COMPUTATIONAL STATISTICS AND GPU ACCELERATION

By

ANDREAS S. BAMPOURIS

A THESIS

Submitted to the Department of Statistics
of the Athens University of Economics and Business
in partial fulfilment of the requirements for
the degree of Master of Science in Applied Statistics

Athens, Greece
July 2025

ΥΠΟΛΟΓΙΣΤΙΚΗ ΣΤΑΤΙΣΤΙΚΗ ΚΑΙ ΕΠΙΤΑΧΥΝΣΗ ΜΕΣΩ GPU

ΑΝΔΡΕΑΣ Σ. ΜΠΑΜΠΟΥΡΗΣ

ΔΙΑΤΡΙΒΗ

Που υποβλήθηκε στο Τμήμα Στατιστικής
του Οικονομικού Πανεπιστημίου Αθηνών
ως μέρος των απαιτήσεων για την απόκτηση
Διπλώματος Μεταπτυχιακών Σπουδών στην Εφαρμοσμένη Στατιστική

Αθήνα, Ελλάδα
Ιούλιος 2025

DEDICATION

To my father, Sotirios,
who first taught me how to use a computer.

ACKNOWLEDGEMENTS

My journey into the world of data has been motivated by my desire to master the algorithms and the mathematics behind modern machine learning. I soon discovered that the discipline of statistics provided one of the most robust vehicles for that pursuit. This thesis represents the culmination of my deep dive into statistics, focusing on its natural intersection with computer science: computational statistics.

The stimulating academic environment at the Athens University of Economics and Business was instrumental to this work. I am profoundly thankful to my supervisor, Professor Athanasios Yannacopoulos, who encouraged and empowered my decision to tackle a topic that might seem unconventional. This gratitude also extends to my professional environment over the years, where I have been fortunate to work with colleagues who have helped foster my growth not only as a professional but also as an academic.

On a personal note, I am eternally grateful to my family and friends for their immense patience throughout this ambitious undertaking. I especially want to thank my father, whose gentle push was crucial in seeing this work through to its completion. Finally, to Giouli: this thesis would not have been possible without your unwavering support and understanding.

VITA

Andreas is a technology enthusiast with a passion for software engineering, applied statistics, and their intersection. He received his diploma in Electrical and Computer Engineering from the University of Patras in 2018. His early academic interests focused on machine learning, pattern recognition, and high-performance parallel computing, topics which he explored in his diploma thesis, Bampouris (2018).

He has since worked for over five years as a Machine Learning Engineer and Data Scientist, building and maintaining large-scale data pipelines and production machine learning systems. He is a proponent of ongoing and self-motivated learning, enjoys coding and tinkering, and has interests ranging from science and technology, to history and linguistics.

ABSTRACT

Andreas S. Bampouris

Computational Statistics and GPU Acceleration

July 2025

Modern statistical methods often become computationally prohibitive as data volumes and model complexity grow. This thesis examines how Graphics Processing Unit (GPU) acceleration can expand the practical scale of such methods. We organize the work around three components: (1) a theoretical analysis of computational bottlenecks in two widely-used but immensely intensive methods, Kernel Methods and Gradient Boosting, and the algorithmic redesign required for efficient GPU execution; (2) an empirical validation of the potential performance gains by benchmarking two state-of-the-art, GPU-accelerated libraries, Falkon and XGBoost, against CPU-based baselines on real-world datasets to quantify speedups and assess effects on predictive accuracy; and (3) an implementation-oriented overview of the enabling software frameworks, developing a massively parallel Markov Chain Monte Carlo (MCMC) sampler in CUDA as an illustrative case study.

Results indicate that substantial performance gains are attainable on commodity GPU hardware with no material loss in statistical accuracy when algorithms are reformulated to exploit fine-grained parallelism and memory hierarchies. More broadly, the findings underscore that scalability in statistics is as much an engineering problem as it is a methodological one: algorithm design, data layout, and hardware architecture must be considered jointly. By moving from theory, to empirical evidence, to the underlying engineering, this thesis aims to bridge the gap between advanced statistical modelling and high-performance computing, and provides the tools to not only leverage but also contribute to this expanding field.

ΠΕΡΙΛΗΨΗ

Ανδρέας Σ. Μπαμπούρης

Υπολογιστική Στατιστική και Επιτάχυνση μέσω GPU

Ιούλιος 2025

Η πρακτική εφαρμογή σύγχρονων στατιστικών μεθόδων καθίσταται συχνά υπολογιστικά απαγορευτική, λόγω του διαρκώς αυξανόμενου όγκου των δεδομένων και της πολυπλοκότητας των μοντέλων. Η παρούσα εργασία εξετάζει πώς η επιτάχυνση μέσω Μονάδων Επεξεργασίας Γραφικών (GPU) μπορεί να διευρύνει το πεδίο εφαρμογής τέτοιων μεθόδων. Η εργασία δομείται σε τρεις άξονες: (1) τη θεωρητική ανάλυση των υπολογιστικών «σημείων συμφόρησης» σε δύο ευρέως διαδεδομένες αλλά και εξαιρετικά απαιτητικές μεθόδους, τις Μεθόδους Πυρήνα (Kernel Methods) και το Gradient Boosting, καθώς και του αλγοριθμικού ανασχεδιασμού που απαιτείται για την αποδοτική τους εκτέλεση σε GPU, (2) την εμπειρική επικύρωση των δυνητικών κερδών απόδοσης, μέσω της συγκριτικής αξιολόγησης δύο βιβλιοθηκών λογισμικού αιχμής σε GPU, των Falkon και XGBoost, έναντι των αντίστοιχων υλοποιήσεών τους σε CPU, ποσοτικοποιώντας την επιτάχυνση σε πραγματικά σύνολα δεδομένων, και (3) την επισκόπηση των πλαισίων λογισμικού που καθιστούν εφικτές τέτοιες υλοποιήσεις, χρησιμοποιώντας ως ενδεικτική μελέτη περίπτωσης την υλοποίηση ενός μαζικά παράλληλου δειγματολήπτη Markov Chain Monte Carlo (MCMC) σε CUDA.

Τα αποτελέσματα καταδεικνύουν ότι η επίτευξη σημαντικών κερδών απόδοσης σε ευρέως διαθέσιμο υλικό GPU είναι εφικτή χωρίς καμία ουσιαστική απώλεια στατιστικής ακρίβειας, υπό την προϋπόθεση ότι οι αλγόριθμοι έχουν ανασχεδιαστεί ώστε να αξιοποιούν αποδοτικά τον παραλληλισμό και τις ιεραρχίες μνήμης. Γενικότερα, τα ευρήματα τεκμηριώνουν ότι η κλιμακωσιμότητα μεθόδων στατιστικής αποτελεί πρόβλημα τόσο μηχανικής λογισμικού, όσο και μεθοδολογίας: ο ανασχεδιασμός του αλγορίθμου, η δομή των δεδομένων, και η αρχιτεκτονική του υλικού απαιτούν συνδυαστική αντιμετώπιση. Προχωρώντας από τη θεωρία στην εμπειρική τεκμηρίωση και, τέλος, στην τεχνολογία της υλοποίησης, η παρούσα εργασία στοχεύει να γεφυρώσει το χάσμα μεταξύ της προηγμένης στατιστικής μοντελοποίησης και της υπολογιστικής υψηλών επιδόσεων, παρέχοντας τα εφόδια όχι μόνο για την αξιοποίηση των GPU, αλλά και για τη συνεισφορά στο ταχέως αναπτυσσόμενο αυτό πεδίο.

Contents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

The field of applied statistics is fundamentally intertwined with computation. Throughout its endeavour to analyse and model data and derive meaningful patterns from it, it has to apply complex mathematical transformations which require significant computational power. This connection has become increasingly critical as we find ourselves in an era of unprecedented data proliferation. From the petabytes of genomic data in bioinformatics to the high-frequency trading data in computational finance and the vast, unstructured text of the internet, the sheer volume and complexity of information have grown exponentially across every domain of science and industry.

This data deluge has catalysed a shift toward more sophisticated statistical methods. Simple linear models, once the bedrock of statistical analysis, often fall short of capturing the intricate, non-linear relationships hidden within these massive datasets. Consequently, practitioners are increasingly adopting more flexible and computationally demanding techniques to extract meaningful insights and power their applications.

However, this progress hinges on a critical trade-off that introduces a significant challenge: the very complexity that makes these models so effective also renders them enormously demanding on computational resources. The practical application of a state-of-the-art algorithm on a dataset with millions or billions of data points can transform a minutes-long analysis into a process that

takes hours, days, or becomes entirely intractable. This computational bottleneck hinders the pace of research and the deployment of cutting-edge models in real-world applications. A statistical method, after all, is of little use if it cannot be practically applied.

Solving this computational bottleneck has necessarily required a parallel evolution in hardware. One of the most transformative breakthroughs in this evolution has been the Graphics Processing Unit (GPU). Since their mainstream introduction in the 1990s, the GPU's unique architecture provided the raw power needed to drive modern statistical techniques. Its impact over the last decade is particularly evident on the field of Artificial Intelligence, where the GPU has enabled the practical implementation and scaling of deep neural networks, a class of architectures that are themselves built upon foundational statistical principles.

This raises a critical question: if GPUs revolutionized AI by accelerating one class of statistical methods, what potential do they hold for the broader landscape of statistical computation? This work is motivated by a dual interest in statistics and computer science and investigates the implementation of diverse classes of statistical methods on GPUs. Our main hypothesis is that the widespread adoption of GPUs can usher in a new era of data analysis. Ultimately, this thesis aims to help bridge the gap between the ambition of modern statistical theory and the limits of current computational practices by leveraging the massive parallel processing power enabled by modern engineering.

1.2 Historical Context

The evolution of computation is inextricably linked to the history of statistics. The two fields have co-evolved, with the need to analyse data and quantify uncertainty serving as a primary catalyst for innovation in calculation for centuries.

Before the 20th century, the practice of statistics was defined by immense manual effort. Pioneers like Karl Pearson, who laid much of the theoretical groundwork for the modern field, conducted their work with pen, paper and logarithmic tables. As the field progressed with figures such as R. A. Fisher, the need to aid the process of calculation became increasingly evident. Yet, the early devices available at the time only automated individual calculations. The theoretical brilliance of those early statisticians was constrained by the painstaking labour required for every analysis, a

Figure 1.1: The Antikythera mechanism, an ancient analogue computer, demonstrates the long-held ambition to mechanize complex calculations.

reality that severely limited the scale and complexity of the problems they could address.

The ambition to overcome these manual constraints has a long history. Precedents for mechanical calculation, such as the Antikythera mechanism, date back to antiquity. However, the true conceptual predecessors to the modern computer emerged in the 19th century with Charles Babbage's Difference Engine and its more ambitious successor, the Analytical Engine. It was for this latter machine that Ada Lovelace wrote what is considered the first computer program, an algorithm to calculate Bernoulli numbers, demonstrating an early understanding that these machines could transcend simple arithmetic and automate complex sequences of calculations.

Practical needs soon drove further innovation. The daunting challenge of processing the 1890 U.S. Census became the first major application of automated data processing. Faced with a data problem that would have taken over a decade to analyse by hand, Herman Hollerith developed an electromechanical tabulating machine that reduced the task to a matter of months. The success of his invention led to the company that would eventually become IBM. Yet, the true turning point arrived a few decades later. The foundational principles of computation from figures like Alan Turing converged with the urgent, immense computational demands of World War II. This led directly to the development of the first general-purpose electronic computers, such as ENIAC, launching the modern computing age.

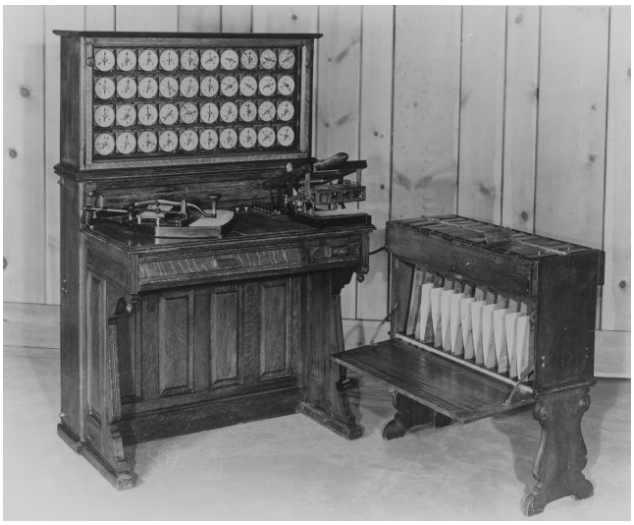


Figure 1.2: Herman Hollerith's electromechanical tabulating machine, created to solve the 1890 U.S. Census crisis and marking a key step in automated data processing.

The post-war era was dominated by the *von Neumann* architecture, which established the Central Processing Unit (CPU) and its sequential model as the heart of the computer. This new paradigm of programmable machines, along with the creation of the first high-level languages designed for scientific computing like *Fortran*, directly enabled the birth of statistical software. At research hubs like Bell Labs, the development of the powerful *C* programming language provided tools to write not only the Unix operating system but also foundational statistical packages like *S*, the predecessor to *R*. For decades, the field advanced in lockstep with Moore's Law, as ever-faster CPUs from companies like Intel, supported by numerical libraries like BLAS, allowed for the analysis of progressively larger datasets.

Figure 1.3: The personal computer democratized computation, enabling statisticians to run powerful software like R and Python directly from their desks.

By the mid-2000s, however, the physical limitations of silicon brought the era of exponential clock speed improvements to an end. To continue increasing performance, the industry pivoted from making single cores faster to putting multiple cores on a single chip. For statisticians and programmers

alike, the "free lunch" of automatically faster software was over. To harness the power of new hardware, algorithms themselves now had to be re-engineered for parallel execution.

It was in this new landscape that the GPU emerged as a transformative force. Originally a niche component designed for rendering video game and cinema graphics by companies like NVIDIA, it was soon discovered that its architecture, built to perform simple calculations on millions of pixels in parallel, was perfectly suited for the data-parallel tasks that defined modern computational statistics. Today, GPUs are at the epicentre of the AI revolution, powering the training and inference of massive generative models, such as Large Language Models (LLMs).

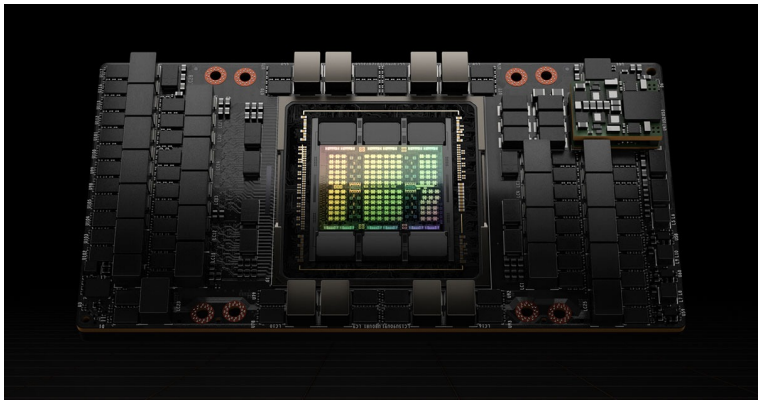


Figure 1.4: The NVIDIA H100 Tensor Core GPU, an architecture designed for massively parallel computation, which has become foundational for training large-scale artificial intelligence models.

1.3 Foundations of the GPU Architecture

The transformative potential of the Graphics Processing Unit (GPU) in accelerating complex statistical calculations stems from its massively parallel design. This section introduces the foundational ideas which are required to appreciate the paradigm shift enabled by the GPU. We examine the key distinctions between the CPU and the GPU, explore the GPU's execution model and memory hierarchy, and conclude by discussing the software abstractions that make its power accessible for statistical computing.

1.3.1 The *von Neumann* architecture and the GPU

The foundation of modern computing is the *von Neumann* architecture, a model centred around a Central Processing Unit (CPU), a unified Memory Unit for both program instructions and data, and Input/Output (I/O) devices, all interconnected by buses. A key characteristic of this design is its specific memory hierarchy and its approach to computation, which is primarily done sequentially, one instruction at a time.

Within this architecture, the CPU is engineered for low-latency access and the rapid execution of a single task or a few concurrent tasks. CPU cores are complex and powerful, equipped with sophisticated control flows and large cache memories. To optimize performance and handle diverse, general-purpose workloads effectively, CPUs employ techniques such as branch prediction and speculative execution, which are crucial for minimizing latency in sequential instruction streams.

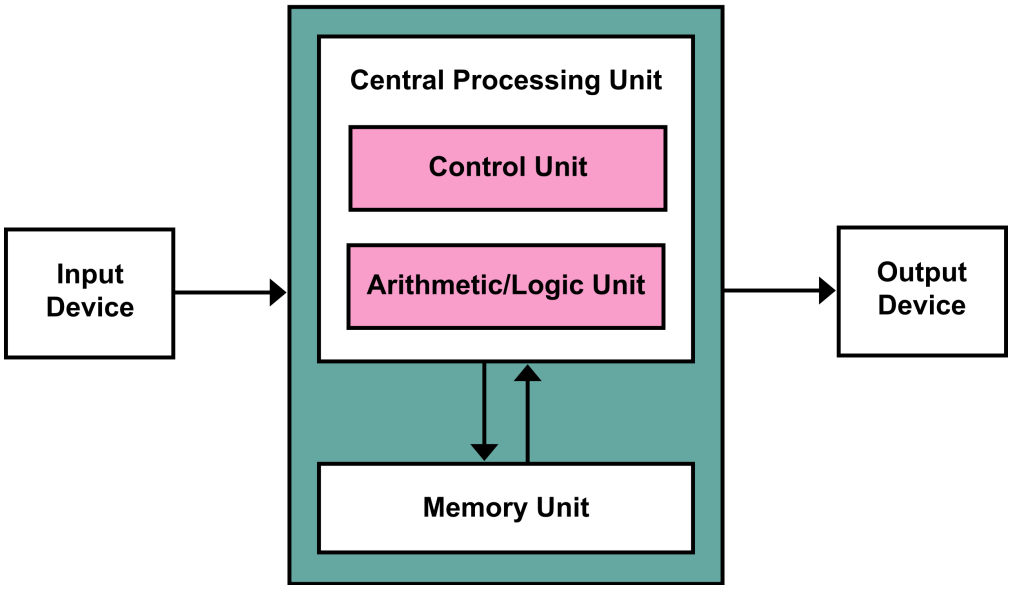


Figure 1.5: Scheme of the standard von Neumann architecture.

A significant limitation of this design is the *von Neumann* bottleneck. This arises because program instructions and data share the same memory space and the same buses to transit *to* and *from* the CPU. Consequently, they cannot be fetched simultaneously, often forcing the CPU to wait for data or instructions to be transferred from memory, thereby constraining overall processing speed.

To address this limitation, particularly for computationally intensive tasks, the GPU emerged to aug-

ment the *von Neumann* model. GPUs act as specialized, massively parallel co-processors allowing the CPU to offload specific types of workloads. This enables the CPU to focus on its strengths: general-purpose computations, sequential tasks, and overall system management.

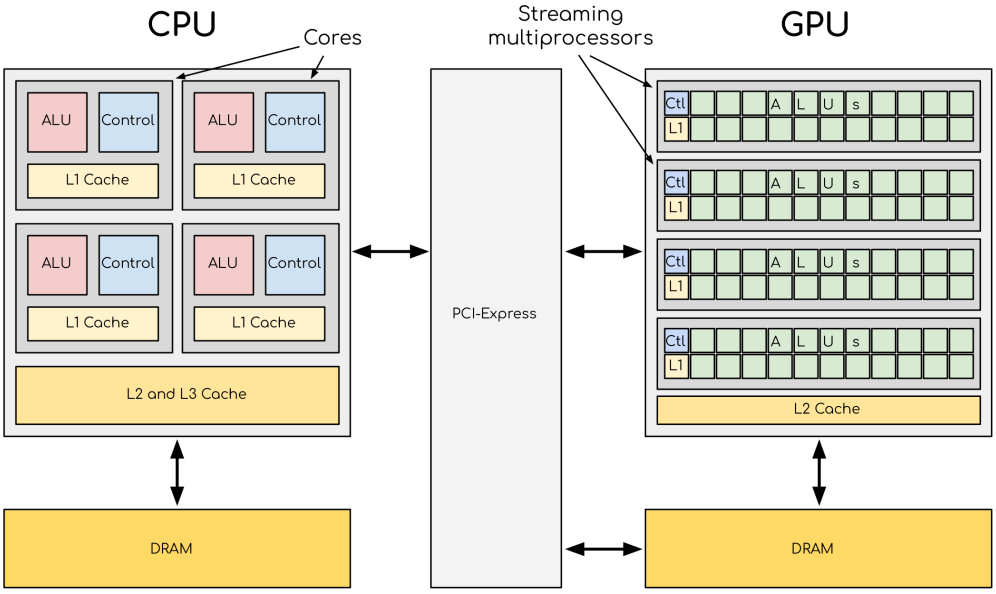


Figure 1.6: Comparison between CPU and GPU architecture.

GPUs help overcome the *von Neumann* bottleneck for suitable tasks by being throughput-oriented. They are engineered for high computational throughput on problems that can be broken down into many independent operations that can be performed simultaneously. This is achieved through a massively parallel architecture. Instead of a few powerful cores like a CPU, a GPU contains hundreds or even thousands of simpler cores. In NVIDIA technology, which we explore on this thesis, these cores are typically grouped into Streaming Multiprocessors (SMs), where each SM acts as an independent processor. This design allows a single GPU to manage and execute thousands of threads concurrently.

Furthermore, GPUs feature dedicated high-bandwidth memory (VRAM), which is optimized for the rapid access and manipulation of large datasets. This dedicated memory allows the GPU to perform its computations without constantly contending with the CPU for access to the main system memory (RAM), thereby bypassing the primary bottleneck for those specific, data-parallel operations.

While GPUs offer this immense parallel power, their design involves trade-offs. They have smaller

local memory per processing unit compared to CPU caches, and data transfers between RAM and the GPU's VRAM can be slow relative to the GPU's internal computational speed. However, for tasks where the GPU excels, the significant speedup from parallel processing often far outweighs the latency introduced by inbound and outbound data transfers.

In this augmented model, the CPU often acts as an orchestrator, managing the workflow and delegating parallelizable portions to the GPU. The GPU executes the computation and returns the results to main memory for the CPU to utilize. Thus, GPUs don't fundamentally replace the *von Neumann* architecture, but extend it to create a powerful heterogeneous computing environment. CPUs continue to excel at complex sequential logic and task management, while GPUs provide the muscle for massive parallel computation, allowing modern systems to tackle increasingly demanding computational challenges.

1.3.2 GPU Parallel Execution Model

GPUs primarily use the Single Instruction, Multiple Thread (SIMT) model. This paradigm allows a single instruction to be issued to a large number of threads, which then execute it concurrently, each operating on distinct data. While building upon principles seen in earlier Single Instruction, Multiple Data (SIMD) architectures, SIMT offers greater flexibility in managing thread execution.

Code designed for GPU execution is encapsulated within a *kernel*, which is essentially a function compiled to run on the GPU *device*. When the *host* CPU launches a kernel, it defines a grid of thread blocks. Each thread block, in turn, is composed of individual threads. All threads within this grid execute the same kernel code. However, each thread possesses a unique identifier (its thread ID), enabling it to access different data elements and follow distinct execution paths within the kernel, allowing for diverse computations despite running the same core program.

The execution of these threads is managed by the SMs on the GPU. Within each SM, threads are organized and scheduled in groups of 32, known as warps. An SM issues instructions to an entire warp at a time, meaning all 32 threads in a warp execute the same instruction in lockstep during a given cycle.

This lockstep execution is highly efficient for uniform computations. However, if threads within

a warp encounter a conditional branch in the code and attempt to take different execution paths, a phenomenon termed branch divergence, the SM typically handles this by serializing the execution of the different paths. This serialization can severely impact performance, making it beneficial to write code that minimizes divergence within a warp.

To manage the high latency of operations like accessing global memory access, SMs are designed for latency hiding. If one warp stalls, for instance, while waiting for data to be fetched from global memory, the SM can rapidly context-switch to another warp that is ready to execute. This ability to hide latency by interleaving the execution of many active warps is a cornerstone of GPU performance because it ensures that the processing units remain highly utilized. Consequently, launching a sufficient number of warps, often organized into multiple thread blocks per SM, is crucial for achieving optimal throughput.

It is important to understand that the execution order of threads or even entire thread blocks is not guaranteed. Without explicit synchronization mechanisms employed by the programmer, there is no way to know when one block will execute relative to another. This inherent parallelism requires careful algorithm design to correctly manage any dependencies between threads in different blocks.

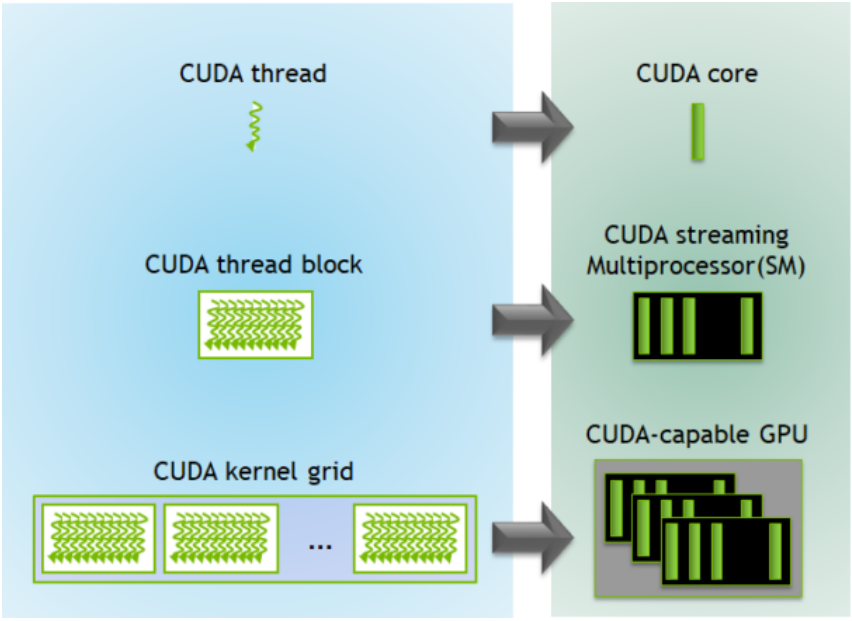


Figure 1.7: Correlation between CUDA programming paradigms and hardware conventions.

1.3.3 GPU Memory Hierarchy

Effective data management is crucial for achieving high performance on GPUs, due to the potential bottlenecks from memory access latency and bandwidth. GPUs feature a distinct and complex memory hierarchy designed to feed their massively parallel processing units. As outlined by authors such as Zhang et al. (2015), the detailed characteristics of this hierarchy can significantly impact performance and may not always be fully disclosed by vendors. A comprehensive understanding of this hierarchy is paramount for optimizing statistical algorithms for GPU execution. An illustrative example of a modern accelerator, the NVIDIA A100 GPU, is depicted in Figure 1.8.

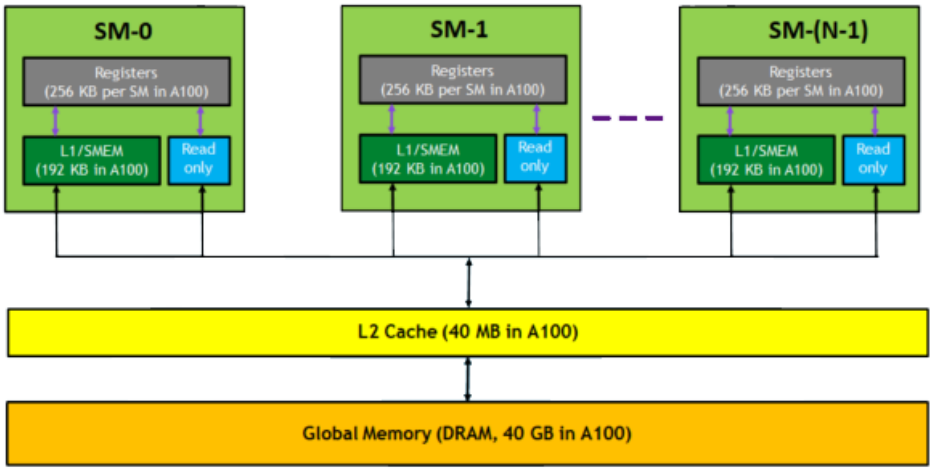


Figure 1.8: Simplified view of the memory hierarchy of an NVIDIA A100 40GB GPU.

The primary levels of this hierarchy are as follows:

1. **Global Memory (VRAM)**: The largest component of GPU memory, often several gigabytes (e.g., NVIDIA A100 features 40GB or 80GB of high-bandwidth DRAM). It is directly accessible by all processing threads and the host CPU. As an off-chip resource, VRAM exhibits the highest access latency. To utilize its high bandwidth efficiently requires what is called coalesced memory access, meaning simultaneous access to contiguous memory locations by threads in a warp. Uncoalesced access can severely degrade performance.
2. **L2 Cache**: A shared L2 cache resides on-die and helps reduce the average latency of global

memory accesses. It services requests from all SMs, and transparently caches frequently used data to improve overall performance.

3. **Per-SM On-Chip Memory**: Each SM contains several faster, lower-latency memory resources on-chip.

- **Registers**: The fastest memory type, private to each thread and located directly on the SM. They are crucial for holding frequently accessed variables, and compilers aim to maximize their use. Efficient intra-warp communication can also be achieved via register data exchange using shuffle instructions.
- **L1 Cache / Shared Memory**: A per-SM fast on-chip memory resource that typically serves as both a hardware L1 cache for global memory accesses and as a programmer-managed Shared Memory. Shared memory offers low-latency, high-bandwidth access for threads *within the same block*, making it invaluable for explicit data sharing, inter-thread communication, and a user-controlled cache to reduce global memory traffic. The balance between L1 cache and shared memory capacity is often configurable.
- **Read-Only Data Caches**: Dedicated read-only caches also exist within each SM for specific data access patterns. Constant Memory efficiently broadcasts identical values to multiple threads, while Texture Memory is optimized for read-only access with spatial locality, often providing hardware-accelerated interpolation.

4. **Local Memory**: Thread-private data that does not fit into registers (e.g., large arrays) is placed in Local Memory. Despite its name, local memory typically resides in the slow off-chip global memory, so its extensive use can significantly hinder performance and should be avoided where possible.

The design of this memory hierarchy is critically dependent on the principle of latency hiding. As previously discussed, the hardware's ability to manage and rapidly switch between thousands of concurrent threads is what allows it to tolerate the high latency of off-chip global memory. This ensures that the computational units remain highly utilized.

Finally, data transfers between the host's (CPU) system memory (RAM) and the device's (GPU) global memory (VRAM), conducted over the PCIe bus, represent another critical performance consideration. These transfers can become a significant bottleneck if not managed efficiently, for exam-

ple, by minimizing their frequency and maximizing the amount of data moved per transfer. A strategic exploitation of this entire memory hierarchy is indispensable for developing GPU-accelerated solutions.

1.3.4 GPU Programming Software Abstractions

The raw computational power of the GPU is made accessible to developers through layers of software abstractions. The most fundamental of these is **CUDA** (Compute Unified Device Architecture), NVIDIA's proprietary parallel computing platform, which allows developers to write **kernels**, which are functions executed on the GPU, in familiar languages like C++. CUDA offers fine-grained control over thread management and memory allocation. For broader hardware support, **OpenCL** provides an open, cross-platform standard for heterogeneous computing. An ecosystem of highly optimized libraries is built on top of these, such as NVIDIA's CUB and Thrust, which provide efficient primitives for common parallel operations like reduction, scanning, and sorting.

For most data scientists and statisticians, however, direct interaction with these low-level APIs may not always be necessary. The widespread adoption of GPUs in machine learning has led to the development of sophisticated high-level libraries, most notably **PyTorch** and **TensorFlow**. These frameworks provide intuitive, Python-based interfaces that abstract away the complexities of kernel writing and memory management. They allow practitioners to define complex models and computations using familiar tensor operations, while the framework's backend automatically translates these operations into optimized GPU kernels. This accessibility has been instrumental in democratizing high-performance computing, enabling researchers to leverage the full power of GPUs without needing to be experts in parallel programming.

1.4 Thesis Objectives and Structure

The central goal of this thesis is to demonstrate how GPU acceleration provides a practical and powerful solution to the computational bottlenecks inherent in modern statistical methods. We aim to bridge the conceptual gap between the theoretical principles of high-performance computing and their concrete application to statistical practice. In this journey we investigate the nuances and trade-

offs of implementing advanced, large-scale statistical methods on modern hardware, moving from theory to application. Our inquiry is built upon a methodological framework of three interconnected pillars, using examples such as Kernel Methods, Gradient Boosting, and Markov Chain Monte Carlo (MCMC) methods to exemplify the challenges and opportunities at the frontier of computational statistics.

The theoretical analysis is conducted on **Chapter 2, "Algorithmic Pathways to GPU-Accelerated Statistics"**. This chapter provides the analytical foundation for the thesis by dissecting key statistical algorithms into their core computational components. By analysing the complexity and memory access patterns of these components, we identify the specific operations that create performance bottlenecks and reveal inherent opportunities for parallelism. This provides a crucial rationale for *why* a given method is amenable to GPU acceleration.

The empirical validation of these concepts is then explored in **Chapter 3, "Benchmarking Statistical Solutions on GPU"**. This chapter presents applied research that benchmarks state-of-the-art, GPU-accelerated libraries against their CPU-based counterparts. Through rigorous experimentation, we quantify the tangible, real-world impact of these parallelization strategies. This not only validates the theoretical analysis but also furnishes concrete evidence of the performance gains available to practitioners.

Finally, a pedagogical overview of the engineering that underpins these solutions is presented in **Chapter 4, "An Introduction to GPU Programming"**. This chapter addresses the practical question of *how* these high-performance tools are constructed. By providing a brief, accessible introduction to GPU programming paradigms, this section serves to demystify the development process for a statistical audience.

Together, these chapters aim to provide a comprehensive analysis of GPU acceleration in modern computational statistics, guiding the reader from architectural foundations to algorithmic strategies and, finally, to practical application.

Chapter 2

Algorithmic Pathways to GPU-Accelerated Statistics

2.1 Introduction: Statistical Motivation

Modern statistical methods, while powerful, are often constrained by their own computational intensity. This creates a persistent challenge in applying sophisticated theoretical models to the large-scale data common in contemporary research and industry. This chapter explores this issue by dissecting the specific algorithmic pathways that enable two common, yet fundamentally different classes of methods to leverage the massive parallelism of GPUs.

The analysis centres on two distinct case studies: Kernel Methods, as implemented in the Falkon library, and Gradient Boosting, via the ubiquitous XGBoost framework. These methods were chosen deliberately to offer a rich comparative analysis. They highlight how different computational bottlenecks demand different optimization philosophies specifically because their primary challenges originate from different aspects of their design.

Kernel Methods are an exercise in dense linear algebra, with performance dictated by the efficiency of matrix operations and decompositions. They ultimately represent a problem of scaling, as their intrinsic polynomial complexity makes the exact algorithm fundamentally infeasible for large datasets, and thus the primary goal is to change the algorithm's complexity class itself. XGBoost, on the other

hand, presents an opportunity for acceleration, since its design makes data aggregation and search the primary bottleneck. Its exact algorithm is computationally feasible, but prohibitively slow for the iterative demands of model development and tuning, making raw speed the primary objective.

This chapter aims to deconstruct the evolution of these two methods from their theoretical foundations to their highly optimized, hardware-aware implementation. In doing so, it demonstrates that effective GPU utilization is not a simple matter of porting code from one hardware architecture to another. Rather, deliberate algorithmic redesign needs to take place in order to reshape statistical computation to align with the core architectural strengths of modern parallel hardware.

2.2 Kernel Methods at Scale with Falkon

Kernel methods are a class of powerful statistical and machine learning techniques that enable non-linear relationships to be modelled efficiently. They achieve this by mapping input data into a high-dimensional feature space, upon which simple and well-understood linear algorithms can be deployed. The ability to treat complex non-linear problems linearly is intuitively propitious; but it comes at high time and space complexities, $O(n^3)$ and $O(n^2)$ respectively.

Novel research on kernel methods (Rudi et al. 2018) comes to remedy this scalability profile through approximation techniques like Nyström, stochastic subsampling, iterative solvers, and preconditioning. Modern implementations of these ideas, like the one brought forth by Meanti et al. (2020) also leverage advancements in computing hardware to accelerate the solution of the underlying optimization problem.

This section begins with a brief introduction to kernel methods, discusses their computational pain points, and provides a glimpse into ways to scale them using GPU acceleration. In particular, we examine the ideas brought forth by the Falkon algorithm, which implements a version of approximate ridge regression that can handle potentially billions of points.

2.2.1 Foundations of Kernel Methods

To set the stage, we briefly review the fundamentals of kernel methods, including Cover's theorem, the kernel trick, common kernel functions, and their canonical applications in classification and

regression.

2.2.1.1 Cover's Theorem and the Kernel Trick

The performance of many statistical and machine learning algorithms relies on the linear separability of the data. In practice, however, real-world data is very often not linearly separable in its original form. This challenge was famously addressed by Cover (1965) in what became known as *Cover's theorem*, which states that a complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimensional space. Simply phrased, linear separability can be introduced by mapping data to a higher-dimensional space. Unfortunately, this transformation is often computationally infeasible in its explicit form, particularly for very high-dimensional spaces.

This is precisely the challenge that kernel methods are designed to overcome. Kernel methods are part of a broader class of methods called nonparametric learning. These methods do not assume a predefined structure or parametric form for a model. They rely on the *kernel trick*, which allows computations in a high-dimensional feature space without explicitly transforming the data into that space. Instead, a kernel function calculates the inner product between data points directly in this higher-dimensional space, saving computational effort.

This "trick" is enabled by a kernel function, K, which mathematically formalizes this shortcut. Whereas typically finding the inner product between the projections of two points $\mathbf{x}$ and $\mathbf{y}$ would first necessitate use of a transformation function ϕ to explicitly map them into a high-dimensional feature space, the kernel function allows direct calculation of the dot product between those projections without ever computing $\phi(\mathbf{x})$ and $\phi(\mathbf{y})$. This relationship becomes the foundation of kernel methods:

$$K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \tag{2.1}$$

The inner product operation within a linear algorithm can systematically be replaced with a chosen kernel function $K(\mathbf{x}, \mathbf{y})$, thus operating in the high-dimensional space implicitly and sidestepping the computational burden of the transformation. This powerful technique is invaluable in applica-

tions such as Support Vector Machines (SVMs), Gaussian Processes, Kernel Principal Component Analysis (Kernel PCA), and more.

2.2.1.2 Kernel Functions and the RBF Kernel

The intuition on why a kernel function is more efficient than an explicit feature transformation $\phi(\cdot)$ can be built by considering the simple case of a Polynomial kernel. Consider two-dimensional input data, so two vectors $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$. A Polynomial kernel of degree 2 can be defined as:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^2 \tag{2.2}$$

Calculating this is computationally cheap. It's a dot product (2 multiplications, 1 addition), followed by one more multiplication. Compare this to the explicit feature map $\phi(\mathbf{x})$ that this kernel corresponds to:

$$\phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2) \tag{2.3}$$

To use the feature map directly, both $\mathbf{x}$ and $\mathbf{y}$ would need to be transformed to this new 3D space, and then the dot product of these projections would have to be computed:

$$\phi(\mathbf{x}) \cdot \phi(\mathbf{y}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2) \cdot (y_1^2, \sqrt{2}y_1y_2, y_2^2) = x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2 \tag{2.4}$$

It can be shown with simple algebra that this result is identical to the right-hand side of (2.2). However, the kernel function $K(\mathbf{x}, \mathbf{y})$ was able to get to it in just three operations, whereas the explicit transformation required creating two new three-dimensional vectors and then performing a more complex dot product. The computational savings from using the kernel become immense for higher-dimensional data. This is the essence of the kernel trick's efficiency.

In practice, a variety of kernels are used depending on the structure of the data and the problem at

hand. One of the most widely used kernel functions is the *Radial Basis Function (RBF) kernel*, also known as the *Gaussian kernel*. It is defined as:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right) \tag{2.5}$$

This is often parameterized using $\gamma = \frac{1}{2\sigma^2}$, leading to the equivalent expression:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma\|\mathbf{x} - \mathbf{y}\|^2\right) \tag{2.6}$$

The RBF kernel's value is based on the Euclidean distance between points $\mathbf{x}$ and $\mathbf{y}$. It acts as a similarity measure, returning the value of 1 if the points are identical and decaying towards 0 as they move farther apart. The hyperparameter $\gamma > 0$ controls the "width" of the kernel, defining how much influence a single training sample has. A small γ results in a broader kernel, where distant points have more influence, while a large γ creates a narrow kernel, making the model more sensitive to the local vicinity of each point. A key theoretical advantage of the RBF kernel is that its corresponding feature space $\phi(\cdot)$ is infinite-dimensional, allowing it to model highly complex, non-linear patterns.

While the RBF kernel is a popular default choice, other common kernel functions include:

- **Linear Kernel**: $K(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$. This is the simplest kernel, representing the standard inner product. It is used when the data is expected to be linearly separable and serves as a fast, fundamental baseline. It does not add any non-linearity.
- **Polynomial Kernel**: $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + c)^d$. This kernel captures polynomial interactions between features up to degree d. The parameter $c \geq 0$ trades off the influence of lower-degree terms versus higher-degree terms.
- **Sigmoid Kernel**: $K(\mathbf{x}, \mathbf{y}) = tanh(\alpha\mathbf{x} \cdot \mathbf{y} + c)$. It is also known as the hyperbolic tangent kernel, and its form is inspired by the activation functions used in neural networks. It essentially mimics the activation function of a single-layer perceptron.

These kernel functions power a variety of tasks like classification, regression, and in general, pattern recognition. Choosing the right kernel and its hyperparameters is a critical aspect of model selection.

2.2.1.3 Kernel Methods in Classification and Regression

Regardless of the specific kernel used, any algorithm that relies on forming the full kernel matrix will face the same computational hurdles. This section showcases two popular applications of kernel methods as canonical examples demonstrating these challenges and motivating the need for scalable approximation techniques: *Support Vector Machines (SVM)*, which demonstrates how kernels enable non-linear decision boundaries in classification, and, importantly for Falkon, *Kernel Ridge Regression (KRR)*, which extends linear regression to complex manifolds.

2.2.1.3.1 Support Vector Machines Support Vector Machines (SVMs) leverage kernel methods to tackle complex classification tasks, transforming non-linearly separable data into manageable decision boundaries. SVMs work by identifying the optimal hyperplane that maximizes the margin between data points of different classes. When data is not linearly separable in its original space, the SVM algorithm employs the kernel trick, as shown in (2.1), to implicitly map data into a higher-dimensional feature space where linear separation becomes feasible. Common kernels, such as the RBF kernel defined in (2.6), enable this transformation by computing pairwise similarities without explicitly calculating the mapped coordinates.

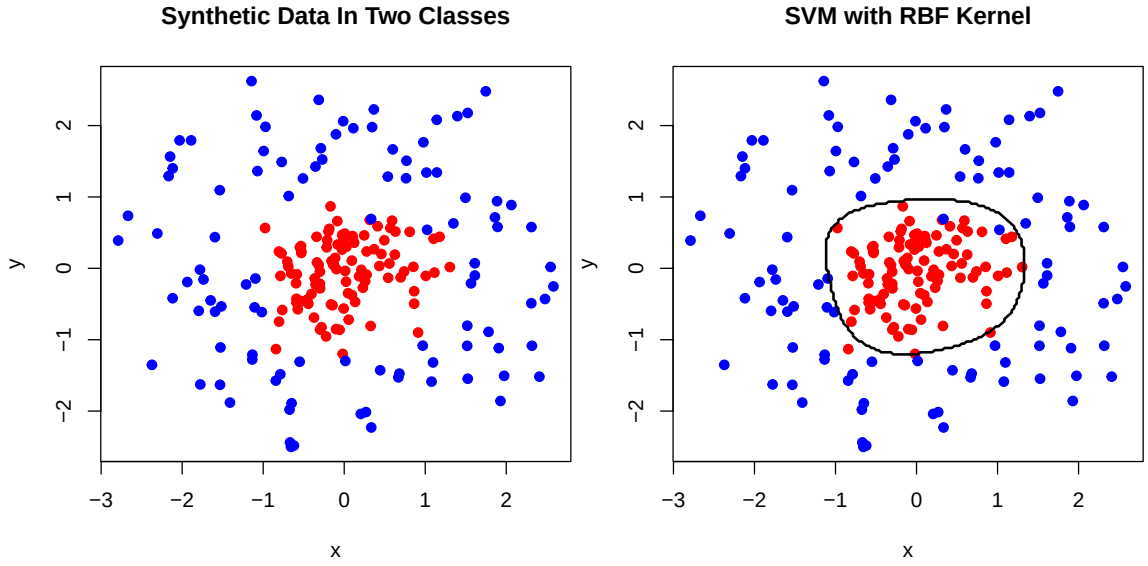


Figure 2.1: SVM classification of synthetic data using an RBF kernel.

Figure 2.1 demonstrates the effectiveness of SVMs with an RBF kernel. The left panel shows synthetic belonging to two classes: Class A, a cluster near the origin (red), and Class B, a ring

encircling it (blue), which are not linearly separable in their original 2D space. The right panel applies an RBF kernel, mapping the data into a higher-dimensional space where a linear hyperplane separates them. Projected back to 2D, this decision boundary appears as a circle, enabling the SVM to classify future points based on their position relative to this boundary.

SVMs can be defined by a primal problem that seeks to find a separating hyperplane with maximum margin. However, the primal problem scales poorly with data dimensionality and sample size. Instead, the problem is typically solved in its dual form, where the kernel trick can be directly applied. Considering kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$, this involves maximizing the following objective function with respect to a set of coefficients α, subject to constraints ($0 \leq \alpha_i \leq C$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$):

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{2.7}$$

A key insight of the dual is that the solution is sparse, meaning that only the coefficients α_i corresponding to the most critical data points (the *support vectors*) will be non-zero. The computational structure of this problem, and its inherent bottleneck, becomes undeniable when expressed in matrix form:

$$\max_{\alpha} \quad \mathbf{1}^\top \alpha - \frac{1}{2} \alpha^\top (\mathbf{YKY}) \alpha \tag{2.8}$$

Here, $\mathbf{Y}$ is a diagonal matrix with entries y_i representing the class labels ($y_i = \pm 1$) and, most critically, $\mathbf{K}$ is the $n \times n$ kernel matrix of all pairwise kernel evaluations. Equation (2.8) decisively shows that the optimization hinges on this kernel matrix. The necessity of computing, storing and operating on $\mathbf{K}$ results in at least $O(n^2)$ time and space complexity. This type of constrained optimization is known as a Quadratic Programming (QP) problem, and it is typically solved using highly specialized algorithms like Sequential Minimal Optimization (SMO) (Platt 1998). However, the reliance of the full kernel matrix makes any such standard solver prohibitive for very large datasets, motivating the search for accelerated and approximate methods.

2.2.1.3.2 Kernel Ridge Regression Kernel Ridge Regression (KRR) extends Ridge Regression to learn non-linear relationships by incorporating the kernel trick. While standard Ridge Regression finds a linear function that fits the data with an added penalty to control model complexity, KRR operates in high-dimensional feature space, allowing it to model complex patterns. Like SVM, KRR relies on a kernel matrix to compute similarities, but its goal is regression rather than classification.

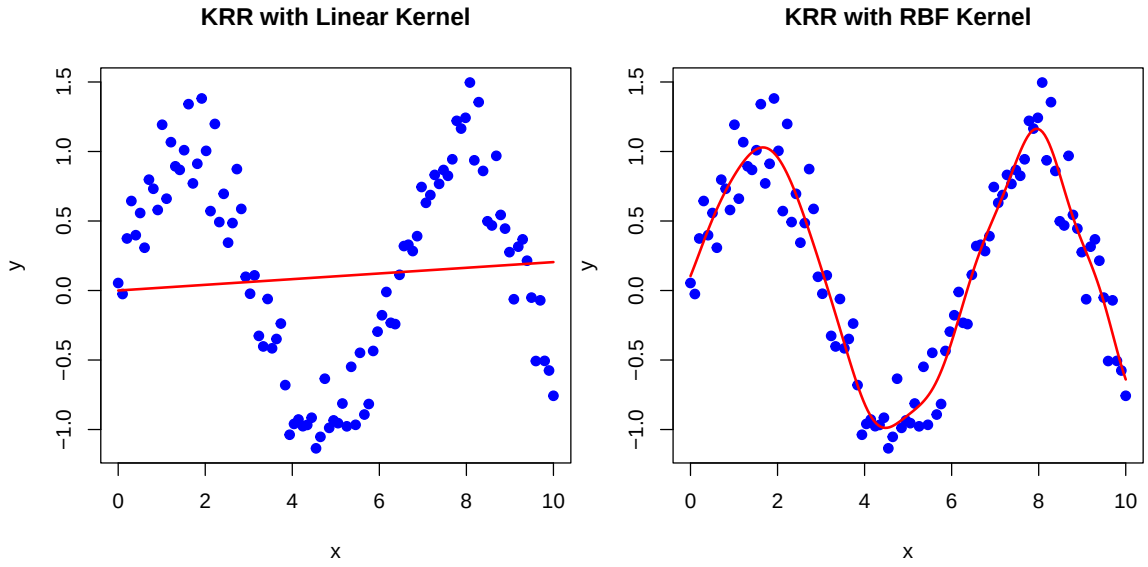


Figure 2.2: Kernel Ridge Regression on synthetic non-linear data following a sinusoidal pattern

Figure 2.2 demonstrates KRR's ability to model non-linear relationships using different kernels. On the left panel, KRR with a linear kernel is shown, equivalent to standard ridge regression. The right panel shows KRR with an RBF kernel, effectively fitting the non-linear data.

The process begins with the standard Ridge Regression objective, but assumes the linear model exists in the feature space mapped by $\phi(\cdot)$:

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \langle \phi(\mathbf{x}_i), \mathbf{w} \rangle)^2 + \lambda \|\mathbf{w}\|^2 \tag{2.9}$$

Here, we seek a weight vector $\mathbf{w}$ in the high-dimensional feature space. The Representer Theorem (Schölkopf et al. 2001) states that the solution $\mathbf{w}^*$ can be expressed as a linear combination of the mapped training points:

$$\mathbf{w}^* = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_i) \tag{2.10}$$

In this context, α is a vector of dual coefficients. Substituting this into (2.9) allows us to reframe the problem in terms of α:

- The inner product becomes $\langle \phi(\mathbf{x}_i), \mathbf{w} \rangle = \sum_{j=1}^{n} \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)$, which is nothing but the i-th entry of the vector $\mathbf{K}\alpha$, and
- The regularization term $\|\mathbf{w}\|^2$ becomes $\alpha^\top \mathbf{K} \alpha$.

This transforms the minimization problem into one that depends only on the kernel matrix $\mathbf{K}$, combining a least-squares loss with a regularization term defined in a Reproducing Kernel Hilbert Space (RKHS), as explained by Hastie et al. (2009, Chapter 5.8):

$$\min_{\alpha} \|\mathbf{y} - \mathbf{K}\alpha\|_2^2 + \lambda \alpha^\top \mathbf{K} \alpha \tag{2.11}$$

Unlike SVM, this objective function is a simple quadratic and can be minimized by taking the derivative with respect to α and setting it to zero. This yields a direct, closed-form solution for the coefficients:

$$\alpha = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{2.12}$$

Here, $\mathbf{y}$ is the vector of target values, $\mathbf{K}$ is the $n \times n$ kernel matrix of pairwise similarities between data points, and λ is a regularization parameter. Once the coefficients α are found, a prediction for a new point $\mathbf{x}_{new}$ is made via $f(\mathbf{x}_{new}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_{new})$.

This formulation decisively shows that, just like SVM, the solution to KRR hinges on the kernel matrix $\mathbf{K}$.

2.2.2 Scalability Challenges of Kernel Methods

The power of kernel methods like SVM and KRR to model complex, non-linear relationships comes at a steep price. Both methods, as demonstrated, are fundamentally bottlenecked by their reliance on the kernel matrix **K**, which encodes all pairwise interactions between data points. This dependency, evident in the dual formulations for SVM (2.8) and KRR (2.11), is the primary barrier to their application on large-scale datasets.

To understand this challenge formally, we can frame the learning problem for most kernel methods within the general theory of Reproducing Kernel Hilbert Spaces (RKHS). The goal is to find a function f within the RKHS (H) that minimizes a combination of empirical loss and a regularization term:

$$\min_{f \in H} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i) + \lambda \|f\|_H^2 \tag{2.13}$$

Here, $l(\cdot, \cdot)$ is a loss function (e.g., square loss for regression, hinge loss for classification) and $\|f\|_H^2$ is the squared norm in the RKHS, which penalizes model complexity. The Representer Theorem states that the optimal solution to this problem will always take the form of a finite expansion over the training data:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i) \tag{2.14}$$

This single theoretical result forms the basis for all of these methods. The general problem in (2.13) becomes a specific algorithm based on the choice of loss function, but the solution for the coefficients α invariably requires the construction of the $n \times n$ kernel matrix:

$$\mathbf{K}_{nn} = [K(\mathbf{x}_i, \mathbf{x}_j)]_{i,j=1}^{n} \tag{2.15}$$

This matrix is the source of the severe scalability issues. Its creation and storage demand $O(n^2)$ memory, while solving the associated system for α (via matrix inversion as in (2.12) for KRR, or QP for SVM) requires $O(n^3)$ time in the worst case. For a dataset of $n = 100,000$, storing the kernel

matrix in 64-bit precision consumes approximately 74.5 GB of RAM, exceeding the capacity of typical hardware. The subsequent $O(n^3)$ computations, which would take an infeasibly long time.

This computational wall effectively renders exact kernel methods obsolete for the "big data" problems common today. To reconcile the modelling power of kernel methods and the demands of modern datasets, it is necessary to turn to approximation techniques and hardware acceleration.

2.2.3 Low-Rank Approximation with Nyström

To overcome the prohibitive costs of forming the full kernel matrix $\mathbf{K}_{nn}$, several approximation techniques have been proposed. The most fundamental of these is the Nyström method, which constructs a low-rank approximation of $\mathbf{K}$ by leveraging a small subset of m landmark points from the original dataset, where $m \ll n$. These landmark points can be selected via uniform random sampling, clustering (e.g., k-means), or other more sophisticated sampling heuristics.

Instead of computing and storing the full $n \times n$ matrix, the Nyström method only requires two smaller matrices: $\mathbf{K}_{nm}$, the $n \times m$ kernel matrix between all n points and the m landmark points, and $\mathbf{K}_{mm}$, the small $m \times m$ matrix of similarities among the landmark points themselves. The full kernel is then approximated as:

$$\mathbf{K}_{nn} \approx \tilde{\mathbf{K}} = \mathbf{K}_{nm}\mathbf{K}_{mm}^{-1}\mathbf{K}_{nm}^{\top} \tag{2.16}$$

For symmetric kernels such as the RBF kernel, it can be shown that $\mathbf{K}_{mn} = \mathbf{K}_{nm}^{\top}$. This low-rank approximation reduces the memory footprint from $O(n^2)$ to $O(nm + m^2)$, which is approximately $O(nm)$ for $m \ll n$. Computationally, solving the system shifts from an $O(n^3)$ matrix inversion or decomposition to a more manageable $O(nm^2 + m^3)$ cost: $O(nm^2)$ for forming the approximation and $O(m^3)$ for inverting $\mathbf{K}_{mm}$.

By substituting this approximation $\tilde{\mathbf{K}}$ into the KRR problem from (2.12), there is no longer a requirement to solve a massive $n \times n$ system. Instead, we are faced with a new, much smaller and more manageable system to solve for the coefficients α:

$$\alpha = (\mathbf{K}_{nm}^\top \mathbf{K}_{nm} + \lambda \mathbf{K}_{mm})^{-1} \mathbf{K}_{nm}^\top \mathbf{y} \tag{2.17}$$

The Nyström method is not without its limitations. Its effectiveness hinges on the choice of the landmark points and the value of m. In practice, it is shown by (Rudi et al. 2015) that m can be as few as $m = O(\sqrt{n})$, but the choice is often subject to empirical validation and domain knowledge. Nonetheless, its ability to reduce kernel methods' complexity from intractable to practical makes it a cornerstone for modern scalability practices, especially when paired with the efficient iterative solvers discussed next. These advanced solvers bypass the direct matrix inversion shown in (2.17), offering a scalable and numerically stable path to a solution.

2.2.4 Conjugate Gradient and Preconditioning in Falkon

The Nyström method provides an efficient, low-rank approximation of the kernel matrix, $\tilde{\mathbf{K}}$, transforming the original KRR problem into a smaller, more manageable linear system. While one could solve this new system by direct matrix inversion, as in (2.17), that approach is often neither the fastest, nor the most numerically stable. State-of-the-art solvers like Falkon instead use a more sophisticated two-part strategy: an iterative solver combined with a powerful preconditioner.

2.2.4.1 Solving Iteratively with CG

Instead of inverting a large matrix directly, we can solve the linear system $\mathbf{A}\alpha = \mathbf{b}$ (where $\mathbf{A} = \tilde{\mathbf{K}} + \lambda\mathbf{I}$ and $\mathbf{b} = \mathbf{y}$) using an iterative method. The Conjugate Gradient (CG) algorithm is ideal for this, as the matrix $\mathbf{A}$ is symmetric and positive definite.

The core of the CG algorithm is a series of matrix-vector products. In each iteration, we must compute $\mathbf{Av} = (\tilde{\mathbf{K}} + \lambda\mathbf{I})\mathbf{v}$ for some vector $\mathbf{v}$. The key operation is the product with the Nyström approximation:

$$\tilde{\mathbf{K}}\mathbf{v} = (\mathbf{K}_{nm}\mathbf{K}_{mm}^{-1}\mathbf{K}_{nm}^\top)\mathbf{v} \tag{2.18}$$

This is computed efficiently from right to left without forming any large matrices:

1. Compute $\mathbf{v}_1 = \mathbf{K}_{nm}^\top \mathbf{v}$.
2. Solve $\mathbf{K}_{mm}\mathbf{v}_2 = \mathbf{v}_1$ for v_2.
3. Compute $\mathbf{v}_3 = \mathbf{K}_{nm}\mathbf{v}_2$.

The cost of each iteration is dominated by matrix-vector products involving $\mathbf{K}_{nm}$ and is approximately $O(nm)$. As per Meanti et al. (2020), for a well-conditioned system, only a few iterations are needed to reach a solution. Specifically, $O(\log n)$ CG steps are sufficient to achieve optimal statistical bounds.

When using $m = \mathcal{O}(\sqrt{n})$ inducing points, the total computational cost to solve the system and achieve these bounds is $\mathcal{O}(n\sqrt{n}\log n)$ in time and $\mathcal{O}(n)$ in memory.

2.2.4.2 Taming Ill-Conditioned Systems

Performance of CG is highly dependent on the properties of the system matrix being solved. A challenge with kernel methods specifically is that kernel matrices are often ill-conditioned. This means that their eigenvalues are spread across a wide dynamic range, which can cause the standard CG algorithm to converge extremely slowly, which negates its benefits.

To overcome this, Falkon employs preconditioning. The goal is to find an easily invertible matrix, the preconditioner, which transforms the original problem into an equivalent one that is better behaved. The linear system that arises from the Nyström approximated KRR problem, as discussed in (2.17), can be expressed as:

$$(\mathbf{K}_{nm}^\top \mathbf{K}_{nm} + \lambda n \mathbf{K}_{mm})\alpha = \mathbf{K}_{nm}^\top \mathbf{y} \tag{2.19}$$

Letting $\mathbf{H} = \mathbf{K}_{nm}^\top \mathbf{K}_{nm} + \lambda n \mathbf{K}_{mm}$, our goal is to solve $\mathbf{H}\alpha = \mathbf{K}_{nm}^\top \mathbf{y}$. The preconditioner selected to address the ill-conditioning of $\mathbf{H}$, must be both effective at improving conditioning and cheap to construct and apply. A naïve preconditioner $\mathbf{P}$ for the Nyström-approximated KRR system might be one that satisfies $\mathbf{P}\mathbf{P}^\top = (\mathbf{K}_{nm}^\top \mathbf{K}_{nm} + \lambda n \mathbf{K}_{mm})^{-1}$, but computing this is as costly as the original problem, as noted by Meanti et al. (2020).

This is where the core insight of Falkon's preconditioner comes into play. To create a practical

preconditioner, the Nyström approximation is applied a second time, this time to approximate the expensive $\mathbf{K}_{nm}^{\top}\mathbf{K}_{nm}$ term within the matrix $\mathbf{H}$ as $\approx \frac{n}{m}\mathbf{K}_{mm}^{2}$. Following Rudi et al. (2018), the implemented preconditioner in Falkon, denoted $\tilde{\mathbf{P}}$, is constructed from the Cholesky factors of $\mathbf{K}_{mm}$ and a related regularized matrix derived from it. Specifically, the preconditioner is defined as:

$$\tilde{\mathbf{P}} = \frac{1}{\sqrt{n}}\mathbf{T}^{-1}\mathbf{A}^{-1} \tag{2.20}$$

Factors $\mathbf{T}$ and $\mathbf{A}$ are obtained via Cholesky decomposition and given, respectively, by:

$$\mathbf{T} = \text{chol}(\mathbf{K}_{mm}) \tag{2.21}$$

$$\mathbf{A} = \text{chol}\left(\frac{1}{m}\mathbf{T}\mathbf{T}^{\top} + \lambda\mathbf{I}_{m}\right) \tag{2.22}$$

The efficient computation and storage of these Cholesky factors, particularly in a memory-constrained GPU environment, are key aspects of Falkon's design, as will be elaborated upon in the discussion of GPU memory optimization. The crucial insight is that operations involving this $\tilde{\mathbf{P}}$ are computationally efficient due to the triangular structure of these Cholesky factors.

By applying this preconditioner, the problem is transformed via a change of variables from the original α to a new vector β, where $\beta = \tilde{\mathbf{P}}^{-1}\alpha$. The CG algorithm then solves for β in a well-conditioned system of the form:

$$\tilde{\mathbf{P}}^{\top}\mathbf{H}\tilde{\mathbf{P}}\beta = \tilde{\mathbf{P}}^{\top}\mathbf{K}_{nm}^{\top}\mathbf{y} \tag{2.23}$$

Once the optimal β is found, the final solution is recovered by transforming back, $\alpha = \tilde{\mathbf{P}}\beta$. This specific preconditioning strategy is essential for Falkon's performance, as it transforms the ill-conditioned Nyström-approximated system into one that CG can solve rapidly, ensuring fast convergence to an accurate solution.

2.2.5 GPU-Optimized Kernel Solvers: The Falkon Implementation

Approximation techniques like the Nyström method and iterative solvers such as conjugate gradient reduce memory complexity of Kernel methods from $O(n^2)$ to approximately $O(nm)$ and time complexity from $O(n^3)$ to $O(nm \log n)$. But even with $m \ll n$, their practical scalability on traditional CPU architectures remains constrained by sequential processing and limited memory bandwidth. GPUs, on the other hand, excel at massively parallel computations and provide significantly higher computational throughput and memory bandwidth, mitigating CPU limitations.

To leverage the power of GPUs, careful implementation tailored to their architecture needs to be made. GPUs thrive on high operational density (i.e., many computations per byte of memory) and low-latency data access. Kernel methods, by design, demand substantial memory due to the kernel matrix, yet they exhibit a low density of operations per byte of memory used. This mismatch poses a challenge when adapting them to GPU architectures, and necessitates strategies that maximize parallelism, minimize memory usage, and reduce data transfers between CPU (host) and GPU (device) memory. Naïve implementations of Nyström or CG solvers, while theoretically efficient, may falter on GPUs if they fail to address these constraints, such as limited onboard memory or the overhead of host-device communication.

Recent advancements have addressed these challenges with efficient, GPU-optimized implementations of kernel methods. Notable examples include GPyTorch and GPflow, which come from the Gaussian process literature, ThunderSVM for Support Vector Machines, and EigenPro for GPU-friendly Kernel Ridge Regression updates. Among these, the Falkon algorithm stands out for its ability to process datasets with billions of points efficiently, achieving dramatic speedups without sacrificing model performance. Falkon builds on the Nyström approximation and preconditioned CG solvers, reformulating them to fully exploit GPU capabilities, and serves as a cornerstone for this section's exploration of GPU-optimized strategies. In particular, this section dives into the GPU-specific tricks employed by Falkon, which enable it to scale KRR to massive datasets.

2.2.5.1 Minimizing Memory Footprint

A primary contribution of the Falkon library is its set of strategies for overcoming the memory bottlenecks that have long constrained kernel methods. While the Nyström approximation reduces

the problem size from n to m, a naïve setup would require storage of the full kernel matrix $\mathbf{K}_{mm}$, the matrix system $\mathbf{H}$, and the preconditioned form $\mathbf{P}^\top\mathbf{HP}$ separately, all of which are $m \times m$ matrices, in addition to the potentially massive $n \times m$ matrix K_{mn}. Falkon's memory allocation is instead composed of just one $m \times m$ matrix, a $1 \times m$ vector for β, and two small buffers.

The Falkon implementation systematically reduces the memory footprint through two main strategies: (i) computing the preconditioner factors in-place using a single memory allocation, and (ii) sidestepping explicit construction of $\mathbf{K}_{nm}$ by processing it in blocks.

2.2.5.1.1 In-Place Preconditioner Computation The most significant memory allocation in Falkon is the $m \times m$ matrix required for the preconditioner. The algorithm cleverly reuses only a single pre-allocated $m \times m$ matrix for the entire multi-step calculation of the factors $\mathbf{T}$ and $\mathbf{A}$. The process proceeds as follows:

1. **Compute $\mathbf{K}_{mm}$**: The $m \times m$ kernel matrix of inducing points is computed (typically in blocks on the GPU) and stored directly into a single pre-allocated $m \times m$ matrix.
2. **First Cholesky Factor $\mathbf{T}$**: Since kernel matrices are symmetric and positive definite, they are perfectly suited for Cholesky decomposition. Falkon performs this decomposition in-place to find the factor $\mathbf{T}$ such that $\mathbf{K}_{mm} = \mathbf{T}^\top\mathbf{T}$ (assuming a lower triangular $\mathbf{T}$). The resulting factor $\mathbf{T}$ overwrites the corresponding triangle of $\mathbf{K}_{mm}$ in memory.
3. **Second Cholesky Factor $\mathbf{A}$**: The same $m \times m$ memory allocation is then reused. The intermediate matrix $\left(\frac{1}{m}\mathbf{TT}^\top + \lambda\mathbf{I}_m\right)$ is computed and stored, leveraging the previously unused opposite triangle of the matrix for storage. An in-place Cholesky decomposition is then performed for this result to find the factor $\mathbf{A}$.

These in-place operations might seem counterintuitive at first, given that reconstructing K_{mm} would then require computing $\mathbf{TT}^\top$. However, the benefit becomes clear once the subsequent use of $\mathbf{T}$ is considered. Specifically, the approximation of the system matrix H in (2.23) involves precisely this structure.

$$\mathbf{H} \approx \frac{n}{m}\mathbf{K}_{mm}^2 + \lambda n \mathbf{K}_{mm} = \frac{n}{m}\mathbf{T}\mathbf{T}^\top\mathbf{T}\mathbf{T}^\top + \lambda n \mathbf{T}\mathbf{T}^\top = n\mathbf{T}(\frac{1}{m}\mathbf{T}^\top\mathbf{T} + \lambda \mathbf{I}_m)\mathbf{T}^\top = n\mathbf{T}\mathbf{A}\mathbf{A}^\top\mathbf{T}^\top \tag{2.24}$$

Here, the inner matrix emerges naturally, capturing both the kernel similarity structure and regularization. This symmetric positive definite matrix itself admits a Cholesky decomposition as in (2.22). Thus, Falkon cleverly uses a single matrix's lower and upper triangles to hold both factor **T** and **A**, effectively reducing memory requirements by nearly half.

2.2.5.1.2 Block-wise (Out-of-Core) Matrix-Vector Products

The second, and arguably more critical, memory optimization is avoiding the construction of the $n \times m$ kernel matrix, $\mathbf{K}_{nm}$. For large n, this matrix would be far too large to store in RAM.

Falkon sidesteps this issue entirely because $\mathbf{K}_{nm}$ is only ever used within matrix-vector products during CG iterations, as in (2.18). The implementation treats this as an "out-of-core" operation by splitting the input data **X** into smaller batches. The matrix-vector product is then accumulated as follows:

- For each batch of data, transfer it to the GPU.
- Compute the corresponding kernel sub-matrix on-the-fly on the GPU.
- Perform the matrix-vector multiplication on the GPU.
- Accumulate the result and discard the kernel sub-matrix.

Crucially, the computed kernel sub-matrices are never stored in the main memory or transferred back from the GPU, drastically reducing memory usage and host-device communication. This block-wise approach is what allows Falkon to handle a virtually unlimited number of samples, n.

2.2.5.2 Out-of-Core (OOC) Operations

The strategies for minimizing RAM usage are crucial, but modern GPU architectures introduce a second, more restrictive memory ceiling: the GPU's own memory (VRAM), which is typically much smaller than the system's main RAM. For instance, a system might have 256GB of RAM, but only 16GB of VRAM per GPU. When the number of inducing points, m, is large (e.g., $m =$

2×10^5), the preconditioner matrix alone can occupy over 150GB, making it impossible to store on a single GPU.

To solve this, Falkon implements out-of-core (OOC) operations. OOC algorithms are designed to work with a large storage layer (main RAM) and a smaller but much faster layer (GPU VRAM), processing data in chunks that fit into the fast memory. Falkon provides its own OOC implementations for key linear algebra routines, as they are not standard in common higher-level machine learning libraries like PyTorch. This OOC design is also the foundation for distributing work across multiple GPUs in parallel.

2.2.5.2.1 Optimized OOC $\mathbf{K}_{nm}$-Vector Multiplication As established in the previous section, the $\mathbf{K}_{nm}$-vector product is handled in a block-wise manner. The OOC implementation further optimizes this by carefully choosing the block size to maximize the ratio of computation to data transfer time for a given amount of GPU memory. This ensures the GPU spends as much time as possible performing calculations rather than waiting for data from the slower main RAM, maximizing accelerator utilization.

2.2.5.2.2 OOC Multi-GPU Cholesky Decomposition A more complex challenge is computing the Cholesky decomposition of the $m \times m$ preconditioner matrix when it is too large for a single GPU's VRAM. Falkon implements a parallel OOC Cholesky algorithm inspired by high-performance computing literature. The high-level process is as follows:

1. **Tiling**: The full matrix in RAM is conceptually split into smaller square tiles that can fit into GPU memory.
2. **Column-wise Processing**: The algorithm proceeds by finalizing one full column of the final Cholesky factor at a time.
3. **Parallel Updates**: For each column, the work is distributed across all available GPUs. The update process involves a sequence of operations on the tiles: a standard Cholesky decomposition on the diagonal tile, triangular system solves for the off-diagonal tiles in that column, and finally, using the results to update the rest of the matrix (the "trailing submatrix"). 4 **Inter-GPU Communication**: This parallel process requires data transfers between GPUs.

By implementing these sophisticated OOC routines, Falkon effectively removes GPU VRAM as a hard limit on the problem size, allowing it to scale to a very large number of inducing centres, m.

2.2.5.3 Leveraging Reduced-Precision Arithmetic

Beyond managing memory, Falkon achieves significant speedups by carefully managing the numerical precision of its calculations. Modern GPUs are designed to achieve peak performance with lower-precision floating-point numbers. For instance, switching from 64-bit (double precision) to 32-bit (single precision) arithmetic can result in a throughput improvement of up to 10x, depending on the specific GPU architecture.

However, naively switching to 32-bit precision can lead to severe numerical stability issues. Meanti et al. (2020) highlights a classic example when computing the Gaussian / RBF kernel as in (2.6). That depends on the squared norm $\|x - y\|^2$. This is often computed by expanding the norm as $\|x\|^2 - 2x^\top y + \|y\|^2$, but in high dimensions, the squared norm terms can become very large, while the cross-term can be a large negative number. Summing these values in low precision can lead to "catastrophic cancellation", where significant digits are lost. This loss of precision can result in a kernel matrix that is no longer perfectly symmetric positive definite, which would cause the Cholesky decomposition to fail.

To gain the speed of low-precision computation without sacrificing numerical stability, Falkon employs a mixed-precision strategy. For operations like the computation of $\mathbf{K}_{mm}$, the matrix blocks are calculated using 32-bit precision, but the crucial intermediate sums are performed in 64-bit precision to maintain accuracy. The final result is then converted back to 32-bit for storage.

This careful, targeted use of high precision only where it is numerically critical allows Falkon to benefit from the immense speed of 32-bit GPU arithmetic for the vast majority of its computations, while ensuring the robustness and success of the overall algorithm.

2.3 Accelerating XGBoost with GPU Histograms

The landscape of supervised learning has been shaped profoundly by the advent of powerful ensemble techniques, with the gradient boosting framework emerging as a particularly robust and effec-

tive methodology. At the forefront of this evolution is XGBoost, a system whose name has become synonymous with state-of-the-art performance. Introduced by Chen & Guestrin (2016), XGBoost represents a highly optimized and scalable implementation of gradient boosting that is the consensus choice for winning solutions in a wide array of classification and regression challenges, from classifying high-energy physics events to predicting store sales and customer behaviour. Its success stems from a principled approach to model building that incorporates sophisticated regularization to prevent overfitting and leverages novel algorithms for computational efficiency.

However, XGBoost's predictive power presents a fundamental paradox: the very process that makes the model so effective is also its greatest computational liability. The sequential construction of decision trees at the core of the algorithm is fundamentally bottlenecked by an expensive, exhaustive search for the optimal splits at each node. For large-scale datasets, this search becomes a computational wall, turning training from a matter of minutes into hours or even days. This high cost is further compounded by the practical need for extensive hyperparameter tuning, where models must be retrained repeatedly, making raw computational speed a critical limiting factor for practitioners.

This chapter argues that surmounting this paradox required a crucial, two-stage evolution that effectively links statistical theory and high-performance hardware. We will first dissect the theoretical formulation of XGBoost to reveal precisely how its regularized, second-order objective leads to the demanding split-finding computation. We will then analyse the first of its key innovations: a fundamental algorithmic leap on CPU architectures, which replaced the exhaustive sorting-based search with a highly efficient histogram aggregation method. Then, we will demonstrate how this algorithmic shift inadvertently created a new computational profile that is perfectly suited for the massively parallel, high-throughput architecture of the GPU, and how a subsequent architectural co-design by Mitchell & Frank (2017) leveraged this synergy to achieve dramatic acceleration, transforming XGBoost into a truly scalable tool for modern computational statistics.

2.3.1 From Decision Trees to Gradient Boosting

To establish the context for its design, we will first trace the lineage of XGBoost from its foundational concepts, decision trees and the gradient boosting framework.

2.3.1.1 Decision Tree as a Base Learner

Decision trees are a versatile class of non-parametric supervised learning models that recursively partition the feature space into a set of disjoint, axis-aligned regions (Hastie et al. 2009, Chapter 9). Their hierarchical structure, which resembles an inverted tree, facilitates interpretable decision-making for both classification and regression tasks. Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^p$ is a p-dimensional feature vector and y_i is the target variable, a decision tree is constructed by iteratively selecting the "best" split for the instances at the current node.

Figure 2.3: A decision tree derived from the 'Play Golf' dataset. The model recursively splits the data based on predictor features like 'Outlook' to classify the outcome.

Each internal node in the tree represents a test on a single feature x_j (with $j \in \{1, \dots, p\}$) against a threshold $\tau \in \mathbb{R}$. An observation $\mathbf{x}_i$ traverses the tree from the root node, and at each internal node, if $x_{ij} \leq \tau$, it proceeds to the left child node R_{t_L}; otherwise, it proceeds to the right child node R_{t_R}. This process continues until a terminal node, called a *leaf*, is reached. All observations falling into the same leaf region R_m are assigned a common prediction value. For regression, that is typically the mean of the y_i values in R_m, while for classification, it is the majority class.

The key question in tree construction is how to determine the "best" split. Algorithms most typically select the split that maximizes the reduction in an impurity measure, $H(R_t)$, after partitioning a node t. Common impurity measures include:

- **Gini Index (Classification)**: Measures the probability of misclassifying a randomly chosen element if it were randomly labelled according to the distribution of labels in the subset. For a region R_t with K classes and $\hat{p}_{t,k}$ representing the proportion of class k observations:

$$H(R_t) = \sum_{k=1}^{K} \hat{p}_{t,k}(1 - \hat{p}_{t,k}) \tag{2.25}$$

- **Entropy (Classification)**: Quantifies the uncertainty in the class labels within the region. This measure uses the concept of Shannon's entropy and is the fundamental component of deviance, a statistical metric of model fit derived from the log-likelihood. Using the same conventions as the Gini index:

$$H(R_t) = -\sum_{k=1}^{K} \hat{p}_{t,k} \log_2 \hat{p}_{t,k} \tag{2.26}$$

- **Mean Squared Error (MSE) (Regression)**: For regression tasks, the impurity of a node is measured by the variance of the target variable. Minimizing node variance is equivalent to minimizing MSE relative to the node's mean. For a region R_t with mean target value $\bar{y}_t = \frac{1}{|R_t|} \sum_{\mathbf{x}_i \in R_t} y_i$:

$$H(R_t) = \frac{1}{|R_t|} \sum_{\mathbf{x}_i \in R_t} (y_i - \bar{y}_t)^2 \tag{2.27}$$

Considering $|R_t|$ as the number of observations in R_t, the impurity reduction, or **Gain**, from splitting region R_t into left child R_{t_L} and right child R_{t_R} is defined as:

$$\Delta H = H(R_t) - \frac{|R_{t_L}|}{|R_t|} H(R_{t_L}) - \frac{|R_{t_R}|}{|R_t|} H(R_{t_R}) \tag{2.28}$$

Algorithms in classical implementations, such as CART (Classification and Regression Trees) by Breiman et al. (1984) employ a greedy approach, searching over all possible features and split points (i.e., all (j, τ)) to maximize this Gain. The recursive partitioning continues until a stopping criterion is met, such as a minimum number of observations per leaf ($|R_t| < n_{\text{min}}$), a maximum tree depth ($d_t \geq d_{\text{max}}$), or negligible impurity reduction ($\Delta H < \epsilon$). For each node, finding the optimal split on a continuous feature requires first sorting the data, an operation with a complexity of $O(n_N \log n_N)$, where n_N is the number of instances in the node. This process, repeated for all p features, establishes the high baseline cost of tree construction, making this greedy search the

source of the method's primary computational bottleneck.

While interpretable, individual decision trees are prone to high variance, meaning that small changes in the training data can lead to significantly different tree structures and predictions. They can also easily overfit by creating overly complex trees that capture noise in the data. Although pruning techniques can mitigate overfitting, these foundational limitations of stability and accuracy have motivated the development of ensemble methods.

2.3.1.2 Ensemble Methods and Tree Boosting

The foundational limitations of single decision trees, namely their high variance and tendency to overfit, motivated the development of ensemble learning. These techniques combine multiple "weak" learners, such as a single decision tree, to create a "strong" learner with improved predictive performance and robustness (Hastie et al. 2009, Chapter 16). The core idea is to leverage the diversity among individual models to reduce variance (as in *bagging*), bias (as in *boosting*), or both.

Bagging (or *Bootstrap Aggregating*) involves training multiple independent models, typically of the same type (e.g., decision trees), on different bootstrap samples (generally, random samples with replacement) of the training data. Predictions from these models are then aggregated, usually by averaging for regression or majority voting for classification. Random Forests, introduced by Breiman (2001), extend bagging by also randomly selecting a subset of features at each split in the tree construction process, further decorrelating the trees and reducing the ensemble's variance.

Boosting, in contrast, builds models sequentially. Each new model attempts to correct the errors made by the ensemble of previously trained models. Early boosting algorithms like AdaBoost by Freund & Schapire (1997) iteratively re-weighted misclassified training instances, forcing subsequent learners to focus on these "harder" examples. *Tree Boosting* specifically applies this sequential error-correction principle using decision trees as the base learners.

Given M decision trees in total, where $h_m(\mathbf{x})$ is the prediction of the m-th tree, and β_m is its weight, the final prediction is a weighted sum of the predictions from all trees in the ensemble:

$$F(\mathbf{x}) = \sum_{m=1}^{M} \beta_m h_m(\mathbf{x}) \tag{2.29}$$

While highly effective at reducing bias, this sequential methodology introduces a significant computational burden not present in parallel methods. The total training time is not merely the cost of building M trees, but the cost of building them in a strict, dependent sequence where the construction of tree m cannot begin until tree $m-1$ is complete. This sequential dependency, combined with the expensive greedy search required to build each individual tree, creates a formidable computational challenge that necessitates the algorithmic optimizations at the heart of modern boosting systems.

2.3.1.3 Gradient Boosting as Functional Gradient Descent

Gradient Boosting [*Machines*] (*GBM*), introduced by Friedman (2001), generalizes boosting to arbitrary differentiable loss functions. It reframes the boosting process as a numerical optimization in function space, where the goal is to find a function $F(\mathbf{x})$ that minimizes the expected value of a chosen loss function $L(y, \widehat{y})$. This is achieved by iteratively adding new weak learners (trees) that move the ensemble prediction in the direction of the negative gradient of the loss function, analogous to gradient descent in parameter space.

Formally, given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^p$ is a p-dimensional feature vector and y_i is the target variable, as well as a differentiable loss function $L(y, \widehat{y})$, the algorithm begins by setting a constant model $F_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^n L(y_i, \gamma)$. For example, the initial prediction may be the mean of y_i for MSE loss. Then, for each iteration $m = 1, \ldots, M$ the algorithm performs the following steps:

1. Compute pseudo-residuals: Calculate the negative gradient of the loss with respect to the current model's predictions for each observation $i = 1, \ldots, n$:

$$r_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x_i}))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} \tag{2.30}$$

These pseudo-residuals represent the direction in which the prediction for observation i should be adjusted to best reduce the loss. They are termed *pseudo* because for arbitrary loss functions they are not the true residuals $y_i - F_{m-1}(\mathbf{x}_i)$. Instead, they act as the negative gradient components, indicating the direction of steepest descent for the loss function with respect to the current prediction

$F_{m-1}(\mathbf{x}_i)$. For MSE loss, $L(y, F) = \frac{1}{2}(y - F)^2$, the pseudo-residual is, indeed, simply $y_i - F_{m-1}(\mathbf{x}_i)$, the ordinary residual.

2. Fit a decision tree: Train a new decision tree, $h_m(\mathbf{x})$, by fitting it to these pseudo-residuals $\{(\mathbf{x}_i, r_{im})\}_{i=1}^n$ using the tree construction process outlined in the previous section. The tree partitions the feature space to minimize the impurity of the residuals.

3. Update the model: Add the new tree to the ensemble, scaled by a learning rate $\eta \in (0, 1]$:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta h_m(\mathbf{x}). \tag{2.31}$$

The learning rate η replaces the more general weight β_m seen in (2.29). It introduces *shrinkage*, meaning that it reduces the contribution of each tree and helps to prevent overfitting. This often necessitates more trees (M) to reach convergence, but typically results in a better-generalized model. Notably, this update step is directly parallel to the parameter update in traditional gradient descent as seen in Table 2.1.

Table 2.1: Analogy between Gradient Descent in parameter space and Gradient Boosting in function space.

Concept	Gradient Descent (Parameter Space)	Gradient Boosting (Function Space)
Goal	Minimize loss $L(\theta)$	Minimize loss $L(y, F(\mathbf{x}))$
Object	Parameters, θ	Function, $F(\mathbf{x})$
Direction	Negative gradient, $-\nabla_\theta L$	Pseudo-residuals, r_{im}
Step	Scaled gradient vector	Weak learner, $h_m(\mathbf{x})$
Update	$\theta_m = \theta_{m-1} - \eta \nabla L$	$F_m = F_{m-1} + \eta h_m(\mathbf{x})$

The iterative process of fitting trees to pseudo-residuals allows gradient boosting to effectively minimize a wide variety of loss functions, making it applicable to regression, classification, and ranking problems. However, this statistical elegance comes with compound computational burden. The framework inherits the expensive, sort-based greedy search required to evaluate potential splits within the inner loop of each tree's construction, layered upon which is the strict sequential fitting of hundreds or thousands of trees in the outer loop of the boosting process. Additionally, the method's sensitivity to hyperparameters often necessitates that this entire, demanding procedure be repeated many times during model tuning. These combined limitations underscore the need for

an optimized implementation that improves scalability and regularization, setting the stage for the specific algorithmic and systematic innovations introduced by XGBoost.

2.3.2 XGBoost Formulation and Computational Challenges

XGBoost (eXtreme Gradient Boosting), introduced by Chen & Guestrin (2016), is a highly optimized implementation of gradient boosting. It extends the gradient boosting framework through a number of innovations and manages to capture complex, non-linear patterns in data, making it robust for tasks where traditional linear models may fail. The performance gap between a well-tuned XGBoost model and more complex ensembles, often involving sophisticated models like deep neural networks, is observed to be small. It is popular in a wide range of real-world applications and appears among the top contenders across academic competitions and industrial benchmarks.

The power of XGBoost stems from a principled approach that combines sophisticated regularization techniques, novel algorithms for tree construction, and an end-to-end system-level design to achieve remarkable efficiency and scalability. The core components of this architecture are:

1. **A Regularized, Second-Order Learning Objective**: XGBoost optimizes a unique objective function that directly penalizes model complexity using L1 and L2 terms. Unlike traditional methods that use only the gradient, it leverages a second-order Taylor expansion to incorporate both gradient and Hessian information, often leading to more accurate and rapid convergence.
2. **Fast, Sophisticated Split-Finding Algorithms**: To solve the primary computational bottleneck of finding splits, XGBoost introduces two techniques: a *Weighted Quantile Sketch*, which replaces the costly, sort-based exact search with a faster, approximate, percentile-based approach, and a *Histogram Algorithm*, which bins feature values to scan aggregated statistics. The latter is XGBoost's key innovation for scalability.
3. **Sparsity-Awareness for Real World Data**: The framework handles missing values intelligently with a sparsity-aware split-finding algorithm. Instead of requiring imputation the model learns an optimal path for missing entries during training, improving both efficiency and model performance on sparse, real-world datasets.
4. **System Optimizations for Scalability**: To handle massive datasets, XGBoost is engineered

with specific system optimizations. These include *Column Blocks* (i.e., column-major, cache-friendly blocks) for parallel data processing, *Cache-Aware Access* patterns to maximize hardware throughput, and *Out-of-Core Computation* for datasets that cannot fit into memory.

These components work in concert to create a high-performance system. The first and most foundational of these innovations is the objective function itself, which explicitly defines the statistical problem that the entire system is built to solve.

2.3.2.1 Regularized, Second-Order Learning Objective

A key innovation in XGBoost is that it explicitly balances model fit with model complexity to prevent overfitting and enhance generalization. While traditional gradient boosting methods rely primarily on shrinkage (via learning rate η) and early stopping, XGBoost integrates complexity control directly into the objective function minimized at each step of the tree building process.

Recall that tree ensemble methods build an additive model as in (2.29), where $h_m(\mathbf{x})$ represents the m-th tree. XGBoost seeks to determine the function h_m at each iteration by optimizing an objective function that considers both the training loss and penalty for the complexity of the new tree. The objective function at iteration m is given by:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i) + h_m(\mathbf{x}_i)) + \Omega(h_m) \tag{2.32}$$

Here, L is a differentiable convex loss function, which may be squared error for regression or log-loss for classification, and $\Omega(h_m)$ is a regularization term that penalizes the complexity of the newly added tree h_m. This term is most typically defined as:

$$\Omega(h_m) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{2.33}$$

In this expression, T is the number of leaves in the tree h_m and w_j is the score assigned to the j-th leaf, representing the prediction for instances in that leaf. Hyperparameter λ is an L2 regularization parameter that shrinks leaf weights towards zero, akin to ridge regression. The parameter γ penalizes the number of leaves, which encourages trees with fewer leaves and acts as a form of

pre-pruning. XGBoost may also optionally add a hyperparameter α that adds an L1 penalty term, $\alpha \sum_{j=1}^{T} |w_j|$, akin to lasso regression. This is less commonly emphasized, as the L1 parameter is often set to zero in practice (Mitchell & Frank 2017).

To optimize this objective, XGBoost employs a second-order Taylor expansion of the loss function around the current prediction $F_{m-1}(\mathbf{x}_i)$. Let $g_i = \partial_{F_{m-1}} L(y_i, F_{m-1}(\mathbf{x}_i))$ be the first-order gradient (i.e., the pseudo-residual in (2.30)) and $h_i = \partial^2_{F_{m-1}} L(y_i, F_{m-1}(\mathbf{x}_i))$ be the second-order gradient (the Hessian). The objective function at iteration m, ignoring constants, can then be approximated as:

$$\mathcal{L}^{(m)} \approx \sum_{i=1}^{n} [g_i h_m(\mathbf{x}_i) + \frac{1}{2} h_i h_m^2(\mathbf{x}_i)] + \Omega(h_m) \tag{2.34}$$

Let $I_j = \{i \mid q(\mathbf{x}_i) = j\}$ be the set of indices assigned into the j-th leaf by the tree structure $q(\mathbf{x})$. Since all instances in a leaf receive the same score, $h_m(\mathbf{x}_i) = w_{q(\mathbf{x}_i)}$, the objective can be rewritten by summing over the leaves:

$$\mathcal{L}^{(m)} \approx \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \tag{2.35}$$

For a fixed tree structure $q(\mathbf{x})$, this is a simple quadratic in terms of the leaf weights w_j. The optimal weight w_j^* for each leaf can be found by setting the derivative with respect to w_j to zero:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{2.36}$$

At this point, it is worth remembering that the value of g_i and h_i depends on the loss function being used. For example, for the simple case of Mean Square Error regression, where the Hessians (h_i) are all 1 and the gradients (g_i) are equal to the residuals, (2.36) simplifies to the average of the residuals in the leaf, regularized by λ. Substituting w_j^* back into the objective function yields the final score for a given tree structure q:

$$\mathcal{L}^{(m)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{2.37}$$

This score measures the quality of a potential tree structure q. Conceptually, each term in the sum can be seen as a "purity" or "similarity" score for a leaf, rewarding homogeneity among the gradients within it. The goal of the tree construction algorithm is to find the tree structure that minimizes this score (or maximizes its negative). The use of the Hessian (h_i) provides more information about the curvature of the loss function, making the optimization analogous to Newton's method and potentially leading to faster convergence and better model fit than first-order methods.

In addition to this objective-based regularization, XGBoost also employs shrinkage and column subsampling. Shrinkage scales the contribution of each newly added tree by a learning rate η as in (2.31), reducing the influence of individual trees and allowing future trees to improve the model. Column subsampling, inspired by Random Forests from Breiman (2001), involves using only a random subset of features when constructing each tree, which further helps prevent overfitting and speeds up computation.

2.3.2.2 Split-Finding Criterion and the Computational Wall

The sophisticated statistical goal of the model is transformed into a difficult discrete optimization problem: finding the tree structure q that minimizes (2.37). The construction of each decision tree $h_m(\mathbf{x})$ within the ensemble is performed greedily, a strategy similar to the recursive partitioning used in classical decision trees like CART. However, while traditional trees maximize the reduction in an impurity measure such as the Gini Index or Entropy (as in (2.28)), XGBoost selects splits that maximize the reduction to the regularized objective function, $\mathcal{L}^{(m)}(q)$, defined in Equation (2.37). As in traditional trees, this maximized reduction is termed the **Gain**.

To translate this principle into a practical criterion, the gain of a potential split is calculated as the improvement in the objective's quality score, penalized by the complexity cost of adding a new leaf. Formally, consider a leaf node I that we are attempting to split. The score of the current node, if left unsplit, is given by:

$$\mathcal{L}_{\text{parent}} = -\frac{1}{2}\frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda} + \gamma \tag{2.38}$$

If the node is split into a left child I_L and a right child I_R, the total objective score is now the sum of the scores for these two new leaves:

$$\mathcal{L}_{\text{children}} = \left(-\frac{1}{2}\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \gamma\right) + \left(-\frac{1}{2}\frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} + \gamma\right) \tag{2.39}$$

As such, the gain from this split is the reduction in the objective function achieved by replacing the single unsplit leaf with the two newly split leaves. Thus, Gain $= \mathcal{L}_{\text{parent}} - \mathcal{L}_{\text{children}}$ post-split turns out to be:

$$\text{Gain} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{2.40}$$

The term γ in this gain formula effectively represents the penalty for adding an additional leaf to the tree. If the calculated gain does not exceed this γ threshold, the split is not performed. This acts as a pre-pruning mechanism, as explained in the context of (2.33), and is integrated within the tree construction stage as opposed to post-pruning, which removes branches after building the tree.

This Gain formula provides a clear, qualitative criterion for every potential split. The central computational task of the tree-building process, therefore, becomes a search problem: for every node, the algorithm must find the feature and split points that yield the maximum possible gain.

The most direct strategy for solving this problem is the **Exact Greedy Algorithm**. This method performs an exhaustive search over all possible split points for every feature. For each continuous feature, the algorithm first sorts the instances in the current node according to their values. It then performs a single linear scan through this sorted data, evaluating every unique value as a potential split threshold. At each candidate split, the running sum of gradients (g_i) and Hessians (h_i) for the resulting left and right partitions are used to calculate Gain via Equation (2.40).

While this approach is exhaustive and guarantees finding the best split according to the objective function, its computational cost is significant. The complexity is dominated by the sorting operation, resulting in a cost of $O(p \cdot n_N \cdot \log n_N)$ for a node containing n_N instances and p features. This high cost renders the exact algorithm impractical for the large-scale datasets common in modern applications and establishes it as the definitive computational wall. This reliance on an irregular, data-dependent operation is the central bottleneck that more scalable, approximate methods are designed to solve.

2.3.3 From Exact Search to Histogram Aggregation

The practical utility of any powerful statistical model is ultimately constrained by its computational feasibility on large-scale data. As established, the sort-based exact greedy algorithm, while statistically sound, confronts a "computational wall" due to its reliance on an expensive, data-dependent sorting operation at every node. Overcoming this barrier required a fundamental shift in strategy within XGBoost's design, moving away from the exhaustive search for the optimal split towards efficient and principled approximation. This section details this crucial algorithmic leap, which forms the first bridge to scalability, beginning with the introduction of the approximate framework and culminating in the highly optimized histogram method that redefines the problem's computational character.

2.3.3.1 Approximate Algorithm and Weighted Quantile Sketch

To address the performance limitations of the exact greedy algorithm, XGBoost employs a highly effective **approximate framework**. Instead of enumerating every unique feature value, this approach considers only a limited set of candidate split points proposed according to the percentiles of the feature distribution. This significantly reduces the computational burden while maintaining high predictive accuracy.

The core innovation that powers this framework is the **Weighted Quantile Sketch**. This is a novel algorithm for proposing candidate split points that are not uniformly distributed across a feature's range, but are instead concentrated in regions where the objective function is most sensitive. To achieve this, the algorithm uses the Hessians (h_i) as per-instance weights. The rationale is derived

directly from the Taylor-approximated objective function in Equation (2.34), which can be viewed as a weighted squared loss with labels $-g_i/h_i$ and weights h_i. The Hessian defines the curvature of the loss for a given instance, meaning that a larger h_i implies that the overall loss is more sensitive to changes in that instance's prediction. By weighting instances by their Hessians, the sketch proposes more candidate splits in areas of the feature space that have a greater impact on the objective, allowing for a more refined search where it matters most.

Formally, for a given feature k, the algorithm defines a rank function $r_k : \mathbb{R} \to [0, 1)$ over the dataset $\mathcal{D}_k = \{(x_{ik}, h_i)\}_{i=1}^n$. This function calculates the weighted proportion of instances whose feature value is less than a certain point z:

$$r_k(z) = \frac{1}{\sum_{(x,h)\in\mathcal{D}_k} h} \sum_{(x,h)\in\mathcal{D}_k, x<z} h \tag{2.41}$$

The Weighted Quantile Sketch algorithm seeks a set of candidate split points $\{s_{k1}, s_{k2}, ..., s_{kl}\}$ such that the difference in rank between consecutive points is approximately constant $|r_k(s_{k,j+1}) - r_k(s_{k,j})| < \varepsilon$. Here, the tolerance ε is an approximation factor that controls the number of candidates (intuitively, it is found that $l \approx 1/\epsilon$). The algorithm to build this sketch avoids a full sort and is instead based on a data structure that supports `merge` and `prune` operations, allowing it to work on data streams with provable theoretical guarantee.

This proposal of candidate splits can be performed in two modes:

- **Global Variant**: The candidate points for each feature are proposed once at the beginning of tree construction based on the initial dataset. The same set of candidates is then used for all splits at all levels of the tree, making this the computationally simplest approach.
- **Local Variant** The candidate points are re-proposed within each node, based only on the subset of data that currently reside in that node. This is more computationally intensive but is also more adaptive, potentially finding better splits deep in the tree.

The approximate algorithm, powered by this sophisticated candidate proposal strategy, provides a powerful and theoretically grounded first step away from brute-force computation.

2.3.3.2 Scaling with the Histogram Method

While the Weighted Quantile Sketch provides a theoretically elegant approximation, the most efficient and widely used implementation of the approximate framework is the **Histogram Method** due to its unmatched efficiency and hardware-friendly computational structure. This approach is the cornerstone of high-performance tree boosting and, as we will see, the critical enabler of GPU acceleration.

The algorithm's strategy replaces the expensive sorting phase with a process that unfolds in three clear steps:

1. **Feature Discretization (Binning)**: Before training begins, the algorithm performs a one-time scan over the dataset to discretize each continuous feature into a small, fixed number of bins (typically 256), mapping the continuous values to low-precision integers representing their bin index. This process is analogous to the step in the approximate framework where continuous features are mapped into buckets based on candidate split points. This initial investment pays significant dividends, as it dramatically reduces the number of split points to evaluate and allows the dataset to be held in a more compact, memory-efficient format.

2. **Histogram Aggregation**: At each node in the tree, the algorithm makes a single pass through the relevant data instances, a process referred to as *streaming*. For each feature, it builds a histogram by iterating through the pre-binned values. Each bin, b, aggregates two crucial statistics: the sum of gradients ($G_b = \sum_{i \in b} g_i$) and the sum of Hessians ($H_b = \sum_{i \in \text{bin}_b} h_i$), for the instances that fall into that bin.

3. **Split-Point Evaluation**: With the histogram built, the algorithm no longer needs to reference the underlying data to find the best split. Instead, it performs a fast linear scan over the histogram bins to find the split point that maximizes the gain formula. To evaluate a potential split after bin k, it needs the gradient and Hessians sums for the proposed left partition (bins 1 to k) and the right partition (bins $k+1$ to B). By using cumulative sums of G_b and H_b across the bins, these values can be retrieved in constant time, allowing the gain to be calculated at every possible split point with minimal computational effort.

The separation of concerns (i.e., a single data scan followed by a much faster scan over the small

aggregated histograms) is the source of the method's efficiency. The computational complexity of finding the best split for a node is reduced from the $O(p \cdot n_N \cdot \log n_N)$ of the exact greedy algorithm to approximately $O(p \cdot (n_N + B))$. Since the number of bins B is a small constant typically much smaller than the number of instances n_N, the dominant cost becomes the single linear scan over the data, a dramatic improvement.

To further optimize this process, XGBoost employs the *histogram subtraction trick*. Once the best split is found for a node, the data is partitioned into two children. Instead of scanning the data twice to build histograms for both the left and right child, the algorithm builds a histogram for only the smaller of the two nodes. The histogram for a larger node can then be calculated for free by taking the parent's known histogram and subtracting the newly computed histogram of its sibling. This saves redundant computation, especially as the tree grows deeper and nodes become smaller.

Ultimately, the computational structure of this histogram-based approach is defined by regular, data-parallel operations on compact data structures. It minimizes the complex, irregular memory access patterns of sorting and is precisely what makes the algorithm an ideal candidate for massive parallelization, laying the groundwork for the architectural leap to GPUs.

2.3.3.3 Algorithmic and System-Level Optimizations

Beyond the core histogram algorithm, the practical success of XGBoost hinges also on several key innovations that handle complexities of real-world data and optimize the use of hardware resources. These refinements include a principled approach to sparse data and a carefully engineered system design.

2.3.3.3.1 Sparsity-Aware Split-Finding XGBoost incorporates an efficient mechanism to handle sparse data natively, which adds to its functionality for real-world datasets where missing values and artifacts of feature engineering (like one-hot encoding) are common. Instead of imputing missing values, the algorithm has a mechanism for learning an optimal "default direction" for each feature at each node. Specifically, when considering a candidate split on a feature x_j for a set of instances I in the current node:

1. The instances are first partitioned into two groups: I_{present} with the non-missing values for x_j,

and I_{missing}, with the missing values.

2. The instances in I_{present} are divided into a potential left I'_L and right I'_R child sets, as per split finding.

3. The optimal direction for instances in I_{missing} is determined by explicitly evaluating via (2.40) both Gain_L and Gain_R for $I_L = I'_L \cup I_{\text{missing}}$ and $I_R = I'_R \cup I_{\text{missing}}$ respectively.

4. The default direction that yields maximum gain is then selected, and the actual split occurs in that direction. The respective gain is recorded for this candidate split to be used to compare against other candidate splits.

While this process does add a computational step, it is highly efficient, as the sums of gradients and Hessians for I_{missing} can be computed once for the current node and then added to the respective sums when calculating gains. This process is typically much faster than most imputation procedures followed by standard tree training, and has the added benefit of being data-driven, potentially uncovering and exploiting patterns in the missingness itself.

2.3.3.3.2 System-Level Design for Scalability Beyond the sophisticated mathematical formulation of its regularized objective and its advanced split finding provisions, XGBoost's performance is also attributed to some careful engineering considerations that optimize resource utilization and enable efficient processing of large-scale datasets that often exceed the capacity of the main memory. While a full treatment of these system-level details fall more within the domain of software engineering, their core ideas are summarized here as they are essential for understanding the system's overall scalability. Specifically:

- **Column Blocks for Parallel Learning**: Data is stored in in-memory units called "blocks". Within each block, data is stored in a compressed column format, with each column sorted by its corresponding feature value. This pre-sorted layout, computed only once, facilitates the parallelization of the split-finding process across CPU threads.

- **Cache-Aware Access**: To mitigate the CPU cache misses that can occur from the non-contiguous memory access required when gathering gradient statistics, XGBoost employs a cache-aware prefetching algorithm. Each thread uses an internal buffer to fetch data in

mini-batches, reducing runtime overhead on large datasets.

- **Out-of-Core Computation**: XGBoost can handle datasets larger than available RAM by dividing the data into blocks and storing them on disk. It improves I/O throughput with two primary techniques: block compression, which trades CPU cycles for reduced disk reading time, and block sharding, which striped data across multiple disks to be read in parallel.

These system-level optimizations ensure that the sophisticated algorithmic features can be practically applied to the large and complex datasets encountered in modern computational statistics.

2.3.4 GPU-Accelerated Tree Construction: XGBoost's `gpu_hist`

2.3.4.1 The Emergent Computational Profile

The transition from the exact greedy algorithm to a fully-featured, histogram-based approach does more than merely optimize the tree building process. It fundamentally reshapes the computational character of the problem. By eliminating the irregular, data-dependent sorting operation at its core, the primary performance bottleneck of the system shifts entirely. The algorithm is no longer constrained by the complex, branching logic of a CPU-intensive sort, but rather by the simple, regular task of streaming through data to aggregate statistics into histograms. This transformation creates a new performance profiles, summarized Table 2.2, which serves as the crucial link to the next stage of hardware acceleration.

Table 2.2: A comparison of XGBoost's tree construction algorithms.

Criterion	Exact Greedy Algorithm	Approximate Quantile Sketch	Histogram Method
XGBoost `tree_method`	`exact`	`approx`	`hist`
Core Operation	Full Data Sorting	Weighted Quantile Summary	Histogram Aggregation
Candidate Splits	All unique values	Few, data-driven	Fixed bin boundaries
Primary Bottleneck	$O(n \log n)$ (Sort-based)	$O(n)$ (Sketch-based)	$O(n)$ (Scan-based)
Algorithm Complexity	High (Irregular)	Medium (Streaming)	Low (Regular)
GPU Suitability	Poor	Better	**Excellent**

The trajectory from complex, data-dependent logic towards simple, parallelizable aggregation is the key to XGBoost's modern scalability. This evolution transforms the problem from being **compute-bound**, where performance is limited by the speed of CPU's logical operations, to being **memory-bandwidth-bound**, where performance is dictated by the rate at which data can be read from mem-

ory. As derived in the context of Equation (2.34), each boosting iteration requires two large-scale, data-parallel computations:

1. The calculation of per-instance first and second-order gradients, g_i and h_i, for all n training instances.
2. The repeated aggregation of these g_i and h_i values into histograms to evaluate the Gain (Equation (2.40)) for every potential split in every growing node.

The very optimizations designed to make XGBoost faster on a CPU unintentionally forged a computational profile that is a relatively poor match for latency-optimized CPU cores but a perfect fit for an entirely different hardware paradigm: the massively parallel, high-throughput architecture of the modern GPU.

This architectural synergy is exploited by the GPU histogram method, the so called `gpu_hist` tree construction algorithm in XGBoost detailed by Mitchell & Frank (2017). This method represents a ground-up redesign of the tree-build process, engineered to execute entirely on the GPU and leverage its massive parallelism. The result is a feature-complete implementation that can reduce training times from hours to minutes, making comprehensive modelling and hyperparameter tuning practically feasible on large-scale datasets.

2.3.4.2 Bridging Algorithm and Architecture

The efficacy of `gpu_hist` stems from the direct mapping between the computational patterns of the histogram algorithm and the core design philosophy of the GPU. As discussed, CPUs are latency-optimized, designed for complex, sequential logic on a few powerful cores. GPUs are throughput-optimized, featuring thousands of simpler cores built to execute the same instruction across vast amounts of data in parallel (the "SIMD" architecture). They excel at hiding the high latency of memory access by orchestrating a massive number of concurrent threads, ensuring compute units remain saturated.

The mathematical structure of the histogram method is a perfect match for this paradigm. The calculation of the Gain in (2.40) relies on sums of gradients ($\sum g_i$) and Hessians ($\sum h_i$). Building a histogram is, therefore, fundamentally a *parallel reduction*, a classic GPU computing pattern where an array of values is reduced to a single value using a binary-associative operator. Thousands of

threads can collaboratively perform this aggregation with high efficiency using a tiered approach of reductions within warps (groups of 32 threads) and thread blocks. In contrast, the complex branching and sorting logic of the Exact Greedy Algorithm is ill-suited to this architecture.

Furthermore, the `gpu_hist` implementation processes all nodes at a given level of the tree concurrently. While many traditional tree algorithms process nodes one by one, leaving the GPU severely underutilized, this level-wise strategy ensures that a large volume of parallel work is always available to keep the hardware fully occupied.

2.3.4.3 Data Layout and Level-Wise Growth on the GPU

The overarching strategy of the `gpu_hist` algorithm is to build trees *level-wise*, processing all nodes at a given depth concurrently. This approach ensures that a sufficient volume of parallel work is always available to keep the thousands of GPU cores occupied, which is essential for hiding memory latency and maximizes hardware utilization. At each level, the algorithm executes a sequence of steps: it finds the best splits for all current leaf nodes, updates the position of every training instance based on these splits, and finally re-groups the data for the next level.

To facilitate this high-throughput strategy, both data organization and the parallel processing model are critical. The work of finding the best split is delegated such that each feature is processed in parallel by a dedicated thread block. To make this efficient, data is organized in GPU device memory in a specific *columnar format*. Each feature is stored as a contiguous block of values, and each value is paired with the ID of the instance it belongs to, which allows it to be mapped back to the corresponding gradient (g_i) and Hessian (h_i) pair for that data point.

This columnar layout is fundamental to performance on GPUs because it enabled *coalesced memory access*, a crucial optimization where a group of threads can read a contiguous segment of memory in a single, efficient transaction. This maximizes memory bandwidth which is vital for the histogram-building step's rapid, sequential scan over all values of a feature. For sparse data, this principle is extended by using specialized compressed formats that maintain a regular memory access pattern suitable for the GPU's SIMD execution model, resulting in a lower memory footprint and higher throughput. After each thread block find the best split for its assigned feature, the results are written to global memory, and a final, fast reduction kernel selects the single best split across all features

for each node.

2.3.4.4 A Hybrid Strategy for Split-Finding

A key insight from Mitchell & Frank (2017) is that the `gpu_hist` implementation is not one single algorithm but a hybrid that intelligently switches its split-finding strategy based on tree depth. This pragmatic engineering choice is designed to balance the high fixed cost of sorting against the limited temporary storage (registers and shared memory) available on the GPU. The algorithm can therefore achieve optimal performance for both shallow and deep trees.

2.3.4.4.1 Interleaved Method (Shallow Trees) For the initial, shallower levels of the tree (e.g., up to depth 5 or 6), data instances remain in their original sorted order, and their assignment to different nodes is tracked separately. This creates a situation where data for different nodes is mixed, or "interleaved", in memory. To find splits in this state required a *multiscan* operation, which is a variation of a parallel prefix sum that must maintain separate running sums for each node bucket.

Because this requires storing a vector of sums, it becomes impractical as the number of nodes (and thus buckets) grows, due to the limited temporary storage on the GPU. The `gpu_hist` implementation uses fast, warp-level scans executed sequentially for each bucket, which is efficient for a small number of nodes but whose runtime increases exponentially with tree depth.

2.3.4.4.2 Sorting Method (Deep Trees) Once the number of nodes grows large, the interleaved approach becomes inefficient. The algorithm then switches to a sorting-based method. It performs a single, highly efficient GPU radix sort to physically re-group all instances by their current node ID. This sorting operation transforms the data so that all instances belonging to the same node are in a contiguous block, or "segment".

This reordering allows the algorithm to use a more scalable *segmented scan* to find the best splits. A segmented scan is a parallel prefix sum modified to handle multiple sequences within a single array. It works by using a custom operator that checks the key (in this case, the node ID) of adjacent elements. If the key changes, the running sum is reset. This approach has constant temporary storage requirements, allowing it to scale to arbitrary tree depths.

2.3.4.5 GPU-Accelerated Sparsity Handling

As detailed in Section 2.3.3.3.1, the algorithm must learn an optimal "default direction" for missing values by evaluating whether sending them left or right yields a greater Gain. The GPU implementation optimizes this process significantly.

Instead of the two-scan method used by the CPU algorithm, the GPU version first performs a single, fast parallel reduction over all the non-missing values in the node to get their combined gradient and Hessian statistics. By subtracting this result from the known total statistics of the parent node, it can derive the statistics for the group of missing values in one step. With these statistics known, the algorithm then only needs to perform a single scan over the non-missing values to find the best split point and determine the optimal default direction. This approach is far more efficient on parallel hardware, minimizing memory transfers and redundant computation.

2.3.4.6 Modern Enhancements and Scalability Beyond a Single GPU

The development of GPU acceleration for XGBoost is ongoing. Recent releases demonstrate a commitment to expanding this high-performance backend. Key advancements include:

Backend and Hardware Support: Introducing of a SYCL backend for broader hardware compatibility beyond NVIDIA's CUDA.

Algorithmic Improvements: Support for multi-output decision trees and auto-scaling of the number of histogram bins to optimize usage of shared memory on modern GPUs (e.g., keeping memory per SM under 8 MiB).

Distributed and Multi-GPU Training: For scaling beyond a single card, XGBoost integrates with distributed computing frameworks like Dask and Spark. It leverages NVIDIA's NCCL library for high-speed, all-reduce communication between GPUs, enabling efficient data-parallel training across multiple GPUs on a single or multiple nodes.

In-Situ Hyperparameter Tuning: Recent versions allow for hyperparameter tuning tasks like cross-validation to be run entirely on the GPU. This avoids the significant overhead of moving data back and forth between the GPU and CPU, dramatically speeding up the model selection process.

2.4 Discussion: Algorithmic Co-Design Insights

The preceding analysis, while focusing on two specific classes of statistical methods, provides a fertile ground for a broader discussion on successfully accelerating statistical algorithms. Comparing Falkon's implementation of Kernel Methods and XGBoost's GPU-accelerated histograms, we identify common patterns with wider implications for the field of computational statistics.

A primary theme emerging from the analysis is the primacy of algorithmic co-design. It becomes clear that hardware acceleration is rarely a "plug-and-play" solution. In both cases, the most critical performance gains were unlocked only after fundamental algorithmic transformation had taken place: the Nyström approximation and iterative CG solver for Kernel Methods, and the histogram method for XGBoost. That goes to show that hardware can only accelerate a process that has first been made algorithmically suitable for parallelism. This initial step reshapes the computational profile of the problem, systematically replacing complex CPU-bound logic like direct matrix inversion or sorting with simple, regular, data-parallel operations such as matrix-vector products and histogram aggregation. This shift in the nature of the problem is a common theme in high-performance computing, and this chapter demonstrates its direct application in a statistical context.

The strategic use of intermediate representation is also a common pattern that emerges. Both solutions rely on transforming raw data into a smaller, computationally convenient structure: the low-rank Nyström matrices for Falkon, and the gradient-and-Hessian histograms for XGBoost. The principle of performing the bulk of the computationally demanding work on these compact representations makes sense, as it minimizes data movement and improves efficiency.

Lastly, it also becomes evident that the path to acceleration requires navigating both numerical and architectural challenges simultaneously. The promise of GPU speed-ups is tied to the use of lower-precision arithmetic, which introduces tangible risk of numerical instability that both Falkon and XGBoost must actively manage with mixed-precision schemes. Likewise, both solutions demonstrate a deep awareness of the hardware's memory hierarchy, using explicit Out-of-Core and memory-aware strategies. All of that is to say that a successful implementation must treat the target hardware not as a black box, but as a complex system whose architectural and numerical properties must be taken into account.

The most profound gains in computational statistics are achieved through an integrated co-design process between the underlying statistical algorithm and hardware architecture. The patterns observed here suggest that reshaping statistical theory with algorithmic innovations, we can create computational structures that modern parallel hardware can exploit at unprecedented scale, opening new avenues for research and application.

Chapter 3

Benchmarking Statistical Solutions on GPU

3.1 Introduction: Experimental Framework

In Chapter 2, we laid the theoretical groundwork for understanding how GPU acceleration can address the computational intensity of modern statistical methods. We explored two distinct classes of algorithms: kernel methods, renowned for their ability to model complex non-linear relationships but often constrained by the $O(n^2)$ or worse complexity of kernel matrix operations, and XGBoost, a leading gradient boosting framework whose sequential tree-building process presents a significant computational challenge. For each, we dissected their inherent scalability bottlenecks and reviewed the algorithmic innovations that create a computational profile amenable to parallelization; Nyström approximation for kernel methods, and histogram-based split-finding for XGBoost.

This chapter transitions from theory to practice. We aim to provide a rigorous, empirical validation of the theoretical claims by demonstrating the tangible impact of GPU acceleration on these methods. We present detailed benchmarks of state-of-the-art, GPU-accelerated libraries against their conventional CPU-based counterparts on large-scale, real-world datasets. A particular focus has been placed on ensuring that the experimental design is transparent and the results are reproducible. As such, all experimental setups, code, and resulting data are meticulously documented and have been made available in the appendices. In this section, we provide a high-level overview of the standardized experimental framework designed and applied to all benchmarks in this chapter.

To ensure the integrity and reproducibility of our findings, a standardized experimental framework was designed. This includes the selection of two challenging, large-scale public datasets, the use of consistent hardware and software environment based on Google Colab, and a clear methodology for measuring both computational performance and statistical accuracy. This framework is detailed in the subsequent sections and provides the foundation for the empirical case studies of Falkon and XGBoost that form the core of this chapter.

3.1.1 Datasets for Benchmarking

To rigorously evaluate the performance of the selected methods, two distinct, large-scale, and publicly available datasets were chosen. These datasets represent challenging, real-world problems in regression and classification, respectively, and their scale is suitable for testing model scalability. The following sections detail the origin, characteristics, and preprocessing pipelines for each.

3.1.1.1 NYC Taxi Fare Prediction (Regression)

This dataset is constructed from the complete 2024 NYC Yellow Taxi trip data, sourced from the New York (N.Y.). Taxi and Limousine Commission (2019) (TLC) public data repository. The dataset presents a challenging large-scale regression problem due to its volume and the complexity of its features. The primary objective of this task is to predict the continuous `fare_amount` target variable.

The raw data consists of monthly Parquet files, which were first downloaded and consolidated into a single data pool. A comprehensive preprocessing pipeline was then executed to clean the data and engineer relevant features.

1. **Feature Engineering**: To enrich the feature set, trip `duration` (in minutes) was calculated from the `tpep_pickup_datetime` and `tpep_dropoff_datetime` timestamps. Additionally, cyclical temporal features, `pickup_day` and `pickup_hour`, were extracted from the pickup timestamp to capture weekly and daily patterns.
2. **Filtering**: A multi-stage filtering process was applied to ensure data quality and remove outliers. Records were retained only if they met the following criteria:

- `fare_amount` between $2.50 and $200.
- `trip_distance` between 0.1 and 100 miles.
- `duration` between 1 and 360 minutes.
- `passenger_count` between 1 and 6.

Following this, any rows with remaining missing values were dropped. This pipeline resulted in a final data pool of over 35 million valid instances.

3. **Data Splitting**: For model training and evaluation, the processed data was first partitioned into a training pool (80%) and a final test pool (20%). To evaluate scalability, training sets of varying sizes were then sampled from the training pool.

4. **Scaling**: As a final preprocessing step, all features in the training and test sets were standardized by removing the mean and scaling to unit variance using the `StandardScaler` from scikit-learn.

The final features used for model training are described in Table 3.1:

Table 3.1: Description of NYC Taxi Fare features used for model training.

Feature Name	Description	Type
`VendorID`	A code indicating the TPEP provider.	Categorical
`passenger_count`	The number of passengers in the vehicle.	Numerical
`trip_distance`	The elapsed trip distance in miles.	Numerical
`RatecodeID`	The final rate code for the trip.	Categorical
`PULocationID`	TLC Taxi Zone ID where the trip began.	Categorical
`DOLocationID`	TLC Taxi Zone ID where the trip ended.	Categorical
`payment_type`	A numeric code indicating how the passenger paid.	Categorical
`duration`	*Engineered:* Total trip duration in minutes.	Numerical
`pickup_day`	*Engineered:* Day of the week (0=Mon, 6=Sun).	Numerical
`pickup_hour`	*Engineered:* Hour of the day (0-23).	Numerical
`fare_amount`	The time-and-distance fare calculated by the meter.	**Target**

3.1.1.2 HIGGS Boson Detection (Classification)

The second dataset is the HIGGS benchmark from the UCI Machine Learning Repository (Dua & Graff 2017, Baldi et al. 2014). It is a well-known dataset for large-scale binary classification, containing 11 million instances derived from Monte Carlo simulations of particle collisions. The

objective is to classify events as either "signal" (class 1), corresponding to the production of a Higgs boson, or "background" (class 0).

The dataset is already well-structured, requiring minimal preprocessing:

1. **Feature Set**: The dataset contains 28 features and one class label. The features are categorized into 21 low-level kinematic properties directly measured by particle detectors and 7 higher-level, derived features. All 28 features were used for the classification task. No additional feature engineering or filtering was necessary.
2. **Data Splitting**: The full dataset was partitioned into a training pool (80%) and a fixed test set (20%). To handle the natural class imbalance and ensure that the training and test sets are representative of the overall data distribution, this split was stratified based on the class label. To test model scalability, training sets of varying sizes were then subsampled from the training pool, again using stratification.
3. **Scaling**: As with the regression dataset, all 28 features in the training and test sets were standardized using `StandardScaler`.

3.1.2 Hardware and Software Environment

All benchmarks were conducted within the Google Colaboratory (Colab) cloud platform to ensure a standardized and reproducible experimental environment. This choice provides on-demand access to specialized computational resources, mitigating the challenges of local hardware procurement and configuration.

The experimental work was carried out across two distinct hardware configurations. Initial testing and exploratory analysis was performed using the free tier of Google Colab. The primary hardware for the GPU-accelerated tests in this setting was an **NVIDIA T4 GPU** with 14.7 GB of VRAM. The corresponding CPU baseline was established using the platform's multi-core **Intel Xeon CPU @ 2.00GHz**, with access to 12.7 GB of system RAM.

The final, reported benchmarks were executed on the more powerful **Google Colab Pro+** tier. This premium configuration featured a high-performance **NVIDIA A100-SXM4-40GB GPU** and an **Intel Xeon CPU @ 2.20GHz**, with access to a substantial **83.5 GB of system RAM**. The upgrade

to the A100 GPU yielded a dramatic performance improvement, with a qualitative speedup of nearly $5\times$ in some test cases compared to the T4. That is both due to the increased performance of the GPU itself, and due to the larger capacity in host and device memory. Despite the availability of superior hardware, the experimental settings from the initial T4 tests (e.g., sample sizes) were intentionally retained. This approach ensures that the benchmark results are reproducible even on the free tier, but given the configurable options of the tests, it can easily be adjusted.

All experiments were performed within a consistent Python ecosystem. The core libraries used throughout the chapter include Falkon, XGBoost, Scikit-learn, PyTorch, NumPy and Pandas. Specific versions for each library are documented within the respective case-study sections to ensure full transparency.

3.1.3 Benchmarking Methodology

A standardized protocol was applied to all experiments. Each GPU-accelerated implementation is benchmark against one or more CPU-based alternatives to allow for direct measurement of speedup and scalability.

3.1.3.1 Performance Metrics

The primary metrics used for evaluating computational performance are:

- **Training Time**: Wall-clock time in seconds to fit the model on the training dataset.
- **Prediction Time**: Wall-clock time in seconds to generate predictions for the held-out test set.

3.1.3.2 Statistical Accuracy Metrics

To ensure that computational performance gains do not compromise predictive quality, the following task-specific metrics are used:

- For **regression tasks**, Root Mean Squared Error (RMSE) and the Coefficient of Determination (R^2) are reported.
- For **classification tasks**, Area Under the ROC Curve (AUC), Accuracy, and the F1-Score are reported.

3.1.3.3 Reproducibility and Uncertainty Quantification

To mitigate variability from system load and stochastic processes within the algorithms, each benchmark configuration was executed multiple times, using parameter `N_RUNS` in the Notebook setup. This parameter was set to `N_RUNS = 5` for the measurements reported on this text. The results for time-based and accuracy metrics are reported as **mean** $\pm$ **standard deviation**. A fixed random state (parameter `RANDOM_STATE = 6`) was used for all stochastic operations, including data splitting and model initialization, to ensure comparability of results across different runs and implementations.

3.2 Approximate Kernel Ridge Regression on GPU with Falkon

Section 2.2 established the theoretical foundations of kernel methods, their inherent scalability limitations, and the algorithmic solutions designed to overcome those bottlenecks, namely the Nyström approximation and preconditioned conjugate gradient (CG) solvers. This section provides a rigorous empirical demonstration of these concepts through a detailed case study of the Falkon library.

3.2.1 Introduction to the Falkon Library

Falkon is an open-source Python library designed to scale up kernel methods to very large datasets with multi-GPU acceleration. It implements solvers for both approximate Kernel Ridge Regression (KRR), which seeks to solve the optimization problem outlined in (2.11), and Kernel Logistic Regression (KLR). As discussed previously, traditional, exact kernel methods face significant computational bottlenecks due to the need to construct and manipulate the $n \times n$ kernel matrix $\mathbf{K}_{nn}$, leading to $O(n^2)$ memory and $O(n^3)$ time complexities. Falkon addresses these challenges by integrating Nyström approximation with a preconditioned CG solver, implemented on a PyTorch backend to leverage GPU parallelism.

The library's core algorithm centres around approximating the full kernel matrix with a much smaller $n \times m$ matrix based on m Nyström centres (with $m \ll n$), approximating $\mathbf{K}_{nn}$ as in (2.16) and effectively reducing the complexity for solving KRR from $O(n^3)$ to roughly $O(nm^2)$. The compu-

tationally intensive operations, such as kernel evaluations and the iterative CG solver used to handle the resulting linear system (akin to solving the preconditioned system (2.23)), are parallelized on the GPU, leveraging strategies such as out-of-core operations and Falkon's specific memory optimization techniques discussed in the previous chapter. The library can be further accelerated by integrating with KeOps, an efficient C++ library for fast kernel evaluations. This design allows Falkon to handle datasets with millions (or even billions) of points, which would be intractable for exact kernel methods.

3.2.2 Experimental Setup

3.2.2.1 Software Configuration

To empirically evaluate the performance benefits and practical utility of GPU acceleration for kernel methods via Falkon, a series of benchmarking experiments were designed. The general protocol was outlined in the previous section. The final software configuration for the experiments presented, achieved after addressing initial version incompatibilities, such as NumPy ABI conflicts which required downgrading the default NumPy and PyKeOps installations and pinning to an older version, comprised the following key library versions:

- **Falkon**: 0.8.5
- **PyTorch**: 2.2.0 (with CUDA 12.1 support)
- **PyKeOps**: 2.2
- **Scikit-learn**: 1.6.1
- **NumPy**: 1.26.4
- **Pandas**: 2.2.2

3.2.2.2 Implementations Compared

Three distinct implementations were compared:

1. **Falkon (GPU)**: The GPU-accelerated configuration with `use_cpu=False` and `keops_active="yes"`. Float32 precision was used for optimal performance on GPU.
2. **Falkon (CPU)**: A direct CPU baseline of the same approximate algorithm, with

`use_cpu=True` and `keops_active="no"`. Float64 precision was used, as lower floating point precision is not numerically stable on the CPU implementation.

3. **Scikit-learn `KernelRidge` (CPU)**: A traditional, non-approximated KRR solver serving as a baseline for exact kernel methods.

3.2.2.3 Hyperparameter Configuration

A consistent set of hyperparameters was chosen for all Falkon runs to ensure a fair comparison:

- **Kernel** (`kernel`): Gaussian Kernel (as in (2.5)) with $\sigma = 15.0$ (equivalently corresponding to $\gamma = \frac{1}{2\sigma^2}$ in (2.6) for `KernelRidge`).
- **Regularization Penalty** (`penalty`): The regularization parameter in the KRR objective (and its Nyström-approximated counterpart in (2.17) was set to $\lambda = 1 \times 10^{-6}$.
- **Number of Nyström Centers** (`M`): To maintain a consistent approximation quality relative to the dataset size, m was set dynamically using the formula $m = \lfloor \log n \sqrt{n} \rfloor$, where n is the number of training samples. This heuristic was informed by Meanti et al. (2020)'s claim of model performance at $m = O(\sqrt{n})$, as a slightly more conservative (i.e., higher) sample size.
- **Maximum Iterations** (`maxiter`): The maximum number of iterations for the CG solver was set to 20.

Note that the experiments were restricted to a specific set of reasonable predictive variables and hyperparameters in order to be able to focus specifically on computational metrics. As such, finding the truly optimal predictive model via cross-validation was outside the scope.

3.2.3 Benchmark 1: Scalability with Sample Size (*n*)

The primary experiment was designed to measure how each implementation scales as the number of training samples (n) increases. The training and prediction times were recorded across stratified subsamples of the **NYC Taxi Fare** dataset, ranging from $1,000$ to $2,000,000$ instances.

3.2.3.1 Execution Time and Resource Consumption

The results of the scalability benchmark are summarized in Table 3.2. It captures the training time, prediction time, Root Mean Squared Error (RMSE) and R^2 score for each configuration across various dataset sizes.

Table 3.2: Scalability Benchmark Results: A comparison of training time (s), prediction time (s), RMSE, and R^2 for Falkon (GPU), Falkon (CPU), and Scikit-learn (CPU) across varying sample sizes (n) with number of centres $m = \lfloor \log n \sqrt{n} \rfloor$.

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
1,000	218	**Falkon (GPU)**	0.40 ± 0.18	0.17 ± 0.02	4.6489	0.9279
1,000	218	Falkon (CPU)	0.33 ± 0.18	0.08 ± 0.01	4.1540	0.9424
1,000	NA	Scikit-learn (CPU)	0.15 ± 0.01	0.84 ± 0.01	4.9229	0.9192
2,000	339	**Falkon (GPU)**	0.35 ± 0.02	0.00 ± 0.00	4.5030	0.9324
2,000	339	Falkon (CPU)	0.21 ± 0.01	0.10 ± 0.00	3.9475	0.9480
2,000	NA	Scikit-learn (CPU)	0.30 ± 0.15	1.64 ± 0.02	4.4838	0.9329
5,000	602	**Falkon (GPU)**	1.16 ± 0.30	0.00 ± 0.00	4.3895	0.9357
5,000	602	Falkon (CPU)	0.34 ± 0.00	0.17 ± 0.00	3.6841	0.9547
5,000	NA	Scikit-learn (CPU)	0.89 ± 0.07	4.13 ± 0.03	4.2894	0.9386
10,000	921	**Falkon (GPU)**	2.67 ± 0.23	0.00 ± 0.00	4.0620	0.9450
10,000	921	Falkon (CPU)	0.94 ± 0.14	0.25 ± 0.00	3.6450	0.9557
10,000	NA	Scikit-learn (CPU)	4.12 ± 0.08	7.50 ± 0.04	4.2117	0.9408
20,000	1400	**Falkon (GPU)**	7.58 ± 0.23	0.00 ± 0.00	4.0692	0.9448
20,000	1400	Falkon (CPU)	2.15 ± 0.14	0.37 ± 0.01	3.8172	0.9514
50,000	2419	**Falkon (GPU)**	0.79 ± 0.02	0.00 ± 0.00	4.0378	0.9456
50,000	2419	Falkon (CPU)	8.31 ± 0.25	0.60 ± 0.00	3.6291	0.9561
100,000	3640	**Falkon (GPU)**	1.62 ± 0.17	0.00 ± 0.00	3.9626	0.9476
100,000	3640	Falkon (CPU)	23.48 ± 0.11	0.90 ± 0.00	3.5804	0.9572

Table 3.2: Scalability Benchmark Results: A comparison of training time (s), prediction time (s), RMSE, and R^2 for Falkon (GPU), Falkon (CPU), and Scikit-learn (CPU) across varying sample sizes (n) with number of centres $m = \lfloor \log n\sqrt{n} \rfloor$. *(continued)*

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
200,000	5458	**Falkon (GPU)**	3.42 ± 0.03	0.01 ± 0.00	3.9818	0.9471
200,000	5458	Falkon (CPU)	68.19 ± 0.84	1.44 ± 0.16	3.5490	0.9580
500,000	9278	**Falkon (GPU)**	9.69 ± 0.14	0.01 ± 0.00	4.0204	0.9461
1,000,000	13815	**Falkon (GPU)**	19.02 ± 0.31	0.01 ± 0.00	3.9845	0.9470
2,000,000	20518	**Falkon (GPU)**	39.62 ± 0.27	0.01 ± 0.00	4.2097	0.9409

3.2.3.2 Analysis of Scalability and Speedup Factors

The results in Table 3.2 and Figure 3.1 clearly demonstrate the performance paradigm shift enabled by GPU acceleration. For small dataset sizes ($n \leq 20,000$), the overhead associated with GPU memory transfers and kernel initializations results in the Falkon (CPU) implementation being faster. At $n = 20,000$, the CPU version completes training in 2.15 ± 0.14 seconds, while the GPU version takes approximately 7.58 ± 0.23 seconds. This is the expected behavior and highlights that for smaller tasks, a simple CPU approach remains efficient.

However, a critical inflection point occurs around $n = 50,000$. At this size, the GPU-accelerated Falkon becomes dramatically faster, completing training in just 0.79 ± 0.02 seconds compared to the CPU version's 8.31 ± 0.25 seconds, yielding a speedup of over 11 $\times$. This reversal is primarily because the core CG solver, the most computationally intensive part of the algorithm, automatically switches from the CPU-based implementation to the GPU-based one at this scale, as observed in the library's debug logs. This trend highlights the GPU's superior ability to handle the parallelizable matrix-vector multiplications as the data size grows.

The performance gap widens significantly at $n = 100,000$ where the GPU version takes only 1.62 ± 0.17 seconds compared to the CPU version's 23.48 ± 0.11 seconds, yielding an approximately 14 $\times$ speedup. Beyond this point, the Falkon (CPU) implementation became too slow to be practical,

and often resulted in memory errors. Meanwhile, the `KernelRidge` baseline failed due to memory errors at just $n = 20,000$. In contrast, the GPU-accelerated Falkon scaled efficiently, successfully training on a dataset of $1,000,000$ samples in just 19.02 ± 0.31 seconds, a task impossible for the other methods on the given hardware.

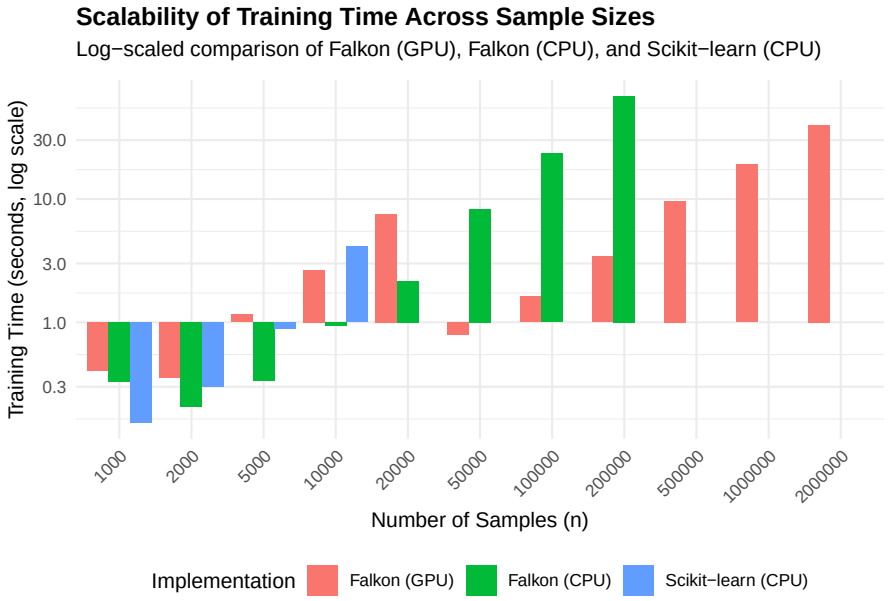


Figure 3.1: Training time scalability comparison across implementations using a log-scaled y-axis.

Critically, this dramatic acceleration did not compromise model quality. Across all successful runs, the RMSE and R^2 scores remained comparable between the GPU and CPU implementations, confirming that the performance gains were achieved without sacrificing predictive performance at $m = \lfloor \log n\sqrt{n} \rfloor$.

3.2.4 Benchmark 2: Accuracy vs. Nyström Centers (m)

While the first benchmark established computational scalability, this second experiment assesses the relationship between the number of Nyström centres (m) and the model's predictive accuracy. This test is crucial for validating that the Nyström approximation, while efficient, can still yield a high-quality predictive model. The number of training samples was fixed at a large scale of $n = 5,000,000$, while m was varied across a range from approximately $0.1\sqrt{n}$ to $10\sqrt{n}$.

3.2.4.1 Overcoming a Data Engineering Bottleneck

Pushing the dataset to a size of $n = 5$ million revealed a critical bottleneck at the current configuration. The initial approach of loading and processing the full source dataset in each iteration of the experiment led to out-of-memory errors. This was not due to a lack of total available RAM, but rather a phenomenon known as memory fragmentation, where the repeated allocation and deallocation of a multi-GB block of memory prevented the system from finding a contiguous block for subsequent loads. To resolve this, a disk-based caching strategy was implemented: once the final, processed PyTorch tensors were created, they were saved to a file and loaded directly in subsequent runs, enabling stable execution of the large-scale benchmark.

3.2.4.2 Accuracy vs. Number of Centers (*m*)

The results of this benchmark, summarized in Table 3.3, reveal a nuanced trade-off between the number of inducing points and model performance.

Table 3.3: Accuracy vs. Nyström Centers: Results from Benchmark 2 showing model accuracy as a function of Nyström centres m with a fixed sample size $n = 5{,}000{,}000$.

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
5,000,000	223	**Falkon (GPU)**	12.20 ± 0.44	0.13 ± 0.01	8.1308	0.7795
5,000,000	447	**Falkon (GPU)**	12.12 ± 0.11	0.00 ± 0.00	7.2994	0.8223
5,000,000	670	**Falkon (GPU)**	12.03 ± 0.02	0.00 ± 0.00	8.4913	0.7595
5,000,000	894	**Falkon (GPU)**	12.12 ± 0.02	0.00 ± 0.00	5.4847	0.8996
5,000,000	1,118	**Falkon (GPU)**	12.34 ± 0.23	0.00 ± 0.00	5.2332	0.9086
5,000,000	1,341	**Falkon (GPU)**	12.40 ± 0.05	0.00 ± 0.00	5.7635	0.8892
5,000,000	1,565	**Falkon (GPU)**	12.58 ± 0.03	0.00 ± 0.00	5.8015	0.8877
5,000,000	1,788	**Falkon (GPU)**	12.71 ± 0.07	0.00 ± 0.00	4.9920	0.9169
5,000,000	2,012	**Falkon (GPU)**	12.91 ± 0.10	0.00 ± 0.00	5.4802	0.8998
5,000,000	2,236	**Falkon (GPU)**	13.02 ± 0.03	0.00 ± 0.00	5.3791	0.9035
5,000,000	4,472	**Falkon (GPU)**	14.90 ± 0.07	0.00 ± 0.00	4.8998	0.9199
5,000,000	6,708	**Falkon (GPU)**	17.48 ± 0.22	0.01 ± 0.00	4.3043	0.9382

Table 3.3: Accuracy vs. Nyström Centers: Results from Benchmark 2 showing model accuracy as a function of Nyström centres m with a fixed sample size $n = 5{,}000{,}000$. *(continued)*

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
5,000,000	8,944	**Falkon (GPU)**	21.22 ± 0.45	0.01 ± 0.00	5.2537	0.9079
5,000,000	11,180	**Falkon (GPU)**	24.78 ± 0.25	0.01 ± 0.00	4.1767	0.9418
5,000,000	13,416	**Falkon (GPU)**	29.34 ± 0.08	0.01 ± 0.00	5.3471	0.9046
5,000,000	15,652	**Falkon (GPU)**	35.11 ± 0.11	0.01 ± 0.00	4.6981	0.9264
5,000,000	17,888	**Falkon (GPU)**	42.38 ± 0.09	0.01 ± 0.00	4.0985	0.9440
5,000,000	20,124	**Falkon (GPU)**	50.88 ± 0.16	0.01 ± 0.00	4.3770	0.9361
5,000,000	22,360	**Falkon (GPU)**	61.14 ± 0.42	0.01 ± 0.00	5.2821	0.9069

There is a clear and significant improvement in accuracy as m initially increases. With a very small number of centres ($m = 223$), the model produces a poor approximation (RMSE of 8.13). However, as m increases towards $\sqrt{n}$ at $m = 2236$, the performance improves dramatically, with the RMSE dropping by over 33.8% to 5.38. The model's performance continues to improve, reaching a peak accuracy at approximately $8\sqrt{n}$, where the RMSE is minimized at 4.10 and the R^2 score is maximized at 0.94.

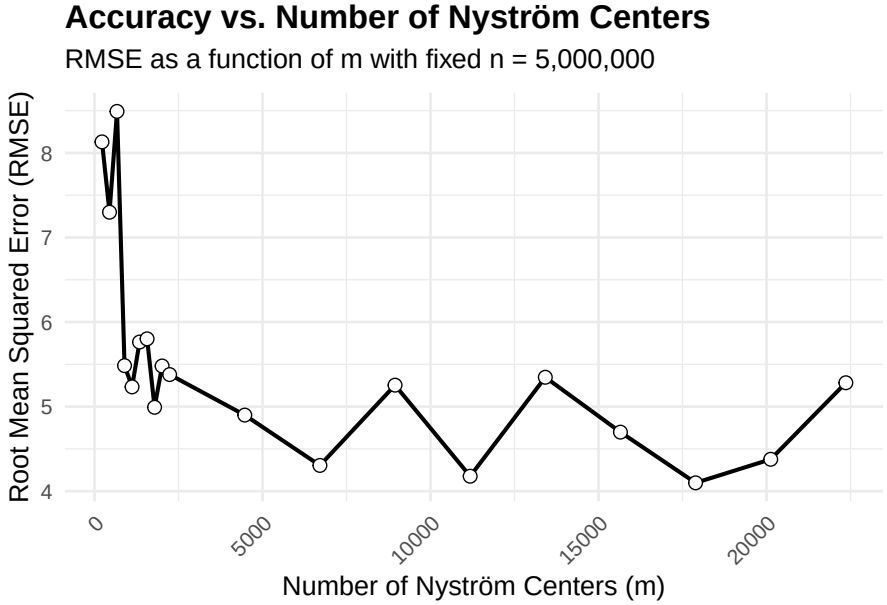


Figure 3.2: Effect of Nyström centres on model accuracy at fixed $n = 5{,}000{,}000$.

Interestingly, increasing m beyond this point does not necessarily yield further improvements. As observed in Figure 3.2, the performance effectively plateaus and fluctuates within a high-accuracy band. This behavior suggests that the model is approaching a practical limit, likely due to the onset of numerical instability. As the matrix of Nyström centres becomes very large, its Cholesky decomposition, a key step in calculating the preconditioner, can become ill-conditioned when performed with single-precision (Float32) arithmetic on the GPU. This can lead to small numerical errors that prevent the solver from converging to a marginally better solution, even with a theoretically better approximation.

This finding empirically validates that while a larger m is generally better, there is a point of diminishing returns where the theoretical benefit is offset by the practical limitations of hardware precision. It also underlines the importance of validating nuanced hardware limits and adjusting settings accordingly.

3.2.5 Discussion

The benchmarks from the NYC Taxi Fare Prediction task provide compelling evidence for the thesis's central argument. Falkon, by combining the Nyström approximation with a GPU-optimized iterative solver, effectively bridges the gap between the high predictive power of kernel methods and the computational demands of large-scale datasets.

The scalability analysis unequivocally demonstrates the practical limits of traditional and CPU-based methods, while highlighting the dramatic speedup (over $50\times$ in some cases) and superior scalability of the GPU-accelerated implementation. This enables the analysis of datasets that are orders of magnitude larger than previously feasible on comparable hardware.

Furthermore, the investigation into the number of Nyström centres (m) provides a crucial practical insight. The results confirm that m is a key hyperparameter that governs a trade-off between computational cost and predictive accuracy. While increasing m initially leads to substantial gains, the experiment reveals a performance plateau, likely due to numerical precision limits on the GPU. This finding suggests that an optimal number of centres often exists that maximizes accuracy without incurring unnecessary computational cost or risking numerical instability.

3.3 Extreme Gradient Boosting on GPU with XGBoost

This section provides an empirical analysis of XGBoost, a state-of-the-art gradient boosting library. We evaluate the performance of its GPU-accelerated `hist` tree construction method, `gpu_hist`, detailed in Section 2.3.4, on the large-scale HIGGS Boson classification dataset. This case study aims to demonstrate the practical speedups achieved by offloading the demanding tasks of gradient calculation and histogram aggregation to parallel hardware, validating the theoretical shift from a compute-bound to a memory-bandwidth-bound algorithm.

3.3.1 Introduction to the XGBoost Library

XGBoost is an open-source Python library that has become a cornerstone in modern applications of computational statistics due to its state-of-the-art performance. As detailed in the previous chapter, its sophisticated algorithms entail significant computational demands that stem from sequential tree construction, exhaustive split finding, and extensive gradient computations. This section showcases how GPU acceleration, particularly the CUDA-based implementation introduced by Mitchell & Frank (2017), offers a tangible solution to these bottlenecks.

The underlying XGBoost system is built on a robust C++ core and integrates seamlessly with Python via its Scikit-learn compatible API. For GPU acceleration, the critical tree construction algorithms are executed entirely on the GPU, thus fully leveraging NVIDIA's CUDA toolkit for memory-efficient and high-speed processing. It is designed for high performance across various datasets and settings, including those with sparse input matrices. GPU-accelerated functionality is available as a plug-in within the standard XGBoost library supporting typical features like classification, regression and ranking with minimal syntactic changes.

3.3.2 Experimental Setup

3.3.2.1 Software Configuration

The general benchmarking protocol outlined earlier has been applied here as well, in order to empirically quantify the performance advantages and practical utility of GPU acceleration for XGBoost. The experiments were conducted on a Google Colab environment using the software versions listed

below:

- **XGBoost**: 2.1.4
- **PyTorch**: 2.6.0 (with CUDA 12.4 support)
- **Scikit-learn**: 1.6.1
- **NumPy**: 2.0.2
- **Pandas**: 2.2.2

3.3.2.2 Implementations Compared

To provide a comprehensive performance picture, five distinct implementations of gradient boosting were compared. These corresponding to the split-finding algorithms analyzed in Chapter 2:

1. **XGBoost (GPU, `hist`)**: The GPU-accelerated configuration using the histogram method (`tree_method = 'hist'`, `device = 'cuda'`), which is the primary focus of this case study.
2. **XGBoost (CPU, `hist`)**: A direct multi-core CPU baseline of the same efficient histogram algorithm (`tree_method = 'hist'`, `n_jobs = -1`)
3. **XGBoost (CPU, `approx`)**: The CPU-based **Approximate Greedy Algorithm** (`tree_method = 'approx'`), which uses a weighted quantile sketch to propose candidate splits.
4. **XGBoost (CPU, `exact`)**: The original CPU-based **Exact Greedy Algorithm** (`tree_method = 'exact'`), which relies on sorting and represents the computational wall described in Section 2.3.2.
5. **Scikit-learn (CPU)**: A baseline using the standard `GradientBoostingClassifier` from `scikit-learn` for a broader comparison against other popular implementations.

3.3.2.3 Hyperparameter Configuration

To isolate the impact of the algorithm and hardware, a consistent set of hyperparameters was used for all runs. These parameters were chosen based on common practices for the HIGGS dataset to ensure a realistic and robust benchmark. Each parameter is directly linked to the theoretical framework established in Chapter 2.

- **Objective** (`objective`): `'binary:logistic'`, specifying the differentiable loss function

$L(y, \hat{y})$ to be minimized.

- **Evaluation Metric** (`eval_metric`) `'auc'`, the metric used for evaluation during training.
- **Learning Rate** (`eta`): η set to `0.05`, controlling shrinkage applied to each new tree as shown in Equation (2.31).
- **Number of Estimators** (`n_estimators`): `300` trees, defining the total size (M) of the ensemble.
- **Maximum Tree Depth** (`max_depth`): Set to `7`, limiting the complexity of each individual tree, $h_m(\mathbf{x})$.
- **Subsample Ratios** (`subsample`, `colsample_bytree`): Set to `0.7` for both row sampling and feature sampling, aiding regularization as inspired by the Random Forest methodology Breiman (2001).
- **Regularization** (`gamma`, `lambda`, `alpha`): Regularization terms corresponding to the penalty terms in the objective function $\Omega(h_m)$ (Equation (2.33)) were kept at their default values with γ set `0`, λ set to `1`, and α set to `0`.

3.3.3 Benchmark: Scalability with Sample Size (*n*)

The primary benchmark for XGBoost was designed to measure how each of the five implementations scales as the number of training samples (*n*) from the HIGGS Boson dataset increases. The dataset was subsampled at various sizes, ranging from $10,000$ to $8,000,000$ instances, to record training and prediction times.

3.3.3.1 Execution Time and Resource Consumption

The results of the scalability benchmark are presented in Table 3.4. It captures the training time, prediction time, and the Area Under the Curve (AUC) score for each configuration across the different dataset sizes.

Table 3.4: XGBoost Scalability Benchmark Results: A comparison of training time (s), prediction time (s), and key classification metrics for various Gradient Boosting implementations across varying sample sizes (n).

n	Implementation	Train Time (s)	Pred. Time (s)	AUC	Accuracy	F_1
10,000	**XGBoost (GPU, hist)**	1.50 ± 0.08	0.64 ± 0.01	0.7812	0.7079	0.7276
10,000	XGBoost (CPU, hist)	1.96 ± 0.70	5.12 ± 0.09	0.7804	0.7073	0.7269
10,000	XGBoost (CPU, approx)	4.89 ± 0.74	5.13 ± 0.14	0.7805	0.7072	0.7269
10,000	XGBoost (CPU, exact)	5.13 ± 0.81	5.10 ± 0.13	0.7803	0.7069	0.7266
10,000	Scikit-learn (CPU)	67.30 ± 0.06	47.65 ± 0.03	0.7802	0.7072	0.7265
50,000	**XGBoost (GPU, hist)**	1.88 ± 0.03	0.65 ± 0.01	0.8056	0.7269	0.7443
50,000	XGBoost (CPU, hist)	3.34 ± 0.69	5.28 ± 0.16	0.8054	0.7270	0.7444
50,000	XGBoost (CPU, approx)	13.59 ± 0.10	5.27 ± 0.11	0.8057	0.7271	0.7445
50,000	XGBoost (CPU, exact)	19.58 ± 0.68	5.19 ± 0.13	0.8058	0.7271	0.7447
50,000	Scikit-learn (CPU)	321.42 ± 0.71	45.01 ± 0.04	0.8049	0.7261	0.7432
100,000	**XGBoost (GPU, hist)**	2.12 ± 0.04	0.66 ± 0.01	0.8138	0.7336	0.7504
100,000	XGBoost (CPU, hist)	3.89 ± 0.46	5.40 ± 0.14	0.8143	0.7341	0.7511
100,000	XGBoost (CPU, approx)	22.52 ± 0.62	5.37 ± 0.10	0.8139	0.7337	0.7507
100,000	XGBoost (CPU, exact)	36.82 ± 0.29	5.26 ± 0.12	0.8131	0.7332	0.7501
100,000	Scikit-learn (CPU)	635.18 ± 1.19	44.07 ± 0.04	0.8123	0.7322	0.7492
250,000	**XGBoost (GPU, hist)**	2.53 ± 0.02	0.64 ± 0.01	0.8224	0.7409	0.7578
250,000	XGBoost (CPU, hist)	6.59 ± 0.84	5.43 ± 0.14	0.8225	0.7409	0.7579
250,000	XGBoost (CPU, approx)	49.71 ± 0.85	5.46 ± 0.16	0.8226	0.7413	0.7582
250,000	XGBoost (CPU, exact)	89.76 ± 1.18	5.28 ± 0.11	0.8211	0.7396	0.7565
500,000	**XGBoost (GPU, hist)**	3.32 ± 0.04	0.65 ± 0.01	0.8268	0.7448	0.7615
500,000	XGBoost (CPU, hist)	10.56 ± 0.83	5.52 ± 0.20	0.8268	0.7447	0.7614
500,000	XGBoost (CPU, approx)	95.76 ± 1.14	5.47 ± 0.10	0.8270	0.7448	0.7615
500,000	XGBoost (CPU, exact)	173.17 ± 0.78	5.35 ± 0.10	0.8252	0.7436	0.7603

Table 3.4: XGBoost Scalability Benchmark Results: A comparison of training time (s), prediction time (s), and key classification metrics for various Gradient Boosting implementations across varying sample sizes (n). *(continued)*

n	Implementation	Train Time (s)	Pred. Time (s)	AUC	Accuracy	F_1
1,000,000	**XGBoost (GPU, hist)**	4.35 ± 0.08	0.65 ± 0.02	0.8296	0.7471	0.7638
1,000,000	XGBoost (CPU, hist)	18.23 ± 0.73	5.46 ± 0.11	0.8297	0.7473	0.7639
2,000,000	**XGBoost (GPU, hist)**	6.17 ± 0.04	0.64 ± 0.01	0.8312	0.7487	0.7652
2,000,000	XGBoost (CPU, hist)	35.07 ± 0.56	5.55 ± 0.12	0.8316	0.7491	0.7656
5,000,000	**XGBoost (GPU, hist)**	11.61 ± 0.02	0.67 ± 0.02	0.8322	0.7498	0.7661
5,000,000	XGBoost (CPU, hist)	84.97 ± 0.72	5.43 ± 0.09	0.8324	0.7498	0.7661
8,000,000	**XGBoost (GPU, hist)**	16.76 ± 0.08	0.66 ± 0.02	0.8323	0.7498	0.7661

3.3.3.2 Analysis of Scalability and Speedup Factors

The results observed in Table 3.4 and visualized in Figure 3.3 provide a clear, multi-layered hierarchy of performance among the tested gradient boosting implementations. At the baseline, the standard Scikit-learn implementation is by far the most computationally expensive. The superiority of the XGBoost framework is immediately apparent, as even its slowest algorithm, the **Exact Greedy Algorithm** (`exact`), is significantly faster than Scikit-learn's `GradientBoostingClassifier`. At $n = 100{,}000$, the `exact` method is over $17\times$ faster (635s vs. 37s).

However, the `exact` method's reliance on sorting leads to poor scaling, and it becomes impractical for larger datasets, with training time requiring nearly 3 minutes for 500,000 samples. The **Approximate Greedy Algorithm** (`approx`) offers only a minor reprieve, proving to be roughly $1.8\times$ as fast as the `exact` method, but still struggling to scale effectively.

The most remarkable leap in performance comes from the **histogram-based algorithm** (`hist`). This method represents a fundamental algorithmic improvement, and this is underscored by a stunning observation: even on its CPU implementation, `hist` trains on 5,000,000 data points faster than the `approx` method handles just 500,000. This clearly demonstrates that the `hist` algorithm

fundamentally breaks the computational bottlenecks that plague the earlier methods.

The final tier of performance is achieved by offloading the `hist` algorithm to parallel hardware. The GPU implementation of the histogram-based algorithm, `gpu_hist`, is substantially faster than its CPU counterpart at every data point. At n = 500,000, the GPU version completes training in just 3.32 seconds compared to the CPU version's 10.56 seconds, a speedup of nearly 3.2×. This advantage widens with the dataset, growing to an 7.3 × at the 5,000,000 sample mark. The GPU-accelerated `hist` method successfully trains a model on 8,000,000 in just 17 seconds, an achievement that highlights the transformative power of combining state-of-the-art algorithms with dedicated hardware acceleration.

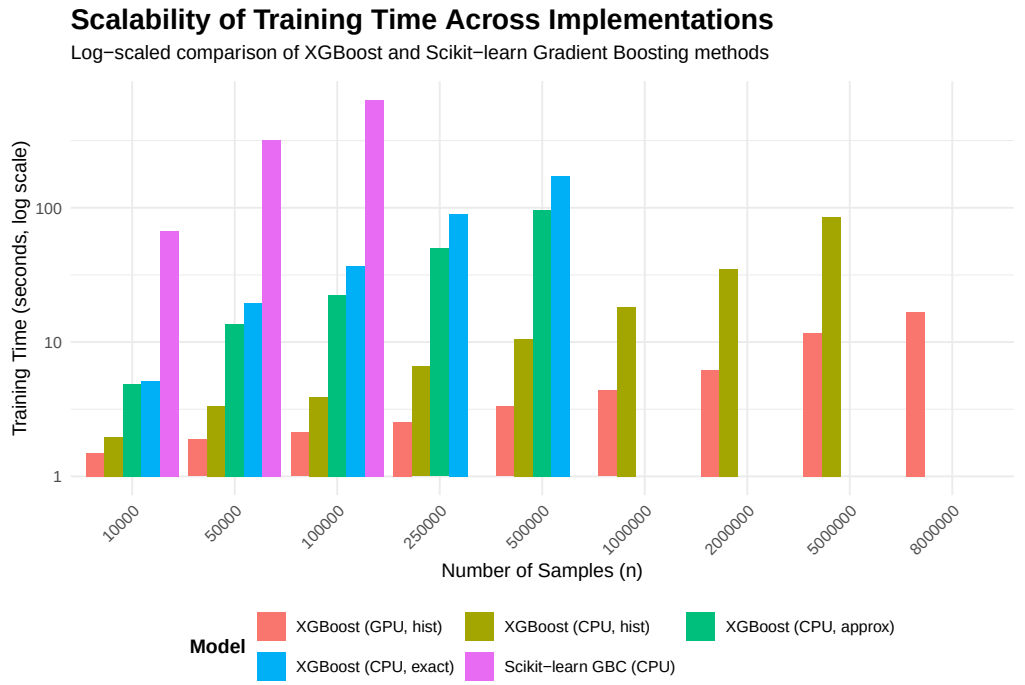


Figure 3.3: Training time scalability comparison across XGBoost and Scikit-learn implementations using a log-scaled y-axis.

3.3.3.3 Predictive Accuracy Assessment

Crucially, these dramatic performance gains are achieved with no loss in predictive accuracy. As seen in Table 3.4 and visualized in Figure 3.4, the AUC scores for all XGBoost implementations are nearly identical across all dataset sizes, confirming that the algorithmic and hardware-based accelerations provide a "free" performance boost without compromising the model's effectiveness.

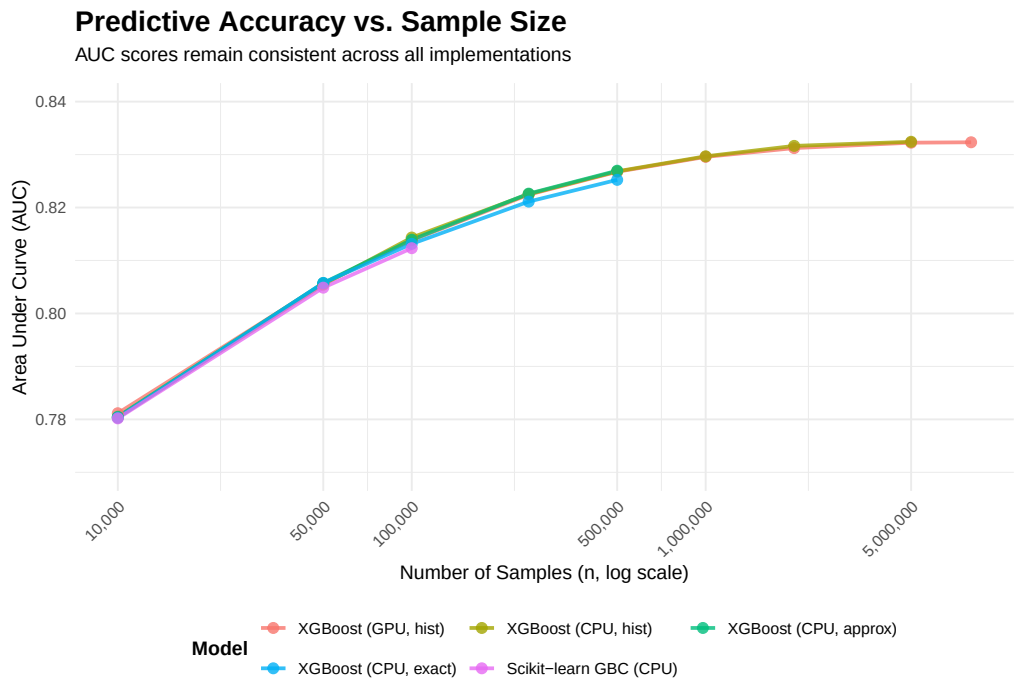


Figure 3.4: Predictive accuracy (AUC) across varying sample sizes. Note the narrow y-axis range, which confirms that all implementations achieve nearly identical performance.

3.3.4 Statistical Diagnostics and Interpretation

The preceding benchmark unequivocally demonstrated that GPU acceleration can deliver dramatic performance gains without degrading predictive accuracy. However, a final critical question for any complex statistical model is whether the resulting model is trustworthy. This section validates this by showing that the final, highly-accelerated XGBoost model is not an opaque "black box", but a powerful and interpretable tool.

3.3.4.1 Confusion Matrix

First, we move beyond a single metric like AUC to examine the model's detailed classification behaviour. The confusion matrix in Figure 3.5 provides this granular view, confirming that the model effectively distinguishes between "signal" and "background" classes, which is the fundamental goal in the HIGGS Boson Detection context.

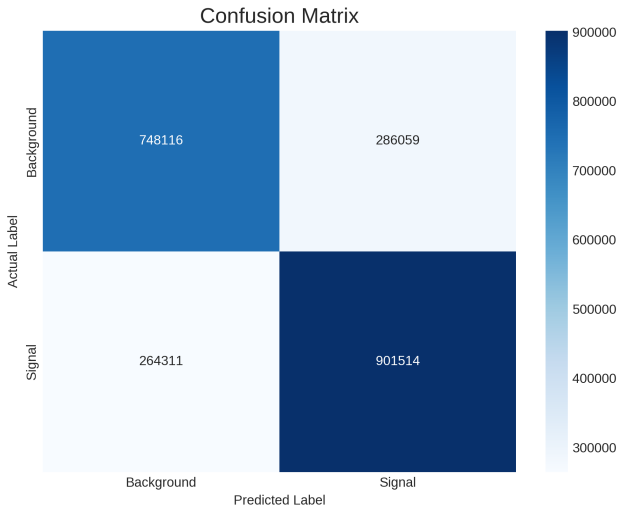


Figure 3.5: Confusion matrix for the final GPU-trained model on the HIGGS test set, detailing the specific counts of correct and incorrect classifications.

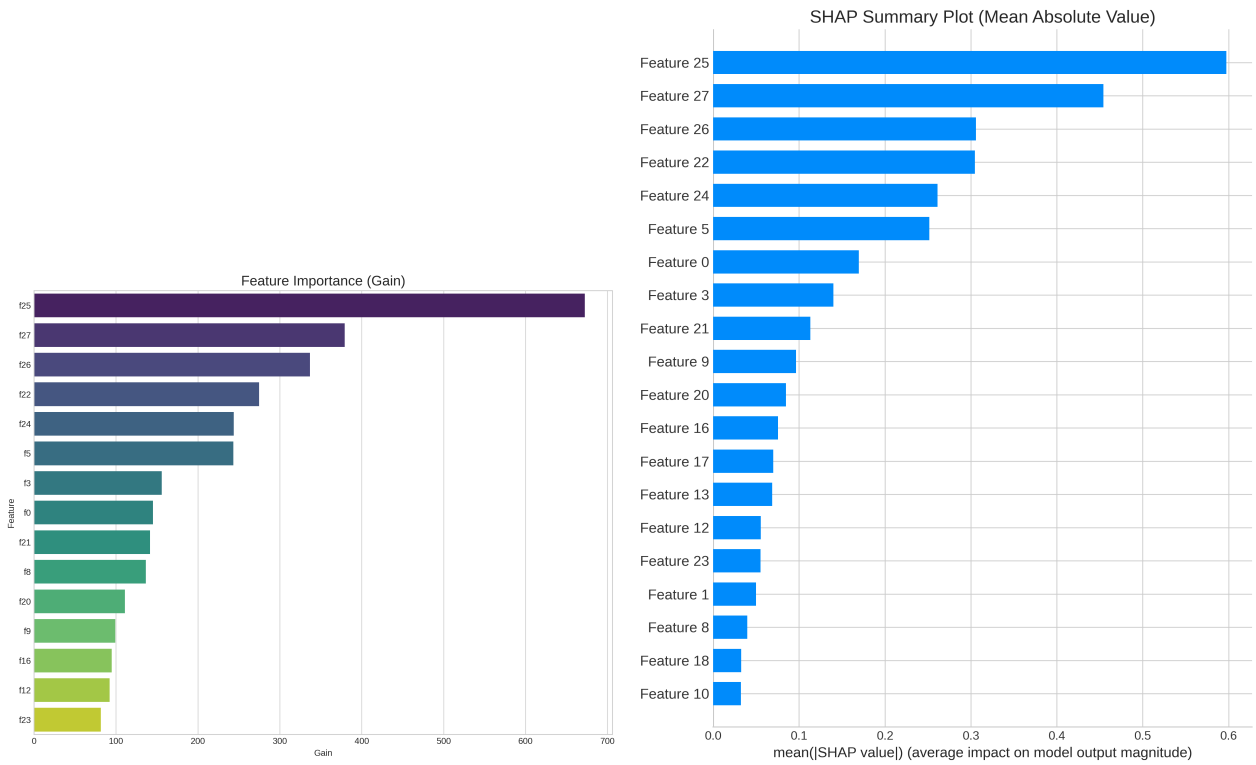
3.3.4.2 Feature Importance Analysis

To understand the model's internal logic, we can inspect which features it relies on most. Figure 3.6 shows the global feature importance, calculated by both Gain (left) and mean absolute SHAP value (right). For clarity, the top 15 features are shown.

Both methods identify feature 25 (which corresponds to `DER_mass_MMC`) as the most influential feature. This analysis demonstrates the mechanics of identifying key predictive variables. A full physical interpretation of their meaning, naturally, requires domain expertise and is beyond the scope of this analysis. The key finding, however, is that the accelerated training process produced a model with a clear and stable feature hierarchy.

3.3.4.3 Instance-Level Interpretation

Lastly, to see how the model makes individual predictions, the SHAP beeswarm plot in Figure 3.7 provides a rich, instance-level view. Each dot shows how a high (red) or low (blue) feature value pushes a single prediction towards "signal" (positive SHAP value) or "background" (negative SHAP value).



(a) Feature Importance by Gain

(b) Feature Importance by SHAP Value

Figure 3.6: Global feature importance plots, ranked by Gain (left) and mean absolute SHAP value (right).

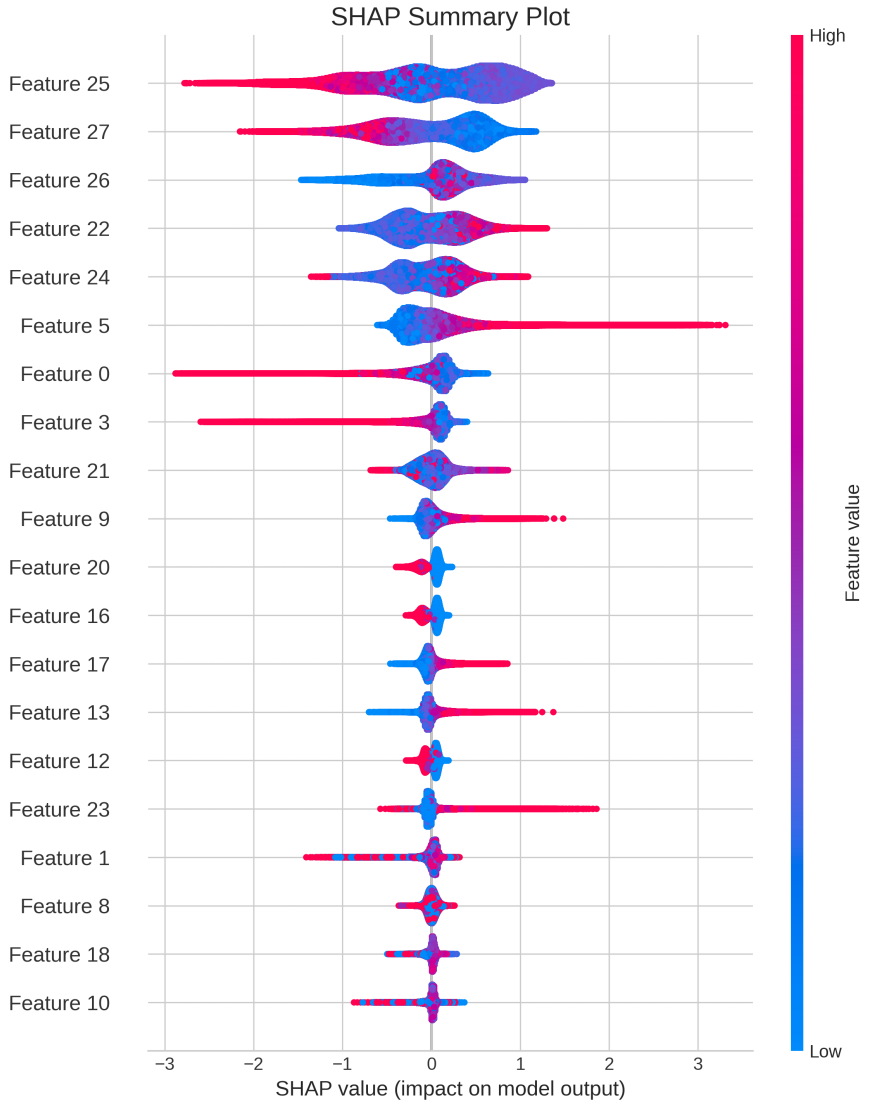


Figure 3.7: SHAP summary plot (dot), illustrating the distribution and direction of feature effects for individual predictions.

3.3.5 Discussion

The empirical results from the HIGGS Boson benchmark provide additional evidence for the central argument: that the evolution of XGBoost's algorithms, culminating in a native GPU implementation, was essential for overcoming the computational bottlenecks inherent in gradient boosting. The benchmarks clearly map out a performance hierarchy, starting from the computationally intensive traditional methods and leading to a highly scalable, GPU-accelerated solution.

The `exact` method, as theoretically predicted, quickly hits a "computational wall", becoming impractical even at moderate data sizes due to its reliance on expensive, repeated sorting operations. While an important step, the `approx` method offers only marginal improvements. The true architectural shift occurs with the `hist` algorithm. By replacing sorting with an efficient histogram aggregation, it fundamentally changes the problem's computational profile from being compute-bound, to memory-bandwidth-bound.

This algorithmic transformation is the critical enabler for hardware acceleration. The `gpu_hist` implementation leverages this new computational profile, offloading the now regular, data-parallel tasks of gradient calculation and histogram building to the thousands of cores on a GPU. The results are dramatic: the final GPU implementation demonstrates a speedup of over $8.5\times$ compared to its multi-core CPU counterpart on the largest dataset. This allows a model to be trained on 8 million in just over a minute.

The subsequent diagnostic analysis confirms that these immense performance gains do not come at the cost of model integrity. The accelerated model is not only as accurate as its slower counterparts but is also fully interpretable, allowing for a complete statistical validation of its decision-making process. Ultimately, the synergy between the histogram algorithm and parallel GPU architecture enables XGBoost to shine as one of the most powerful and agile models in large-scale computational statistics.

3.4 Discussion: Empirical Findings

The empirical benchmarks conducted on Falkon and XGBoost provide compelling, practical evidence to support the central arguments of this thesis. Across both case studies, a clear performance

ladder emerged, with traditional, exact algorithms proving to be computationally non-competitive. Their failure to scale empirically confirms their status as a "computational wall" for modern, large-scale datasets. This highlights a critical insight: algorithmic innovation is a prerequisite for speed. The most profound performance leap in XGBoost, for instance, was not from the GPU itself, but from the adoption of the `hist` algorithm. This algorithmic transformation was the key enabler, as it reshaped the computational profile from a CPU-bound sorting problem to a memory-bandwidth-bound aggregation problem, making it perfectly suited for the GPU's parallel architecture.

This synergy between an optimized algorithm and parallel hardware is what unlocks state-of-the-art performance. While CPU methods can be faster for smaller tasks before the "inflection point" where GPU overhead is overcome, the GPU implementations consistently deliver a dramatic, additional layer of acceleration on large datasets. This speedup fundamentally enhances the statistical rigor of the entire modelling process. The ability to train a model in minutes instead of hours makes computationally expensive but more robust techniques, such as k-fold cross-validation, practically feasible. It allows for a more exhaustive exploration of the hyperparameter space, leading to better-tuned models. This newfound agility facilitates a more iterative and scientific approach to modelling, where multiple feature engineering strategies and competing architectures can be rapidly tested.

Crucially, the experiments confirm that these immense performance gains do not come at the cost of model integrity. The accuracy metrics for both Falkon and XGBoost remained stable and comparable between the fast GPU versions and their slower CPU counterparts. Furthermore, the diagnostic analysis of the final XGBoost model underscores that the end result is not an opaque "black box", but a trustworthy and fully interpretable statistical model. The experiments also revealed important practical trade-offs, such as the point of diminishing returns for approximation quality in Falkon, where increasing the number of Nyström centres eventually plateaued due to the numerical precision limits of the hardware.

Ultimately, the empirical findings of this chapter are clear. GPU acceleration does more than just make existing workflows faster, but rather expands the scope of what is statistically sound and practically achievable. While the public datasets used here are essential for reproducible science, the true value of this acceleration is most profound for the unique, high-stakes datasets found in scientific and industrial research. By empowering practitioners to build more robust, better-validated,

and more deeply understood models, this technology enables more ambitious research and more reliable data-driven discovery.

Chapter 4

An Introduction to GPU Programming

4.1 Introduction: Building a Massively Parallel MCMC Sampler with CUDA C++

Chapter 2 established that rigorous theoretical and algorithmic analysis is required in order to proceed with hardware acceleration of statistical methods. Then, Chapter 3 provided an empirical validation of the real-world impact of GPU-accelerated implementations of methods such as Kernel Ridge Regression and XGBoost through state-of-the-art libraries that deliver transformative performance gains. To truly "bridge the gap" between theory and practice, however, one last crucial question must be answered: *How are these powerful, high-performance tools actually constructed?*

This final chapter addresses that question by shifting perspective from that of an analyst to that of an architect. It is designed as a pedagogical bridge, moving beyond the use of existing tools to demystify the very development process of those tools. We will pursue this examination by constructing a complete statistical application from the ground up using CUDA C++, in order to illustrate the granular, low-level control over memory and execution that is essential for developing novel or highly specialized algorithms. The pedagogical vehicle for this demonstration will be the Markov Chain Monte Carlo (MCMC) sampler, a cornerstone of modern Bayesian inference.

4.1.1 Embarrassingly Parallel Bayesian Inference

Bayesian analysis is founded on Bayes' theorem, which provides a mathematical framework for updating beliefs about parameters, θ, in light of observed data, $\mathbf{y}$. The objective is to characterize the *posterior distribution* $p(\theta|\mathbf{y})$, which is proportional to the product of the *likelihood* $p(\mathbf{y}|\theta)$ and the *prior* $p(\theta)$:

$$p(\theta|\mathbf{y}) = \frac{p(\mathbf{y}|\theta)p(\theta)}{p(\mathbf{y})} \propto p(\mathbf{y}|\theta)p(\theta) \tag{4.1}$$

For most non-trivial models, the posterior distribution is analytically intractable due to the high-dimensional integral required to compute the *marginal likelihood* (or *evidence*) $p(\mathbf{y})$. MCMC methods offer a powerful numerical solution by constructing a Markov chain whose stationary distribution is the desired posterior, $p(\theta|\mathbf{y})$, thereby allowing us to generate representative samples and approximate its properties.

The Metropolis-Hastings algorithm is a general and widely-used MCMC method that achieves this. From a current state θ_t, the algorithm iteratively proposes a new state θ' from a proposal distribution $q(\theta'|\theta_t)$. This proposed move is then accepted with a carefully constructed probability α:

$$\alpha(\theta', \theta_t) = \min\left(1, \frac{p(\theta'|\mathbf{y})q(\theta_t|\theta')}{p(\theta_t|\mathbf{y})q(\theta'|\theta_t)}\right) \tag{4.2}$$

While statistically powerful, this presents a significant computational challenge. The algorithm is inherently sequential due to its Markovial property, meaning that each step in the chain depends directly on the state of the previous one. A single chain, therefore, cannot be parallelized. The key insight, however, is that running many *independent* chains is an embarrassingly parallel problem. This structure makes MCMC an ideal candidate for GPU acceleration and a perfect case study for CUDA C++ development.

To maintain focus on the engineering challenges, we will implement a sampler for a simple, illustrative case: inferring the scalar mean μ, of a Normal distribution given N data points, $\mathbf{y} = \{y_1, \ldots, y_N\}$, with a known variance, σ^2. The statistical model is defined by:

- **Likelihood**: The data is assumed to be drawn from a Normal distribution centred at μ.

$$p(\mathbf{y}|\mu, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y_i|\mu, \sigma^2) \tag{4.3}$$

- **Prior**: Our initial belief about the mean μ is also described by a Normal distribution.

$$p(\mu) = \mathcal{N}(\mu|\mu_0, \tau_0^2) \tag{4.4}$$

Although this conjugate model has a closed-form analytical solution, its simplicity makes it an ideal vehicle for showcasing the key engineering challenges of any parallel statistical simulation. The following subsections are structured as a methodical progression, assembling the conceptual and practical building blocks required to implement our massively parallel MCMC sampler. In the conclusion, we will briefly contrast this low-level approach with high-level frameworks like **PyTorch**, which enable rapid prototyping by abstracting away hardware complexities.

4.2 CUDA Development Workflow

4.2.1 The Host-Device Model

The first and most critical concept in CUDA programming is that a program is architected for two distinct processors: the **Host** (the CPU) and the **Device** (the GPU). This Host-Device model is the fundamental paradigm upon which all CUDA applications are built. In this model, the Host acts as the orchestrator of the overall workflow, managing program logic, I/O, and sequential tasks. The Device, in contrast, serves as a massively parallel computational workforce, executing specific, data-parallel tasks that have been offloaded to it by the Host.

As detailed in Section 1.3.3, a direct consequence of this dual-processor architecture is the existence of two separate memory systems: the Host's main system memory (RAM) and the Device's onboard memory (VRAM). There is no unified memory space, and therefore the programmer is responsible for explicitly managing all data transfers between these two locations. The typical workflow involves the Host preparing data, dispatching it along with computational instructions to the Device,

and later retrieving the results.

To facilitate this interaction, CUDA extends the C++ language with several keywords and a specific syntax. The most important of these are:

- `__global__`: A function specifier that declares a function, known as a **kernel**, that runs on the Device but is called from the Host.
- `<<...>>`: An execution configuration syntax, used only on the Host, to launch a kernel on the Device. It specifies the number of parallel threads to be created for the kernel's execution.

A simple "Hello, World!" program serves as a practical demonstration of these concepts.

```
#include <stdio.h>


/*
 * Kernel: A function that runs on the device (GPU).
 * The __global__ specifier marks it as such.
 * This kernel will be executed by many threads in parallel.
 */
__global__ void hello_from_gpu()
{
  printf("Hello, World! from the GPU!\n");
}


/*
 * Host: The main function that runs on the CPU.
 * It orchestrates the program and launches kernels on the device.
 */
int main()
{
  // 1. A message from the host CPU.
  printf("Hello from the host CPU before launching the kernel.\n");
```

```
    // 2. The Host launches the kernel on the Device.
    // We launch a "grid" of 1 block, containing 4 threads.
    hello_from_gpu<<<1, 4>>>();

    // 3. Host must wait for the device to finish its work before exiting.
    // cudaDeviceSynchronize() is a barrier that pauses the host
    // until all previously launched device tasks are complete.
    cudaDeviceSynchronize();

    // 4. A final message from the host CPU.
    printf("Kernel launch finished. Hello from the host CPU again.\n");

    return 0;
}
```

When compiled with the NVIDIA CUDA Compiler (NVCC), this program's output demonstrates the core principles of the Host-Device model. The Host's `printf` statements execute sequentially and predictably. The kernel launch `hello_from_gpu<<1, 4>>()` dispatches the task to the GPU, creating four parallel threads, each of which executes the `printf` command inside the kernel, resulting in four distinct messages from the GPU. Crucially, the Host does not wait for the Device to complete its work by default. The call to `cudaDeviceSynchronize()` creates an explicit synchronization point, pausing the Host's execution until all four GPU threads have finished. Without this, the `main` function could finish and exit before the GPU had a chance to execute its task.

4.2.2 Hierarchy of Threads, Blocks, and Grids

The previous example launch four parallel threads using the `<<1, 4>>` syntax, but this simple case obscures two critical questions. First, how does this model scale to the thousands or millions of threads required for large-scale statistical problems? Second, how does each individual thread, when operating as part of this massive workforce, determine which specific piece of data it is responsible for processing?

The answer lies in CUDA's hierarchical organization of threads. When a kernel is launched, the "army" of parallel threads is organized into a three-level hierarchy:

- **Grid**: The highest level of the hierarchy. A grid encompasses all the threads generated by a single kernel launch. It can be thought of as the entire workforce assigned to a given computational task.
- **Blocks**: The grid is partitioned into a one-, two-, or three-dimensional array of thread blocks. A block is a collection of threads that can in more advanced use cases, cooperate by sharing fast on-chip memory and synchronizing their execution. For our purposes, we will simply treat a block as a convenient grouping of threads.
- **Threads**: The fundamental unit of parallel execution. Each thread runs the same kernel code independently.

To enable each thread to perform a unique task, CUDA provides built-in variables that allow a thread to identify its position within this hierarchy. For a one-dimensional organization, the most important of these are:

- `threadIdx.x`: The index of the thread within its block.
- `blockIdx.x`: The index of the block within the grid.
- `blockDim.x`: The number of threads in each block.
- `gridDim.x`: The number of blocks in each grid.

Using these variables, each thread can compute a unique global index within its grid, which maps it to a specific element in a large data array. The standard formula for this calculation is:

```
int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
```

For example, a thread with `blockIdx.x = 2` and `threadIdx.x = 5`, in a grid where blocks have `blockDim.x = 8` threads, would calculate its global index as `2 * 8 + 5 = 21`.

A parallel vector addition provides a canonical example of this principle. Adding two vectors, **A** and **B**, is an embarrassingly parallel problem, as each element-wise addition, $C_i = A_i + B_i$, is completely independent of all others. The kernel to perform this task leverages the global index to assign each thread to one such addition.

```
/*
 * Kernel to add two vectors A and B into a result vector C.
 * Assumes d_A, d_B, and d_C are pointers to arrays on the GPU.
 * N is the number of elements in the vectors.
 */
__global__ void vectorAdd(const float *d_A, const float *d_B, float *d_C,
                          int N) {

    // 1. Calculate the unique global index for this thread.
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. A crucial "bounds check". We often launch more threads than
    // needed for hardware reasons. This check ensures we do not
    // access memory outside the bounds of our arrays.
    if (idx < N) {
        // 3. Perform the addition for the element this thread is
        // responsible for.
        d_C[idx] = d_A[idx] + d_B[idx];
    }
}
```

On the host side, one must calculate the required grid dimensions to ensure that at least N threads are launched to cover the entire dataset. A common practice is to choose a block size that is a multiple of 32 (the number of threads in a *warp*, as explained in Section 1.3.2) for hardware efficiency, and then calculate the number of blocks needed.

```
// Define the size of our data
int N = 1024 * 1024; // Let's process over a million elements

// Define the number of threads per block.
int THREADS_PER_BLOCK = 256; // Multiple of 32 for hardware efficiency
```

```
// Calculate number of blocks needed in the grid to cover all N elements.
// This is a standard integer arithmetic trick to compute
// ceil(N / THREADS_PER_BLOCK).
int BLOCKS_PER_GRID = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

// Assume d_A, d_B, d_C are device pointers that have already been set up.
// Launch the kernel with our calculated grid and block dimensions.
vectorAdd<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, N);

// Synchronize to make sure the kernel finishes.
cudaDeviceSynchronize();
```

This hierarchical model allows CUDA to scale from a handful of threads to millions, while the global index mechanism provides a robust way to map this massive parallelism to large datasets. While the `vectorAdd` example illustrates the concept, it is not yet a complete, functional program, as the device pointers (`d_A`, `d_B`, `d_C`) are assumed to exist. A simple, self-contained program that demonstrates the calculation of thread indices without performing any data manipulation is included in the code repository indicated in Appendix A. Instead of `vectorAdd`, it uses a simpler `demoIndices` kernel, serving as a practical confirmation of the indexing logic discussed here.

4.2.3 Managing Memory Between Host and Device

We have designed a `vectorAdd` kernel that is, in principle, capable of adding two large vectors. However, that kernel cannot yet function because it has no data to operate on. As a consequence of the Host and the Device having their own distinct memory systems (system RAM and GPU VRAM, respectively), data does not implicitly flow between them. To make our program functional, we must learn to explicitly manage this **data pipeline**.

This process involves three fundamental steps: allocating memory on the Device, copying data from the Host to the Device, and, after computation, copying the results back from the Device to the Host. CUDA provides a specific set of functions, which mirror standard C memory management,

to control this pipeline.

- `cudaMalloc(void **devPtr, size_t size)`: The device equivalent of `malloc()`. It allocates `size` bytes of memory in the GPU's global memory. `devptr` is a pointer to a device pointer, which will hold the address of the newly allocated memory.
- `cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)`: The workhorse function for moving data. It copies `count` bytes of data from a source `src` to a destination `dst`. The `kind` parameter specifies the direction of the transfer, most commonly `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.
- `cudaFree(void *devPtr)`: The device equivalent to `free()`. It deallocates a block of GPU memory that was previously allocated with `cudaMalloc`. If not used appropriately, memory leaks may occur.

With these functions, we can now construct the complete, end-to-end `vectorAdd` program. We will examine its implementation step-by-step, following the typical lifecycle of a CUDA application.

4.2.3.1 Host-Side Setup

First, on the Host, we define the problem size and allocate standard system memory for our input vectors, `h_A` and `h_B` (prefix `h_` for host), and our result vector, `h_C`. We then initialize the input vectors.

```
// --- 1. Host-side Setup ---
int N = 1024 * 1024;
size_t size = N * sizeof(float);

// Allocate host memory for vectors A, B, and C
float *h_A = (float *)malloc(size);
float *h_B = (float *)malloc(size);
float *h_C = (float *)malloc(size);

// Initialize host vectors
for (int i = 0; i < N; ++i)
```

```
{
  h_A[i] = 1.0f;
  h_B[i] = 2.0f;
}
```

4.2.3.2 Device Memory Allocation and Data Transfer

Next, we allocate the corresponding memory buffers on the Device using `cudaMalloc`. We use the prefix `d_` to denote these device pointers. Once the device memory is allocated, we copy the input data from the host vectors to their device counterparts using `cudaMemcpy` with the `cudaMemcpyHostToDevice` kind.

```
// --- 2. Device-side Memory Allocation ---
// Declare device pointers
float *d_A, *d_B, *d_C;
// Allocate memory on the GPU for each vector
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);

// --- 3. Copy Input Data from Host to Device ---
printf("Copying input data from Host to Device...\n");
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

4.2.3.3 Kernel Execution

With the data now resident on the GPU, we can launch our `vectorAdd` kernel. We calculate the necessary grid and block dimensions and then call the kernel, passing it the device pointers. The `cudaDeviceSynchronize()` call is crucial, as it forces the host to wait until the device has completed its computation.

```
// --- 4. Launch the Kernel ---
int THREADS_PER_BLOCK = 256;
int BLOCKS_PER_GRID = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

printf("Launching vectorAdd kernel...\n");
vectorAdd<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, N);

// Block host execution until the device kernel finishes
cudaDeviceSynchronize();
printf("Kernel execution finished.\n");
```

4.2.3.4 Retrieving Results

After the kernel finishes, the result vector `d_C` exists in the GPU's memory. To use it on the CPU, we must copy it back to our host vector `h_C` using `cudaMemcpy` with the `cudaMemcpyDeviceToHost` kind.

```
// --- 5. Copy Result Data from Device to Host ---
printf("Copying result data from Device to Host...\n");
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

4.2.3.5 Verification and Cleanup

Finally, back on the host, we can verify the results to ensure the computation was correct. It is imperative to then free all allocated memory on both the Device (with `cudaFree`) and the Host (with `free`) to prevent memory leaks.

```
// --- 6. Verification (on the host) ---
bool success = true;
// Check all elements for correctness
for (int i = 0; i < N; ++i)
{
```

```
  if (h_C[i] != 3.0f)
  {
    printf("Error at index %d: Expected 3.0, got %f\n", i, h_C[i]);
    success = false;
    break;
  }
}
if (success)
{
  // Print one of the results to show it worked
  printf("Verification successful! e.g., h_C[100] = %f\n", h_C[100]);
}

// --- 7. Cleanup ---
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
// Free host memory
free(h_A);
free(h_B);
free(h_C);
```

This complete, functional program demonstrates that the separate memory space of the Host and Device necessitate an explicit data pipeline, controlled by the programmer. This workflow is the fundamental pattern underpinning the vast majority of CUDA applications and is the final prerequisite for building our MCMC sampler.

4.3 Parallel Randomness and the CUDA Library Ecosystem

The `vectorAdd` example was entirely deterministic. Given the same inputs, it will always produce the same output. However, many fundamental methods in statistics are stochastic. Simulation-based techniques such as the Bootstrap, permutation tests, and our chosen case study, MCMC, are central to modern statistical practice. To implement these on a GPU, we require a robust and statistically sound source of random numbers that is safe to use in a massively parallel environment.

This requirement reveals a critical pitfall and an important opportunity. A programmer accustomed to a CPU-only environment might be tempted to use a standard C function like `rand()` inside a CUDA kernel. This approach is fundamentally flawed and leads to incorrect results. Standard random number generators (RNG) typically rely on a single, hidden global state that is updated each time a number is generated. If thousands of GPU threads call a function simultaneously, they create a "race condition", all attempting to read and update this single state at once. This not only destroys the statistical properties of the resulting number sequences but also makes the simulation entirely non-reproducible.

The solution is to use libraries specifically designed for the parallel architecture of the GPU. The CUDA toolkit includes a rich ecosystem of high-performance libraries that provide optimized implementations of common computational primitives. These libraries, such as cuBLAS (for linear algebra), cuFFT (for Fourier Transforms), and cuDNN (for deep neural networks), are the foundational building blocks upon which many industry-standard, GPU-accelerated tools, including PyTorch, TensorFlow, and even XGBoost, are built. For our purposes, the most relevant of these is the `cuRAND` library.

`cuRAND` is designed to solve the parallel RNG problem by adhering to a simpler but powerful principle: instead of a single global generator, each thread is given its own independent RNG state. This ensures that the streams of random numbers produced by each thread are statistically independent and that there is no interference or contention between threads. The key components for using `cuRAND` are:

- `curandState_t`: A struct that holds the state of a single RNG. In practice, we allocate an array of these states on the GPU, one for each thread.

- `curand_init()`: A device-side function used within a kernel to initialize the state for each thread. It is crucial to seed each thread's generator uniquely to ensure the random number streams don't overlap. A combination of a user-provided seed and the thread's global index is a standard approach.
- `curand_uniform()` / `curand_normal()`: Device-side functions that take a pointer to a thread's state, generate a random number from the specified distribution, and update the state for the next call.

The following example demonstrates how to use `cuRAND` to generate an array of `N` random numbers, where each number is generated by a separate, independent thread. The process requires two distinct kernels: one to set up the generator states and a second to use them.

4.3.1 Initializing RNG States with a Setup Kernel

The first kernel is responsible for initializing a unique `curandState_t` for each thread in the grid.

```
/*
 * Kernel 1: Initializes the cuRAND state for each thread.
 * Each thread gets a unique state based on its ID and a seed.
 */
__global__ void setup_kernel(curandState_t *states, unsigned long seed)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Initialize the generator state for this thread
  curand_init(seed,          // The seed for the generator
              idx,           // A unique sequence number for each thread
              0,             // A 0 offset
              &states[idx]); // The address of the state to initialize
}
```

4.3.2 Generating Numbers with a Parallel Kernel

This second kernel uses the initialized states to generate the random numbers. For efficiency, each thread copies its state from slow global memory to a fast register (`localState`), performs the generation, and then copies the updated state back.

```
/*
 * Kernel 2: Uses the initialized states to generate random numbers.
 */
__global__ void generate_kernel(float *output, curandState_t *states,
                                int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        // Copy the state from global memory to a register for this thread
        curandState_t localState = states[idx];

        // Generate a random float between 0.0 and 1.0
        output[idx] = curand_uniform(&localState);

        // Copy the updated state back to global memory
        states[idx] = localState;
    }
}
```

4.3.3 Orchestrating the cuRAND Workflow

The host code manages the allocation of memory for both the output array and the `cuRAND` states, and then launches the two kernels in sequence.

```
int N = 1024;
size_t size = N * sizeof(float);
```

```
// --- 1. Host-side Setup ---
float *h_output = (float *)malloc(size);

// --- 2. Device-side Memory Allocation ---
float *d_output;
curandState_t *d_states;
cudaMalloc(&d_output, size);
cudaMalloc(&d_states, N * sizeof(curandState_t));

// --- 3. Kernel Configuration ---
int THREADS_PER_BLOCK = 256;
int BLOCKS_PER_GRID = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

// --- 4. Launch Setup Kernel ---
// Use the current time as a seed for the random number generator
setup_kernel<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(d_states, time(NULL));

// --- 5. Launch Generation Kernel ---
generate_kernel<<<BLOCKS_PER_GRID,
                  THREADS_PER_BLOCK>>>(d_output, d_states, N);
cudaDeviceSynchronize();

// --- 6. Copy results back to host ---
cudaMemcpy(h_output, d_output, size, cudaMemcpyDeviceToHost);

// --- 7. Verification ---
printf("Generated random numbers (first 10):\n");
for (int i = 0; i < 10; ++i)
{
```

```
  printf("h_output[%d] = %f\n", i, h_output[i]);
}

// --- 8. Cleanup ---
cudaFree(d_states);
cudaFree(d_output);
free(h_output);

return 0;
```

This example illustrates the necessity of using specialized, hardware-aware libraries like `cuRAND` to perform even fundamental statistical operations in a parallel environment. The ability to generate statistically valid random numbers for each thread independently is the final conceptual tool required before we can assemble our complete MCMC sampler.

4.4 Assembling a Massively Parallel MCMC Sampler

With the foundational concepts of the CUDA programming model established, we are now ready to assemble them to solve our statistical problem. This section deconstructs a complete, end-to-end MCMC sampler that leverages the GPU to run thousands of independent Markov chains in parallel. This approach transforms a traditionally time-consuming, sequential task into a high-throughput computation.

The final program consists of three key pieces of device code: a helper function to compute the log-posterior, a kernel to initialize the random number generators, and the main MCMC kernel. Orchestration occurs within the Host-side `main` function.

4.4.1 MCMC Device Code: Kernels and Helpers

The core of the application resides in the code executed on the GPU. This is where the statistical computation happens. The Device-side logic is in the kernels below, in addition to the `cuRAND` setup kernel presented in the previous section.

4.4.1.1 The `log_posterior` Device Function

To keep our main kernel clean and modular, we first define a `__device__` function. Unlike a `__global__` kernel, a `__device__` function cannot be called from the host, but only by other kernels or function on the device. It serves as a reusable helper function. This one calculates the log-posterior for a given value of μ by summing the log-prior and the log-likelihood

```
// A __device__ function can only be called from the device (e.g., from a
// kernel).
__device__ float log_posterior(float mu, const float *d_data, int N,
                               float sigma2, float mu0, float tau2_0)
{
  // 1. Calculate log prior: log( N(mu | mu0, tau2_0) )
  float log_prior = -0.5f * powf(mu - mu0, 2) / tau2_0;

  // 2. Calculate log likelihood: log( N(data | mu, sigma2) )
  float log_likelihood = 0.0f;
  for (int i = 0; i < N; ++i)
  {
    log_likelihood += -0.5f * powf(d_data[i] - mu, 2) / sigma2;
  }

  return log_prior + log_likelihood;
}
```

4.4.1.2 The Main `mcmc_kernel`

This is the workhorse of our application. A single launch of this kernel executes the *entire* MCMC simulation. Each thread is assigned to a single chain and runs a `for` loop for the required number of iterations. This design is highly efficient as it minimizes the overhead of launching many small kernels from the host.

Inside the kernel, each thread:

1. Identifies its unique chain using the `global_idx` pattern.
2. Initializes its own `curandState` and starting parameter `current_mu`.
3. Enters the main MCMC loop. In each iteration, it proposes a new `mu`, computes the log-posteriors for the current and proposed values by calling our `log_posterior` helper, and performs the Metropolis-Hastings acceptance check.
4. After all iterations are complete, it stores the final sample of its chain in the global output array.

```
// --- The MCMC Kernel ---
__global__ void mcmc_kernel(float *d_output, const float *d_data,
                            curandState_t *states, int N_data, int N_chains,
                            int N_iters, int N_burn_in, float sigma2,
                            float mu0, float tau2_0, float prop_sigma)
{

  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < N_chains)
  {
    curandState_t local_rand_state = states[idx];
    float current_mu = mu0; // Start each chain at the prior mean

    // Main MCMC loop
    for (int i = 0; i < N_iters + N_burn_in; ++i)
    {
      // Propose a new mu using the Normal distribution from cuRAND
      float proposed_mu = current_mu +
        curand_normal(&local_rand_state) * prop_sigma;

      // Calculate log posterior for current and proposed mu
      float log_post_current = log_posterior(current_mu, d_data, N_data,
```

```
                                                sigma2, mu0, tau2_0);
      float log_post_proposed = log_posterior(proposed_mu, d_data, N_data,
                                                sigma2, mu0, tau2_0);

      // Acceptance check in log-space
      float log_alpha = fminf(0.0f, log_post_proposed - log_post_current);
      if (logf(curand_uniform(&local_rand_state)) < log_alpha)
      {
        current_mu = proposed_mu;
      }
    }

    // After burn-in and iterations, store the final sample of the chain
    d_output[idx] = current_mu;
    states[idx] = local_rand_state; // Save the updated random state
  }
}
```

4.4.2 Host Code: Orchestrating the Sampler

The host is responsible for setting up the problem, managing the GPU, and analysing the final results.

4.4.2.1 Problem Setup and Host Data Management

First, we define all the parameters for our simulation and generate synthetic data on the host. This data will serve as the "observed" data for our model.

```
// --- 1. Problem Setup ---
int N_data = 1000;          // Number of data points
int N_chains = 1024 * 16; // Number of parallel MCMC chains to run (16,384)
int N_iters = 2000;         // MCMC iterations per chain
```

```
int N_burn_in = 500;       // Burn-in iterations to discard

// True parameters for data generation
float true_mu = 10.0f;
float sigma2 = 4.0f;

// Priors for mu ~ N(mu0, tau2_0)
float mu0 = 0.0f;
float tau2_0 = 100.0f;

// MCMC proposal width
float prop_sigma = 1.0f;

// --- 2. Host-side Data Generation and Memory Allocation ---
float *h_data = (float *)malloc(N_data * sizeof(float));
float *h_output = (float *)malloc(N_chains * sizeof(float));

// Generate synthetic data from the true model
srand(time(NULL));
for (int i = 0; i < N_data; ++i)
{
  // A simple way to get a standard normal-like random number
  float u1 = rand() / (float)RAND_MAX;
  float u2 = rand() / (float)RAND_MAX;
  float rand_std_normal = sqrtf(-2.0f * logf(u1)) * cosf(2.0f * M_PI * u2);
  h_data[i] = true_mu + sqrtf(sigma2) * rand_std_normal;
}
```

4.4.2.2 Device Memory Management and Kernel Launch

Next, we follow the data pipeline pattern: 1. allocate memory on the device with `cudaMalloc`, 2. copy the host data to the device with `cudaMemcpy`, and 3. set up the `cuRAND` states by launching the `setup_kernel`. Finally, we launch our main `mcmc_kernel`. Note that we only launch it **once**.

```
// --- 3. Device-side Memory Allocation ---
float *d_data, *d_output;
curandState_t *d_states;
cudaMalloc(&d_data, N_data * sizeof(float));
cudaMalloc(&d_output, N_chains * sizeof(float));
cudaMalloc(&d_states, N_chains * sizeof(curandState_t));

// --- 4. Copy data and Setup Random States ---
cudaMemcpy(d_data, h_data, N_data * sizeof(float), cudaMemcpyHostToDevice);

int THREADS_PER_BLOCK = 256;
int BLOCKS_PER_GRID = (N_chains + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

setup_kernel<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(d_states, time(NULL));

// --- 5. Launch the MCMC Kernel ---
printf("Launching %d parallel MCMC chains...\n", N_chains);
mcmc_kernel<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(
    d_output, d_data, d_states, N_data, N_chains, N_iters, N_burn_in,
    sigma2, mu0, tau2_0, prop_sigma);
cudaDeviceSynchronize();
printf("MCMC simulation finished.\n");
```

4.4.2.3 Retrieving Results and Cleanup

After the simulation is complete, we copy the final samples from each chain back to the host and perform a simple analysis by calculating the posterior mean. Finally, we free all allocated memory on both the device and the host.

```
// --- 6. Copy results back and analyze ---
cudaMemcpy(h_output, d_output, N_chains * sizeof(float),
           cudaMemcpyDeviceToHost);

float posterior_mean = 0.0f;
for (int i = 0; i < N_chains; ++i)
{
  posterior_mean += h_output[i];
}
posterior_mean /= N_chains;

printf("\n--- Analysis ---\n");
printf("Posterior Mean of mu (from %d samples): %f\n", N_chains,
       posterior_mean);
printf("True Mean of mu was: %f\n", true_mu);

// --- 7. Cleanup ---
cudaFree(d_data);
cudaFree(d_output);
cudaFree(d_states);
free(h_data);
free(h_output);
```

This complete example demonstrates how the various components of the CUDA programming model are integrated to solve a non-trivial statistical problem. By reframing the task from running one long chain to running thousands of shorter ones, we can leverage the massive parallelism

of the GPU to dramatically reduce the time required for Bayesian inference.

4.5 Discussion: The GPU Computing Stack from CUDA to PyTorch

This chapter navigates the intricate process of building a high-performance statistical application from first principles. By assembling an MCMC sampler in CUDA C++, we engaged directly with the GPU, manually managing memory, organizing thousands of threads, and ensuring the statistical validity of parallel random number generation. This detailed construction provides a necessary perspective on how modern high-performance tools are truly built.

The high performance of GPU implementations seen in many modern libraries is often achieved through this exact kind of custom-written, low-level code. The highly-optimized XGBoost library, for instance, relies on a suite of CUDA kernels to execute its `gpu_hist` method efficiently. The development of such novel and specialized tools required a direct engagement with the underlying hardware, and **CUDA** provides the granular control over memory and its thousands of processing cores that is essential for this work.

This foundational layer of low-level engineering enables a second, complementary paradigm: high-level abstraction. Frameworks like **PyTorch** build upon these powerful, low-level primitives to offer a highly productive and accessible programming environment. They abstract away the immense complexity of hardware management, allowing developers to focus on statistical and algorithmic logic. The simplicity of the high-level API is a direct result of the complexity handled by the underlying, often CUDA-based kernels.

To illustrate this layered structure, consider again the calculation of the log-posterior. The explicit loops and manual indexing in our CUDA C++ kernel are replaced by a single, expressive line in PyTorch.

```
import torch


# Assume 'd_data' is our dataset, now as a PyTorch tensor
```

```
# and we are evaluating a batch of 'proposed_mu' values in parallel.
N_data = 1000
N_chains = 16384
sigma2 = 4.0
mu0 = 0.0
tau2_0 = 100.0

# Ensure tensors are on the GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
d_data = torch.randn(N_data, device=device) * 2 + 10 # Example data
proposed_mu = torch.randn(N_chains, device=device)    # Example proposals

def log_posterior_pytorch(mu, data):
    # Log-prior: N(mu | mu0, tau2_0)
    log_prior = -0.5 * (mu - mu0)**2 / tau2_0

    # Log-likelihood: N(data | mu, sigma2)
    # The sum is broadcast automatically across all chains
    log_likelihood = -0.5*torch.sum((data.unsqueeze(0)-mu.unsqueeze(1))**2,\
                                    dim=1) / sigma2

    return log_prior + log_likelihood

# This single call triggers pre-compiled, highly-optimized CUDA kernels
log_p = log_posterior_pytorch(proposed_mu, d_data)

print(f"Computed log-posterior for {log_p.shape[0]} chains\
      on device: {log_p.device}")
```

When this Python code is executed, PyTorch calls its own optimized, pre-compiled kernels, them-

selves written in a low-level language like CUDA C++, to perform the computation on the GPU. The low-level control of the engineer enables the rapid abstraction of the statistician and the data scientist.

Therefore, the spectrum of GPU development is not a matter of choosing one tool over another, but of understanding how they fit together. The ecosystem thrives on both: developers who write low-level CUDA code to build the foundational engines of computation, and statisticians and scientists who use high-level tools to rapidly build, test, and deploy models. By offering this glimpse into the creation process, we complete the narrative that has taken us from theoretical rationale and empirical proof to the enabling technology itself. We are thus equipped with a holistic view of high-performance statistical computing.

Chapter 5

Conclusion and Future Directions

5.1 Summary of Key Findings

The quest for computational power is not external to statistics but is a challenge that has defined it from its very beginning. The central finding of this work is that effective hardware acceleration is not a matter of simply porting existing solutions, but rather a process of deliberate algorithmic co-design. As demonstrated through the case studies of Falkon and XGBoost, the most profound performance gains are unlocked only after fundamentally reshaping statistical algorithms to align with the architectural strengths of the GPU.

By replacing direct matrix inversions with Nyström-based iterative solvers and exhaustive sorting with histogram aggregations, these methods were transformed into regular, data-parallel tasks perfectly suited for the GPU's high-throughput design. Low-level optimizations like deep memory hierarchy awareness and strategic use of mixed-precision arithmetic further unlocked the full performance potential of the hardware.

The empirical benchmarks confirmed the success of this approach, revealing orders-of-magnitude speedups that were achieved without sacrificing statistical accuracy. This work empirically validates that a hardware-aware perspective is essential for breaking through the "computational wall" that has historically limited the scale of statistical modelling.

5.2 Implications for Statistical Practice

The architectural characteristics of GPUs, specifically their massive parallelism, high memory bandwidth, and hierarchical memory system, render them exceptionally well-suited for accelerating a wide range of computationally intensive statistical methods. Any tasks that exhibit substantial data parallelism, where the same set of operations can be performed independently across numerous data elements, is a prime candidate for GPU acceleration.

For statistics, this opens up new possibilities for many foundational and advanced techniques. This includes:

- **Large-scale linear algebra operations**, such as matrix multiplication and vector operations, which are fundamental to a vast number of statistical models.
- **Resampling techniques** like bootstrapping and permutation tests, which can be executed concurrently by running thousands of independent analyses in parallel, as demonstrated with the MCMC sampler in Chapter 4.
- **Likelihood calculations** in complex models with large datasets can be parallelized by computing contributions from individual data points simultaneously.
- **Distance computations**, which are prevalent in methods such as k-nearest neighbors and Kernel Methods, are inherently data-parallel.
- **Ensemble learning algorithms**, such as Random Forests and Gradient Boosting Machines, can often be accelerated by building or evaluating their many individual models in parallel.

By effectively mapping the parallelizable components of these algorithms onto the GPU architecture, it is possible to achieve significant reductions in computation time. This acceleration not only enables more complex analysis on larger datasets but also facilitates faster research iterations and more thorough model exploration.

5.3 Directions of Future Research

The work presented here opens several promising avenues for future research. While this thesis focused primarily on single-GPU implementations, a natural next step is to explore **distributed, multi-GPU training** to tackle problems that exceed the memory capacity of a single card. Most importantly, there is a significant opportunity to develop highly-optimized parallel libraries for advanced statistical domains where GPU acceleration is still underexplored.

The author of this work intends to continue research in this field by focusing on creating such libraries for specialized methods. Promising areas include **complex non-linear optimization** and **advanced Bayesian models**, such as hierarchical, state-space, or mixed-effects models, building upon the parallel sampling concepts introduced in Chapter 4. The goal is to leverage GPU power in areas of statistics that have not yet fully benefited from it, further closing the gap between cutting-edge theory and practical, scalable application.

Appendix A

Thesis Repository and Code

All the code, data, and supplementary materials for this thesis are publicly available in a GitHub repository, which can be accessed at the following URL:

https://github.com/andbamp/stats-gpu

This repository is designed to ensure the full reproducibility of all the results and analyses presented in this work. It is organized as follows:

- **`/` (root)**: Contains the `bookdown` source files (`.Rmd`) for each chapter of the thesis, along with configuration files and the bibliography.
- **`docs/`**: The rendered thesis document in PDF format.
- **`data/`**: Data used for the benchmarks in Chapter 3.
- **`notebooks/`**: Jupyter notebooks used to run the benchmarks for Falkon and XGBoost.
- **`py_src/`**: Python code used by the benchmark notebooks.
- **`src/`**: CUDA C++ source code for the MCMC sampler developed in Chapter 4.
- **`img/`**: Images and figures used in the thesis.

The `README.md` file in the repository root provides detailed instructions for setting up the environment and reproducing the thesis document and all empirical results. Readers are strongly encouraged to explore the repository's materials to gain a more profound, hands-on understanding of the algorithms and implementations that are discussed throughout this thesis.

Appendix B

Bibliography

Baldi, P., Sadowski, P. & Whiteson, D. (2014), 'Searching for exotic particles in high-energy physics with deep learning', *Nature communications* **5**(1), 4308.

Bampouris, A. S. (2018), Development of a software library for machine learning applications on parallel computers, Technical report, Patras, Greece. Diploma Thesis.

Breiman, L. (2001), 'Random forests', *Machine Learning* **45**(1), 5–32.

Breiman, L., Friedman, J. H., Olshen, R. A. & Stone, C. J. (1984), *Classification and Regression Trees*, Wadsworth & Brooks/Cole Advanced Books & Software.

Chen, T. & Guestrin, C. (2016), 'XGBoost: A scalable tree boosting system', *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* pp. 785–794.
URL: *https://doi.org/10.1145/2939672.2939785*

Cover, T. M. (1965), 'Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition', *IEEE Transactions on Electronic Computers* **EC-14**(3), 326–334.

Dua, D. & Graff, C. (2017), 'UCI machine learning repository'.
URL: *http://archive.ics.uci.edu/ml*

Freund, Y. & Schapire, R. E. (1997), 'A decision-theoretic generalization of on-line learning and an application to boosting', *Journal of Computer and System Sciences* **55**(1), 119–139.

Friedman, J. H. (2001), 'Greedy function approximation: A gradient boosting machine', *Annals of Statistics* **29**(5), 1189–1232.

Hastie, T., Tibshirani, R. & Friedman, J. (2009), *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics, 2 edn, Springer, New York, NY.
URL: *https://doi.org/10.1007/978-0-387-84858-7*

Meanti, G., Carratino, L., Rosasco, L. & Rudi, A. (2020), 'Kernel methods through the roof: handling billions of points efficiently'.
URL: *https://arxiv.org/abs/2006.10350*

Mitchell, R. & Frank, E. (2017), 'Accelerating the XGBoost algorithm using GPU computing', *PeerJ Computer Science* **3**, e127.
URL: *https://doi.org/10.7717/peerj-cs.127*

New York (N.Y.). Taxi and Limousine Commission (2019), 'New York City Taxi Trip Data, 2009-2018'.
URL: *https://doi.org/10.3886/ICPSR37254.v1*

Platt, J. C. (1998), Sequential minimal optimization: A fast algorithm for training support vector machines, Technical Report MSR-TR-98-14, Microsoft Research.
URL: *https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/*

Rudi, A., Camoriano, R. & Rosasco, L. (2015), Less is more: Nyström computational regularization, *in* 'Advances in Neural Information Processing Systems 28', pp. 1656–1664.

Rudi, A., Carratino, L. & Rosasco, L. (2018), 'Falkon: An optimal large scale kernel method'.
URL: *https://arxiv.org/abs/1705.10958*

Schölkopf, B., Herbrich, R. & Smola, A. J. (2001), A generalized representer theorem, *in* 'Computational Learning Theory', Vol. 2111 of *Lecture Notes in Computer Science*, Springer, pp. 416–426.

Zhang, X. S., Roy, M. K., Stefanidis, K., Jaleel, A., Dong, S. & Owens, J. D. (2015), 'Demystifying gpu microarchitecture through microbenchmarking', *arXiv preprint arXiv:1509.02308* .