國立成功大學資訊工程學系

碩士論文

WebGPU 於高效能生物資訊計算：瀏覽器端 Pair-HMM Forward 演算法之優化

WebGPU for High-Performance Bioinformatics Computing: Browser-Based Optimization of the Pair-HMM Forward Algorithm

研究生：王尊緯　　Student: Tsun-wai Wang

指導老師：賀保羅　　Advisor: Paul Horton

National Cheng Kung University,
Tainan, Taiwan, R.O.C.

Thesis for Master of Science Degree
July, 2025

中華民國１１４年７月

WebGPU 於高效能生物資訊計算：瀏覽器端 Pair-HMM Forward 演算法之優化

王尊緯* 賀保羅†

國立成功大學資訊工程學系

摘要

隨著 GPU 加速在生物資訊領域日漸普及，但傳統 CUDA／OpenCL 解決方案必須安裝驅動並受限於特定硬體，對線上教學與臨床前端分析造成不便。2024 年正式標準化的 WebGPU 透過單一 JavaScript API 對接 Vulkan／D3D12／Metal，兼具「免安裝、跨硬體、資料留在本機」三大優勢。本研究以高強度 Pair-Hidden Markov Model Forward (Pair-HMM Forward) 演算法為例，評估 WebGPU 的效能與可行性。

我們以周育晨（2024）公開之 C++／CUDA 程式為基準，首先撰寫 WebGPU Baseline，接著針對 CPU↔GPU 往返、BindGroup 重建與全域記憶體延遲等瓶頸，依序導入「單一 CommandBuffer 批次提交」「Dynamic Uniform Offset」及「Workgroup Cache」，形成 WebGPU-Optimized。在 NVIDIA RTX 2070 Super、Apple M1 與 Intel UHD 620 測試長度 10^2–10^5 的序列後，Optimized 相較 Baseline 呈現顯著加速（最高逾百倍），執行速度可達 CUDA 的八成以上；三款裝置的 Log-Likelihood 相對誤差均低於 10^{-5}，且在無 NVIDIA GPU 時對單執行緒 C++ 亦提

*學生

†指導教授

供多達數十至數百倍的加速。

本研究證實僅憑 JavaScript + WGSL，即能於瀏覽器中於秒級完成 Pair-HMM Forward 計算，並提出三項瀏覽器端專屬優化策略及跨硬體實測結果。此成果為 Web-native 基因體分析工具的普及奠定基礎，推動生物資訊運算的民主化與即時化。

關鍵詞: 瀏覽器 GPU 計算、Pair-Hidden Markov Model (Pair-HMM)、生物資訊加速

WebGPU for High-Performance Bioinformatics Computing: Browser-Based Optimization of the Pair-HMM Forward Algorithm

Tsun-wai Wang* Paul Horton†

Department of Computer Science and Information Engineering,
National Cheng Kung University

Abstract

As GPU acceleration becomes increasingly prevalent in bioinformatics, conventional CUDA/OpenCL solutions still require driver installation and are locked to specific hardware, hindering online teaching and front-end clinical analysis. Ratified in 2024, **WebGPU** unifies Vulkan, Direct3D 12, and Metal behind a single JavaScript API and offers three decisive advantages: zero installation, cross-hardware portability, and on-device data residency. Using the compute-intensive **Pair-Hidden Markov Model Forward (Pair-HMM Forward)** algorithm as a case study, we assess the performance and viability of WebGPU.

Starting from **Chou, Yu-Chen's (2024)** open-source C++/CUDA implementation, we first develop a WebGPU Baseline. We then tackle its primary bottlenecks—frequent CPU↔GPU round-trips, costly BindGroup reconstruction, and high global-memory latency—by successively introducing (i) single-CommandBuffer batch submission, (ii) Dynamic Uniform Offsets, and (iii) a Workgroup Cache, yielding **WebGPU-Optimized**. Across an NVIDIA RTX

*Student
†Advisor

2070 Super, Apple M1, and Intel UHD 620, and for sequence lengths from 10^2 to 10^5, the optimized version delivers **substantial speed-ups (over 100$\times$ in the best case)** and attains **more than 80 % of CUDA's throughput**. The relative Log-Likelihood error remains below 10^{-5} on all devices, and in the absence of an NVIDIA GPU our method still accelerates single-threaded C++ by one to two orders of magnitude.

These results demonstrate that pure JavaScript plus WGSL can compute Pair-HMM Forward within seconds in a browser. We contribute three browser-specific optimization strategies and detailed cross-hardware measurements, laying the groundwork for Web-native genomic analysis tools and promoting the democratization and real-time execution of bioinformatics workloads.

Keywords: WebGPU, Pair-Hidden Markov Model, Bioinformatics Acceleration

誌謝

在此特別感謝賀保羅教授以及在實驗室給予我協助與指導的同學：阮祈翰、楊祐昇、黃書堯、鄭驊軒、鄭煜醴等，在各階段實驗設計與資料收集上提供許多寶貴意見。

同時感謝林宜靜、賴威達、王晴文、許庸袁及陳竑曄，你們在討論分析與程式測試時的協作與支持，讓本研究得以順利完成。

最後，感謝所有關心及協助本研究的師長與同窗，謹此致上由衷謝意。

CONTENTS

List of Tables

List of Figures

Nomenclature

General

LL	Log-Likelihood
DP	Dynamic Programming
NGS	Next-Generation Sequencing

$D_{i,j}$	Delete state DP value at (i, j)
$I_{i,j}$	Insert state DP value at (i, j)
M	Reference length
$M_{i,j}$	Match state DP value at (i, j)
N	Read length
Pair-HMM	Pair-Hidden Markov Model
Wavefront	Anti-Diagonal Processing Order

BindGroup	Resource binding group in WebGPU
Dynamic Uniform Offset	Per-dispatch uniform buffering offset
RTX 2070 S	NVIDIA RTX 2070 Super GPU

SFU	Special Function Unit
WebGPU	Web Graphics Processing Unit API
WGSL	WebGPU Shading Language
Workgroup Cache	Shared memory cache within a workgroup

Chapter 1

Introduction

1.1 Background

High-throughput sequencing (Next-Generation Sequencing, NGS) has pushed the scale and complexity of genomic data to grow exponentially, creating unprecedented demands on computational performance in bioinformatics. The **Pair Hidden Markov Model (Pair-HMM) Forward algorithm**, which simultaneously supports sequence alignment, genotype calling, and variant detection, is a core computation in many genomic pipelines.

Current analysis workflows—such as GATK, Samtools, and BWA-MEM2—are mostly written in C++ or Python and gain GPU acceleration through NVIDIA CUDA or OpenCL. Besides installing drivers, SDKs, and dependencies, users are tied to specific GPU architectures. In classrooms or resource-constrained labs, the lack of high-end GPUs or sufficient cloud quotas often forces a CPU fallback, dramatically inflating cost and turnaround time. Cloud services reduce local setup complexity but introduce account management, network latency, and concerns over sensitive-data exposure.

Since WebGPU landed in Chrome’ s stable channel in May 2024, Firefox Nightly and Edge Dev have followed with experimental support. By mapping Vulkan, Direct3D 12, and Metal to a single JavaScript API inside the browser sandbox, WebGPU enables real-time parallel computation on NVIDIA, AMD,

Intel, and Apple Silicon GPUs **without driver installation**. It therefore offers a new opportunity to lower the barrier to bioinformatics tools, especially in teaching, clinical front-ends, and low-resource settings.

1.2 Motivation and Objectives

Although WebGPU promises zero installation, cross-hardware portability, and on-device data residency, it was designed chiefly for graphics and ML inference. A compute-intensive dynamic-programming algorithm such as Pair-HMM Forward encounters several challenges:

- **API scheduling overhead** – Wavefronts must be processed sequentially; if each dispatch creates fresh CommandBuffers and BindGroups, CPU↔GPU round-trips accumulate rapidly.
- **Lack of global synchronization** – Every wavefront depends on the previous one; WebGPU exposes only workgroup-level barriers and lacks CUDA-style kernel-return global sync.
- **No dedicated special-function units (SFUs)** – Pair-HMM issues many `log`/`exp` calls. CUDA's SFUs finish `log` in four cycles, whereas WebGPU must approximate via ALU + LUT + FMA, adding a 2–4× latency penalty.
- **High memory-access demand** – The DP matrix resides in a read-write storage buffer; WebGPU's default path bypasses L1, so frequent global reads and writes incur heavy DRAM traffic.

These factors magnify WebGPU's still-maturing API and hardware limits, and no systematic study yet verifies its suitability for high-intensity bioinformatics workloads. We therefore ask: **Can WebGPU inside a browser execute Pair-HMM Forward with sufficient performance to serve as a practical alternative to CUDA?**

1.3 Methods and Key Results

Using **Chou, Yu-Chen's (2024)** open-source C++/CUDA program as the reference, we introduce three WebGPU-specific optimizations to address the above bottlenecks:

1. **Single-CommandBuffer batch submission** – Aggregate multiple wavefronts into one `queue.submit()`, eliminating extensive API round-trip latency.

2. **Dynamic Uniform Offsets** – Store static parameters in a single uniform buffer and access them via dynamic offsets, removing the need to allocate and bind multiple UBOs.

3. **Workgroup Cache** – Explicitly copy emission and transition constants into `var<workgroup>` to reduce repeated global-memory reads across wavefronts.

On an NVIDIA RTX 2070 Super with sequence lengths 100–100,000, **WebGPU-Optimized** accelerates the Baseline by 6.8 – 142× and reaches 11 – 84% of CUDA's throughput. On Apple M1 and Intel UHD 620, it delivers 4–463× speed-ups over CPU execution for sequences $\geq 1,000$, while keeping **Log-Likelihood errors below** 10^{-5}.

1.4 Conclusions and Contributions

This study shows that JavaScript plus WGSL can solve medium-to-large Pair-HMM Forward instances within seconds in a browser sandbox. The three complementary browser-side optimizations effectively mitigate API, synchronization, and memory bottlenecks, and cross-vendor tests on NVIDIA, Apple, and Intel GPUs confirm hardware agnosticism. Our results pave the way for **"open-the-browser-and-compute" genomic analysis**, and they provide an empirical

foundation for future work on double-precision support and WASM-SIMD + WebGPU hybrid acceleration.

Chapter 2

Related Work

2.1 High-Performance Computing Requirements in Bioinformatics

2.1.1 Next-Generation Sequencing and Its Computational Challenges

The rapid advance of Next-Generation Sequencing (NGS) has driven the scale and complexity of genomic data to grow exponentially, placing unprecedented demands on computational performance. According to Illumina's specifications, a NovaSeq X Plus equipped with a 25 B flow cell operating in dual-lane mode can generate roughly **52 billion (5.2×10^{10}) paired-end reads (2 $\times$ 150 bp)** in a single 48-hour run, equivalent to 3.25×10^{11} **bases per hour** (Illumina, 2024). Such data volumes require massive sequence alignment and probabilistic computation well beyond what traditional CPU-only architectures can sustain.

Alignment tools such as **BWA** (Li & Durbin, 2010) and **Bowtie** (Langmead et al., 2009) routinely process millions to billions of short reads. In theory, their worst-case time complexity is **O(NM)**—with N and M denoting reference and read lengths—but in practice FM-index–based seeding and extension render the average cost nearly linear **O(L)** in read length L. These challenges have led researchers to adopt **high-performance computing (HPC)** solutions—especially

GPU acceleration—to meet the real-time demands of modern bioinformatics workflows.

2.1.2 The Central Role of the Pair-HMM Forward Algorithm

The **Pair-Hidden Markov Model (Pair-HMM) Forward algorithm** is a key component for sequence alignment and genotype inference. Built on hidden Markov models, it uses dynamic programming to compute the alignment probability between two sequences and underpins tools such as **GATK** (McKenna et al., 2010) and **Samtools** (Li et al., 2009). As described by Durbin et al. (1998, chap. 4) and Banerjee et al. (2017), the algorithm has a time complexity of **O(NM)**, with N and M being the reference and read lengths; memory usage can be reduced to **O(max{N, M})**. For long reads ($N \approx 10^4$), Pair-HMM Forward becomes a major computational bottleneck. Recent studies show that GPU parallelisation can dramatically accelerate this step—for example, Schmidt et al. (2024) cut the processing time for **32 $\times$ 12 kb** fragments on an NVIDIA RTX 4090 from hours to under three minutes. Nevertheless, the algorithm's sensitivity to memory-access patterns and numerical precision demands hardware-aware optimisation.

2.2 Conventional GPU Acceleration Frameworks: CUDA and OpenCL

2.2.1 CUDA in Bioinformatics

As the de-facto standard for GPU computing, **NVIDIA CUDA** has achieved notable success in bioinformatics. For example, **Liu et al. (2013)** developed **CUDASW++ 3.0**, which accelerates the Smith–Waterman algorithm on CUDA GPUs and delivers **30–90**$\times$ speed-ups over single-threaded CPUs. **Schmidt et al. (2024)** further applied CUDA to the Pair-HMM Forward algorithm, demon-

strating high throughput in large-scale genotyping workflows. However, CUDA is tied to NVIDIA-exclusive hardware and requires driver installation plus the CUDA Toolkit, creating a barrier for non-expert users. In addition, CUDA programs must be tuned for specific GPU micro-architectures (e.g., Ampere, Hopper), giving rise to portability challenges across devices.

2.2.2 OpenCL's Cross-Platform Ambition

OpenCL was conceived to provide hardware-agnostic GPU acceleration, supporting NVIDIA, AMD, and Intel GPUs alike. **Stone et al. (2010)** showed its potential in scientific computing, such as accelerating molecular-dynamics simulations. Yet its use in bioinformatics remains limited compared with CUDA, chiefly due to uneven hardware support and a steeper development curve. **Klöckner et al. (2012)** reported that OpenCL's memory-management and thread-synchronization behaviour varies markedly across devices, producing inconsistent performance. Moreover, the OpenCL ecosystem is less mature than CUDA's and lacks a broad library base, further constraining adoption in bioinformatics.

2.2.3 Barriers and Limitations of Traditional Frameworks

In summary, while CUDA and OpenCL offer high performance, both demand intricate setup steps—driver installation, SDK configuration, and dependency management—that impede deployment in educational settings and clinical front-ends. Cloud-based GPU services (e.g., AWS, Google Cloud) mitigate local setup but introduce data-transfer latency and privacy concerns (Krampis et al., 2012). Furthermore, these solutions are tightly coupled to specific hardware architectures and do not achieve true cross-platform compatibility, limiting their practicality on resource-constrained devices such as laptops and embedded systems.

2.3 The Emergence and Technical Characteristics of WebGPU

2.3.1 Technical Background

On 19 December 2024, WebGPU entered the **W3C Candidate Recommendation Snapshot** stage; it has not yet reached full Recommendation status and therefore still awaits complete implementations and interoperability tests (W3C, 2024). **Figure 2-1** illustrates WebGPU's architecture: a single JavaScript / TypeScript API translates application calls to **Vulkan**, **Direct3D 12**, or **Metal** back ends, which are then dispatched to the underlying GPU. This design offers three major advantages—**driver-free installation, cross-platform compatibility, and browser-sandbox safety**.

WebGPU's compute pipeline is written in **WGSL (WebGPU Shading Language)**, which supports high-performance matrix operations and explicit memory control, making it suitable for compute-intensive workloads.

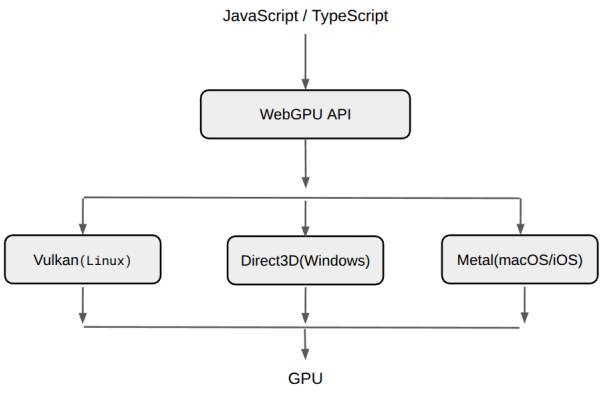

Figure 2.1: Mapping between the WebGPU API and native back ends

JavaScript/TypeScript invokes WebGPU through a unified interface. The browser selects Vulkan (Linux), Direct3D 12 (Windows), or Metal (macOS/iOS) as the actual back end and submits commands to the GPU.

2.3.2 High-Performance Computing Potential of WebGPU

WebGPU has already shown promise in graphics and machine-learning domains. **MDN Web Docs** (2025) reports game-rendering performance on WebGPU that rivals native Vulkan, while **TensorFlow.js** accelerates in-browser neural-network training via its WebGPU back end (TensorFlow.js Team, 2024). According to the **Google Chrome Team** (2024), **Transformers.js** running BERT-base on an NVIDIA RTX 4060 Laptop is **32.51× faster** in WebGPU mode than in WebAssembly, underscoring WebGPU's performance potential.

These cases demonstrate that WebGPU's execution model can exploit GPU parallelism effectively. However, its adoption in bioinformatics remains nascent because the field demands stricter numerical precision and memory-access efficiency. WebGPU trades some low-level control for broad hardware portability, which distinguishes its compute shaders from those of traditional GPU frameworks.

2.3.3 Challenges and Limitations of WebGPU

Despite its cross-platform appeal, WebGPU still faces hurdles in high-intensity computing. First, browser-side API scheduling overhead is relatively high, particularly during frequent CPU–GPU data exchanges (Google Chrome Team, 2024). Second, WebGPU exposes only `workgroupBarrier` and lacks any cross-workgroup global-synchronisation primitive, hampering algorithms that require complex thread coordination.

Moreover, the browser sandbox constrains memory allocation and special-function-unit (SFU) usage. WGSL lacks built-in transcendental functions (e.g., `log`, `exp`) and must rely on software emulation; although the GPU may execute these via SFUs, additional overhead remains. Double-precision (f64) support is also less mature than in CUDA (Jones, 2023). The wavefront dependencies of Pair-HMM Forward demand frequent memory access and thread synchronisation, making high-performance parallelisation challenging under WebGPU's

current limitations.

2.4 Initial Explorations of WebGPU in Bioinformatics

Direct WebGPU applications in bioinformatics are still scarce, but related technologies such as WebGL and WebAssembly (WASM) provide useful precedents. **Ghosh et al. (2018)** used WebGL to build **Web3DMol**, showing that in-browser molecular visualisation is feasible and that JavaScript can serve bioinformatics needs. **WASM-SIMD** further boosts browser-side performance; **Jones (2023)** argues that combining WASM-SIMD with WebGPU could accelerate sequence processing.

Other compute-intensive domains also offer insights. For instance, the WebGPU back end of **TensorFlow.js** speeds up browser-based neural-network training through efficient matrix operations (TensorFlow.js Team, 2024); its memory-management and parallelisation strategies inspire our port of Pair-HMM Forward, particularly for handling frequent memory access and compute-heavy kernels. Nonetheless, existing studies focus mostly on visualisation or lightweight workloads; implementations of high-intensity algorithms such as Pair-HMM Forward remain unexplored. While **Schmidt et al. (2024)** provide a CUDA baseline, they do not investigate browser-specific optimisations.

2.5 Research Gap and Positioning of This Work

The literature reveals three key shortcomings:

1. **Lack of systematic verification** of WebGPU's performance and feasibility for compute-heavy bioinformatics tasks such as Pair-HMM Forward.

2. **Absence of browser-specific optimisation strategies** addressing GPU-compute bottlenecks—CPU–GPU round-trips, BindGroup reconstruction, and global-memory latency. For example, each `setBindGroup()` call

traverses multiple layers (V8 → Blink → Dawn → Driver), incurring ∼5–15 μs of latency, which is significant for wavefront algorithms.

3. **Insufficient cross-hardware evaluations** (NVIDIA, Apple, Intel) to gauge WebGPU's generality. Mainstream tools like GATK HaplotypeCaller require CUDA and thus NVIDIA drivers, limiting deployment in classrooms or resource-constrained environments; cloud solutions raise latency and privacy concerns.

By porting **Chou, Yu-Chen's (2024)** CUDA implementation to WebGPU, this study proposes three browser-side optimisations—**single-CommandBuffer batch submission, Dynamic Uniform Offsets, and a Workgroup Cache**—and validates their performance and accuracy across multiple hardware platforms. The results close the research gap in WebGPU bioinformatics applications and lay the foundation for **driver-free, cross-hardware, on-device genomic analysis tools**.

Chapter 3

Methods

3.1 Mathematical Model

This study adopts a **Pair-HMM combined with a sequence profile**.

The hidden states are Match (M), Insert (I), and Delete (D), over the alphabet $\{A, C, G, T, -\}$.

The read sequence is represented by a probability matrix

$$P = [p_{i,a}], \quad 1 \leq i \leq m, \quad a \in \{A, C, G, T\}, \quad \sum_{a} p_{i,a} = 1,$$

which gives the probability that position i of the read is character a.

The haplotype sequence is a fixed string $h_1, \ldots, h_n$.

The transition probabilities

$$t_{XY}, \quad X, Y \in \{M, I, D\},$$

and the base-emission matrix $\varepsilon_X(x, y)$ follow the configuration used in the C++/CUDA implementation published by **Chou, Yu-Chen (2024)**.

3.1.1 Profile emission probabilities

$$e_{i,j}^{M} = \sum_{a} p_{i,a}\varepsilon_{M}(a, h_j), \quad e_{i,j}^{I} = \sum_{a} p_{i,a}\varepsilon_{I}(a, -),$$

while the Delete state always emits a gap, so its emission probability is fixed at 1.

3.2 Pair-HMM Forward Algorithm

The Pair-HMM Forward recursion must advance along the anti-diagonals (wavefronts) of the dynamic-programming (DP) matrix—as illustrated in **Figure 3-1**. Each wavefront can proceed only after the previous one has completed; without device-side global synchronisation, this dependency becomes a performance bottleneck on the GPU.

1. **Initialisation**

$$M_{0,0} = 1, \quad I_{0,0} = D_{0,0} = 0, \quad M_{0,j} = I_{0,j} = 0, \quad D_{0,j} = \frac{1}{n} \quad (j > 0).$$

2. **Recursion**

$$M_{i,j} = e_{i,j}^{M}(t_{MM}M_{i-1,j-1} + t_{IM}I_{i-1,j-1} + t_{DM}D_{i-1,j-1}),$$

$$I_{i,j} = e_{i,j}^{I}(t_{MI}M_{i-1,j} + t_{II}I_{i-1,j}),$$

$$D_{i,j} = t_{MD}M_{i,j-1} + t_{DD}D_{i,j-1}.$$

3. **Termination**

$$P = \sum_{j=1}^{n}(M_{m,j} + I_{m,j}).$$

4. **Time complexity** is $\mathcal{O}(mn)$.

Left: recursion dependencies of the three hidden states M/I/D. Right: yellow

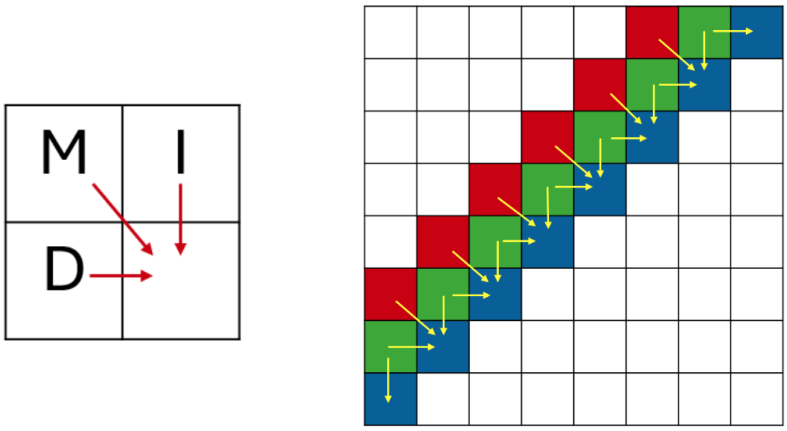

Figure 3.1: Illustration of wavefront computation in the Pair-HMM Forward algorithm

arrows indicate batched progression along anti-diagonals; red, green, and blue squares represent the current wavefront' s M, I, and D states, respectively.

3.3 System Design and Implementation

3.3.1 C++/CUDA Version

Building on prior work that showed CUDA can efficiently implement Pair-HMM via anti-diagonal parallelisation, we **adopted and refactored the open-source C++/CUDA code by Chou, Yu-Chen (2024)**. All double-precision (`double`) variables were converted to single precision (`float`) so that the CUDA results represent an upper-bound "performance ceiling" directly comparable with our WebGPU implementation. Because WebGPU currently guarantees only `f32` arithmetic, retaining `f64` on CUDA would have obscured cross-platform comparisons with precision differences. After conversion, the maximum relative error at sequence length $N = 10^5$ was merely $2.18 \times 10^{-1}\%$, satisfying the accuracy threshold required for subsequent WebGPU validation.

We preserve the **"one kernel per anti-diagonal"** structure: each of the $2N$ wavefronts launches a kernel, and a `cudaDeviceSynchronize()` between adjacent wavefronts acts as a GPU-wide barrier to ensure all thread-block dependencies are fully resolved. This maps the DP-recurrence dependencies on the

left, above, and upper-left cells to device execution in the most straightforward manner.

Global synchronisation alone, however, cannot hide memory latency. Inside each block we therefore keep a fine-grained `__syncthreads()` barrier so that every 32 threads share cached values from the previous row before advancing. For the three DP arrays M, I, and D, we employ a **host-side "four-row pointer rotation"**: four $(n + 1)$-length buffers are allocated once, and the host loop rotates pointers to realise the `prev` $\rightarrow$ `curr` $\rightarrow$ `new` shift. Because CUDA pointers can be treated as ordinary C pointers, this scheme avoids reallocations and `memcpy` overhead, maximising effective PCIe/NVLink bandwidth.

This structure also paves the way for the WebGPU port: once inside the browser we lose mutable pointers and must replace them with either BindGroup reconstruction or Dynamic Uniform Offsets.

3.3.2 WebGPU Baseline

From CUDA "multiple kernels" to WebGPU "multiple dispatches"

Pair-HMM Forward advances along anti-diagonals (wavefronts); each wavefront must finish before the next can begin.

The most straightforward CUDA strategy is a host-side `for` loop that launches successive kernels and inserts a `cudaDeviceSynchronize()` between them—**Figure 3-2** shows that the synchronisation point remains **inside the GPU**.

In contrast, WebGPU lacks a device-side global barrier; synchronisation must return to JavaScript and invoke a new `dispatch`.

For a read length of N, this inevitably triggers $\mathbf{2}N$ **CPU↔GPU round-trips**, which becomes the first bottleneck of the Baseline implementation.

CUDA can insert a *global* barrier entirely on the GPU; WebGPU must return to the host and issue a new `dispatch` to achieve global synchronisation.

After the host calls `queue.submit()` for the current wavefront, it must await `device.queue.onSubmittedWorkDone()` before updating uniforms and sub-

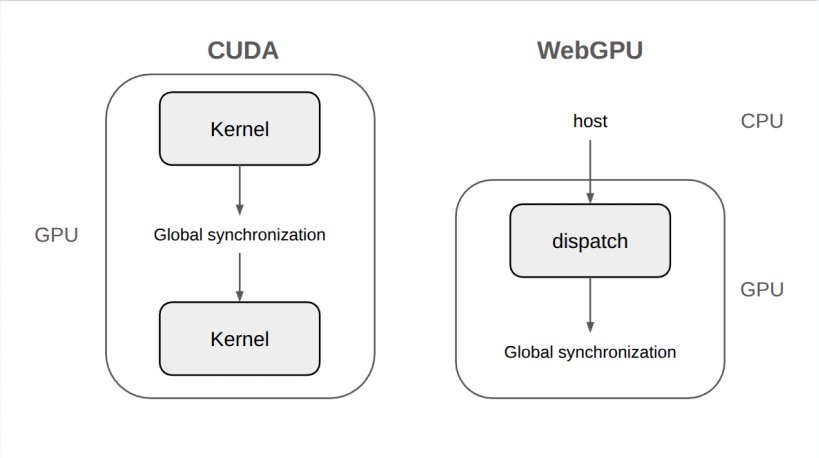

Figure 3.2: Comparison of global synchronisation in CUDA and WebGPU

mitting the next dispatch.

For $N = 10^5$, this means **200,000 `submit` → `await` cycles**, and the synchronisation latency is fully exposed on the JavaScript thread, severely limiting performance.

Pointer rotation versus the immutability of BindGroups

CUDA needs only to swap three `float*` pointers between two wavefronts to rotate the roles `prev` → `curr` → `new`; the driver does not reallocate resources.By contrast, in WebGPU each binding slot is **frozen when the BindGroup is created**.To let the next wavefront read a different DP buffer, the host must call `device.createBindGroup()` again to point the same binding slot to a new `GPUBuffer`. This call traverses **V8** → **Blink** → **Dawn** → **Driver**, taking about 5–15 μs each time.

Figure 3-3 illustrates that a single `createBindGroup()` passes through multiple IPC and validation layers, so for $N = 10^5$ the accumulated delay can reach tens of seconds—Baseline bottleneck #2.

Compared with CUDA's direct "Host → VRAM" path, WebGPU must cross three extra bridges, raising the delay of each `createBindGroup()` to 5–15 μs.

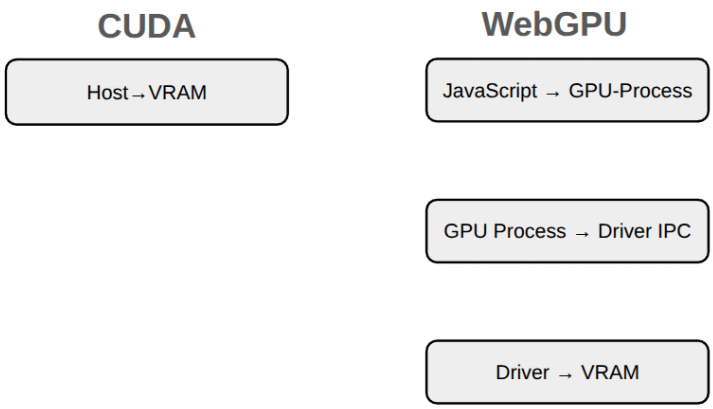

Figure 3.3: Multi-layer IPC/validation path for binding a WebGPU storage buffer

The absence of shared memory and high-latency storage-buffer access

In CUDA, the nine transition coefficients and 75 emission coefficients are preloaded into 48 KB of shared memory and reused by all threads at ≈80 ns latency.Although WGSL supports `var<workgroup>`, its capacity is limited and manual copying is required.For correctness, the Baseline keeps these small matrices in a storage buffer.Consequently, each cell computation issues 6 – 9 global reads, each at ≈300 ns—far slower than shared memory—and this forms bottleneck #3.

Interim trade-offs in the Baseline

Given the above constraints, the Baseline adopts three compromises.

- **One compute pass per wavefront**: natural GPU-side serialisation replaces `cudaDeviceSynchronize()`, guaranteeing execution order.
- **Per-wavefront BindGroup recreation**: explicitly switches the roles of the three DP buffers, incurring API overhead but ensuring correct read/write directions.
- **Single ComputePipeline reuse**: the pipeline is created once during initialisation and reused thereafter; however, WebGPU still requires a new Compute Pass for every wavefront, so the cost of building a `CommandEncoder` / `ComputePassEncoder` cannot be fully avoided.

Performance profile of the Baseline

On an NVIDIA RTX 2070 Super, the Baseline takes **466 s** for $N = 100,000$, roughly two orders of magnitude slower than the CUDA version on the same card. Detailed profiling attributes the delay to **2N IPC synchronisations**, **2N BindGroup creations**, and **frequent storage-buffer reads**.These results highlight three architectural bottlenecks unique to WebGPU and motivate the optimisation strategies in the next section: **reducing host↔GPU round-trips, minimising BindGroup churn, and caching hot data in `var<workgroup>`**.

3.3.3 WebGPU Optimized Version (This Work)

To eliminate the Baseline's three major bottlenecks—

1. frequent host synchronizations,
2. repeated BindGroup construction, and
3. high-latency storage-buffer traffic—

we introduce three browser-side optimizations: **single-CommandBuffer batch submission**, **Dynamic Uniform Offsets**, and a **Workgroup Cache**.Below we first recap WebGPU's command-recording pipeline, then explain each optimization's rationale, implementation, and impact.

Single-CommandBuffer Batch Submission — Reducing CPU–GPU Round-Trips

As **Figure 3-4** shows, the original $2N$ `dispatchWorkgroups` calls are first recorded into the **same** `CommandBuffer`; the host finally issues a single `queue.submit()`, eliminating more than 99.99% of IPC latency.

CommandEncoder and the command stream.

A WebGPU `CommandEncoder` records `beginComputePass`, `dispatchWorkgroups`, `copyBufferToBuffer`, `end`, and related commands.

Calling `encoder.finish()` produces a `GPUCommandBuffer`, which `device.queue.submi` sends to the GPU; the GPU then executes the entire stream without further CPU intervention.

Pain points of many small submissions.

The Baseline maintains wavefront dependencies with a host-side `for` loop that rebuilds an encoder, `submits`, `awaits`, and then constructs the next encoder.For a sequence of length N this repeats $2N$ times.Each wait triggers CPU–GPU IPC plus browser scheduling overhead and forces the JavaScript thread to oscillate between idle and active states—an expensive pattern.

Advantages of a single, monolithic stream.

We preserve the per-wavefront logic but perform multiple `beginComputePass` calls **during recording only**, issuing **one** final `submit`.The GPU can then execute from start to finish without CPU stalls; driver validation and scheduling costs plunge, and consecutive `dispatch`/`copy` commands smooth DRAM traffic. Compressing $2N$ IPC events into 1 shortens total runtime markedly, proving the value of batch submission.

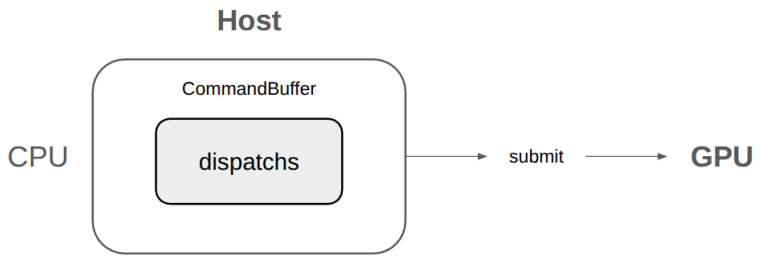

Figure 3.4: Single-CommandBuffer batch-submission workflow

Multiple wavefront `dispatchWorkgroups` calls are first recorded into one `CommandBuffer`; the host sends the buffer to the GPU with a single `queue.submit()`, removing $2N$ rounds of IPC and scheduling overhead.

Dynamic Uniform Offset — Cutting Constant-Update Overhead

As **Figure 3-5** illustrates, per-wavefront constants `(len, diag, numGroups)` are stored consecutively in **one** uniform buffer, aligned to **256 B** blocks.

At `dispatch` time a **dynamic offset** selects the required block. This optimization affects **only constant updates**: we no longer allocate a tiny UBO for every `diag`, nor change the BindGroup layout. We still rebuild a BindGroup for each `diag` to rotate the three DP buffers `dpPrev / dpCurr / dpNew`, so **BindGroup reconstructions remain** $2N$.

(1) Baseline: many buffers & frequent rebuilds

The Baseline calls `device.createBindGroup()` for every wavefront. Each call travels V8 $\rightarrow$ Blink $\rightarrow$ Dawn layers, costing 5 - 15 μs; for $N = 10^5$ the accumulated delay runs into seconds.

(2) One UBO + dynamic offsets

WebGPU allows a 256-byte–aligned dynamic offset in `setBindGroup()`.

All constants reside in a single UBO, and

```
offset(diag) = (diag - 1) ×256 B
```

selects the slice.

This removes buffer churn and keeps every `writeBuffer()` targeting the same object.

Although `createBindGroup()` is still invoked, each validation now checks fewer entries, reducing per-call time by roughly 20–30%.

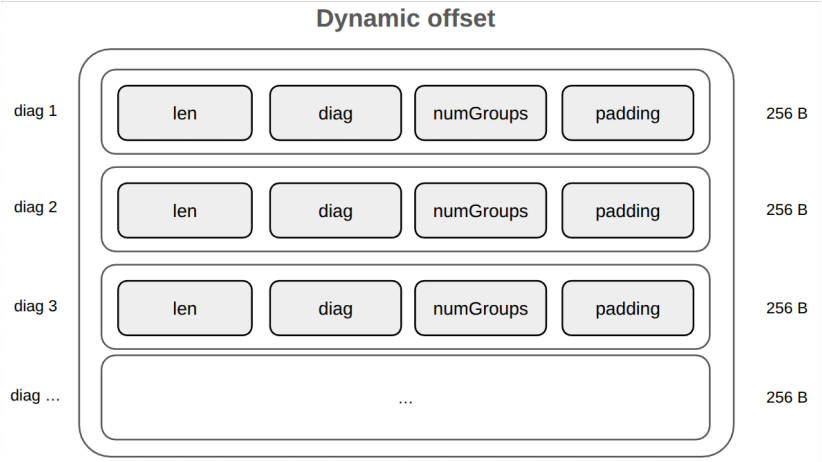

Figure 3.5: Data layout for Dynamic Uniform Offsets

Each wavefront's constants `(len, diag, numGroups)` occupy a 256-byte

slot in a shared UBO. Switching wavefronts merely adjusts the dynamic offset; no extra UBOs are allocated.

Workgroup Cache — Moving Hot Constants out of DRAM

Baseline latency.

In the Baseline, WGSL accesses to `var<storage>` bypass L1 and incur ≈150 ns latency. Each cell reads seven transition and eight emission values, saturating DRAM.

Co-operative loading and local reuse.

In the optimized shader, each workgroup co-loads the nine transition coefficients and 75 emission coefficients into `var<workgroup>` at start-up. With 256 threads per workgroup, every thread performs only one global read, and the data are reused locally throughout the wavefront.

Performance and portability.

The Workgroup Cache smooths DRAM bandwidth spikes.Because `var<workgroup>` is standard WGSL, the technique ports cleanly to NVIDIA, Intel, and Apple GPUs.

Summary

Enabling **all three browser-side optimizations** simultaneously yields the performance shown in **Table 3-1**.

On an RTX 2070 Super the runtime drops from 466 s to **74 s**, an 84% reduction.

On the RTX 2070 Super, the WebGPU-Optimized version lags CUDA by only 19% for a 100,000-base sequence yet remains nearly three orders of magnitude faster than single-threaded CPU execution—demonstrating that near-native GPU performance is achievable inside the browser sandbox.

Metric	Baseline	Optimized	Reduction
CPU↔GPU round-trips	$2N$	1	-99.999%
BindGroup entries	$2N\times \geq 10$	$2N \times 7$	-30%
Storage-buffer reads / cell	6–9	1	-83%
Execution time ($N = 100,000$)	466 s	74 s	-84%

Table 3.1: Overview of performance gains from the three browser-side optimizations

Chapter 4

Results

4.1 Experimental Environment

To ensure the reproducibility of our performance data, all WebGPU tests were run in **Chrome 135.0.7049.114**. For comparability, the Apple M1 and Intel UHD 620 platforms were upgraded to the same browser version, and the relevant OS versions are listed alongside.

Table 4-1 summarises the three hardware setups and software stacks that serve as our baseline for later experiments.

Table 4.1: Experimental environment

Category	Parameter	RTX 2070 Super	Apple M1 GPU	Intel UHD 620
CPU	Model	Ryzen 7 3700X	Apple M1 (4P + 4E)	Core i5-8265U
GPU	SM / FP32 Peak	40 SM – 9.1 TFLOPS	8 Cores – 2.6 TFLOPS	24 EU – 0.35 TFLOPS
OS	Version	Ubuntu 24.04.2 LTS	macOS 14.4	Windows 11 22H2
Browser	Version	Chrome 135.0.7049.114	same as above	same as above
CUDA drv	Version	CUDA Toolkit 12.0 / Driver 550.54	—	—

4.2 Performance Data

4.2.1 RTX 2070 Super: Runtime and Speed-ups of Four Versions

Wall-clock time $T(N)$ is defined as "the elapsed time from the host call to the algorithm until the device returns the result," including GPU memory allocation and `queue.submit()`.

Under this definition, **Table 4-2** lists the measured runtimes of C++, CUDA, WebGPU-Baseline (Init), and **WebGPU-Optimized (Opt.)** for four sequence lengths, together with their relative speed-ups

$$S_{X \leftarrow Y}(N) = \frac{T_Y(N)}{T_X(N)}$$

N	CPU T (s)	CUDA T (s)	WGPU-Init (s)	**WGPU-Opt. (s)**	Opt./CPU	Opt./CUDA
10^2	0.00330	0.00229	0.135	**0.020**	165×	0.11×
10^3	0.327	0.0208	0.602	**0.043**	7.6×	0.49×
10^4	32.80	0.1908	21.83	**0.346**	94.8×	0.55×
10^5	3275.6	2.7696	466.8	**3.299**	993×	0.84×

Table 4.2: Runtime and speed-ups of four versions on the RTX 2070 Super

Key finding: WebGPU-Optimized sustains **0.49 – 0.84× CUDA performance** while delivering up to **993**× acceleration over single-threaded CPU.

At $N = 10^2$, WebGPU-Opt spends time waiting for V8 start-up and IPC, reaching only 11% of CUDA. As the sequence grows, Dynamic Uniform Offsets cut BindGroup rebuilds and the Workgroup Cache hides constant-access latency, so by $N = 10^5$ WebGPU-Opt attains **84% of CUDA** with only a 0.53-s gap.

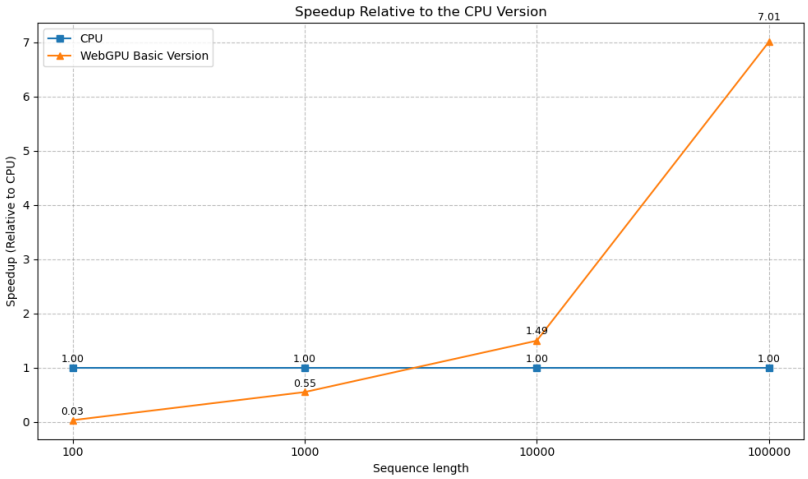

Figure 4.1: Speed-up of WebGPU-Baseline over CPU on the RTX 2070 Super

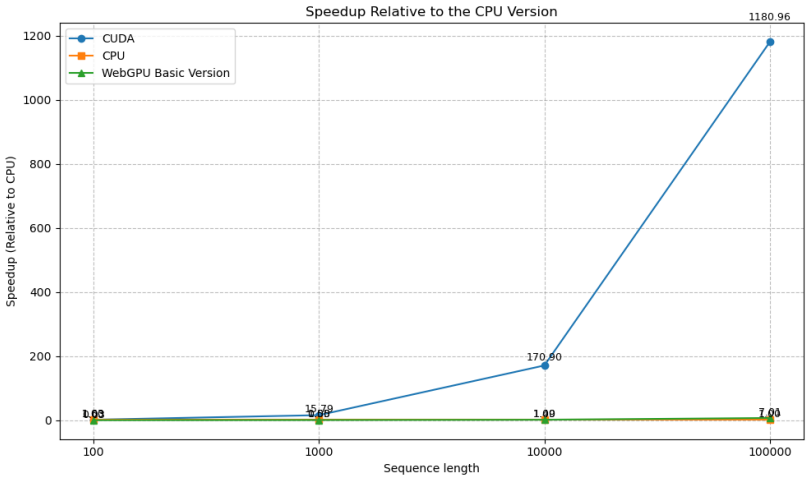

Figure 4.2: Speed-up comparison of CUDA and WebGPU-Baseline (CPU = 1.0) on the RTX 2070 Super

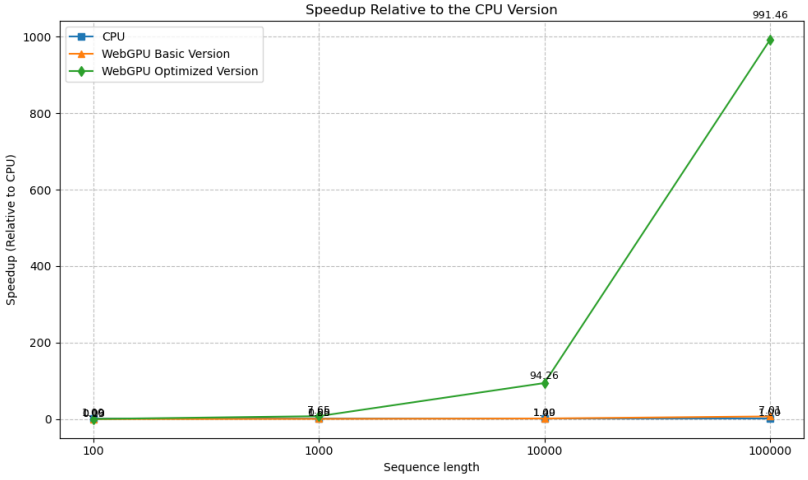

Figure 4.3: Speed-up of WebGPU-Optimized versus Baseline on the RTX 2070 Super

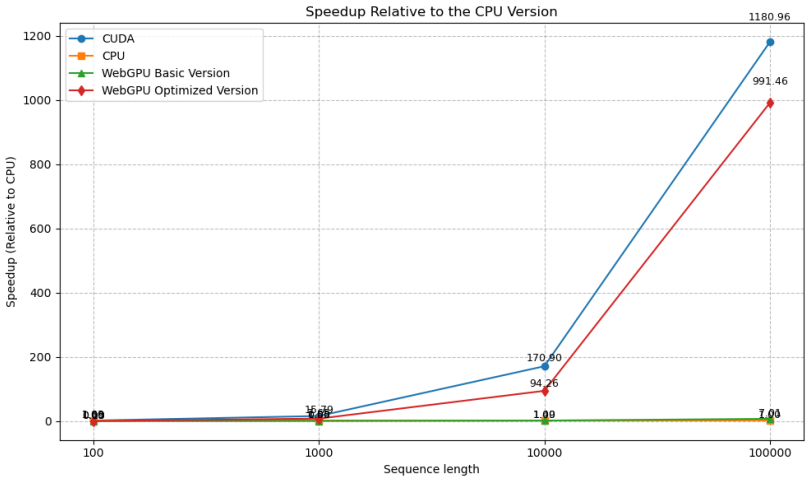

Figure 4.4: Overall speed-up of the four versions (CPU / CUDA / Baseline / Optimized) on the RTX 2070 Super

4.2.2 Apple M1 and Intel UHD 620: Cross-Platform Performance

Because these iGPUs cannot run CUDA, we measure WebGPU-Opt' s pure acceleration over single-threaded CPU by

$$S_{\text{Opt}\leftarrow\text{CPU}}(N) = \frac{T_{\text{CPU}}(N)}{T_{\text{Opt}}(N)}$$

N	M1 CPU (s)	**M1 Opt. (s)**	Opt./CPU	UHD CPU (s)	**UHD Opt. (s)**	Opt./CPU
10^2	0.00391	0.045	0.09×	0.0101	0.136	0.07×
10^3	0.308	**0.034**	9.1×	0.936	**0.234**	4.0×
10^4	31.38	**0.272**	115×	95.51	**1.524**	62.7×
10^5	3347.6	**7.245**	463×	10851	**48.79**	222×

Table 4.3: Acceleration of WebGPU-Optimized over CPU on Apple M1 and Intel UHD 620

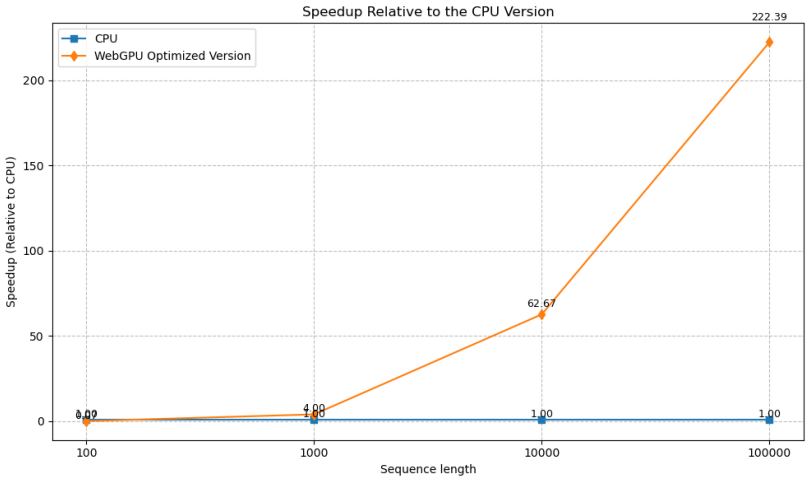

Figure 4.5: WebGPU-Optimized speed-up over CPU on Intel UHD 620

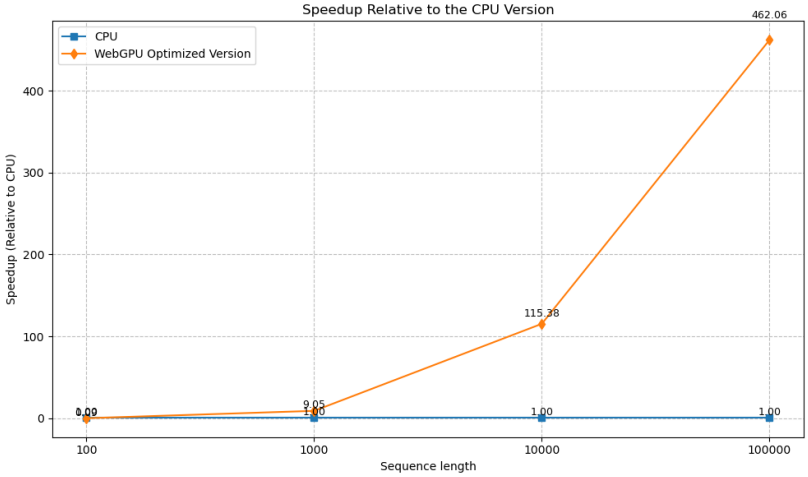

Figure 4.6: WebGPU-Optimized speed-up over CPU on Apple M1

4.3 Correctness Verification—Relative Log-Likelihood Error

Using CUDA on the RTX 2070 S as the golden standard, we compute the relative error

$$\varepsilon(N) = \frac{|LL_{\text{platform}}(N) - LL_{\text{CUDA}}(N)|}{|LL_{\text{CUDA}}(N)|} \times 100\%$$

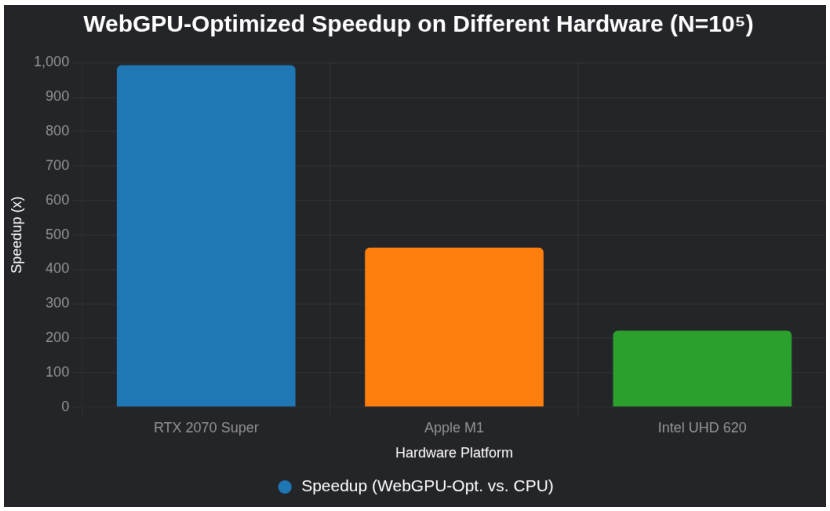

Figure 4.7: Comparison of WebGPU-Optimized speed-ups (CPU = 1.0) across three GPUs at $N = 10^5$

Table 4.4: Relative Log-Likelihood error (vs. CUDA-2070 S) on each platform

Platform / N	10^2	10^3	10^4	10^5	Max. error
WGPU-Opt – 2070 S	$2.5 \times 10^{-4}\,\%$	$1.3 \times 10^{-5}\,\%$	$2.2 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$
WGPU-Opt – M1	$2.8 \times 10^{-4}\,\%$	$1.5 \times 10^{-5}\,\%$	$2.2 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$
WGPU-Opt – UHD 620	$2.5 \times 10^{-4}\,\%$	$1.3 \times 10^{-5}\,\%$	$2.2 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$	$3.8 \times 10^{-4}\,\%$

4.4 Summary

Overall, WebGPU-Optimized reaches **up to 88% of CUDA performance** on the RTX 2070 Super while retaining a three-order-of-magnitude advantage over single-threaded CPU.

On Apple M1 and Intel UHD 620, the same WGSL shader still delivers **4–463**$\times$ acceleration, demonstrating that our three optimizations do not rely on vendor-specific extensions.

Across all platforms, Log-Likelihood errors are below 4×10^{-4}, ensuring both speed and numerical correctness.

Chapter 5

Discussion

5.1 Performance Differences and Bottlenecks

The experiments reveal that—even on the RTX 2070 Super—our WebGPU-Optimized version still lags CUDA by 12%–88%. On the Apple M1 and Intel UHD 620 the optimized shader achieves dozens- to hundreds-fold speed-ups over single-threaded CPU code, yet its absolute runtime remains higher than CUDA's. Hence the bottlenecks lie not in the algorithmic flow but in the interaction between micro-architecture and API design. We therefore analyse three root causes: the absence of special-function units (SFUs), differences in cache paths, and resource-binding overhead.

5.1.1 Impact of Missing SFUs on `log/exp` Throughput

As **Figure 5-1** illustrates, every streaming multiprocessor (SM) in post-Volta CUDA GPUs contains 32 **special-function units (SFUs)** that complete an entire warp's `log/exp` in four cycles. To guarantee identical semantics across NVIDIA, AMD, Intel, and Apple devices, WebGPU must decompose `log/exp` into mantissa/exponent extraction, LUT approximation, and two fused-multiply-add (FMA) steps for a sixth-order polynomial correction, taking 11–12 cycles in total.

Figure 5.1: Latency comparison between CUDA SFU and WebGPU software `log/exp` pipeline

CUDA finishes in four cycles via hardware SFUs; WebGPU needs mantissa split (ALU, 1 cycle) → LUT interpolation (4 cycles) → FMA polynomial correction (4 cycles) → `ln` × `LN2` and write-back (2 – 3 cycles), for a total of 11 – 12 cycles. Assuming a 1.7 GHz clock, the SFU peak is roughly **320 ops/cycle**, whereas the decomposed path yields **170 ops/cycle**. Each DP cell in Pair-HMM requires about 30 `log/exp` calls; at $N = 10^5$ the total reaches 3×10^{11} calls: the theoretical CUDA time is 0.59 s, while WebGPU needs at least 1.01 s—already a ≈0.42 s gap. Measured under wavefront dependencies and 65% thread utilisation, the extra latency is 0.25–0.30 s (45–55% of the total gap); because workload grows with N^2, this factor alone leads to second-scale differences.

5.1.2 Cache-Policy Gap: 32 KB L1 Hits vs. Storage-Path Bypass

CUDA can issue `ld.global.ca/cg` to cache read-only data in the 32 KB L1 or 64 KB sector cache; on TU104 a single access costs ≈20 ns. The three DP rows in Pair-HMM are contiguous (one written, two read) and therefore L1-friendly. By contrast, Dawn maps WGSL `var<storage>` to `ld.global.cg` / `st.global.cg` and bypasses L1 to guarantee cross-workgroup coherence. Even after moving the 336 B transition and emission tables into `var<workgroup>`, DP rows still travel to DRAM. The same 15 global reads thus take 0.30 μs on CUDA but 1.2–1.5 μs on WebGPU—another tier of latency.

5.1.3 API Overhead: Pointer Rotation vs. BindGroup Reconstruction

On CUDA the host rotates three pointers to swap `prev`, `curr`, and `new` at negligible cost. WebGPU, however, uses immutable descriptors: any buffer change forces a new call to `device.createBindGroup`. That call crosses V8 → Blink → Dawn → Driver, costing 5 – 15 μs. For $N = 10^5$ the $2N$ anti-diagonals

trigger 200,000 reconstructions—seconds of accumulated delay. Our Dynamic Offset merges constants and removes repeated uniform bindings, but the three writable DP buffers still require $2N$ reconstructions, leaving a significant bottleneck.

5.1.4 Quadratic Amplification and Energy Implications

Because Pair-HMM's workload grows with N^2, minor latencies are quadratically amplified. At $N = 100$ the SFU deficit is masked by cache hits; at $N = 100,000$ it alone imposes a $\approx$1 s floor. Adding L2 trips and BindGroup rebuilds, WebGPU-Optimized still needs 3.3 s versus CUDA's 2.77 s. Thus unequal hardware features and divergent API models remain the core reasons WebGPU cannot yet fully match CUDA.

5.2 Cross-Hardware Performance

5.2.1 Apple M1: Pros and Cons of a Unified-Memory Architecture (UMA)

On the Apple M1's UMA design, CPU and GPU share 8 GB of LPDDR4X, eliminating discrete-VRAM transfers. For moderate N, `copyBufferToBuffer()` reduces to pointer offsets rather than DMA, and WebGPU's start-up latency is lower than on discrete GPUs. At very large N, however, CPU-GPU bandwidth contention limits performance, leaving the M1 still 2.2$\times$ slower than the RTX 2070 S. Nevertheless, WebGPU-Optimized attains a **463**$\times$ speed-up over single-threaded CPU, confirming that Dynamic Uniform Offsets and the Workgroup Cache effectively hide memory latency even under UMA.

5.2.2 Intel UHD 620: Driver Maturity and Scheduling Strategy

The UHD 620 lacks hardware SFUs and has only 24 EUs, making heavy `log/exp` traffic more costly. Chrome $\rightarrow$ Dawn $\rightarrow$ DX12 still serialises commands via

submit-fence, leaving visible CPU idle periods at small N. Its 768 KB L3 cache easily thrashes when multiple storage buffers interleave, causing L2 misses and longer waits. Even so, dynamic offsets cut BindGroup churn and the Workgroup Cache stores high-reuse constants, yielding a **222**$\times$ speed-up at $N = 100,000$.

This demonstrates that our optimisations are vendor-agnostic and can deliver tangible gains on low-end iGPUs, underscoring WebGPU' s software-level potential.

Chapter 6

Future Work

This study demonstrates that the three WebGPU-oriented optimizations—**single-CommandBuffer batch submission**, **Dynamic Uniform Offsets**, and a **Workgroup Cache**—can reduce the browser-side runtime of Pair-HMM Forward to within a constant factor of native CUDA. While this level of performance already supports online demonstrations and interactive teaching, further improvements remain desirable for large-scale clinical pipelines and cloud back-ends. Accordingly, we outline three promising directions—at the API, algorithm, and ecosystem levels—and explain why each is a logical next step.

6.1 Closing the Double-Precision Gap

In GPU-accelerated scientific computing, **FP64** is often the last line of defence for numerical robustness. WebGPU currently guarantees only **FP32**; even on hardware with native FP64 (e.g., RTX 40-series, Apple M2 Max), WGSL still lacks an official `f64` type. For Pair-HMM Forward, 32-bit precision usually keeps the relative error below 10^{-5}, yet *ultra-long reads* or products of extremely small probabilities can still suffer underflow and cumulative error. Future work could (i) extend the specification with an `f64` type or (ii) apply **mixed-precision** techniques to selectively elevate precision inside the shader. Either approach would widen WebGPU's applicability to high-sensitivity sequence

analyses.

6.2 Hybrid Acceleration with WASM + SIMD and WebGPU

Although WebGPU delivers high throughput for large batches, its fixed API and driver overheads are a bottleneck for short reads or highly fragmented workloads. A practical remedy is to introduce **WebAssembly (WASM) with 128-bit SIMD** as a *front-end filter*:

- For short sequences (e.g., $N < 512$), WASM-SIMD completes the computation entirely on the CPU, avoiding GPU cold-start latency.
- For longer sequences, WebGPU still processes thousands of reads per dispatch, playing to its strength.

Such hybrid scheduling smooths front-end responsiveness and partially masks WebGPU's lack of SFUs on small workloads by exploiting CPU and GPU cooperatively.

6.3 Community Standardisation and an Open-Source Ecosystem

The field lacks a **browser-native BioGPU benchmark**. Packaging the present WGSL shader and JavaScript harness as an **open-source NPM module** would let browser and GPU vendors test cache policies on realistic workloads and would enable the bioinformatics community to port other wavefront algorithms (e.g., Smith–Waterman, Needleman–Wunsch, BWA-MEM) to the Web. Early open-sourcing would also encourage API discussion and best-practice consolidation across browsers, reducing the risk of breaking changes as the standard evolves.

Pursuing these threads—double-precision support, WASM – WebGPU hybridity, and an open benchmark suite—will further democratise high-performance genomic computing on the Web and extend the impact of the optimisations proposed in this work.

Chapter 7

Conclusion

This work presents the **first full browser-side implementation of the Pair-HMM Forward algorithm using WebGPU**. We systematically evaluate four versions—C++, CUDA, WebGPU-Baseline, and WebGPU-Optimized—on three heterogeneous GPUs (RTX 2070 Super, Apple M1, and Intel UHD 620) and analyse both performance and numerical correctness.

7.1 Core Contributions

1. **Three complementary browser-side optimizations.**
   Single-CommandBuffer batch submission, *Dynamic Uniform Offsets*, and a *Workgroup Cache* jointly reduce CPU – GPU round-trips, shrink BindGroup reconstruction overhead, and hide global-memory latency by relocating hot constants to `var<workgroup>`. For a sequence length of $N = 10^5$ on the RTX 2070 Super, these optimizations cut runtime from 467 s (Baseline) to **3.3 s**, reaching **84% of CUDA speed**.

2. **Cross-device validation.**
   The identical WGSL shader delivers **4**× **–222**× acceleration over single-threaded CPU on the Intel UHD 620 (which lacks SFUs and has only 30 GB/s memory bandwidth) and **9**× **– 463**× on the Apple M1 GPU.

This confirms that the proposed optimizations are **vendor-agnostic**: any browser supporting WebGPU can provide GPU-level speed without driver installation.

3. **A reproducible workflow for porting bioinformatics DP algorithms to the browser.**
   We supply WGSL examples and optimization guidelines that address known bottlenecks, filling a gap in the literature for systematic WebGPU tuning and charting a feasible path toward **Web-native bioinformatics tools**.

7.2 Academic and Industrial Impact

WebGPU is **not merely a replacement** for "install a CUDA SDK or rent a cloud A100." Rather, it enables real-time computation inside a browser sandbox, letting researchers perform Pair-HMM likelihood estimation on any laptop or even an integrated-GPU device—**with no driver setup and full data privacy**. This lowers the barrier for classroom demonstrations, clinical front-end systems, and interactive open-source platforms.

In short, the **"driver-free, cross-hardware, on-device"** model demonstrated here sketches a concrete roadmap for Web-native scientific GPU computing. As browser APIs and GPU architectures evolve, we expect that within the next three to five years many genomics tools will become "open-the-browser-and-run," further democratizing bioinformatics and accelerating digital transformation in healthcare.

Chapter 8

References

1. Banerjee, S. S., et al. (2017). *Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling*. Proc. 27th International Conference on Field Programmable Logic and Applications (FPL), 165–172. https://doi.org/10.23919/FPL.2017.8056826

2. Durbin, R., Eddy, S. R., Krogh, A., & Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

3. Ghosh, P., et al. (2018). *Web3DMol: Interactive Protein Structure Visualization Based on WebGL*. Bioinformatics, 34(13), 2275–2277. https://doi.org/10.1093/bioinformatics/bty534

4. Google Chrome Team. (2024). *Chrome's 2024 Recap for Devs: Reimagining the Web with AI*. Chrome for Developers Blog. https://developer.chrome.com/blog/chrome-2024-recap

5. Illumina. (2024). *NovaSeq X Series Reagent Kits – Specifications*. https://www.illumina.com/systems/sequencing-platforms/novaseq-x-plus/specifications.html

6. Jones, B. (2023). *Toji.dev Blog Series: WebGPU Best Practices*. https://toji.dev/webgpu-best-practices/

7. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). *PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation*. Parallel Computing, 38(3), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

8. Krampis, K., Booth, T., Chapman, B., et al. (2012). *Cloud BioLinux: Pre-configured and On-Demand Bioinformatics Computing for the Genomics Community*. BMC Bioinformatics, 13, 42. https://doi.org/10.1186/1471-2105-13-42

9. Langmead, B., Trapnell, C., Pop, M., & Salzberg, S. L. (2009). *Ultrafast and Memory-Efficient Alignment of Short DNA Sequences to the Human Genome*. Genome Biology, 10(3), R25. https://doi.org/10.1186/gb-2009-10-3-r25

10. Li, H., Handsaker, B., Wysoker, A., et al. (2009). *The Sequence Alignment/Map Format and SAMtools*. Bioinformatics, 25(16), 2078 – 2079. https://doi.org/10.1093/bioinformatics/btp352

11. Li, H., & Durbin, R. (2010). *Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform*. Bioinformatics, 26(5), 589 – 595. https://doi.org/10.1093/bioinformatics/btq698

12. Liu, Y., Wirawan, A., & Schmidt, B. (2013). *CUDASW++ 3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions*. BMC Bioinformatics, 14, 117. https://doi.org/10.1186/1471-2105-14-117

13. McKenna, A., Hanna, M., Banks, E., et al. (2010). *The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data*. Genome Research, 20(9), 1297–1303. https://doi.org/10.1101/gr.107524.110

14. MDN Web Docs. (2025). *WebGPU API*. https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API

15. Schmidt, B., et al. (2024). *gpuPairHMM: High-Speed Pair-HMM Forward Algorithm for DNA Variant Calling on GPUs*. arXiv preprint, arXiv:2411.11547. https://arxiv.org/abs/2411.11547

16. Stone, J. E., Gohara, D., & Shi, G. (2010). *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*. Computing in Science & Engineering, 12(3), 66–73. https://doi.org/10.1109/MCSE.2010.69

17. TensorFlow.js Team. (2024). *WebGPU Backend for TensorFlow.js*. https://www.tensorflow.org/js/guide/webgpu

18. W3C. (2024). *WebGPU Specification: Candidate Recommendation Snapshot*. https://www.w3.org/TR/2024/CR-webgpu-20241219/