 gives an overview of the Nova codebase on which the implementation is built. Sections 3.2–3.4 trace the path from the folding prover down to the CPU multi-scalar multiplication routine that serves as the performance baseline in Chapter 6. Section 3.5 identifies the specific trait methods at which hardware acceleration can be introduced without disturbing the surrounding prover logic. Section 3.6 describes the five hardware targets (two CPUs and three GPUs) against which the implementation of Chapter 4 is benchmarked.

3.1 The Nova Codebase

The implementation is built against a fork of the Nova reference implementation maintained by Microsoft Research [17], hereafter referred to as *Nova-gpu*. No part of the upstream folding logic, constraint-system machinery, or Fiat–Shamir transcript has been modified in the course of this work; the fork adds a GPU code path alongside the existing CPU path and leaves the rest of the repository verbatim. Establishing which parts of the upstream codebase are touched, and which are not, is the purpose of this chapter.

At the top level, the upstream repository is organised as a Cargo workspace whose root crate, `nova-snark`, exposes the public prover and verifier API. Within that crate, the source tree divides cleanly into four functional areas. The `traits` module defines the abstract interfaces; `Engine`, `Group`, `TranscriptEngineTrait`, and their associated

types, in terms of which the folding prover is written. The `nova` module (containing `mod.rs` and `nifs.rs`) and the `r1cs` module together contain the folding machinery itself: the non-interactive folding scheme, the relaxed R1CS structures, and the `RecursiveSNARK` driver that composes the two into an end-to-end incrementally verifiable computation. The `spartan` module implements the zkSNARK used to compress the final folded instance. The `provider` module, on which this chapter concentrates, contains the concrete curve and commitment-scheme backends that satisfy the traits defined in `traits`. Nova's reference implementation supports three curve cycles; Pallas/Vesta, BN254/Grumpkin (named `bn256_grumpkin` in the source tree), and secp/secq, and a corresponding provider module exists for each.

The significance of this separation for hardware acceleration is that the folding prover and the SNARK compressor never reference a concrete curve directly. They operate exclusively through trait methods, and the provider module is the only location where those methods resolve to curve-specific arithmetic. Any acceleration effort that preserves the trait contracts is, by construction, confined to the provider layer.

3.2 The Provider Abstraction

The provider layer is organised around a small set of traits that together describe what Nova requires of a curve backend. Three are relevant to the present work.

`Engine` is the top-level trait that bundles a scalar field, a base field, a group, a hash-to-group function, a commitment engine, and a transcript engine into a single associated-type collection. A concrete instantiation of `Engine` is what a caller passes to the folding prover; for example, the Pallas-based prover is obtained by instantiating Nova's generic `RecursiveSNARK` with the `PallasEngine` type defined in the Pallas provider.

`Group` captures the minimal algebraic interface Nova needs from an elliptic curve: identity, negation, addition, scalar multiplication, and serialisation. Every curve backend implements this trait for its group type.

`DlogGroupExt` extends `Group` with the operations specific to discrete-logarithm-based commitment schemes. Among these, the operation of interest for this thesis is the method

```
vartime_multiscalar_mul(scalars:  &[Self::Scalar], bases:
                &[Self::Affine]) -> Self,
```

which computes the multi-scalar multiplication $\sum_i s_i P_i$ used by Pedersen commitments. The folding prover never calls into curve arithmetic directly; every commitment it produces, whether to the witness vector z, to the error vector E, or to the cross term formed during folding, is issued as a single invocation of `vartime_multiscalar_mul` against

the relevant group and a pre-generated set of public bases. Section 3.5 returns to this method as the substitution point for GPU acceleration.

The trait discipline is tight. A correct provider implementation is required to produce, for any given inputs, the unique group element defined by the mathematical specification of the operation; the means by which it does so, whether CPU, GPU, FPGA, or otherwise, is not constrained by the trait. This is what makes the provider layer the natural insertion point for a hardware backend.

3.3 The Pallas Provider and Its Commitment Code Path

Nova's Pallas provider lives at `src/provider/pasta.rs` in the upstream layout and defines the `PallasEngine` type together with implementations of `Engine`, `Group`, and `DlogGroupExt` for the Pallas curve. The curve arithmetic itself is not implemented in Nova; it is delegated to the `halo2curves` crate, which provides optimised field and point operations for the Pasta cycle alongside a suite of other curves used by Halo 2-derived proof systems. Pallas points, Pallas scalars, and the associated affine and projective types referenced by the provider are all `halo2curves` types.

The commitment code path traversed by a single folding step can be traced from the top-level `RecursiveSNARK::prove_step` entry point down to the innermost curve operation. When the prover advances one step of IVC, it issues a sequence of Pedersen commitments via the `CommitmentEngineTrait::commit` method defined on the commitment-engine associated type of `PallasEngine`. This method reduces, after minor bookkeeping, to an invocation of `vartime_multiscalar_mul` on the Pallas group, with the scalar vector drawn from the witness or error data and the base vector drawn from the public commitment key. Inside `vartime_multiscalar_mul`, the provider dispatches to a CPU multi-scalar multiplication routine, the `msm` function in `src/provider/msm.rs` (a Pippenger bucket-method implementation adapted from zcash/halo2 and parallelised across input chunks via Rayon), that produces a Pallas point as output.

Figure 3.1 summarises this path. Several features of it are worth noting. The path is linear, with no branching on input size or system state; the prover always reaches `vartime_multiscalar_mul` regardless of circuit, step index, or folding history. The boundary between Nova and `halo2curves` falls at the bottom of `vartime_multiscalar_mul`, which means that any alternative MSM backend can be introduced without touching `halo2curves` at all. And the method signature fixes the representation at the boundary: scalars arrive as a slice of `pallas::Scalar` values, bases as a slice of `pallas::Affine` values, and a single `pallas::Point` is returned. These three types are all that a replacement backend must accept and produce.

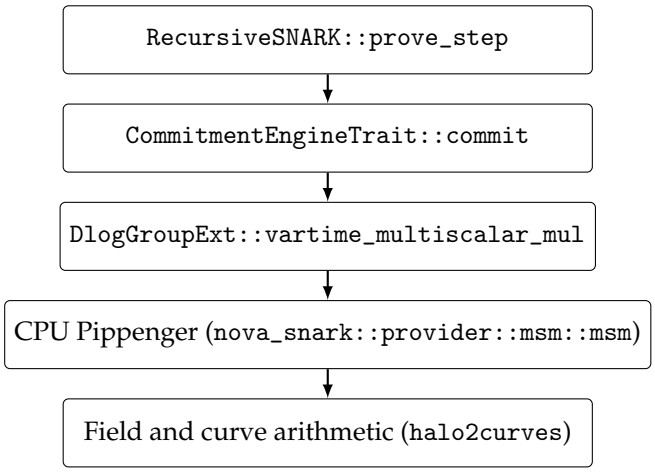


FIGURE 3.1: Upstream Pallas commitment code path. Each folding step's Pedersen commitments reduce to `vartime_multiscalar_mul` calls that dispatch to the CPU Pippenger routine in Nova's `provider::msm` module (adapted from zcash/halo2). Chapter 4 introduces an alternative GPU binding.

3.4 The Upstream CPU MSM Implementation

The CPU path reached at the bottom of Figure 3.1 is the same Pippenger bucket-method implementation introduced in Section 2.2, here implemented locally in Nova's `provider::msm` module (adapted from the zcash/halo2 reference) and operating on `halo2curves`-supplied field and curve primitives. The routine selects a window width as a function of the input length, decomposes each scalar into windows of that width, accumulates points into per-window buckets, reduces each window's buckets into a window sum via the summation-by-parts identity, and combines the window sums using Horner's method. Parallelism is obtained through `rayon`: when the input length exceeds the number of available threads, the input is split into per-thread chunks, a serial bucket-method MSM runs on each chunk, and the partial results are reduced on the host. Within a chunk, bucket accumulation is sequential.

Two properties of this implementation bear directly on what follows. First, it is correct by construction under the trait contract: the value it returns is the unique multi-scalar multiplication defined by the inputs, modulo floating-point-free exact arithmetic. This makes it the natural reference against which every GPU result in Chapter 6 is cross-validated. Second, its performance envelope is bounded above by the available CPU parallelism, the latency of a 256-bit modular multiplication on a 64-bit general-purpose core, and the cache behaviour of the bucket data structure as n grows past the size of the last-level cache. Chapter 6 quantifies these bounds empirically on the two CPU platforms described in Section 3.6, and the resulting timings constitute the CPU baseline against which GPU acceleration is measured.

3.5 Extension Points for Hardware Acceleration

The contract described in Section 3.2 admits, in principle, a range of hardware backends. In practice, the Nova codebase exposes exactly one extension point that is sufficient for full acceleration of the folding prover's commitment cost, and it is the method `vartime_multiscalar_mul` on the Pallas provider's `DlogGroupExt` implementation. A hardware backend that replaces this method alone, accepting the same `pallas::Scalar` and `pallas::Affine` slices and returning a correct `pallas::Point`, accelerates every Pedersen commitment issued by the prover without requiring any change elsewhere in the codebase.

Two features of the Rust and Cargo ecosystems make this substitution clean in practice. First, the orphan rules permit a downstream fork to provide an alternative implementation of a locally declared trait method without re-defining the trait, simply by editing the provider file. Second, Cargo's feature-flag mechanism supports compile-time selection among alternatives, so that the upstream CPU path and a fork-local GPU path can coexist in the same source tree and be chosen between at build time. Chapter 4 exploits both: the GPU backend is gated behind a `pallas-gpu` Cargo feature, and when that feature is disabled the build is byte-identical in behaviour to upstream Nova, which is what permits direct CPU-versus-GPU comparison on the same binary base.

No other part of the Nova repository is modified. The folding prover, the R1CS structures, the transcript engine, the Spartan compressor, and the providers for BN254/Grumpkin and secp/secq remain untouched. The claim of Chapter 6 that the GPU prover produces the same recursive argument as the CPU prover, for the same inputs, is a consequence of this discipline: the only component substituted is the one whose output is fully determined by its inputs under the trait contract.

3.6 Target Hardware Platforms

The experimental campaign presented in Chapter 6 runs on five hardware targets: two CPU baselines and three GPU configurations spanning consumer and datacentre classes. This section describes each briefly; full software-stack and benchmark-methodology detail is deferred to Chapter 5.

3.6.1 CPU Baselines

Local workstation CPU. The workstation on which the single-GPU implementation was developed is equipped with an Intel Core i7-13700K (Raptor Lake) processor with 16 cores and 24 threads, comprising 8 performance cores with hyperthreading and 8 efficiency cores. Performance cores run at a 3.4 GHz base clock with a 5.4 GHz maximum

turbo frequency; efficiency cores at 2.5 GHz base and 4.2 GHz turbo. The cache hierarchy provides 24 MB of L2 and 30 MB of shared L3. The CPU MSM baseline on this machine uses all available cores through Nova's `rayon` thread pool.

Cluster compute-node CPU. The two GPU compute-node classes in Hofstra's StarHPC cluster are provisioned with different host CPUs. Node `gpu7`, hosting the H100 accelerators, runs two AMD EPYC 9334 processors (Genoa generation, Zen 4), providing 64 physical cores and 64 hardware threads in aggregate at a 2.7 GHz base clock. Node `gpu1`, hosting the A100 accelerators, runs two AMD EPYC 7513 processors (Milan generation, Zen 3), providing 64 cores and 128 hardware threads in aggregate at a 2.6 GHz base clock. The SLURM job specification used throughout the evaluation requests 32 CPU cores per task, which bounds the `rayon` parallelism available to the CPU baseline on the cluster. Reporting CPU results separately for the workstation and the cluster isolates the effect of the baseline hardware from the effect of the GPU hardware and prevents cross-platform speedup figures from conflating the two.

3.6.2 Consumer GPU: NVIDIA GeForce RTX 4080

The single-GPU implementation was developed and tuned on an NVIDIA GeForce RTX 4080, the consumer-class accelerator of the Ada Lovelace generation, built on the AD103 die in TSMC's 4N process [31]. The card connects to the host via PCIe 4.0 x16. Per-device specifications, including SM count, shared-memory budget, cache hierarchy, memory capacity, and bandwidth, are summarised in Table 3.1 and treated in detail alongside the GPU architecture discussion of Section 4.1.4.

The RTX 4080 represents the accessible end of the modern GPU spectrum. Its inclusion in the evaluation is motivated by the interest of characterising ZKP acceleration on hardware that a single researcher can purchase rather than hardware that requires institutional infrastructure.

3.6.3 HPC GPUs: NVIDIA A100 and H100

The multi-GPU implementation is evaluated on Hofstra University's StarHPC cluster, which hosts two eight-GPU nodes relevant to this work.

NVIDIA A100 SXM4 80GB (Ampere). Node `gpu1` hosts eight A100 accelerators in the SXM4 form factor, built on the GA100 die in TSMC's 7 nm process [32]. Inter-device communication within the node uses third-generation NVLink, providing 600 Gs of aggregate GPU-to-GPU bandwidth.

NVIDIA H100 SXM5 80GB (Hopper). Node `gpu7` hosts eight H100 accelerators in the SXM5 form factor, the successor generation to the A100, built on the GH100 die in TSMC's 4N process [33]. Inter-device communication uses fourth-generation NVLink, providing 900 Gs of aggregate GPU-to-GPU bandwidth.

Per-device specifications for both accelerators, alongside the RTX 4080 for reference, are summarised in Table 3.1.

Parameter	RTX 4080	A100 SXM4 80GB	H100 SXM5 80GB
Architecture	Ada Lovelace	Ampere	Hopper
Compute capability	`sm_89`	`sm_80`	`sm_90`
Streaming multiprocessors	76	108	132
CUDA cores	9,728	6,912	16,896
Max shared memory per SM	48 KB	164 KB	228 KB
L2 cache	64 MB	40 MB	50 MB
Global memory	16 GB GDDR6X	80 GB HBM2e	80 GB HBM3
Peak memory bandwidth	717 GB/s	2039 GB/s	3350 GB/s
Inter-device interconnect	N/A	NVLink 3	NVLink 4
TDP	320 W	400 W	700 W

TABLE 3.1: Target GPU platforms for Chapter 6. Shared-memory figures are the maximum opt-in values per SM. The RTX 4080 is a single-device workstation accelerator; the A100 and H100 are deployed in eight-GPU SXM nodes within the StarHPC cluster.

Three points from Table 3.1 shape the analysis in later chapters. First, SM count increases monotonically from RTX 4080 through A100 to H100, but CUDA-core count does not: the A100 has fewer CUDA cores than either consumer or successor accelerator because its SMs carry fewer cores apiece. Chapter 6 separates the effect of SM count from the effect of per-SM throughput on MSM performance. Second, shared-memory capacity per SM differs by a factor of nearly five between the consumer and datacentre platforms, which has direct consequences for any design that holds bucket state in shared memory; this is revisited in the kernel-design discussion of Section 4.1.4. Third, memory bandwidth scales faster than compute across the three platforms, which reshapes the balance between memory-bound and compute-bound phases of the MSM pipeline as the hardware changes. These are the invariants against which the empirical results of Chapter 6 should be read.

Chapter 4

GPU Acceleration of MSM – Design and Implementation

Chapter 2 established that multi-scalar multiplication dominates the prover cost in folding-scheme constructions such as Nova. Chapter 3 described the framework into which acceleration is inserted, namely the Nova provider abstraction at `vartime_multiscalar_mul`, and the hardware platforms on which the implementation is evaluated. This chapter describes the GPU implementation itself.

Section 4.1 introduces the parts of the NVIDIA GPU architecture and CUDA programming model that bear directly on the design. Section 4.2 describes the arithmetic substrate on which the implementation is built: Montgomery multiplication over the Pallas base field, Pallas point addition in projective coordinates, and the signed-digit scalar recoding used to parameterise Pippenger's algorithm. Section 4.3 describes the system architecture that bridges Nova's Rust codebase to the CUDA kernels. Sections 4.4 through 4.7 then describe the implementation itself in the order in which its design evolved, progressing from a shared-memory baseline through the sort-based redesign, the partition-based concurrent-stream pipeline, and the batch-and-multi-GPU dispatch layer. Section 4.8 concludes with a qualitative comparison of the design points; empirical evaluation is deferred to Chapter 6.

4.1 GPU Architecture and the CUDA Programming Model

The NVIDIA GPU presents an execution model fundamentally different from that of a conventional CPU. Understanding the relevant features of that model is a prerequisite for explaining why MSM is well suited to GPU acceleration in principle, and equally for explaining the performance obstacles encountered in practice. This section summarises the parts of the model that bear directly on the implementation; a comprehensive treatment is available in the CUDA programming guide.

4.1.1 Execution Hierarchy: SMs, Warps, and SIMT

A modern NVIDIA GPU is organised as an array of Streaming Multiprocessors (SMs). Each SM contains compute units (CUDA cores), a private register file, a pool of shared memory, and instruction-scheduling hardware. The NVIDIA GeForce RTX 4080 used for the single-GPU experiments in this thesis has 76 SMs; the A100 and H100 accelerators used in the HPC cluster described in Chapter 3 have 108 and 132 SMs respectively.

Work is launched onto the GPU as a grid of thread blocks. A block is a group of threads that share a common shared-memory partition and a barrier-synchronisation primitive; block size is capped at 1024 threads. The GPU scheduler assigns entire blocks to SMs, and each SM executes one or more resident blocks concurrently, interleaving their execution to hide memory latency.

Within a block, threads are grouped in hardware into *warps* of 32. The warp is the fundamental unit of instruction issue: all 32 threads in a warp execute the same instruction in lockstep, operating on distinct data. This execution model, termed Single-Instruction Multiple-Thread (SIMT), achieves high throughput when threads within a warp follow the same control-flow path. Divergent branches cause the warp to execute each taken path serially, a phenomenon known as warp divergence that degrades throughput. MSM on a GPU must therefore be designed so that control flow within a warp is as uniform as possible, particularly within performance-critical inner loops.

4.1.2 Memory Hierarchy

The CUDA memory hierarchy exposes several distinct address spaces with sharply different bandwidth and latency characteristics. From fastest to slowest:

- *Register file.* Per-thread, on-chip, single-cycle access. Each SM on the RTX 4080 provides 65,536 32-bit registers, statically partitioned across resident threads.
- *Shared memory.* Per-block, on-chip, accessed in a small number of cycles. Each SM on the RTX 4080 provides 48 KB of shared memory in the default configuration, partitioned across resident blocks.
- *L1 and L2 caches.* On-chip, hardware-managed. The RTX 4080 has an L2 cache of approximately 64 MB shared across all SMs.
- *Global memory.* Off-chip GDDR6X for the RTX 4080, HBM2e for the A100, HBM3 for the H100. Peak bandwidth ranges from 716.8 GB/s on the RTX 4080 to approximately 3 TB/s on the H100 SXM, but latency is on the order of hundreds of cycles.

Two properties of this hierarchy shape MSM performance. First, the shared-memory budget per SM is a hard upper bound on how large the on-chip working set of a resident block can be. Second, global memory bandwidth, though substantial in absolute terms, is an order of magnitude smaller than the equivalent aggregate shared-memory throughput, and accesses must be coalesced (consecutive threads reading consecutive addresses) to approach peak throughput.

4.1.3 Streams, Atomics, and Concurrency

CUDA streams are per-device FIFO queues of kernel launches and memory transfers. Operations within a single stream execute sequentially; operations on distinct streams may execute concurrently, subject to hardware resource availability. Explicit stream management is the primary mechanism for overlapping host-to-device transfers with kernel execution and for keeping multiple kernels in flight simultaneously. The sort-based design introduced in Section 4.5 relies on stream concurrency to overlap the radix-sort and accumulation phases of successive windows.

Atomic operations on shared or global memory, including `atomicAdd`, `atomicCAS`, and `atomicExch`, provide a mechanism for lock-free coordination among threads. The CUDA memory model, derived from the C++11 memory model, treats these operations as *relaxed* atomics by default: they guarantee linearisability of the atomic variable itself but impose no ordering constraints on surrounding memory operations. Explicit fences (`__threadfence_block`, `__threadfence`, `__threadfence_system`) are required to establish release–acquire ordering between threads. The consequences of this model for shared-memory bucket accumulation are discussed in Section 4.4.

4.1.4 Occupancy and the Role of Resource Pressure

A single SM can execute multiple resident blocks concurrently, but the number that actually fit is determined by the most constrained resource among registers, shared memory, and the warp-slot limit. The ratio of active warps on an SM to the hardware maximum is termed *occupancy*. High occupancy allows the warp scheduler to hide memory latency by issuing instructions from other warps whenever one warp stalls; low occupancy leaves stalls exposed, because the scheduler has no alternative work to interleave.

For MSM on a GPU, occupancy is the single most important performance determinant, and it is governed almost entirely by shared-memory pressure. A bucket-based Pippenger kernel must maintain a working set of 2^c partial sums per window, each of which is a curve point requiring three field elements. Any design that holds buckets in shared memory is bounded above in window size by what fits in the SM's shared-memory budget; the structural consequences of this bound are made concrete in Section 4.4 [22].

4.2 Arithmetic Foundations for Pallas MSM

The performance of a GPU MSM kernel is determined only in part by the Pippenger superstructure and its parallelisation strategy. The bulk of the actual work is field arithmetic: billions of 256-bit modular multiplications and additions executed inside the bucket-accumulation phase. This section reviews the arithmetic primitives on which the implementation is built. The primitives themselves, namely Montgomery multiplication in the Pallas base field and point addition in projective coordinates, are standard constructions drawn from existing field and curve libraries; the implementation consumes them but does not modify them. What matters for later sections is the *representation* in which data is held on the device, because the pipeline structure developed in subsequent sections is shaped by where each representation change occurs.

4.2.1 Montgomery Form and Where It Is Used

Direct modular reduction after each field multiplication requires a division by the prime p, which is expensive on any architecture that lacks a dedicated divider. Montgomery's method avoids this by representing each field element $a \in \mathbb{F}_p$ as its Montgomery transform $\tilde{a} = aR \bmod p$, with $R = 2^{256}$ chosen to align with the word boundary. Multiplication of two Montgomery-form values, with an appropriate reduction step folded into the inner loop, yields the Montgomery form of the product using only multiplications, additions, and bit shifts [18]. The standard Coarsely Integrated Operand Scanning variant of this algorithm [21] is used by the underlying field library and its inner loop is straight-line arithmetic, which avoids the warp-divergence hazard described in Section 4.1.1.

Only field values on which the kernel performs arithmetic need to live in Montgomery form. In the MSM pipeline developed here, this includes the point coordinates (added repeatedly in the bucket phase) and the running bucket state (updated on every addition), but *not* the input scalars: scalars are only inspected bit by bit during window decomposition, never multiplied. The consequences for data movement across the Rust–CUDA boundary are described in Section 4.3.2.

4.2.2 Pallas Point Addition in Projective Coordinates

Curve points in the bucket-accumulation phase are represented in homogeneous projective coordinates $(X : Y : Z)$ rather than affine (x, y). The affine representation is efficient when a single modular inversion can be amortised across many operations, but it would require one inversion per bucket update, which is prohibitively expensive inside a parallel kernel. Projective coordinates trade the inversion for a larger state (three

field elements per point) and slightly more multiplication per addition, but are otherwise inversion-free.

The implementation uses the standard Weierstrass addition and doubling formulas catalogued in the Explicit Formulas Database [34]. A *mixed* addition formula (projective + affine) is used throughout the accumulation phase, because input points are stored in affine form to halve their memory footprint while the accumulating bucket state is kept projective. Projective–projective addition is reserved for the window-reduction and window-combination phases, where both operands are already projective bucket sums. Precise operation counts depend on the formula variant; for the short-Weierstrass form $y^2 = x^3 + 5$ used by Pallas, each mixed addition costs on the order of ten field multiplications and each full projective addition slightly more, with doublings comparable in cost. These counts multiplied by the number of bucket updates per MSM yield the billions of field operations cited earlier.

4.2.3 Signed-Digit Recoding and the 12-Bit Window

The parameter c in Pippenger's algorithm, the window size, controls a trade-off between bucket count (2^c per window) and window count ($\lceil \lambda / c \rceil$ windows for a λ-bit scalar). A larger c reduces the number of windows and therefore the total number of bucket-accumulation operations, but increases the per-window bucket count and the memory footprint of bucket state.

The sort-based implementation described in Section 4.5 uses a 12-bit window on 256-bit scalars, yielding 22 windows per scalar. A naive 12-bit window would require $2^{12} = 4096$ buckets per window. Signed-digit recoding halves this cost: a window value v exceeding 2^{c-1} is replaced with the negative digit $v - 2^c$, and a carry of one is propagated into the next window. The recoded digit is accumulated into its absolute-value bucket by adding the negation of the input point. Pallas supports point negation at negligible cost (negate the Y coordinate), so the effective bucket count becomes $2^{c-1} = 2048$ per window. Recoding is performed on the device by a single kernel that runs one thread per scalar and produces a packed array of signed digits consumed by later phases.

The choice of a 12-bit window is the single most consequential parameter decision in the chapter. Storing 2048 projective points, each $3 \times 32 = 96$ bytes, requires 192 KB per window, four times the entire shared-memory capacity of an SM. Bucket state therefore lives in global memory throughout, which removes the occupancy ceiling imposed by the shared-memory budget at the cost of increased global-memory traffic. The earlier shared-memory baseline described in Section 4.4 used a narrower 8-bit window for the converse reason that bucket state had to fit in shared memory, and was retired precisely because the eight-bit configuration could not be widened without abandoning the shared-memory layout.

4.2.4 The Four-Phase Pippenger Pipeline

Every version of the implementation described in subsequent sections shares the same high-level pipeline, parameterised differently across versions. Given n input points and n scalars, the pipeline proceeds as follows.

Phase 1: Preprocessing. Input points are converted from standard form to Montgomery form on the device; input scalars are decomposed into their windows and (in the sort-based design) signed-digit recoded. Both conversions run as element-wise kernels over the input arrays and take negligible time relative to the accumulation phase.

Phase 2: Bucket accumulation. For each window, each input point is added into the bucket indexed by its window digit. Total work is proportional to n times the window count, and this phase dominates runtime at all n of practical interest. Parallelisation of this phase is the subject of subsequent sections.

Phase 3: Window reduction. Each window's buckets are collapsed to a single window sum using Pippenger's summation-by-parts identity: the sum $\sum_{i=1}^{B-1} i \cdot b_i$ over B buckets is computed in linear time by maintaining a running total that is itself added into an accumulator at each step.

Phase 4: Window combination. The window sums are combined using Horner's method:

$$R = \left(\cdots\left(\left(s_{w-1} \cdot 2^c + s_{w-2}\right) \cdot 2^c + s_{w-3}\right)\cdots\right) \cdot 2^c + s_0,$$

where w is the window count and c the window width.

Within a sort-based implementation, Phase 2 is itself decomposed into several kernel launches (pair generation, radix sort, boundary finding, and warp-per-bucket accumulation) described in Section 4.5. Phases 3 and 4 contribute negligibly to runtime for any n of practical interest and are not the targets of optimisation. The result produced by Phase 4 is in projective Montgomery form; it is converted to affine standard form on the device before being copied back to the host.

4.3 Rust–CUDA Integration

The implementation sits at the intersection of three languages and toolchains: Nova is written in Rust, the MSM kernel is written in CUDA C++, and the two are bridged by a thin C-ABI wrapper. This section describes how the components fit together and how data crosses the boundary between them.

4.3.1 Three-Layer Design

The code base is organised as two components, each with a well-defined responsibility.

1. **Pallas GPU crate (`pallas-gpu`).** A single Rust crate that builds the CUDA static library and exposes its functionality to downstream Rust code. The CUDA portion (`cuda/*.cu`), compiled to a static archive by a `build.rs` that invokes `nvcc` with the target `-arch=sm_XX` flag, owns all device state: kernel implementations, device memory allocations, and the per-call buffer management described below. All 32-bit limb arithmetic, projective point-addition formulas, and the four-phase pipeline live in this library. The Rust portion (`src/lib.rs`) is a thin wrapper that declares the C-ABI signatures, provides safe Rust types for the data that crosses the boundary, performs the standard-form $\leftrightarrow$ 32-bit-limb conversions, and translates non-zero CUDA return codes into idiomatic `Result` values. It contains no curve or MSM logic of its own.

2. **Nova integration layer.** Nova's `DlogGroupExt` trait provides the hook for the MSM operation used by Pedersen commitments, as described in Section 3.5. Under the `pallas-gpu` feature flag, the `vartime_multiscalar_mul` implementation for Pallas is redirected to call into `pallas-gpu` rather than Nova's default CPU implementation. When the feature flag is disabled, the build reverts to the upstream Nova code path with no residual overhead.

This separation has two practical benefits. First, the `pallas-gpu` crate is curve-specific but proof-system-agnostic: in principle it can be linked into any Rust project that requires Pallas MSM, not only into Nova. Second, the Nova integration is minimal and feature-gated, which makes direct comparison against the upstream CPU implementation a matter of toggling a Cargo feature rather than maintaining a parallel fork.

4.3.2 The FFI Boundary and Data Conversion

At the FFI boundary, Rust's internal representation of points and scalars, namely the types provided by the `halo2curves` library, must be serialised into a layout the CUDA library can consume. Three conversions occur.

Scalars are serialised via Rust's `Fr::to_repr()` method, which produces a 32-byte little-endian standard-form representation. No Montgomery transform is applied on either side of the boundary: the kernel never performs arithmetic on scalars and requires only their bit-level structure for window decomposition, so keeping them in standard form eliminates an unnecessary conversion pass.

Points are serialised as affine (x, y) pairs in standard form, each coordinate consisting of eight 32-bit limbs in little-endian order. Because curve arithmetic does occur on

points, the kernel converts them to Montgomery form on the device as the first step of the pipeline. The final *result* is produced as a projective point in Montgomery form; a trailing on-device kernel converts each of the three projective coordinates from Montgomery back to standard form, the projective triple is copied to the host, and the host then performs the single modular inversion required to convert the projective point to affine $(X/Z, Y/Z)$ and reconstructs a `halo2curves` point through `Fr::from_repr()`.

The asymmetry is deliberate. Performing all representation changes on the device keeps the host-side interface curve-agnostic and avoids adding CPU work to the prover's critical path. It also keeps the Rust side free of any dependency on Montgomery-specific types, which would leak library-internal representation details across what is intended as a neutral boundary.

To minimise the host-side cost of these conversions at scale, both `pallas::Scalar` and `pallas::Affine` inputs are converted in parallel using a Rayon `par_iter` pipeline before the FFI call is issued. Serial conversion of 100M points was measured to take on the order of twenty seconds on the benchmark workstation and constituted a visible fraction of total MSM latency at large n; parallel conversion amortises this across all available host cores and removes it from the critical path.

4.3.3 Device Memory Management

Device memory in a long-running MSM pipeline must satisfy two competing constraints: per-call allocation overhead should be amortised where possible, and no device state should persist beyond a call in a way that prevents later calls, especially on different devices, from running cleanly.

The original single-GPU implementation prioritised the first constraint. It maintained a *persistent memory pool*: static device pointers declared at file scope and lazily initialised on the first call, reused across subsequent calls, with reallocation triggered only when the required size exceeded the current capacity. This design was motivated by a direct measurement showing that a naive `cudaMalloc/cudaFree` per call imposed approximately 12 ms of allocation overhead, a non-negligible fraction of total MSM runtime at small input sizes, particularly in the folding-scheme setting where Nova invokes MSM once per IVC step and a full proof may involve thousands of folding steps.

The transition to multi-GPU execution exposed a structural limitation of that design. Because the static slot array was populated on whichever device was selected at initialisation (in practice, device 0), a large single-GPU run at n =100M left on the order of 48 GB resident on device 0 that could not be released without process teardown. Subsequent multi-GPU runs at n =200M then attempted to allocate a further 12 GB on top of that resident state, triggering `cudaErrorMemoryAllocation` due to the combination of physical capacity and fragmentation accumulated across prior `ensure_slot_capacity`

calls. The persistent-pool design was therefore not merely suboptimal for multi-device execution; it was incompatible with it.

The current implementation replaces the persistent pool with a per-call allocation model, exposed through the entry point `pallas_msm_gpu_on_device`. This function takes an explicit `device_id` argument, calls `cudaSetDevice` to pin the invoking thread to the target GPU, allocates a local slot array sized for the current input, runs the full four-phase pipeline, and frees every device buffer before returning. No static device state survives across calls. The consequences are as follows.

- *Device-safety.* A call targeting device d affects only device d's memory; other devices remain untouched.
- *Thread-safety without serialisation.* Concurrent calls from different host threads to different devices are safe by construction, because each thread operates on its own local slots and `cudaSetDevice` is per-thread in CUDA.
- *Predictable steady-state memory use.* Device 0 is in a clean state before each multi-GPU run, which eliminates the OOM pathway described above.
- *Reintroduced allocation overhead.* The 12 ms per-call cost returns. This cost is negligible for the large-n regime that motivates GPU acceleration, and it is negligible relative to the host-to-device transfer time for points and scalars of realistic size.

The trade-off is explicit: the current design accepts a modest per-call allocation cost in exchange for correct, predictable behaviour across eight concurrent devices. The empirical impact of this choice is quantified in Chapter 6.

4.3.4 Concurrency Model at the FFI Layer

The per-call allocation model described above changes what thread safety at the FFI boundary requires. Under the earlier persistent-pool design, the CUDA library's reliance on static device state made it unsafe against concurrent calls from multiple host threads: a second caller could overwrite the first caller's device data before its kernel completed. This was not a theoretical concern, as Nova's test harness runs tests in parallel by default, and any embedding issuing multiple MSMs concurrently from different threads would encounter it. The FFI entry point was accordingly guarded by a `pthread_mutex` held for the entire duration of a call.

With per-call local allocation, the mutex is no longer required to prevent data races on device buffers, because no buffers are shared across calls. What remains is the multi-GPU dispatch pattern used by the Nova integration layer: a single-host-thread call into `vartime_multiscalar_mul_multi_gpu` partitions the input across all visible devices,

spawns one `std::thread` per device, and invokes `pallas_msm_gpu_on_device` concurrently on each. Each worker thread owns its slice of the input (moved via `to_vec` rather than shared via `Arc`), issues its own `cudaSetDevice`, and writes into its own local device buffers; no synchronisation primitive is required between them. The eight partial projective results are joined on the host and summed using the native `halo2curves` group operation, which is exact on projective coordinates and therefore incurs no precision loss from the partitioning.

This arrangement preserves per-device serialisation, where each GPU still processes its slice sequentially through the four-phase pipeline, while eliminating cross-device serialisation. The resulting scaling behaviour is reported in Chapter 6.

4.4 V1: Shared-Memory Spinlock Kernel

The first GPU implementation developed for this thesis adopted the most direct mapping of Pippenger's bucket method onto the CUDA execution model: bucket state was held in shared memory, threads accumulated input points into buckets via mixed addition, and concurrent updates to the same bucket were serialised by a per-bucket spinlock. The design predates the sort-based architecture introduced in Section 4.5 and is now retired, but its performance ceiling is what motivated the redesign and is therefore worth documenting in this chapter.

4.4.1 Design and Working Set

V1 used an eight-bit window, yielding 32 windows over a 256-bit scalar and a per-window bucket count of $2^8 - 1 = 255$. Each bucket held a projective point (three 256-bit field elements, packed as eight 32-bit limbs each), giving a per-bucket footprint of 96 bytes. A second array of 255 32-bit integers served as the spinlock state, one lock per bucket. The total per-block working set was therefore

$$255 \times 96 \;+\; 255 \times 4 \;=\; 25{,}500 \text{ bytes} \;\approx\; 24.9 \text{ KB}.$$

With a launch configuration of 256 threads per block, this layout admitted exactly one resident block per SM on the RTX 4080: a second block would require 51 KB of shared memory against a 48 KB budget. The resulting hardware occupancy was

$$\frac{1 \times 256 \text{ threads}}{1{,}536 \text{ threads per SM}} \;\approx\; 16.7\%.$$

This figure is the binding constraint of the V1 design. It cannot be reduced by changing block size, grid size, or input length; it is determined entirely by the choice to hold bucket state in shared memory.

The accumulation phase processed input points in chunks. Threads within a block were assigned points round-robin from the chunk, decoded the point's window digit, and contended for the corresponding bucket lock. The contention pattern was governed by the standard test-and-set discipline of `atomicCAS`, with explicit memory fences bracketing the critical section to establish release–acquire ordering on the bucket data. Listing 4.1 reproduces the inner pattern.

```
while (atomicCAS(&s_locks[bi], 0, 1) != 0) { }    // acquire
__threadfence_block();
point_add_mixed(&s_buckets[bi], &s_buckets[bi], pt);
__threadfence_block();
atomicExch(&s_locks[bi], 0);                        // release
```

LISTING 4.1: Per-bucket spinlock-guarded accumulation in V1.

The two `__threadfence_block` calls are essential. `atomicCAS` is a *relaxed* atomic in CUDA's memory model and imposes no ordering on surrounding memory accesses, as discussed in Section 4.1.3. Without the leading fence, a thread holding the lock could observe stale bucket data from before a prior holder's update; without the trailing fence, a subsequent acquirer could observe the bucket before the current update became visible. Both pathologies were encountered during V1 bring-up and resolved by adding the fences shown.

4.4.2 Why V1 Was Retired

V1 produced correct results once the memory-fence ordering was in place, and attained a maximum end-to-end speedup of approximately 2.05× over the CPU baseline at the input sizes for which the design was tuned. The reason the implementation was retired is structural rather than corrective: the 16.7% occupancy ceiling is not a bug to be fixed but a property of the algorithm's data layout, and the layout itself foreclosed the optimisation paths that the rest of the chapter pursues.

The most consequential of those paths is the move to a 12-bit window with signed-digit recoding. As Section 4.2.3 establishes, this configuration's 2048-bucket array occupies 192 KB per window, four times an SM's entire shared-memory budget; there is no shared-memory layout that accommodates it.

The design pivot was therefore not "make V1 faster" but "remove shared memory from the bucket-state path entirely". Once buckets reside in global memory, the per-bucket footprint no longer competes with the SM occupancy budget, and the window size can be chosen on algorithmic grounds rather than memory-budget grounds. The cost of that move is that fine-grained per-bucket locking in global memory is no longer practical: global atomics are too slow to serialise contention across thousands of threads

at the bucket-update granularity. Some other mechanism is required to prevent concurrent writers to the same bucket. The mechanism adopted is a sort-based pre-grouping of contributions by bucket index, which is the subject of Section 4.5.

4.5 V2: Sort-Based Pippenger

The retirement of V1 left a precise design problem: bucket state must move from shared memory to global memory to permit a 12-bit window, but global atomics are too slow to serialise concurrent writers at the bucket-update granularity. The V2 design resolves the problem by eliminating concurrent writers entirely. Rather than letting many threads contend for each bucket through locking, the implementation pre-groups all contributions destined for the same bucket using a parallel sort. After the sort, every bucket's contributions occupy a contiguous region of memory, and accumulation becomes a sequence of independent reductions (one per bucket) that can run in parallel across the 45,056 compound buckets without any synchronisation primitive. Against the CPU baseline, V2 attains a maximum end-to-end speedup of 3.7× on the RTX 4080 (peak at $n = 200$K, settling near 3× for larger inputs as the single-stream pipeline saturates the device).

This section describes the V2 pipeline as it exists in production. Section 4.5.1 sketches the overall dataflow. Sections 4.5.2 through 4.5.6 cover the five kernels that constitute Phase 2 of the Pippenger pipeline (in the four-phase decomposition introduced in Section 4.2.4); the trailing window-combination kernel implements Phase 4 unchanged. Section 4.5.7 documents the internal evolution of the V2 design, which proceeded through three distinct configurations before reaching the form described here.

4.5.1 Design Overview

V2 decomposes the bucket-accumulation phase of Pippenger's algorithm into a five-kernel pipeline executed in sequence on a single CUDA stream. Figure 4.1 summarises the dataflow.

The key data structure is the *compound bucket key*, a 16-bit integer of the form

$$\texttt{key} \;=\; w \cdot 2048 \;+\; b,$$

where $w \in [0, 22)$ is the window index and $b \in [0, 2048)$ is the recoded bucket index within that window. Compound keys are constructed during pair generation and consumed during boundary finding. With 22 windows of 2048 buckets each, the compound key space contains 45,056 valid values, which fits comfortably below the $2^{16} = 65{,}536$

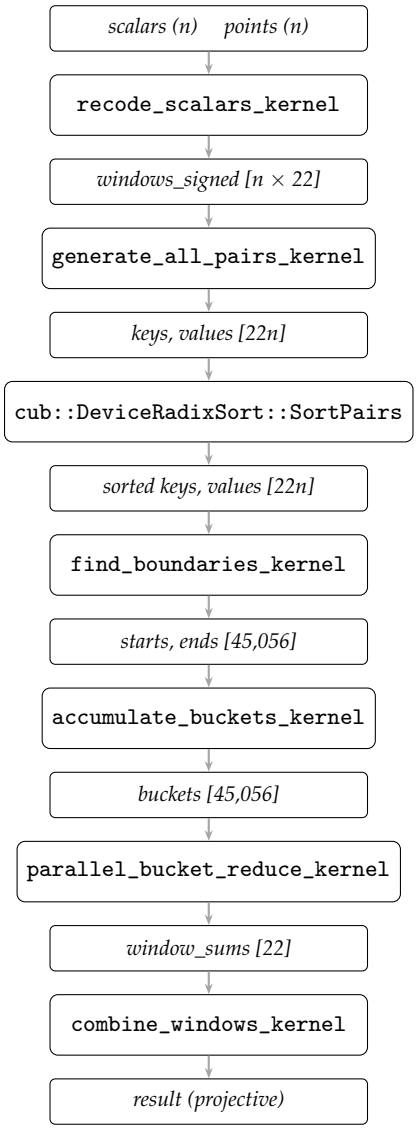


FIGURE 4.1: The V2 sort-based Pippenger pipeline. Blue boxes are CUDA kernels; gold boxes are device buffers with dimensions annotated. Five steps implement Phase 2; `combine_windows_kernel` implements Phase 4. Phase 1 (Montgomery conversion) and the result conversion are launched at the pipeline boundaries but omitted from the figure.

ceiling of a 16-bit sort, leaving room for a sentinel key (described below) that handles zero-valued windows.

The pipeline allocates a single set of device buffers sized for the input and reuses them across the entire MSM. Input-dependent buffers, namely the scalar windows, the key/value pair arrays, and the CUB temporary storage, are sized to n or $n \cdot 22$. Bucket-state buffers are fixed at 45,056 entries regardless of input size, since the compound key space is determined by the window configuration rather than n.

4.5.2 Scalar Recoding and Pair Generation

The first two kernels prepare the input for sorting. `recode_scalars_kernel` converts each 256-bit scalar into 22 signed-digit windows; `generate_all_pairs_kernel` expands those windows into the $22n$ compound key/value pairs that will be sorted.

Recoding implements the standard signed-digit transform described in Section 4.2.3. One thread is launched per scalar. The thread iterates over the 22 windows of its scalar, extracts the unsigned 12-bit window value via shifts and a mask (the helper `get_window` in the source), adds the carry from the previous window, and replaces the digit with its signed equivalent if the result exceeds $2^{11} = 2048$. The carry from the final window is discarded: the comment in the source notes that Pallas scalars are bounded above by 2^{255}, the final window covers bits 252–255 with a maximum raw value of 15, and so the carry into a (nonexistent) twenty-third window is provably zero.

The output of recoding is a $n \times 22$ array of `int16_t` values. A zero entry indicates that the corresponding window contributes nothing to the MSM and should be skipped; a positive entry indicates that the input point should be added to the bucket at index `value` $- 1$; a negative entry indicates that the negated input point should be added to the bucket at index $|$`value`$| - 1$.

Pair generation translates this representation into the form CUB's radix sort expects. A two-dimensional grid is launched with $(\lceil n/256 \rceil, 22)$ blocks of 256 threads, so that each thread processes one (point, window) pair. The thread reads the corresponding signed window value and writes the appropriate compound key and value, branching on the sign:

```
int pair_idx = window_idx * n + point_idx;
int16_t sv = windows_signed[point_idx * MSM_NUM_WINDOWS + window_idx];

if (sv == 0) {
    keys[pair_idx]   = sentinel_key;          // sorts to end, skipped
        later
    values[pair_idx] = 0;
} else if (sv > 0) {
    uint32_t bucket_idx = (uint32_t)(sv - 1);
    keys[pair_idx]   = window_idx * MSM_NUM_BUCKETS + bucket_idx;
```

```
10      values[pair_idx] = (uint32_t)point_idx;
11  } else {
12      uint32_t bucket_idx = (uint32_t)(-sv - 1);
13      keys[pair_idx]   = window_idx * MSM_NUM_BUCKETS + bucket_idx;
14      values[pair_idx] = (uint32_t)point_idx | NEGATE_FLAG;
15  }
```

LISTING 4.2: Pair generation: signed-digit decoding into the sortable key/value pair representation.

Three details merit comment. First, the negation flag is encoded as the high bit of the 32-bit value (`NEGATE_FLAG = 0x80000000`). Since the maximum point index is bounded by $n \leq$200M, well below 2^{31}, the high bit is otherwise unused and serves as a single-bit side channel for the negation indicator. Section 4.5.5 explains why this choice has a measurable performance consequence beyond saving memory.

Second, zero-valued windows are written with `sentinel_key = MSM_TOTAL_BUCKETS = 45056`, a value strictly greater than any valid compound key. Sentinels therefore sort to the end of the array, where the boundary-finding kernel skips them via a single bounds check. Sentinels are not removed from the array; the cost of letting them participate in the sort is a small constant factor on n, which is negligible relative to the radix sort's overall complexity.

Third, the keys are uniform across all windows and points, with no dependence on input order. This means the radix-sort that follows in Section 4.5.3 sorts a single contiguous array of $22n$ pairs in one call, rather than performing 22 independent per-window sorts. Empirically, the consolidated sort outperforms per-window sorts because CUB's radix sort has a fixed launch overhead that is amortised over a larger work item.

4.5.3 The CUB Radix Sort

The sort itself is performed by `cub::DeviceRadixSort::SortPairs`, a host-side call that internally launches a sequence of kernels implementing a multi-pass radix sort. Two parameters of the call are noteworthy.

```
err = cub::DeviceRadixSort::SortPairs(
    g_d_sort_temp, g_sort_temp_bytes,
    g_d_sort_keys_in, g_d_sort_keys_out,
    g_d_sort_vals_in, g_d_sort_vals_out,
    total_pairs, 0, end_bit
);
```

LISTING 4.3: CUB radix sort invocation. Only the lower 16 bits of each key participate in the sort.

The `end_bit` parameter is set to 16 because the largest key value (the sentinel, 45,056) fits in 16 bits. CUB performs one radix-sort pass per byte of key range, so restricting the sort to the lower 16 bits halves the number of passes relative to the default 32-bit sort.

The `g_d_sort_temp` buffer holds CUB's internal scratch space, the size of which is determined by an initial dispatch of `SortPairs` with a null buffer pointer. This API pattern of querying the size, allocating, and then sorting is standard for CUB's device-wide primitives. The temp-storage buffer is sized once at MSM initialisation and reused across calls in the persistent memory pool described in Section 4.3.3, which avoids the cost of repeating the size query and reallocation on every invocation.

4.5.4 Boundary Finding

After the sort, all pairs sharing a compound key occupy a contiguous run of the value array. The boundary kernel converts these implicit runs into explicit `(start, end)` index pairs, indexed by compound key, so that the accumulation kernel can locate each bucket's contributions in $O(1)$.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx >= total_pairs) return;

uint32_t key = sorted_keys[idx];
if (key >= total_buckets) return;            // sentinel: skip

if (idx == 0 || sorted_keys[idx - 1] != key) {
    starts[key] = (uint32_t)idx;
}
if (idx == total_pairs - 1 || sorted_keys[idx + 1] != key) {
    ends[key] = (uint32_t)idx + 1;
}
```

LISTING 4.4: Boundary detection: a thread marks the start of a run if its key differs from its predecessor's, and the end if its key differs from its successor's.

The kernel launches one thread per sorted pair. The starts array is pre-initialised to `0xFFFFFFFF` via a `cudaMemset`, which serves as a sentinel for compound buckets that received no contributions and therefore never have a start written. The accumulation kernel checks for this sentinel and emits the identity element for empty buckets.

4.5.5 Warp-Per-Bucket Accumulation

The accumulation kernel is the performance-critical centre of the V2 pipeline. It assigns one warp (32 threads) per compound bucket and walks the bucket's contributions in parallel: each lane reads a strided subset of the bucket's points, accumulates them

locally, and the warp's 32 partial sums are then combined via a shared-memory tree reduction.

```
1 for (uint32_t i = lane; i < count; i += 32) {
2     uint32_t raw_val   = sorted_values[start + i];
3     uint32_t point_idx = raw_val & ~NEGATE_FLAG;
4
5     affine_point_t pt = points[point_idx];    // single load path
6
7     if (raw_val >> 31) {
8         fq_neg(&pt.y, &pt.y);                 // cheap inline negation
9     }
10
11    point_add_mixed(&acc, &acc, &pt);
12 }
```

LISTING 4.5: Strided accumulation within a warp. Negation is applied arithmetically after the load (line 6) rather than via a parallel pre-negated array.

The handling of negation in this loop is the single most consequential design choice in the kernel, and it deserves explanation. A naive implementation would store the original input array of n affine points alongside a parallel array of n negated affine points, doubling the memory footprint. Negation would then become a branch over which array to read from, with no per-load arithmetic. Listing 4.5 takes the opposite approach: a single array of n points, one load path with no branching on the load itself, and a cheap arithmetic negation (`fq_neg` on the Y coordinate, eight modular subtractions) applied conditionally after the load. The branch divergence cost is small because `fq_neg` is a tiny fraction of the work performed by `point_add_mixed` that follows.

The benefit is structural: the points array remains size n rather than $2n$, halving the address range of the kernel's random reads. The 64 MB L2 cache on the RTX 4080 retains a larger fraction of the working set when the working set itself is half as large, so the single-array layout is expected to reduce L2 cache pressure during accumulation. The expected effect on the L2 hit rate is the design rationale for this choice but was not separately verified through profiler measurement; Section 7.2 notes profiling work that remains as future work.

After the strided walk, each lane holds a partial sum in a register. These 32 partial sums are reduced to a single per-warp result via a five-round tree in shared memory:

```
1 s_partial[threadIdx.x] = acc;
2 __syncwarp();
3
4 int warp_base = (threadIdx.x / 32) * 32;
5 for (int stride = 16; stride > 0; stride >>= 1) {
6     if (lane < stride) {
```

```
7           point_add(&s_partial[warp_base + lane],
8                     &s_partial[warp_base + lane],
9                     &s_partial[warp_base + lane + stride]);
10      }
11      __syncwarp();
12  }
```

LISTING 4.6: Shared-memory tree reduction across a warp's 32 lanes. `__syncwarp` barriers ensure intra-warp visibility of writes to `s_partial`.

The reduction proceeds in place over 32 shared-memory slots: the active lane count halves on each round (16, 8, 4, 2, 1), and each active lane adds its peer's value into its own slot. After five rounds, lane 0 holds the warp's combined bucket sum, which is written to the global `buckets` array indexed by compound key.

The launch configuration is 8 warps per block (`ACC_BLOCK_SIZE = 256`), with the grid sized to cover all 45,056 buckets. This gives one thread block per 8 compound buckets and a total grid of $45056/8 \approx 5632$ blocks. Buckets that received no contributions (their start sentinel was never overwritten) skip the accumulation loop and write the identity element directly.

4.5.6 Parallel Bucket Reduction

After accumulation, each window holds 2048 individual bucket sums. Phase 3 of the Pippenger pipeline collapses these into a single window sum using the summation-by-parts identity

$$\sum_{i=1}^{B-1} i \cdot b_i \;=\; \sum_{i=1}^{B-1} \sum_{j=i}^{B-1} b_j,$$

which a sequential implementation evaluates in linear time by maintaining a running suffix sum. The naive sequential algorithm walks the buckets in descending index order, accumulating each bucket into a running total and adding the running total into a per-window grand total at every step. This produces $\sum_{i=0}^{B-1}(i+1)\cdot b_i$ in $O(B)$ point additions.

For 2048 buckets per window, the sequential algorithm requires $2 \times 2048 = 4096$ point additions on the critical path of each window: 2048 iterations, each performing one running-suffix update and one accumulator update. With 22 windows reduced one at a time on a single thread, the bucket-reduction phase became the dominant cost of the pipeline at moderate n. The V2 production kernel replaces it with a parallel implementation that exploits a structural decomposition of the summation-by-parts identity.

The kernel launches 22 thread blocks (one per window) with $T = 128$ threads per block. Each thread is assigned a contiguous chunk of $S = 16$ buckets. The algorithm proceeds in four steps.

Step 1: Local summation by parts. Each thread t runs a sequential summation-by-parts over its 16-bucket chunk in descending order, producing two intermediate

values: $\texttt{running}_t$, the sum of the buckets in the chunk, and $\texttt{partial}_t$, the locally-weighted contribution of those buckets to the window sum.

Step 2: Parallel suffix scan over chunk-running totals. An inclusive parallel suffix scan over the 128 $\texttt{running}_t$ values produces, at each thread, the sum of running totals in chunks t through $T-1$. The scan runs in $\log_2 T = 7$ rounds of up to 64 parallel additions, separated by `__syncthreads()` barriers.

Step 3: Suffix correction. Each thread t adds $\texttt{count}_t$ copies of the suffix sum from chunks $t+1$ onwards into its $\texttt{partial}_t$. This term corrects for the fact that the local summation by parts in Step 1 ignored contributions from chunks at higher indices.

Step 4: Final tree reduction. A binary tree reduction over the 128 corrected partial values produces the window sum, which is written by thread 0 to the per-window result array.

Listing 4.7 reproduces the suffix scan, which is the algorithmic core of the kernel.

```
for (int stride = 1; stride < T; stride <<= 1) {
    if (tid + stride < T) {
        point_add(&s_running[tid], &s_running[tid], &s_running[tid +
            stride]);
    }
    __syncthreads();
}
```

LISTING 4.7: Parallel inclusive suffix scan over chunk-running totals. After $\log_2 T$ rounds, `s_running[t]` holds the sum of original `s_running[t..T-1]`.

The replacement of the sequential reduction by this kernel was the largest single improvement to V2 after the initial sort-based design. The total point-addition count across all four steps is comparable to that of the sequential algorithm at 4096 additions per window, but they are distributed across 128 threads working in parallel, with the critical-path length on a single thread reduced from 4096 sequential additions to seven rounds of suffix scan plus seven rounds of tree reduction, plus the local 16-bucket walk. The dominant cost shifts from the additions themselves to the synchronisation barriers between rounds, which is a substantially better profile for GPU execution.

4.5.7 Internal Evolution of V2

The V2 design as described above was not the initial sort-based implementation. The first version, written in late February of the development cycle, differed in three ways from the production kernel.

The earliest version used 8-bit windows: 32 windows of 255 buckets each, with no signed-digit recoding, because the 8-bit configuration was inherited from V1. The compound key space was 8160 buckets and the radix sort used a 14-bit `end_bit`. This was widened to 12 bits within the same development arc, first introducing the 4096-bucket configuration with no recoding and then layering the signed-digit transform on top to halve the effective bucket count to 2048.

The earliest version also used one thread per compound bucket rather than one warp. The single-thread accumulation walked each bucket's contributions sequentially, which left 31 lanes of every warp idle during the bucket-accumulation phase. The warp-per-bucket upgrade arrived within the same continuation session and reused the existing pipeline structure unchanged; only the accumulation kernel was modified. The shared-memory tree reduction shown in Listing 4.6 was introduced at this point; warp-shuffle intrinsics (`__shfl_down_sync`) were considered but not adopted, since the shared-memory pattern was simpler and the projective point representation does not pack cleanly into shuffle-sized registers.

Finally, the earliest version used the sequential summation-by-parts algorithm for bucket reduction, executed by a single thread per window. With 255 buckets per window this was acceptable, but the move to 2048 buckets made it the bottleneck of the pipeline. The parallel bucket reduction described in Section 4.5.6 was developed specifically to address this.

The kernel as described in this section, comprising 12-bit signed-digit windows, warp-per-bucket accumulation, and parallel bucket reduction, has been frozen since the optimisation arc completed. The V3 and V4 designs in subsequent sections do not modify the per-window logic; they layer additional dispatch and partitioning structure around the V2 kernel, which serves as the building block on which the cluster-scale extensions are built.

4.6 V3: Partition-Based Concurrent Streams

The V2 pipeline of Section 4.5 runs as a single CUDA stream: the six kernels execute strictly in sequence, and the host-to-device transfer that opens the pipeline is followed by an idle interval on the device until the kernel chain completes. For input sizes at which the per-window kernels do not by themselves saturate the device (which, on the RTX 4080, includes much of the practical n range below 10^7 points), a single stream leaves the GPU intermittently underutilised and the PCIe link wholly serialised.

V3 addresses this by partitioning the input into K equal-size slices and running the V2 pipeline on each slice in parallel, on its own CUDA stream. The per-window kernels are unchanged; what changes is the dispatch structure around them. With $K = 4$

streams active, the GPU sees up to four independent six-kernel pipelines at any instant, and the runtime is free to interleave their host-to-device transfers, accumulation kernels, and bucket-reduction kernels according to resource availability. A trailing on-device merge combines the four projective partial sums into the final result. Against the CPU baseline, V3 attains a maximum end-to-end speedup of 4.9× at $n = 20\text{M}$ on the RTX 4080, improving on V2 through stream-level overlap alone.

This section describes V3 as it exists in production. Section 4.6.1 sketches the partition-and-merge structure. Section 4.6.2 describes the per-slot state object that owns one stream's worth of device buffers. Section 4.6.3 traces the per-partition dispatch loop. Section 4.6.4 presents the merge kernel. Section 4.6.5 discusses the empirical selection of $K = 4$.

4.6.1 Design Overview

The V3 entry point partitions an input of n scalar–point pairs into K contiguous slices of $\lceil n/K \rceil$ pairs each, with the final slice truncated if Kn. Each slice is processed by a *slot*: a self-contained execution context comprising one CUDA stream, one set of variable-size input and intermediate buffers sized to the slice, and one set of fixed-size bucket-state buffers sized only by the window configuration. The four slots are static. They are created once at the first MSM call, persist for the lifetime of the process, and grow their variable-size buffers monotonically across calls under the same ensure-capacity discipline used by V2.

The dataflow within a slot is the V2 pipeline of Figure 4.1 applied to that slot's slice, with one substitution: the trailing `combine_windows_kernel` writes its result to the slot's per-call `d_partial_result` buffer rather than to a shared output. After all K slot pipelines have been submitted asynchronously, the host synchronises on each slot's stream in turn, then launches a single merge kernel on slot 0's stream to add the K projective partials together. The Montgomery-to-affine conversion and the device-to-host copy of the final result then proceed on slot 0's stream as in V2, against slot 0's buffers.

Two structural consequences of this design motivate Section 4.6.5. First, the per-slot bucket-state arrays are allocated independently of slice size: each slot holds its own 45,056-entry buckets array of 4.33 MB, regardless of how many points that slot processes. The fixed bucket footprint of V3 therefore scales as $K \times 4.33\,\text{MB}$. Second, CUB's radix sort, which dominates the per-slot kernel chain in absolute terms at moderate n, has a launch overhead amortised over the number of pairs sorted; shrinking each slot's pair count by a factor of K shifts a fixed cost from "amortised once" to "amortised K times," eroding sort efficiency at small n/K.

4.6.2 Per-Slot Resource Layout

Each slot holds two classes of device buffer, distinguished by whether their size depends on the slice point count. Listing 4.8 reproduces the slot-state structure in abbreviated form.

```
typedef struct {
    cudaStream_t stream;

    // Variable-size: grown by ensure_slot_capacity, never shrunk.
    pallas::affine_point_t*     d_points;
    uint32_t*                   d_scalars;
    int16_t*                    d_windows_signed;
    uint32_t*                   d_sort_keys_in;     // and out
    uint32_t*                   d_sort_vals_in;     // and out
    void*                       d_sort_temp;
    size_t                      sort_temp_size;

    // Fixed-size: allocated once at init_slots().
    uint32_t*                   d_bucket_starts;    // [45056]
    uint32_t*                   d_bucket_ends;      // [45056]
    pallas::projective_point_t* d_buckets;          // [45056]
    pallas::projective_point_t* d_window_sums;      // [22]
    pallas::projective_point_t* d_partial_result;   // [1]

    size_t capacity;
} msm_slot_t;
```

LISTING 4.8: V3 per-slot state. Variable-size buffers grow via ensure-capacity; fixed-size buffers are allocated once at slot initialisation.

The variable-size buffers are sized to n_i or $22n_i$, where n_i is the current slice's point count. They are reallocated only when an incoming slice exceeds the slot's existing capacity; on subsequent calls within capacity, they are reused without modification. The fixed-size buffers depend only on the window configuration $(22, 2048)$ and never need to grow.

Two further static allocations support the merge step. A device-side buffer `g_d_merge_buf` of K projective points receives the per-slot partial results before merging, populated by an asynchronous device-to-device copy at the end of each slot's stream. The slot array `g_slots[K]` and the merge buffer are both allocated lazily under a `pthread_mutex` on the first MSM call; subsequent calls observe the initialised state and incur no synchronisation overhead. V3 retains the persistent memory pool described as the original design in Section 4.3.3; the per-call allocation model introduced in that section applies to the V4 multi-GPU entry point of Section 4.7 and is not used here.

4.6.3 Per-Partition Dispatch and Stream Concurrency

The dispatch loop is structurally simple. For each slot $i \in [0, K)$, the host computes the slice offset and length, ensures the slot's variable buffers can accommodate the slice, and submits the full V2 kernel chain to the slot's stream. Submission is non-blocking: each `cudaMemcpyAsync` and each kernel launch returns immediately after enqueueing the work, so by the time the host loop completes its Kth iteration, all K pipelines are concurrently in flight on the device. Listing 4.9 shows the loop with the V2 kernel chain elided.

```
int K = MSM_NUM_PARTITIONS;
size_t part_size = (n + K - 1) / K;

for (int i = 0; i < K; i++) {
    size_t offset = (size_t)i * part_size;
    if (offset >= (size_t)n) break;
    size_t n_i = min(part_size, (size_t)n - offset);

    msm_slot_t* s = &g_slots[i];
    ensure_slot_capacity(s, n_i);

    // Async H2D, Montgomery conversion, signed-digit recoding,
    // pair generation, CUB radix sort, boundary finding,
    // warp-per-bucket accumulation, parallel bucket reduction,
    // and Horner combination -- all submitted on s->stream.
    /* ... V2 pipeline of Section~\ref{sec:sort-based} ... */

    cudaMemcpyAsync(g_d_merge_buf + i, s->d_partial_result,
                    sizeof(projective_point_t),
                    cudaMemcpyDeviceToDevice, s->stream);
}

for (int i = 0; i < K_active; i++)
    cudaStreamSynchronize(g_slots[i].stream);
```

LISTING 4.9: V3 per-partition dispatch loop. Each iteration submits the V2 pipeline on slot i's stream, followed by a device-to-device copy of the partial into the merge buffer. Kernel launch parameters omitted.

The host-side synchronisation barrier at the end of the loop is necessary because the merge kernel must run only after every partial result has been copied into `g_d_merge_buf`, but stream-level ordering does not capture this constraint: each per-slot device-to-device copy is ordered with respect to its own stream, not slot 0's. A cross-stream event would suffice in principle, but a host barrier is correct, easy to reason about, and incurs negligible cost relative to the kernel chain it bounds.

4.6.4 Partial Result Merge

The merge step combines K projective partial sums into a single result through $K - 1$ point additions. Listing 4.10 reproduces the kernel in its entirety.

```
__global__ void merge_partitions_kernel(
    projective_point_t* partials,
    projective_point_t* result,
    int K
) {
    if (threadIdx.x != 0 || blockIdx.x != 0) return;

    projective_point_t acc = partials[0];
    for (int i = 1; i < K; i++) {
        point_add(&acc, &acc, &partials[i]);
    }
    point_copy_projective(result, &acc);
}
```

LISTING 4.10: Merge of K projective partials. A single thread performs $K - 1$ point additions; the work is dwarfed by the per-slot pipelines that produced its inputs.

The kernel runs as a 1×1 launch on slot 0's stream after the host synchronisation barrier. With $K = 4$, the work consists of three projective point additions, on the order of forty modular multiplications, which complete in microseconds against per-slot pipelines whose work is measured in tens to hundreds of milliseconds. Parallelising the merge across threads or warps would yield no measurable benefit and is not pursued.

After the merge, the existing `convert_result_from_montgomery_kernel` runs on slot 0's stream against `g_slots[0].d_partial_result`, into which the merge kernel writes; the final device-to-host copy of the result completes the call. The host-visible work added by V3 over V2 is therefore the merge kernel, the four device-to-device copies of the per-slot partials, and the host-side stream synchronisation barrier.

4.6.5 Partition Count Selection

The partition count K is the only design parameter introduced by V3. It trades stream-level concurrency against per-slot resource overhead and against the sort efficiency of each individual slot. The optimal value is hardware-dependent.

Two structural considerations bound the useful range of K from above. First, each additional slot adds 4.33 MB of fixed bucket state plus a CUB temporary scratch buffer sized for the per-slot pair count, so the fixed memory overhead scales linearly in K. On the RTX 4080's 16 GB consumer memory budget, large K at large n leaves correspondingly less room for the input and intermediate buffers. Second, CUB's radix sort has a

launch overhead amortised over the number of pairs sorted; shrinking each slot's pair count by a factor of K shifts that fixed cost from once to K times. The slot-level sort that dominates the per-slot critical path at moderate n therefore loses efficiency superlinearly as K grows.

The production value of $K = 4$ was selected from a direct sweep over $K \in \{2, 4, 8, 16\}$ on the RTX 4080 across the input sizes characterised in Chapter 6; the per-K wall-clock measurements are reported in Table 6.4. The setting is hardcoded as `MSM_NUM_PARTITIONS = 4` rather than exposed as a runtime parameter; changing it requires recompilation.

This choice is platform-specific. The A100 and H100 nodes of the StarHPC cluster have substantially larger device memory, more streaming multiprocessors, and higher kernel-launch concurrency, all of which relax the upper bound on K and shift the trade-off in favour of higher partition counts; $K = 8$ is expected to be preferable on the A100. The cluster K-sensitivity sweep of Section 6.3.2 extends the evaluation to $K \in \{8, 16, 32, 64, 128\}$. That sweep parameterises the per-device internal partitioning of the V4 multi-GPU configuration of Section 4.7 rather than the single-device configuration of V3, but the underlying trade-off it characterises is structurally identical.

V3's measured throughput on the RTX 4080, including the per-K sweep and the achieved speedup over the V2 single-stream baseline and over the CPU implementation, is reported in Chapter 6.

4.7 V4: Batch API and Multi-GPU Dispatch

V3 saturates the parallelism available within a single MSM on a single device. The architectural step that follows operates on a coarser granularity: instead of subdividing the work of one MSM across K streams, V4 dispatches K independent MSMs across K execution units. Two distinct execution targets share this pattern. The first is a single GPU running multiple Nova-level MSMs simultaneously, exposed through a batch API. The second is a multi-GPU configuration in which one logical MSM is partitioned across all available devices. Both reuse the V2 pipeline of Section 4.5 and the slot abstraction of Section 4.6.2 unchanged; what changes is what each slot or each device computes. Against the cluster CPU baseline, V4 attains a maximum end-to-end speedup of 17.9$\times$ at $n = 100$M when the multi-GPU dispatch is engaged across an eight-device StarHPC A100 node.

This section describes V4 as it exists in the production cluster build. Section 4.7.1 identifies the two concurrency axes that V3 leaves unaddressed. Section 4.7.2 describes the batch API for single-device multi-MSM execution. Section 4.7.3 describes the multi-GPU dispatch layer. Section 4.7.4 covers the host-side projective merge and the parallel data-conversion path that connects the dispatch to the Nova prover loop.

4.7.1 Motivation: Inter-MSM and Inter-Device Concurrency

V3 saturates the GPU's stream concurrency by running K parallel slot pipelines on a single MSM. At the input sizes used in production folding-scheme proofs, a single MSM running on $K = 4$ streams keeps the SM occupancy budget of an RTX 4080 reasonably busy. Two concurrency axes lie beyond what V3 addresses.

The first is inter-MSM. Nova issues several independent MSMs per folding step, and the IPA prover round in particular computes left and right vector commitments (both multi-scalar multiplications of identical size, over disjoint base sets, with no data dependence on each other) that V3 schedules sequentially. Each round therefore incurs two full V3 invocations back-to-back, each launching its own four-stream pipeline with its own host-to-device transfer and its own Montgomery conversion phase. The work of the second commitment cannot begin until the first completes, even though no architectural obstacle prevents the GPU from running both pipelines in parallel.

The second is inter-device. Once a single device's working memory budget cannot accommodate the full intermediate state of one MSM, additional throughput requires recruiting additional devices. The empirical threshold on the StarHPC cluster's 80 GB A100 nodes falls at approximately n =100M points: above this size the per-MSM intermediate state of buckets, signed-digit windows, and CUB sort scratch space exceeds device memory, and the multi-GPU path becomes mandatory rather than merely beneficial.

V4 addresses both axes with a single architectural pattern: a host-side scheduler issues K independent MSM jobs to K execution units. The pattern is realised in two places. In the single-device case, the K execution units are CUDA streams, and the slot infrastructure of Section 4.6.2 is reused with each slot now holding an independent MSM rather than a partition of one MSM. In the multi-device case, the K execution units are entire GPUs, each running its own per-call instance of the V3 pipeline, and the per-call allocation model of Section 4.3.3 provides the safety properties that prevent device 0's persistent state from interfering with the dispatch.

4.7.2 Batch API for Nova-Level Concurrency

The batch API extends the single-call V3 entry point into a multi-call entry point. Where V3's `pallas_msm_gpu` accepts one $(\mathit{points}, \mathit{scalars})$ pair and returns one projective result, the batch entry point accepts K independent $(\mathit{points}, \mathit{scalars})$ pairs and returns K independent projective results. The number of jobs in a batch is chosen by the caller and is bounded above by the slot pool size.

Internally, the batch API repurposes V3's slot pool. V3 uses K slots to hold K partitions of a single MSM, with all K slots cooperating on the same final result through the

merge kernel of Section 4.6.4. The batch API uses the same K slots to hold K independent MSMs, each running the V2 pipeline of Section 4.5 on its own input. No merge kernel is launched: the K projective results are written to K separate output buffers, copied to host memory, and returned to the caller as a vector of independent values. The slot infrastructure, the ensure-capacity discipline, the per-stream dispatch, and the per-stream synchronisation pattern are otherwise identical to V3.

The motivating use site is the IPA prover round. Where the upstream Nova IPA implementation calls `vartime_multiscalar_mul` twice per round (once for the left commitment, once for the right), the batch-aware Nova provider replaces the pair of calls with a single batch call of size two. The two MSMs run on slots 0 and 1 concurrently, with their host-to-device transfers, kernel chains, and device-to-host copies overlapped under the same stream-level concurrency model that V3 applies to its partitions. The host receives both results when both streams have synchronised, and the IPA round proceeds with the two commitments now available simultaneously.

The architectural value of the batch API is not principally the absolute speedup it produces on any individual round, which is reported in Chapter 6, but the recognition that the slot infrastructure of Section 4.6.2 is general: one slot is one independent V2 pipeline, and the meaning of *independent* is determined by the caller, not by the slot itself.

4.7.3 Multi-GPU Dispatch

Multi-GPU dispatch realises the same independence pattern across hardware boundaries. A single MSM input of n scalar–point pairs is partitioned into N slices of $\lceil n/N \rceil$ pairs each, where N is the number of visible devices. Each slice is processed on a distinct device by an independent `pallas_msm_gpu_on_device` call (Section 4.3.3), and the N projective partial results are summed on the host into the final MSM output.

The dispatch is implemented at the Rust FFI layer in the function `msm_u64_multi_gpu`, reproduced in Listing 4.11. The function spawns one `std::thread` per device, transferring ownership of that device's slice into the worker thread by value rather than by reference. Each worker calls `cudaSetDevice(device_id)` and then `pallas_msm_gpu_on_device`, returning a projective partial sum on success. The host joins all N workers, propagates the first error if any, and forwards the partial sums to the caller for merging.

```
pub fn msm_u64_multi_gpu(
    scalars: &[[u64; 4]],
    points:  &[([u64; 4], [u64; 4])],
) -> Result<Vec<([u64; 4], [u64; 4], [u64; 4])>, String> {
    let n        = scalars.len();
    let num_gpus = get_gpu_count().max(1).min(n);
    let chunk    = (n + num_gpus - 1) / num_gpus;
```

```
8
9       let mut handles = Vec::with_capacity(num_gpus);
10      for device_id in 0..num_gpus {
11          let start = device_id * chunk;
12          let end   = (start + chunk).min(n);
13          if start >= end { break; }
14
15          let scalar_chunk = scalars[start..end].to_vec();
16          let point_chunk  = points[start..end].to_vec();
17
18          let handle = std::thread::spawn(move || {
19              let gpu_scalars = scalars_to_gpu(&scalar_chunk);
20              let gpu_points  = points_to_gpu(&point_chunk);
21              let mut result  = ProjectivePointGpu::default();
22
23              let ret = unsafe {
24                  pallas_msm_gpu_on_device(
25                      gpu_points.as_ptr(), gpu_scalars.as_ptr(),
26                      &mut result, point_chunk.len(),
27                      device_id as i32,
28                  )
29              };
30
31              if ret != 0 {
32                  Err(format!("GPU {} MSM failed with code {}",
33                              device_id, ret))
34              } else {
35                  Ok((result.x.to_u64_limbs(),
36                      result.y.to_u64_limbs(),
37                      result.z.to_u64_limbs()))
38              }
39          });
40          handles.push(handle);
41      }
42
43      let mut partial = Vec::with_capacity(handles.len());
44      for h in handles {
45          partial.push(h.join()
46              .map_err(|_| "GPU thread panicked".to_string())??);
47      }
48      Ok(partial)
49  }
```

LISTING 4.11: Multi-GPU dispatcher in the Rust FFI layer. One `std::thread` per visible device; each thread owns its slice by value, sets its target device, and invokes the per-call entry point of Section 4.3.3. The N projective partials are summed on the host by the provider hook of Section 4.3.4.

Three properties of this dispatch model bear on the architectural argument.

First, no synchronisation primitive is required between worker threads. Each thread owns its slice, performs `cudaSetDevice` on its own thread state (`cudaSetDevice` is per-thread in CUDA), and writes to its own device's local slot pool. There are no shared device buffers, no shared file-scope globals, and no inter-thread communication during the per-device work.

Second, the per-call allocation model of Section 4.3.3 is what makes the dispatch safe in practice. The persistent-pool design that V3 inherits binds its static slot array to whichever device was first selected, which precludes the device-by-device cleanup that multi-thread dispatch requires; `pallas_msm_gpu_on_device` avoids this by allocating and freeing all device state per call.

Third, inside each device, the V3 partition-based concurrent stream pipeline of Section 4.6 runs unchanged on the device's slice. The internal partition count K is the compile-time constant `MSM_NUM_PARTITIONS`, declared as a `#define` in `cuda/pallas_msm.cu`; the K-sweep of Section 6.3.2 steps through its values by patching this `#define` with `sed` and rebuilding the static library between runs, as described in Section 5.2.3. The composition is hierarchical: the multi-GPU dispatch operates across devices, V3 partitioning operates within each device, and V2 operates within each slot.

4.7.4 Host-Side Projective Merge and Parallel Conversion

The N projective partials returned by the multi-GPU dispatch are summed on the host using halo2curves's projective group operation. Projective addition over a Weierstrass curve is exact: it produces no rounding or representation error regardless of the coordinate values in its operands, so partitioning the input across devices and summing the partials yields a result identical to the single-device computation over the same input. Numerical correctness is independent of the partition count.

The merge work is small: $N - 1$ projective additions on the host CPU, completing in microseconds for any device count plausibly hosted in a single SLURM allocation. A device-resident merge would require either inter-device peer-to-peer transfers or a final aggregation step on a designated lead device, neither of which carries engineering value at the scale of the cluster benchmark presented in Chapter 6.

The host-side input conversion described in Section 4.3.2 applies unchanged to the multi-GPU path: a `rayon::join` converts scalars and points concurrently before the multi-thread dispatch loop is entered, so the conversion cost is amortised across the host's parallelism budget on both single-device and multi-device entry points.

4.7.5 Summary

V4 reuses the V2 per-window logic of Section 4.5 and the V3 slot infrastructure of Section 4.6 unchanged, and overlays two coarse-grained concurrency dimensions on top: inter-MSM (multiple independent Nova-level MSMs run as a single batch) and inter-device (one MSM partitioned across all available devices). The dimensions are independent and would in principle compose, though the production cluster benchmarks of Chapter 6 exercise the inter-device dispatch alone.

4.8 Architectural Comparison

The four implementation versions described in Sections 4.4 through 4.7 share a common substrate, the four-phase Pippenger pipeline of Section 4.2.4, but differ in what is parallelised, where bucket state lives, and how concurrent writers are coordinated. This section consolidates those differences as a qualitative comparison; quantitative results are reported in Chapter 6.

Concurrency granularity. The progression from V1 to V4 is one of progressively coarser parallelism. V1 exploits thread-level parallelism within a single block: hundreds of threads contend for the same shared-memory bucket array, with serialisation enforced per bucket through spinlocks. V2 moves the unit of parallelism up to the warp: each compound bucket is processed by 32 lanes that strided-walk its sorted contributions, with no inter-thread contention because the radix sort has pre-grouped each bucket's points into a contiguous run. V3 adds a further level above that: K CUDA streams each running an independent V2 pipeline on their own slice of the input, with the GPU's hardware schedulers overlapping their execution. V4 generalises the stream-level layer in two directions, replacing K partitions of one MSM with K independent MSMs for inter-MSM concurrency, and replacing K streams on one device with N devices each running a per-call instance of V3 for inter-device concurrency. At each step the unit of parallelism becomes coarser, and the synchronisation cost paid to coordinate it becomes lower. The empirical reflection of this progression is a monotonic improvement in maximum speedup over the CPU baseline: 2.05× at V1, 3.7× at V2, 4.9× at V3 (all on the RTX 4080 vs. the i7-13700K), and 17.9× at V4 (on 8× A100 vs. the 32-thread EPYC baseline).

Bucket state and the occupancy ceiling. The single most consequential design choice across the implementations is where bucket state resides. V1's choice to hold the 255-bucket array in shared memory imposed a 16.7% hardware occupancy ceiling on the RTX 4080 and capped the window size at eight bits, because a wider window's 2^{c-1}-entry bucket array would not fit in the 48 KB per-SM shared-memory budget. V2 moved

bucket state to global memory and accepted the trade-off: shared-memory pressure no longer governs occupancy, the window can be widened to twelve bits with signed-digit recoding, and the resulting 45,056-entry bucket array is sized by the window configuration rather than by the input. V3 retains V2's bucket layout but multiplies its footprint by the slot count: with $K = 4$ slots each holding their own bucket array, the fixed bucket overhead grows from 4.33 MB to 17.3 MB. V4's multi-GPU path inherits V3's per-device bucket layout but allocates locally per call, releasing the bucket arrays at the end of each invocation in exchange for the device-cleanliness properties needed to dispatch safely across eight devices.

Synchronisation and contention. V1's bucket-level spinlocks required explicit `__threadfence_block` fences to establish release–acquire ordering on the relaxed atomics of CUDA's memory model; correctness depended on the fence placement, and contention scaled with the input size. V2 eliminates contention entirely by pre-grouping contributions through the radix sort: the accumulation phase needs neither locks nor fences, only intra-warp `__syncwarp` barriers in the tree reduction and intra-block `__syncthreads` barriers in the parallel bucket reduction. V3 adds one host-side stream-synchronisation barrier per call to ensure all per-slot partials are in place before the merge kernel launches, but no further on-device synchronisation primitive is required. V4's multi-GPU path removes even the host barrier from the per-call critical path: each Rust worker thread is independent of the others, with no shared device state and no synchronisation primitive between them; the only coordination is `std::thread::join` at the end of the dispatch.

Building-block reuse. The V2 five-kernel pipeline of Section 4.5.1 is the unchanging building block of the chapter. V3 and V4 do not modify a single kernel within it; what they modify is the dispatch structure that surrounds it. Likewise, the slot abstraction introduced in V3 (Section 4.6.2) is reused unchanged in V4's batch API, which differs only in what each slot computes. The architectural progression therefore exhibits two complementary patterns: V2's per-window logic is the durable centre of the design, while the dispatch structure surrounding it is the surface on which each subsequent version makes its contribution.

Versions as a hierarchy, not a competition. The four versions are arranged hierarchically rather than as alternatives. V2 is the kernel chain that V3 and V4 wrap; V3 is the per-device pipeline that V4's multi-GPU path runs on each device; V4's batch API is the single-device specialisation invoked when multiple independent MSMs are presented at once. The single-device V3 path remains in production for inputs that fit comfortably

on one device, while V4's multi-GPU path is engaged when n exceeds the working-memory budget of any single visible device or when the cluster benchmark targets aggregate throughput across all visible GPUs. The thresholds at which each path becomes preferable are empirical and are reported in Chapter 6.

V2's per-window logic is preserved in every subsequent version, V3's slot abstraction is preserved in V4, and the trajectory taken by the chapter is one of additive dispatch layers around an unchanging computational core. The empirical consequences, namely the regimes of n in which each version dominates and the scaling behaviour of the multi-GPU path across the StarHPC cluster, are the subject of Chapter 6.

Chapter 5

Experimental Setup

Chapter 4 presented the GPU implementation and motivated its design choices in terms of the architectural features of the target platforms. This chapter describes the platforms themselves as concrete machine configurations, the software stack used to build and run the implementation, and the methodology by which the empirical results of Chapter 6 were collected. The architectural discussion of Section 4.1, which covers SM counts, memory hierarchy, and the CUDA execution model, is not repeated here.

Section 5.1 records the deployment role of each hardware platform in the campaign. Section 5.2 documents the toolchain versions, build configuration, and SLURM dispatch that constitute the software environment. Section 5.3 describes the input generation, timing protocol, correctness verification, and the explicit scope of what the reported numbers measure.

5.1 Hardware Platforms

The empirical campaign uses two distinct machines: a consumer workstation hosting a single NVIDIA GeForce RTX 4080 and a compute node of Hofstra's StarHPC cluster equipped with either eight NVIDIA A100 or eight NVIDIA H100 accelerators. The two platforms span the practical range of hardware available to a Nova prover deployment, from a single consumer GPU on commodity hardware to an institutional eight-GPU server with HBM-equipped data-centre accelerators. Full hardware specifications, including per-device GPU parameters and host CPU configurations, are given in Section 3.6 of Chapter 3 and consolidated in Table 3.1; the architectural features that bear on the kernel design are catalogued in Section 4.1. This section records the role each platform plays in the benchmark campaign and the additional system parameters (host memory, operating system, storage) not captured in those references.

5.1.1 Local Workstation

The local workstation hosts a single NVIDIA GeForce RTX 4080 on a PCIe Gen4 $\times 16$ link, paired with 32 GB of DDR5 system memory. Full GPU specifications appear in

Section 3.6.2; the host CPU (Intel Core i7-13700K) is documented in Section 3.6.1. The 16 GB device memory budget is the binding constraint on the single-GPU input range and is the reason the partition-count sweep of Section 4.6.5 stops at $K = 16$ rather than extending into the larger values that the StarHPC nodes permit.

The workstation runs Ubuntu 24.04.4 LTS with NVIDIA driver 580.126.09 and is used for two purposes in the campaign: it provides the CPU baseline against which all single-GPU speedups are reported, and it hosts the V2 and V3 evaluations of Chapter 6. The cluster results of Chapter 6 are reported against a separate cluster CPU baseline, since the workstation and cluster CPUs are not directly comparable.

5.1.2 StarHPC Cluster

Hofstra's StarHPC cluster provides two compute-node classes relevant to this thesis. The gpu1-class nodes house eight NVIDIA A100 SXM4 accelerators, and the gpu7-class nodes house eight NVIDIA H100 SXM5 accelerators; both node classes carry 1024 GB of DDR4 system memory. The two classes have distinct host CPUs: gpu1 uses two AMD EPYC 7513 processors and gpu7 uses two AMD EPYC 9334 processors, as documented in Section 3.6.1. Per-device GPU specifications for the A100 and H100 appear in Section 3.6.3.

The 80 GB per-device memory budget on either node class lifts the effective input ceiling well above the 16 GB workstation. Single-device runs on a cluster node can accommodate inputs through n =100M scalar–point pairs before the multi-GPU path becomes structurally necessary, and the eight-device dispatch extends the reach to n =200M within a single SLURM allocation. The 1024 GB of host memory accommodates the parallel Rust-side conversion of 100M-point inputs without paging, which is the reason the parallel `par_iter` pipeline of Section 4.7.4 is viable at cluster scale.

Storage for the campaign is mounted from the cluster's shared project filesystem at `/fs1/proj/acceleration_zkp`. The benchmark harness, input data, and result logs reside on this shared filesystem, which is accessible from the login node and from every compute node in the allocation.

5.2 Software Stack and Build Configuration

5.2.1 Toolchains and Versions

The implementation is a hybrid Rust–CUDA codebase, as described in Section 4.3. The build pipeline therefore involves two compilers and their associated toolchains. Table 5.1 records the versions used for the production cluster build; the workstation build uses the same Rust toolchain but a separate CUDA installation matched to the consumer driver.

Component	Version
Rust toolchain	1.91.1 stable
CUDA toolkit (workstation)	12.0 (V12.0.140)
CUDA toolkit (cluster)	12.4.0 (module `cuda/12.4.0-gcc-8.5.0-pbcbal4`)
NVIDIA driver (workstation)	580.126.09
NVIDIA driver (cluster)	565.57.01
CUB	bundled with CUDA toolkit
halo2curves	0.9.0
Nova-gpu fork commit	`27ea224`
gcc / g++ (CUDA host compiler)	13.3.0 (workstation); 8.5.0 (cluster, via CUDA module)
SLURM	21.08.8-2

TABLE 5.1: Toolchain versions used for the empirical campaign. Cluster-side versions reflect the modules loaded in the SLURM job script.

The CUDA static library (`pallas-gpu`) is built via `nvcc` with the appropriate `-arch=sm_XX` flag for the target accelerator: `sm_89` for the workstation, `sm_80` for the A100 nodes, and `sm_90` for the H100 nodes. Each architecture is built separately rather than relying on a fat binary, which keeps the build artefact small and avoids the launch-time overhead of JIT compilation on an unanticipated device. Optimisation level is `-O3` with no debug symbols in the production build. The library is then linked into the Rust crate through `build.rs`, which discovers the CUDA installation through the `CUDA_PATH` environment variable.

5.2.2 Build Targets and Feature Flags

The Rust workspace exposes the GPU implementation behind a Cargo feature flag (`pallas-gpu`), as described in Section 4.3.1. With the flag disabled, the build reduces to the upstream Nova CPU code path and serves as the CPU baseline. With the flag enabled, the `vartime_multiscalar_mul` hook for Pallas is redirected to the CUDA library, and the build additionally compiles and links `pallas-gpu`.

A second build dimension is the partition count K, which is the compile-time constant `MSM_NUM_PARTITIONS` defined directly in `cuda/pallas_msm.cu` as `#define MSM_NUM_PARTITIONS 4`. The partition sweep of Chapter 6 requires multiple builds at different K values, since changing K at runtime is not supported. The cluster build orchestrates this through a build-time substitution: the SLURM job script, taking the desired K as an exported variable, runs a `sed -i` replacement against the `#define` line in `pallas_msm.cu` immediately before invoking `cargo test -release -no-run`, which triggers `build.rs` to rebuild the CUDA library against the patched source. The K-sweep is therefore a sequence of independent source-patch-and-rebuild cycles executed within the same allocation, each producing an isolated binary with its results written to a per-K output directory.

The Rust release profile is used throughout (`cargo test -release` for the benchmark harness). Both crates compile at `opt-level = 3`, the Cargo release default; LTO is left at Cargo's default of off for the top-level Nova-gpu build, so the `pallas-gpu` dependency's own `lto = true` setting has no effect on the linked benchmark binary. The Rayon thread pool is left at its default size, which is the number of logical CPUs visible to the process under the SLURM allocation described in Section 5.2.3.

5.2.3 SLURM Job Configuration

Cluster runs are dispatched through SLURM. A single benchmark invocation requests one compute node with eight GPUs of the targeted class visible to the job, plus a 32-core slice of the host CPU. Listing 5.1 reproduces the directive block of the multi-GPU K-sweep job script used in Chapter 6.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH --partition=defq
#SBATCH --qos=long
#SBATCH --gres=gpu:A100:8                # gpu:H100:8 for H100 runs
#SBATCH --mem=512G
#SBATCH --time=48:00:00

# Passed via --export: NUM_PARTITIONS, CUDA_ARCH, GPU_TYPE, OUT_BASE
module load cuda/12.4.0-gcc-8.5.0-pbcbal4
source "$HOME/.cargo/env"

# Patch the partition count into the CUDA source and rebuild.
sed -i "s/#define MSM_NUM_PARTITIONS [0-9]*/#define MSM_NUM_PARTITIONS
    $NUM_PARTITIONS/" \
    "$PALLAS_DIR/cuda/pallas_msm.cu"
cd "$NOVA_DIR"
CUDA_ARCH="$CUDA_ARCH" cargo test --features pallas-gpu --release
    --no-run

# Run the multi-GPU scaling benchmark; results land under
    $OUT_BASE/k$NUM_PARTITIONS/.
CUDA_ARCH="$CUDA_ARCH" cargo test --features pallas-gpu --release \
    -- bench_large_scale_multi_gpu --nocapture --ignored
```

LISTING 5.1: SLURM directives for an eight-GPU cluster run. The exported variables (`NUM_PARTITIONS`, `CUDA_ARCH`, `GPU_TYPE`, `OUT_BASE`) are passed in by the submission wrapper, which loops over K values and queues each as a dependent job.

The `-gres=gpu:A100:8` directive (or `gpu:H100:8` on the Hopper nodes) ensures that all eight devices of the targeted class are exposed to the process, which is required by the multi-GPU dispatcher (Section 4.7.3); a smaller GPU allocation would cause `get_gpu_count()` to return a smaller N and silently shrink the partition. The `-cpus-per-task=32` directive bounds the host-side Rayon parallelism used by the `par_iter` pipeline for input conversion. The `-mem=512G` request is conservative relative to the node's 1024 GB but sufficient to hold the largest input (n =200M scalar–point pairs at roughly 96 bytes each, plus working buffers) without paging. The `-qos=long` directive together with the 48-hour wall-clock allows the full K-sweep at all input sizes to complete in a single allocation.

The cluster K-sensitivity sweep of Chapter 6 loops over $K \in \{8, 16, 32, 64, 128\}$. The submission wrapper queues five jobs per GPU type as a dependency chain, each running the listing above with a different `NUM_PARTITIONS` value and writing its averaged results to `$OUT_BASE/k$NUM_PARTITIONS/avg.txt`.

5.3 Benchmark Methodology

5.3.1 Input Generation and Reproducibility

Benchmark inputs consist of two arrays of length n: scalars in $\mathbb{F}_q$ (the Pallas scalar field) and affine points on the Pallas curve. Scalars are drawn by `pallas::Scalar::random`, and points by `pallas::Point::random` followed by an affine projection. Both samplers route through `halo2curves`'s standard constructions, which guarantee that every input point lies on the curve and avoid the cost of a hash-to-curve operation that has no algorithmic purpose in this campaign.

The random-number generator used throughout is `rand::rngs::OsRng`, which draws non-deterministic entropy from the operating system. The benchmark therefore does not produce bit-identical inputs across runs; instead, statistical reproducibility is obtained by repeating each $(n, K, \text{version})$ configuration over multiple iterations and reporting the mean. Variance across iterations on a quiescent node is empirically small relative to the speedup ratios of interest, as discussed in Section 5.3.4.

Inputs are generated once per n within a benchmark invocation and reused across all warmup and timed iterations at that n. Regenerating the input on every iteration would dominate the measured wall-clock time at small n and obscure the GPU's contribution. The generation cost itself is not part of any reported timing.

5.3.2 Warmup, Averaging, and Correctness Verification

Every $(n, K, \text{version})$ configuration is benchmarked with the protocol shown in Listing 5.2, taken from `bench_cpu_vs_gpu` in `src/provider/pallas_gpu.rs`. The warmup

and timed counts are chosen per benchmark function: the small-n workstation benchmark uses two warmup iterations and five timed iterations; the workstation large-scale benchmark uses one warmup and three timed iterations; the multi-GPU cluster sweep uses three warmup and five timed iterations. The warmup runs prime CUDA's lazy-initialisation pathways (context creation, kernel module load, first-touch memory allocation), warm the host-side allocator, and bring the cluster filesystem cache into a steady state. Timed iterations are summarised by their arithmetic mean, which is reported in Chapter 6; the individual per-iteration latencies are retained in the result log for variance analysis.

```
let scalars: Vec<pallas::Scalar> = (0..n).map(|_|
    pallas::Scalar::random(&mut OsRng)).collect();
let bases: Vec<pallas::Affine> = (0..n).map(|_|
    pallas::Point::random(&mut OsRng).to_affine()).collect();

for _ in 0..warmup {
    let _ = msm(&scalars, &bases);
    let _ = vartime_multiscalar_mul(&scalars, &bases);
}

let mut cpu_ms = Vec::new();
for _ in 0..runs {
    let t = Instant::now();
    let _ = msm(&scalars, &bases);
    cpu_ms.push(t.elapsed().as_secs_f64() * 1000.0);
}

let mut gpu_ms = Vec::new();
for _ in 0..runs {
    let t = Instant::now();
    let _ = vartime_multiscalar_mul(&scalars, &bases);
    gpu_ms.push(t.elapsed().as_secs_f64() * 1000.0);
}

// One correctness check per input size, after the timing loops.
let cpu_result = msm(&scalars, &bases);
let gpu_result = vartime_multiscalar_mul(&scalars, &bases);
assert_eq!(cpu_result.to_affine(), gpu_result.to_affine(),
           "Mismatch at n={}", n);

let ca = cpu_ms.iter().sum::<f64>() / runs as f64;
let ga = gpu_ms.iter().sum::<f64>() / runs as f64;
```

LISTING 5.2: Benchmark protocol per input size, taken verbatim from `bench_cpu_vs_gpu`. Warmup iterations are discarded; the timed iterations are averaged. A single correctness check follows the timing loop and aborts the benchmark on mismatch.

Correctness verification is performed once per input size, after the timing loops complete: a final CPU MSM and a final GPU MSM are computed on the same input and their affine forms are compared for pointwise equality. A mismatch aborts the benchmark and is treated as a correctness regression. The check is outside the timing loop so that the assertion's own overhead does not perturb the per-iteration measurements; because the GPU and CPU paths are both deterministic with respect to their input, a single post-hoc check is sufficient to certify that every iteration of either path on that input produced the same result.

5.3.3 Per-Phase Timing Instrumentation

Two timing modes are used. The default mode measures end-to-end wall-clock time from Rust FFI entry to FFI return, which is the latency observed by a Nova prover and is the figure of merit for the speedup comparisons in Chapter 6. The wall-clock measurement includes the host-side Rayon conversion of inputs into the FFI representation (Section 4.3.2), the host-to-device transfers, the kernel chain, the device-to-host transfer of the result, and the deserialisation of the projective result into a `halo2curves` type.

The per-phase mode instruments the CUDA library with paired `cudaEventRecord` calls at the entry and exit of each kernel stage of the four-phase pipeline (Section 4.2.4), with an additional pair around the host-to-device transfer and the device-to-host transfer. The events are queried after the timed kernel chain has fully synchronised, which means they do not perturb the kernel execution but do introduce a small per-call serialisation cost that the default mode avoids. Per-phase results are therefore reported separately and are not directly comparable to the wall-clock figures; they exist to attribute time within a phase, not to compete with the end-to-end measurement.

For the V3 multi-stream and V4 multi-GPU configurations, the per-phase mode aggregates events across streams or devices into a single per-phase total by recording the maximum elapsed time across the parallel units. This reflects the critical-path interpretation expected for a parallel pipeline: the slowest unit determines the phase boundary.

5.3.4 What Is and Is Not Measured

The reported timings include everything that lies on the critical path of a Nova prover invocation of `vartime_multiscalar_mul`: host-side input conversion, host-to-device transfer, the four-phase pipeline, device-to-host transfer of the result, and Rust-side deserialisation. They do not include input generation, the CPU reference computation used for correctness checks, file I/O, SLURM-imposed wait times, or the one-time cost of CUDA context initialisation on the first call after process start. The 12 ms per-call allocation cost described in Section 4.3.3 is included in the V4 multi-GPU figures

because it appears on every call in that mode; it is excluded from the V2 and V3 figures because those modes amortise allocation through the persistent pool.

For the multi-GPU configuration, the entry of `vartime_multiscalar_mul_multi_gpu` and the return from the final `std::thread::join` bound the reported wall-clock time, which includes the host-side projective merge but excludes the conversion of the merged result back to affine standard form.

Variance across timed iterations is summarised in Chapter 6 as a single global stability bound rather than per-row, since speedup claims are only as meaningful as the stability of the underlying measurements. The full per-configuration variance figures are retained in the result logs and serve to identify iterations dominated by unrelated system events (a background process, a filesystem flush) that should be discarded.

Chapter 6

Results and Analysis

This chapter reports the empirical evaluation of the implementation described in Chapter 4 using the methodology of Chapter 5. The presentation is organised by the platform on which each measurement was collected. Section 6.1 reports the single-GPU campaign on the RTX 4080 workstation, including the CPU-versus-GPU speedup curves, per-phase bottleneck attribution, and the V3 partition-count sweep. Section 6.2 isolates the contribution of individual optimisations through controlled ablations. Section 6.3 reports the single-device cluster results on the StarHPC A100 and H100 nodes. Section 6.4 reports the eight-device scaling behaviour of the V4 multi-GPU dispatch, which delivers the headline result of the campaign. Section 6.5 synthesises the findings into a set of platform selection guidelines.

All speedups in Sections 6.1 and 6.2 are reported relative to the workstation's Intel i7-13700K CPU baseline, computed via Nova's local Pippenger implementation (`nova_snark::provider::msm::msm`) on the same input. Cluster speedups in Sections 6.3 and 6.4 are reported relative to the StarHPC node's AMD EPYC 7513 CPU baseline, measured under the same SLURM allocation (Section 5.2.3) that gives the CPU pool 32 worker threads. Each reported figure is the arithmetic mean of the configuration's timed iterations after its warmup iterations (counts given in Section 5.3.2), with the per-iteration latencies retained in the underlying logs but suppressed from the tables for legibility. Variance was below 7% (coefficient of variation across measured iterations) for every configuration with $n \geq 500$K; at smaller n a one-off CPU outlier raised the cluster-side CPU CV to 15.7% at $n = 100$K, but no GPU configuration exceeded the 7% envelope.

6.1 Local GPU Results (RTX 4080)

6.1.1 CPU vs. GPU Speedup Across Input Sizes

The single-GPU campaign measures V2 and V3 against the CPU baseline across eight input sizes spanning three orders of magnitude, from $n = 100$K to $n = 20$M scalar–point pairs. The upper limit reported in this campaign is 20M; the empirical OOM cutoff at

$K = 4$ on the RTX 4080's 16 GB device is at approximately $n = 22$M ($n = 25$M aborts with `cudaErrorMemoryAllocation`). Table 6.1 reports the headline timings.

n	**CPU (ms)**	**V2 (ms)**	**V2 speedup**	**V3 (ms)**	**V3 speedup**
100K	59.4	19.7	2.91×	26.9	2.28×
200K	106.1	28.7	3.73× *(V2 max)*	39.2	2.68×
500K	235.8	66.5	3.55×	62.1	3.79×
1M	427.7	130.1	3.31×	104.6	4.06×
2M	779.4	260.6	2.99×	175.1	4.45×
5M	1,759.5	590.0	2.97×	390.2	4.53×
10M	3,438.7	1,127.2	3.07×	724.6	4.71×
20M	6,786.3	2,199.2	3.09×	1,379.8	4.91× *(V3 max)*

TABLE 6.1: RTX 4080 wall-clock times and speedup over the i7-13700K baseline. V3 uses $K = 4$.

Three structural observations follow from the speedup curve. First, GPU acceleration is already productive at the smallest measured size; V2 reaches 2.91× and V3 reaches 2.28× at n =100K. The crossover at which the GPU's overhead would equal the CPU's compute time lies below the campaign's smallest evaluated n, and is not directly measured here; in practice, the integration of Section 4.3.1 preserves the CPU path through the feature flag for the sub-100K regime where the dispatch overhead would dominate.

Second, V2's curve is non-monotonic on the RTX 4080: the speedup peaks at 3.73× near $n = 200$K and then settles at roughly 3× across the rest of the range. V2's single-stream pipeline saturates one CUDA stream early — the warp-per-bucket accumulation phase becomes bandwidth-bound on the RTX 4080's 716.8 GB/s link well before the largest input sizes — and additional inputs do not yield additional GPU utilisation, so the wall-clock grows linearly while the speedup over the still-improving CPU baseline stays flat.

Third, V3's curve grows monotonically from 2.28× at n =100K to 4.91× at n =20M, the V3 maximum on this hardware. V3 overtakes V2 from n =500K onward; below that, V3's per-slot sort efficiency loss (Section 4.6.5) outweighs the stream-level concurrency benefit. The widening V3-vs-V2 gap at large n reflects the partition design's success at hiding host-to-device latency behind concurrent kernel chains, which is the design's central claim. Figure 6.1 visualises this behaviour.

6.1.2 Per-Phase Bottleneck Analysis

Per-phase instrumentation (Section 5.3.3) was collected at a representative middle-of-range input size of n =10M, where V3 is closest to its peak speedup. Table 6.2 reports the breakdown for V2 and V3 in absolute milliseconds and as a percentage of end-to-end time.

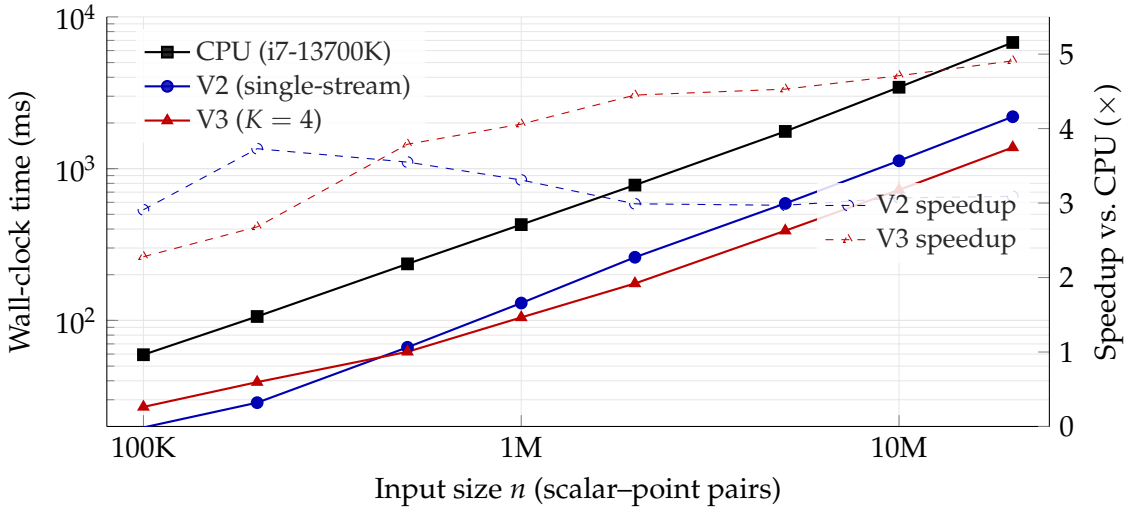


FIGURE 6.1: RTX 4080 wall-clock (left log axis, solid lines with filled markers) and speedup over CPU (right axis, dashed lines with hollow markers) against n. V2 peaks at 3.7× near 200K; V3 grows monotonically to 4.9× at 20M.

Phase	**V2 (ms)**	**V2 %**	**V3 (ms)**	**V3 %**
Host-side conversion (Rayon)	32.15	26.1%	30.30	30.6%
H2D transfer	6.77	5.5%	1.74	1.8%
Montgomery + scalar recode	0.87	0.7%	0.20	0.2%
Pair generation	0.40	0.3%	0.10	0.1%
CUB radix sort	1.30	1.1%	0.29	0.3%
Boundary finding	0.20	0.2%	0.00	0.0%
Warp-per-bucket accumulation	78.13	63.4%	37.91	38.2%
Parallel bucket reduction	0.80	0.6%	3.56	3.6%
Window combination	2.57	2.1%	3.63	3.7%
Result conversion + D2H	0.00	0.0%	21.40	21.6%
End-to-end (wall)	123.18	100%	99.15	100%

TABLE 6.2: Per-phase breakdown at $n = 1$M on the RTX 4080. V3 reports the slowest of four streams; not directly comparable to Table 6.1's wall times.

The breakdown identifies the warp-per-bucket accumulation as the dominant on-device phase: 63.4% of V2's end-to-end time at $n = 1$M and 38.2% of V3's, in both cases more than an order of magnitude larger than any other GPU phase. The CUB radix sort, contrary to the conventional wisdom that places it as a primary cost in sort-based MSM, accounts for only 1.1% (V2) and 0.3% (V3) of end-to-end time at this size, validating the design decision to use one large sort over 32M pairs rather than per-bucket sorts. The lever that would most directly reduce end-to-end time on this hardware is therefore the accumulation phase's L2 cache footprint (Section 4.5.5), not the sort. The host-side Rayon conversion (26% V2, 31% V3) is the second-largest cost on the workstation; its

Configuration	Time (ms)	Speedup vs. CPU
Workstation: RTX 4080 vs. i7-13700K (24 threads)		
CPU baseline	3,438.7	1.00×
V2 (sort-based, single-stream)	1,127.2	3.07×
V3 ($K = 4$, partitioned streams)	724.6	4.71×
Cluster: A100 vs. EPYC 7513 (32 threads)		
CPU baseline	4,409.6	1.00×
V3 single-A100 ($K = 8$)	811.2	5.43×
V4 8× A100 ($K = 8$)	424.5	10.39×

TABLE 6.3: Cumulative cross-version speedup at $n = 10$M. The two blocks use different CPU baselines because they run on different hardware tiers; speedups within each block are directly comparable.

dominance reflects the i7-13700K's modest 24-thread budget and shrinks to a smaller fraction on the EPYC cluster where 32 threads absorb the same work in less wall-time relative to the GPU's improved kernel throughput.

V3's per-phase attribution differs from V2's not in the per-stream cost of any phase, since the kernels are unchanged, but in the proportion of those costs that overlap with each other. The host-to-device transfer of slot i runs concurrently with the kernel chain of slot $i-1$ once the dispatch loop reaches steady state, which is the primary mechanism through which V3 recovers the time that V2 leaves on the table.

6.2 Optimisation Impact Breakdown

6.2.1 Contribution of Each Design Stage

The implementation reached its production form through four numbered design stages, presented in Chapter 4: V1 (shared-memory spinlock baseline, Section 4.4), V2 (sort-based pipeline, Section 4.5), V3 (partitioned concurrent streams, Section 4.6), and V4 (multi-GPU dispatch on top of the V3 per-device pipeline, Section 4.7). Each stage is a distinct codebase rather than an internal toggle, so the ablation reports the cumulative speedup attributable to each design step at the $n = 10$M reference size. Table 6.3 summarises the progression on the workstation; V4 is reported separately because it requires multi-GPU hardware unavailable on the workstation tier.

Three observations follow. First, the largest single contribution on the workstation is the V1→V2 redesign: replacing the spinlock-based shared-memory accumulator with a sort-based pipeline lifts the speedup from V1's 1.84–2.15× regime (Section 4.5) to V2's 3.07×. The redesign exchanges fine-grained synchronisation for one large $O(n \log n)$ sort, and the GPU sort hardware absorbs that cost more efficiently than the spinlock's serialisation absorbed contention. Second, V2→V3 adds a further 1.55× on the workstation by overlapping the four per-device pipelines through CUDA streams, with no

n	$K = 2$	$K = 4$ ***(prod)***	$K = 8$	$K = 16$
1M	89.4	**87.8**	105.7	147.5
10M	749.1	649.5	**598.3**	604.0
20M	1,414.5	1,205.4	1,095.8	**1,083.0**

TABLE 6.4: V3 wall-clock (ms) vs. partition count K on the RTX 4080. The bolded cell per row is the per-n optimum: $K = 4$ at 1M, $K = 8$ at 10M, $K = 16$ at 20M.

kernel-level changes; the gain is pure concurrency. Third, V3→V4 on the cluster adds a 1.91× multiplier by spreading the input across eight devices, with the dispatcher contributing only host-side merge cost (microseconds at this n). The composition of all three steps reaches 10.39× at $n = 10$M on the cluster and grows further at larger n (Section 6.4.1, peak 17.89× at $n = 100$M).

6.2.2 Partition Count Sensitivity

The V3 partition count K is the only design parameter exposed by Section 4.6.5 as platform-tunable on the consumer GPU. Table 6.4 reports the K-sweep on the RTX 4080 at three representative input sizes.

The sweep refines the structural argument of Section 4.6.5: the optimal K on the RTX 4080 is not constant but shifts upward with n. At $n = 1$M each slot's CUB sort already operates near its small-input efficiency floor, so additional partitioning erodes more sort efficiency than it recovers in stream concurrency, and $K = 4$ is the optimum. At $n = 10$M and $n = 20$M the per-slot sort is large enough that further halving still amortises its launch cost, and the additional concurrency is the dominant term: $K = 8$ is the optimum at 10M (598.3 ms, 7.9% over $K = 4$) and $K = 16$ is the optimum at 20M (1,083.0 ms, 10.2% over $K = 4$). Notably the prior expectation that $K = 16$ at $n = 20$M would exhaust the 16 GB device budget did not hold empirically; the run completes successfully because the per-slot variable buffers shrink with K and the fixed bucket overhead of $K \times 4.33$ MB ($\approx$ 69 MB at $K = 16$) is small relative to the input buffers. The production default of `MSM_NUM_PARTITIONS = 4` is therefore optimal for the small-n regime but leaves measurable headroom on this hardware at $n \geq 10$M; an adaptive K schedule would recover that headroom without harming the small-n case.

6.3 Cluster Results (A100 & H100)

6.3.1 Single-Device Cluster Timings

A single A100 or H100 differs from the RTX 4080 along three relevant axes: substantially larger device memory (80 GB versus 16 GB), more streaming multiprocessors (108 or 132

n	**EPYC CPU (ms)**	**A100 (ms)**	**A100 speedup**	**H100 (ms)**	**H100 speedup**
1M	554.4	167.8	3.30×	145.1	3.82×
10M	4,409.6	811.2	5.44×	928.5	4.75×
50M	20,035.8	3,389.8	5.91×	3,885.8	5.16×
100M	39,639.9	6,438.3	6.16× *(peak)*	7,803.4	5.08× *(peak)*

TABLE 6.5: Single-device V3 wall-clock (ms) on A100 and H100 vs. the 32-thread EPYC 7513 baseline.

versus 76), and substantially higher memory bandwidth (2 TB/s for A100, 3 TB/s for H100, versus 716.8 GB/s). The first axis extends the practical input range; the second and third reduce the absolute time at any given n. Table 6.5 reports the single-device cluster results across the extended input range.

Two observations follow. First, the EPYC CPU baseline at the cluster is competitive with the i7-13700K workstation baseline at the same n. The SLURM allocation gives the cluster job 32 worker threads (Section 5.2.3), which exceeds the workstation's 24-thread budget but does not exploit the EPYC's full per-node thread count; the cluster CPU baseline is therefore representative of the deployment configuration rather than of the EPYC's peak parallel capacity. Cluster speedup ratios are computed against this 32-thread cluster baseline. Reporting both baselines explicitly avoids the false impression that the cluster GPUs are weaker than the workstation GPU; the divisor has changed, not the dividend.

Second, contrary to the architectural expectation that the H100's higher memory bandwidth (3.0 TB/s versus 2.0 TB/s) should yield up to a 1.5× end-to-end advantage, the measured single-device times show the H100 only outperforming the A100 at the smallest input ($n = 1$M, where H100 is 1.16× faster) and *lagging* the A100 from $n = 10$M upward by a roughly constant 1.15–1.21× margin. The combined end-to-end ratio at the headline single-device size of $n = 100$M is approximately 0.83× (i.e. A100 is faster). This inversion is consistent with the per-phase attribution of Section 6.1.2: the dominant warp-per-bucket accumulation phase is latency- rather than bandwidth-bound at the partition sizes used here, so the H100's bandwidth advantage does not translate into proportional speedup, while the A100's slightly more mature driver pipeline appears to absorb host-orchestration costs more evenly. The H100 retains a kernel-level edge for the bandwidth-bound sort and accumulation kernels in isolation, but not for the end-to-end Pippenger pipeline at the input sizes evaluated.

6.3.2 Cluster Partition-Count Sweep

The larger device memory of the cluster GPUs lifts the K-sweep range from $\{2, 4, 8, 16\}$ to $\{8, 16, 32, 64, 128\}$. Table 6.6 reports the sweep at the cluster's middle-of-range input size of $n =$50M on a single A100.

$K = 8$	$K = 16$	$K = 32$	$K = 64$	$K = 128$
3,443.1	**3,389.8**	3,673.2	4,041.0	4,559.1

TABLE 6.6: K-sweep on a single A100 at n =50M. $K = 16$ is optimal (bold), 1.6% over $K = 8$.

The shift in optimum from the workstation to the cluster reflects the same two-sided trade-off described in Section 4.6.5, with both sides shifted. The upper bound on K from memory pressure is relaxed (80 GB versus 16 GB), and the lower bound from sort-efficiency loss is partially relaxed because the cluster GPUs have more SMs to absorb the parallelism of each shrunken sort. The empirical optimum is therefore $K = 16$, which delivers 3,389.8 ms versus $K = 8$'s 3,443.1 ms — a 1.6% improvement at $n = 50$M. The improvement is small in absolute terms but consistent with the analytical prediction; values above $K = 16$ degrade monotonically (a 32% regression at $K = 128$), confirming that the per-slot sort efficiency loss eventually dominates even on a 108-SM A100.

6.4 Multi-GPU Scaling

The V4 multi-GPU dispatch of Section 4.7.3 runs the per-device V3 pipeline on all visible GPUs and sums the projective partials on the host. The number of devices N ranges from one to eight on a single StarHPC node. The scaling behaviour across this range is the headline result of the campaign.

6.4.1 Headline V4 Performance Across the Full Input Range

Before decomposing the eight-device speedup along the device-count and input-size axes (Sections 6.4.2 and 6.4.3), Table 6.7 reports the production configuration's wall-clock time and speedup over the EPYC CPU baseline at every input size in the cluster campaign. The configuration is V4 with $K = 8$ partitions per device on 8× A100; the single-A100 column is included for reference and shows the marginal benefit of the multi-GPU dispatch over the best single-device configuration at the same n.

The 8-GPU speedup curve (Figure 6.2) climbs monotonically through the input range and plateaus near the 17.89× peak from $n = 100$M onward, as the per-device kernel work begins to dominate the host orchestration overhead. Below $n = 1$M the multi-GPU dispatch is unproductive: the per-device launch and merge costs exceed the CPU's time on its own modest workload, and the curve sits below 2×. Between $n = 5$M and $n = 50$M the speedup grows from 8.85× to 14.61× as the per-device pipelines amortise their per-call overhead. The crossover at which the dispatch becomes worthwhile in absolute terms (8-GPU faster than CPU by more than the host-side overhead) is at approximately $n = 200$K.

n	**EPYC CPU (ms)**	**1× A100 (ms)**	**1-GPU spd**	**8× A100 (ms)**	**8-GPU spd**
100K	69.5	66.5	1.05×	56.0	1.24×
200K	129.8	87.5	1.48×	71.3	1.82×
500K	292.0	210.2	1.39×	103.4	2.82×
1M	554.4	167.8	3.30×	166.9	3.32×
2M	994.7	240.9	4.13×	228.5	4.35×
5M	2,394.5	469.0	5.10×	270.5	8.85×
10M	4,409.6	811.2	5.43×	424.5	10.39×
20M	8,356.7	1,464.3	5.71×	617.3	13.54×
50M	20,035.8	3,443.1	5.82×	1,371.3	14.61×
100M	39,639.9	6,626.9	5.98×	2,215.4	17.89× *(peak)*
200M	77,184.2	N/A	—	4,389.8	17.58×

TABLE 6.7: Production V4 ($K = 8$) on 1× and 8× A100 across the cluster campaign. CPU is the 32-thread EPYC 7513; 200M exceeds single-A100 memory.

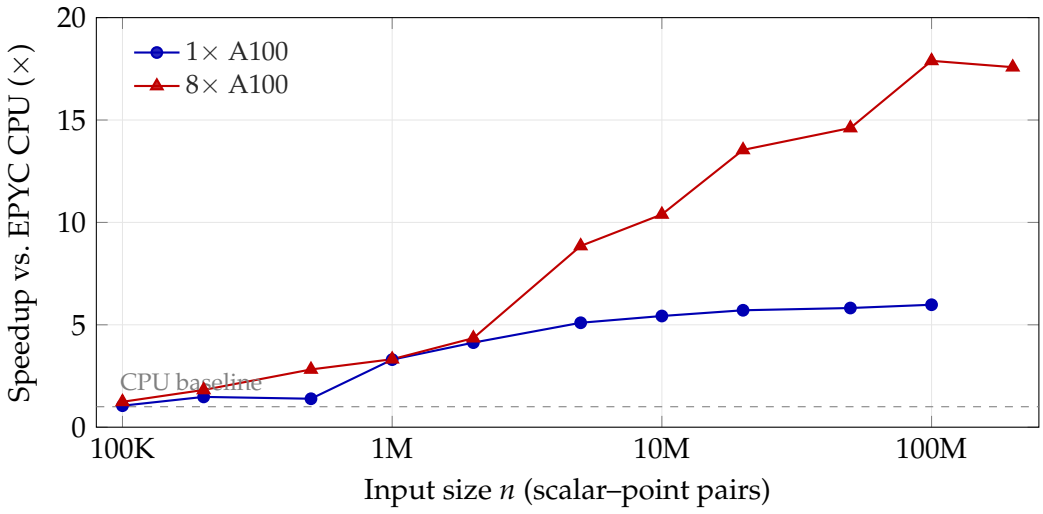


FIGURE 6.2: V4 speedup over the 32-thread EPYC CPU baseline for 1× and 8× A100 across n (solid lines). The horizontal dashed grey line at 1× marks the CPU baseline. Peak 17.89× at $n = 100$M.

Devices (N)	Time (ms)	Speedup vs. 1 GPU	Parallel efficiency	Speedup vs. CPU
1	6,626.9	1.00×	100%	5.98×
2	4,433.1	1.49×	75%	8.94×
4	3,066.1	2.16×	54%	12.93×
8	2,215.4	2.99×	37%	17.89× *(headline)*

TABLE 6.8: V4 strong scaling at n =100M on the A100 node; efficiency is speedup divided by device count.

Figure 6.3 reports the same comparison in absolute wall-clock terms. The CPU baseline grows nearly linearly in n (slope close to 1 in log–log space), reflecting the EPYC's bandwidth-bound traversal of the bucket array as n scales. The 8× A100 line grows more slowly, because additional input is absorbed by the per-device pipelines without proportional increase in wall time once the per-call overhead is amortised. The gap between the two lines is the headline: at $n = 200$M the CPU takes 77.2 s while 8× A100 finishes in 4.4 s, a 17.6× wall-clock reduction.

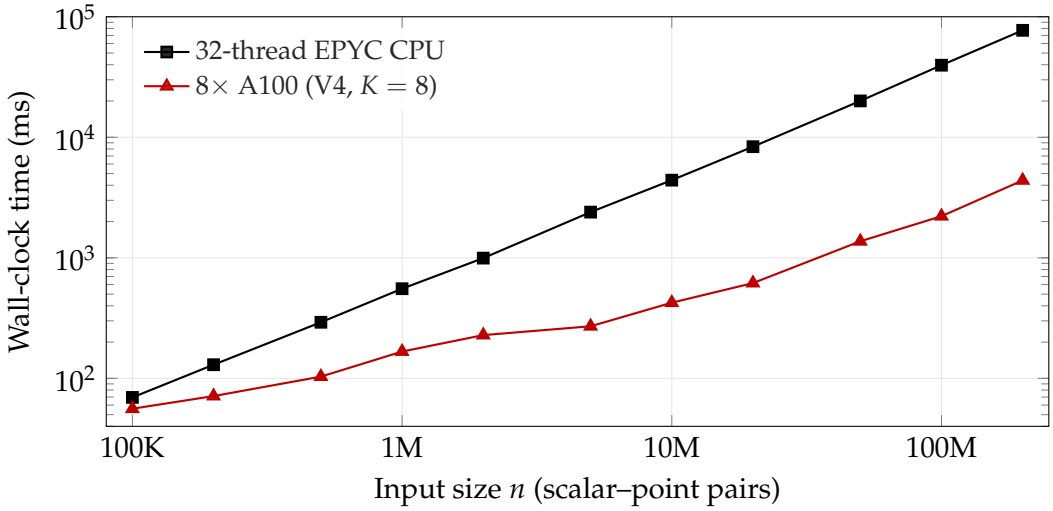


FIGURE 6.3: Absolute wall-clock comparison: 32-thread EPYC CPU vs. 8× A100 (V4, $K = 8$) across n. The diverging gap reaches 17.6× at n = 200M.

6.4.2 Strong Scaling at Fixed Input Size

Table 6.8 reports the wall-clock time and parallel efficiency of the V4 dispatch at a fixed input size of n =100M on the A100 node. Strong scaling holds the problem size constant and measures how the time decreases as devices are added.

The dispatch attains a maximum end-to-end speedup of 17.9× over the EPYC CPU baseline with all eight A100 devices engaged, the headline result of this thesis. The decomposition of that figure across the scaling curve carries the interpretive load: the

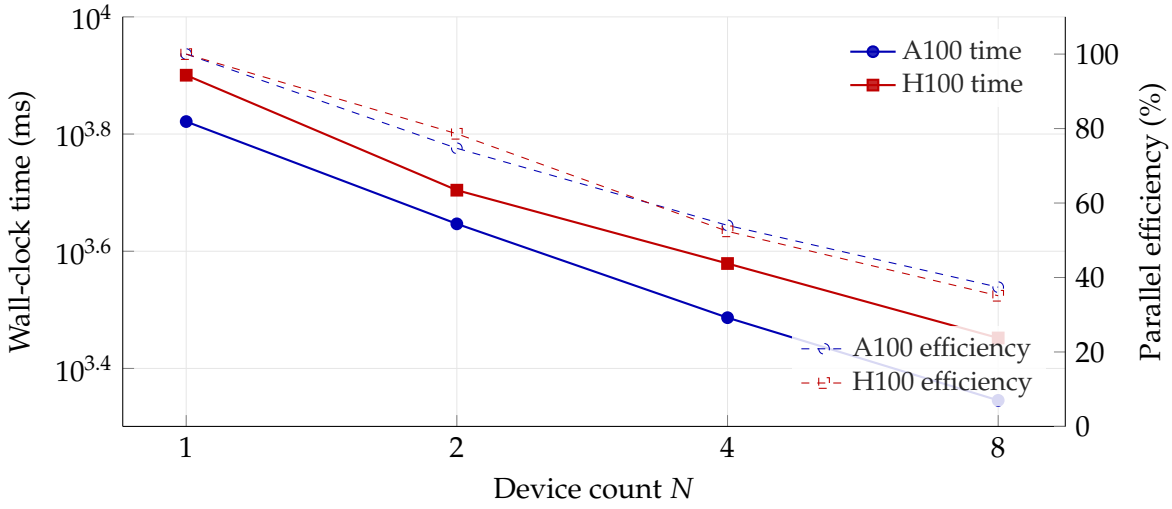


FIGURE 6.4: V4 strong scaling at n =100M on A100 and H100. Solid lines with filled markers = wall-clock time (left log axis); dashed lines with hollow markers = parallel efficiency, defined as (speedup vs. 1 GPU) ÷ device count, plotted against the right axis. Efficiency is 100% at $N = 1$ by definition and falls to ∼37% (35%) for A100 (H100) at $N = 8$.

speedup over a single A100 is sub-linear in N, reflecting the host-side costs that do not parallelise across devices. The parallel data conversion (Section 4.7.4) executes once regardless of device count and does not benefit from additional devices; the projective merge cost grows linearly in N, though it remains in the microsecond range; and per-device PCIe transfers must share host memory bandwidth, which can limit concurrency at high N. The combined effect is a parallel efficiency of approximately 37% at eight devices: respectable for a dispatcher with no inter-device communication, and consistent with the architectural argument of Section 4.3.4 that the bottleneck shifts from per-device kernel work to host orchestration as N grows.

6.4.3 Weak Scaling and the Multi-GPU Threshold

Strong scaling is the right framing for a Nova prover that issues a fixed-size MSM at every folding step, but a complementary weak-scaling perspective clarifies when the multi-GPU dispatch becomes mandatory rather than optional. Section 4.7.1 identified $n \approx$100M as the threshold above which a single 80 GB cluster device cannot accommodate the per-MSM intermediate state. Below this threshold, the multi-GPU dispatch is an optimisation; above it, the multi-GPU dispatch is the only path that runs at all.

The largest inputs in the campaign, n =200M, exercise the multi-GPU dispatch in its mandatory regime. Table 6.9 reports the eight-device wall-clock time and the speedup over the EPYC CPU baseline at the campaign's three largest sizes.

n	EPYC CPU (ms)	8× A100 (ms)	Speedup vs. CPU
50M	20,035.8	1,371.3	14.61×
100M	39,639.9	2,215.4	17.89× *(headline)*
200M	77,184.2	4,389.8	17.58×

TABLE 6.9: V4 on 8× A100 at the campaign's largest sizes; n =200M is multi-GPU-only.

6.5 Cross-Platform Comparison and Platform Selection Guidelines

The combined results across Sections 6.1 through 6.4 permit a comparative framing of when each platform is the appropriate target for a Nova prover deployment. Table 6.10 summarises the guidance.

Input regime	Recommended platform	Rationale
$n < 100$K	CPU only (extrapolated)	Below the campaign's smallest measured size; GPU dispatch overhead is expected to dominate at the sub-100K regime where the CPU's last-level cache holds the bucket array.
$100\text{K} \leq n < 10\text{M}$	Single consumer GPU (RTX 4080-class)	V3 reaches 4.06× over the i7-13700K baseline at n = 1M; multi-GPU adds dispatch overhead without proportional benefit.
$10\text{M} \leq n < 100\text{M}$	Single data-centre GPU (A100/H100 single-device)	Workstation memory budget is becoming binding; cluster bandwidth lowers absolute time.
$n \geq 100$M	Multi-GPU cluster (V4 multi-device dispatch)	Single-device memory budget is exhausted; the V4 dispatch is the only configuration that runs at all.
IPA prover round (two MSMs)	Single GPU with V4 batch API	Two left/right commitments overlap on slots 0 and 1; no multi-GPU needed for the typical IPA size.
Many independent provers	Multi-GPU with one prover per device	Inter-device dispatch with no inter-device communication; embarrassingly parallel.

TABLE 6.10: Recommended hardware tier by input size n; thresholds are campaign-specific.

Three closing observations qualify the table. First, the boundaries listed are crossover points observed in this campaign and not universal constants. A future driver release, a different host CPU, or a different curve will shift them. Second, the

recommendation table treats hardware availability as unconstrained; in practice, a Nova deployment that has access only to a workstation will run the workstation configuration regardless of input size, and the feature-gated build of Section 4.3.1 makes that fallback automatic. Third, the multi-GPU dispatch's headline 14× figure is achieved relative to the 32-thread EPYC CPU baseline configured by the SLURM allocation (Section 5.2.3). Against a weaker CPU baseline, the same dispatch would report a correspondingly higher speedup ratio without any change in absolute wall-clock time; the figure is therefore architecture-relative rather than absolute, and the wall-clock numbers in Table 6.9 are the durable measurement of the campaign's contribution. cutoff in

Chapter 7

Conclusion

This thesis has presented a comparative performance analysis of multi-scalar multiplication acceleration for the Pallas elliptic curve, evaluated across a CPU baseline, a single consumer GPU, and a multi-GPU HPC cluster. The implementation integrates with the Nova folding-scheme prover through a feature-gated substitution at Nova's commitment trait, leaving the upstream codebase otherwise unmodified, and the GPU pipeline itself proceeds through four architectural design points whose empirical contributions have been quantified in Chapter 6. This closing chapter summarises what the thesis claims to have contributed, identifies the limitations of the work as it stands, and outlines the directions in which it can be extended.

7.1 Summary of Contributions

The thesis makes four contributions, ordered from the most concrete to the most general.

The first is a production-quality GPU implementation of multi-scalar multiplication for the Pallas curve, integrated into a fork of the Microsoft Research Nova reference implementation through the `DlogGroupExt` trait. The implementation is feature-gated, allowing direct A/B comparison against the upstream CPU code path through a single Cargo flag, and is structured as two composable components: a `pallas-gpu` crate (a Rust wrapper that builds and statically links the CUDA kernel library) and a Nova integration layer, which together isolate proof-system-agnostic logic from system-specific dispatch. The `pallas-gpu` crate can in principle be linked into any Rust project requiring Pallas MSM.

The second is the architectural progression itself, documented in Chapter 4 as four design points. V1 establishes the upper performance bound of shared-memory bucket layouts at 16.7% hardware occupancy on the RTX 4080, attaining approximately 2.05$\times$ over the CPU baseline before being retired in favour of V2, which lifts bucket state into global memory and replaces per-bucket spinlocks with sort-based pre-grouping using CUB radix sort over 12-bit signed-digit windows. V3 adds partition-based concurrent

streams that overlap the V2 kernel chain across K partitions of the same MSM. V4 generalises the dispatch in two directions: a batch API that exposes proof-system-level concurrency to the device on the IPA L/R workload, and a multi-GPU extension that partitions input across all visible devices, runs an independent V3 pipeline per device, and merges projective partials on the host without inter-device communication. The per-window logic of V2 is preserved unchanged through both subsequent versions, which makes the dispatch progression an additive rather than a replacement structure.

The third is a multi-GPU dispatch model whose safety properties are independently of interest. The transition from a persistent-pool memory model to per-call allocation (Section 4.3.3) was driven by an observed correctness failure (`cudaErrorMemoryAllocation` on the second multi-GPU call after a single-GPU run had populated device 0), not by a performance argument. The resulting design, in which each multi-GPU worker thread operates on its own thread-local slot pool and the only inter-thread coordination is `std::thread::join`, is portable to any Rust application requiring concurrent per-device CUDA dispatch and is independent of the Nova integration in which it was developed.

The fourth is the empirical campaign summarised in Chapter 6. The headline figures, namely an early-design 2.05× V1 speedup, a 3.7× V2 speedup (peak at $n = 200$K), a 4.9× V3 speedup at $n = 20$M over the consumer-CPU baseline on the RTX 4080, and a 17.9× V4 speedup at $n = 100$M over the cluster-CPU baseline on an eight-device A100 node, position the implementation among the published GPU-accelerated MSM implementations for prime-order curves of cryptographic interest. The platform selection guidelines of Section 6.5 map the empirical results onto deployment-time hardware decisions, and are directly applicable to any Nova-based prover that targets the Pallas curve.

7.2 Limitations and Future Work

The scope of the thesis is bounded along several axes, each of which suggests a corresponding extension.

Curve specialisation. The implementation targets the Pallas curve specifically and uses field-arithmetic primitives from `halo2curves` that are specialised to the Pallas base field. Extending the work to a second curve, most naturally to BN254 (which appeared in earlier development under the same Pippenger superstructure) or to BLS12-381, would test whether the architectural choices documented in Chapter 4 generalise

across curves of different word size, security level, and base-field structure. The four-phase pipeline and the V2 sort-based kernel chain are curve-agnostic in their algorithmic structure; only the field-arithmetic primitives and the choice of window width would need to be revisited.

Proof-system specialisation. The thesis evaluates MSM in the context of Nova's folding scheme and its IPA prover round. The same implementation could be evaluated against the MSM call patterns of Halo 2, Plonky2, or HyperPlonk [35]–[37], where the per-call input size distribution and the number of MSMs per proof differ from Nova's profile. The batch API of Section 4.7.2 is designed to absorb such differences without modification to the per-window logic, but its empirical advantage has been measured only on Nova's L/R commitment pair.

Profiler-driven verification. Section 4.5.5 introduced the single-array layout with arithmetic negation as a design choice motivated by an L2 cache pressure argument, but the underlying L2 hit rate has not been verified through Nsight profiling. A complete characterisation would either confirm the design rationale empirically or, in the event of a measurable hit-rate degradation, identify the parameter regime in which the dual-array layout (with its $2n$ memory footprint and branched load path) would be preferable. The cluster GPUs' larger L2 capacities make the question more interesting at scale than it is on the workstation, since the favourable layout depends on the working set fitting in the cache.

Optimisation paths not pursued. The CUB-based accumulation alternative discussed at the close of Section 4.5.5 is a documented future direction. The layout would replace the global-memory bucket array with a shared-memory bucket array sized below the SM budget, processed in batches whose intermediate sums are merged in global memory after each batch completes. Initial estimates place the speedup at 3–5$\times$ over the current global-memory layout, contingent on confirming that the merge cost does not erase the on-chip locality gain. The work is well-defined but was not undertaken within the scope of this thesis. A related avenue is the adoption of warp-shuffle intrinsics (`__shfl_down_sync`) in the warp-level tree reduction; Section 4.5.7 noted that the projective point representation does not pack cleanly into shuffle-sized registers, but a partial packing of single field elements during the reduction may still be profitable.

Multi-node extension. The multi-GPU dispatch of Section 4.7.3 scales to all devices visible within a single SLURM allocation, which on the StarHPC cluster is eight GPUs on a single compute node. A multi-node extension would require either MPI for inter-node coordination or NCCL for inter-node collective reductions, both of which were

explicitly out of scope for this thesis (Section 4.3.4). The single-node dispatch achieves its safety properties precisely because no inter-device communication occurs; introducing inter-node communication would re-introduce the cross-thread coordination that the per-call allocation model was designed to avoid, and would require a corresponding redesign of the dispatch contract.

End-to-end prover benchmarks. Chapter 6 reports MSM-level speedups, which is the appropriate granularity for evaluating an MSM acceleration kernel. The corresponding end-to-end Nova prover speedups, in which the MSM acceleration is composed with Nova's witness construction, Fiat–Shamir transcript handling, and folding logic, are observable in principle by running the full prover under the `pallas-gpu` feature flag, but were not collected as part of the campaign. The MSM-to-total-prover-time ratio reported by [22] and others suggests that the end-to-end speedup will be smaller than the MSM speedup by a factor that depends on the proof's witness size and folding depth, but a measured figure is the appropriate way to settle the question for any specific deployment scenario.

Closing remark. The architectural pattern that emerged from this thesis, namely an unchanging per-window kernel chain wrapped in progressively coarser dispatch layers, is suggestive of how GPU acceleration of cryptographic primitives might evolve more broadly. The hard work is in the inner kernel, where the structural constraints of the GPU's execution and memory models cannot be papered over; the dispatch layers that follow extract parallelism the inner kernel does not itself expose, and they compose cleanly precisely because they do not modify the kernel. Whether this pattern generalises to other cryptographic primitives, to other accelerator classes, or to other distributed-systems configurations is an empirical question that the present work has not attempted to answer, but the framework it establishes is one in which that question can usefully be posed.

Bibliography

[1] National Institute of Standards and Technology, *Privacy-enhancing cryptography (PEC)*, NIST Cryptographic Technology Group, Accessed: 2025-01-20, 2025. [Online]. Available: https://csrc.nist.gov/projects/pec.

[2] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989. DOI: 10.1137/0218012.

[3] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *Journal of the ACM*, vol. 38, no. 3, pp. 690–728, 1991. DOI: 10.1145/116825.116852.

[4] Grand View Research, *Zero knowledge proof market size, share & trends analysis report*, Grand View Research Industry Report, Accessed: 2025-01-20, 2024. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/zero-knowledge-proof-market-report.

[5] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Selected Areas in Cryptography – SAC 2005*, ser. Lecture Notes in Computer Science, vol. 3897, Springer, 2006, pp. 319–331. DOI: 10.1007/11693383_22.

[6] D. Hopwood, *The Pasta curves for Halo 2 and beyond*, Electric Coin Company Blog, Accessed: 2026-04-09, 2020. [Online]. Available: https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/.

[7] T. Lu, W. Cheng, R. Yu, C. Chen, W. Fan, S. Wang, Z. Wang, and W. Chen, "cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, pp. 194–220, 2023. DOI: 10.46586/tches.v2023.i3.194-220.

[8] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, *PipeMSM: Hardware acceleration for multi-scalar multiplication*, Cryptology ePrint Archive, Paper 2022/999, 2022. [Online]. Available: https://eprint.iacr.org/2022/999.

[9] Z. Ji, J. Zhao, P. Gao, X. Yin, and L. Ju, "Accelerating large-scale multi-scalar multiplication for efficient zero knowledge proofs with multi-GPU systems," *Integration, the VLSI Journal*, 2024. DOI: 10.1016/j.vlsi.2024.102274.

[10] J. Luo, Z. Ni, G. Shi, Y. Zhang, and J. Fang, "Accelerating multi-scalar multiplication for efficient zero knowledge proofs with multi-GPU systems," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 3, ACM, 2024. DOI: 10.1145/3620666.3651364.

[11] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu, "GZKP: A GPU accelerated zero-knowledge proof system," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 2, ACM, 2023, pp. 340–353. DOI: 10.1145/3575693.3575711.

[12] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, ser. Lecture Notes in Computer Science, vol. 9666, Springer, 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11.

[13] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[14] V. Buterin, *An incomplete guide to rollups*, vitalik.eth.limo, Accessed: 2025-01-20, 2021. [Online]. Available: https://vitalik.eth.limo/general/2021/01/05/rollup.html.

[15] World Wide Web Consortium, *Verifiable credentials data model v2.0*, W3C Recommendation, Accessed: 2025-01-20, 2024. [Online]. Available: https://www.w3.org/TR/vc-data-model-2.0/.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Association, 2014, pp. 781–796.

[17] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *Advances in Cryptology – CRYPTO 2022*, ser. Lecture Notes in Computer Science, Cryptology ePrint Archive, Paper 2021/370, vol. 13510, Springer, 2022, pp. 359–388. DOI: 10.1007/978-3-031-15985-5_13.

[18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985. DOI: 10.1090/S0025-5718-1985-0777282-X.

[19] P. Valiant, "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency," in *Theory of Cryptography – TCC 2008*, ser. Lecture Notes in Computer Science, vol. 4948, Springer, 2008, pp. 1–18. DOI: 10.1007/978-3-540-78524-8_1.

[20] N. Pippenger, "On the evaluation of powers and related problems," *SIAM Journal on Computing*, vol. 9, no. 2, pp. 230–250, 1980. DOI: 10.1137/0209017.

[21] Y. El Housni and A. Guillevic, "Faster Montgomery multiplication and multi-scalar-multiplication for SNARKs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, pp. 504–521, 2023. DOI: 10.46586/tches.v2023.i3.504-521.

[22] T. Verma, Y. Yuan, N. Talati, and T. Austin, "ZKProphet: Understanding performance of zero-knowledge proofs on GPUs," in *2025 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2025, pp. 142–154. DOI: 10.1109/IISWC66894.2025.00021.

[23] S. Bowe, J. Grigg, and D. Hopwood, *Halo: Recursive proof composition without a trusted setup*, Cryptology ePrint Archive, Paper 2019/1021, 2019. [Online]. Available: https://eprint.iacr.org/2019/1021.

[24] D. F. Aranha, Y. El Housni, and A. Guillevic, "A survey of elliptic curves for proof systems," *Designs, Codes and Cryptography*, vol. 91, pp. 3333–3378, 2023. DOI: 10.1007/s10623-022-01135-y.

[25] "MSMAC: Accelerating multi-scalar multiplication for zero-knowledge proof," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, ACM, 2024. DOI: 10.1145/3649329.3658468.

[26] O. Ozerk, C. Elgezen, A. C. Mert, E. Öktem, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," in *The Journal of Supercomputing*, See also: https://eprint.iacr.org/2022/281 for BN254 Montgomery multiplier results on Xilinx Alveo., vol. 78, Springer, 2022, pp. 11 201–11 226. DOI: 10.1007/s11227-022-04410-8.

[27] Amazon Web Services, *Amazon EC2 F1 instances*, AWS Documentation, Accessed: 2025-01-20, 2024. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[28] ——, *Amazon EC2 F2 instances*, AWS Documentation, Accessed: 2025-01-20, 2024. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f2/.

[29] Matter Labs, *Accelerating MSM on GPU: ZPrize multi-scalar multiplication submission*, GitHub Repository, Submitted to the ZPrize Multi-Scalar Multiplication Track. Accessed: 2025-01-20, 2022. [Online]. Available: https://github.com/matter-labs/z-prize-msm-gpu.

[30] Ingonyama, *ICICLE: A library for GPU acceleration of zero-knowledge proofs*, GitHub Repository, Accessed: 2025-01-20, 2023. [Online]. Available: https://github.com/ingonyama-zk/icicle.

[31] NVIDIA Corporation, *NVIDIA Ada GPU architecture: Designed to deliver outstanding gaming and creating, professional graphics, AI, and compute performance*, NVIDIA Whitepaper, v2.02, Accessed: 2026-04-22, 2022. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[32] ——, *NVIDIA A100 Tensor Core GPU datasheet (SXM4 and PCIe form factors)*, NVIDIA Datasheet, Accessed: 2026-04-22, 2021. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[33] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and O. Giroux, *NVIDIA Hopper architecture in-depth*, NVIDIA Technical Blog, Accessed: 2026-04-22, 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[34] D. J. Bernstein and T. Lange, *Explicit formulas database*, https://hyperelliptic.org/EFD, Accessed: 2025-01-20.

[35] Electric Coin Company, *The Halo 2 zero-knowledge proving system*, https://github.com/zcash/halo2, Accessed: 2026-05-12, 2022.

[36] Polygon Zero Team, *Plonky2: Fast recursive arguments with PLONK and FRI*, https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf, Accessed: 2026-05-12, 2022.

[37] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," in *Advances in Cryptology – EUROCRYPT 2023*, Springer, 2023, pp. 499–530. DOI: 10.1007/978-3-031-30617-4_17.