Parallelizing Pauli Paths (P3):

GPU-Accelerated Quantum Circuit Simulation

Final Project Report

Arul Rhik Mazumder
arulm@andrew.cmu.edu

Daniel Ragazzo
dragazzo@andrew.cmu.edu

15-418/618 Parallel Computer Architecture and Programming
Carnegie Mellon University

December 15, 2025

https://arulrhikm.github.io/Parallelizing-Pauli-Paths/report.html

Abstract

We present a GPU-accelerated implementation of the Pauli path propagation algorithm for quantum circuit simulation. Our CUDA implementation achieves speedups of up to 626× over a single-threaded CPU baseline on circuits with 1000+ initial Pauli words, with an average speedup of 213× across our stress test suite. Key contributions include a multi-block dynamic work distribution algorithm, efficient inter-block coordination through word-count-only transfers, and parallel truncation using prefix sums. We provide a comprehensive analysis of performance characteristics, scaling behavior, and limiting factors across diverse circuit configurations.

1 Summary

We implemented a GPU-accelerated Pauli path propagation algorithm for quantum circuit simulation using CUDA on NVIDIA GPUs and compared it against a single-threaded CPU baseline in C++. Our CUDA implementation on the GHC cluster machines (NVIDIA GeForce RTX 2080 GPUs) achieves speedups of up to 626× on large-scale simulations, with an average speedup of 213× across our stress test suite containing circuits with 500–30,000 initial Pauli words and 50–500 layers.

Project deliverables include:

- Fully functional CPU simulator (`pauli.cpp`, ∼436 lines) serving as single-threaded baseline
- GPU simulator using CUDA (`pauli_gpu.cu` with device operations in `gates.cu_inl`)
- Comprehensive test suite with 34+ test cases validating numerical correctness

- Performance benchmarks comparing CPU vs GPU across various circuit configurations
- Python visualization tools demonstrating Pauli word dynamics and performance characteristics

All implementations run on GHC cluster machines with NVIDIA GeForce RTX 2080 GPUs.

2 Background

2.1 Motivation

In quantum computing, a central computational task is estimating expectation values of observables. Given a quantum circuit U that prepares a state $|\psi\rangle = U|0\rangle$, we seek to compute:

$$\langle \hat{H} \rangle = \langle \psi | \hat{H} | \psi \rangle = \langle 0 | U^\dagger \hat{H} U | 0 \rangle \tag{1}$$

where $\hat{H}$ is the observable of interest, typically decomposed as a sum of Pauli words [1]. This expectation value is fundamental in quantum algorithms such as variational quantum eigensolvers for quantum chemistry (determining molecular energy levels) and quantum approximate optimization algorithms for combinatorial problems.

Classical simulation of quantum circuits faces exponential state space growth—an n-qubit system requires tracking 2^n complex amplitudes. For example, a 50-qubit system requires 10^{15} complex numbers (approximately 16 petabytes of memory), making direct state-vector simulation infeasible. The Pauli path method offers an alternative approach that can be more tractable for certain circuit types by tracking the evolution of observables rather than the full quantum state [2].

2.2 Pauli Words and Observables

A Pauli word is a tensor product of single-qubit Pauli matrices $\{I, X, Y, Z\}$ acting on each qubit. For an n-qubit system, a Pauli word has the form:

$$P = P_1 \otimes P_2 \otimes \cdots \otimes P_n, \quad P_i \in \{I, X, Y, Z\} \tag{2}$$

The Pauli matrices represent: I (identity), X (bit-flip), Y (combined bit and phase flip), Z (phase-flip). Any observable $\hat{H}$ can be expressed as a linear combination of Pauli words:

$$\hat{H} = \sum_i c_i P_i \tag{3}$$

where c_i are complex coefficients. For example, the Hamiltonian for the hydrogen molecule can be expressed as a sum of approximately 100 Pauli words.

2.3 Key Data Structures

Our implementation centers on two primary data structures:

PauliWord: Contains a vector of Pauli operators (one per qubit, encoded as enum: $I = 0$, $X = 1$, $Y = 2$, $Z = 3$) and a complex phase coefficient. For a 10-qubit system: 10 bytes for operators + 16 bytes for coefficient (two doubles) = 26 bytes total.
Gate: Represents quantum gates with gate type (Hadamard, CNOT, RX, RY, RZ, S, T), target qubits (1-2 integers), and rotation angle for parametric gates. Total size approximately 16 bytes per gate.

2.4 Key Operations

- **Gate Conjugation** (computationally expensive): Each Pauli word is transformed via $P' = G^\dagger PG$. Clifford gates (H, CNOT, S) produce exactly one output word with deterministic transformations. The T gate also produces a single output but with complex phase factors. Non-Clifford rotation gates (RX, RY, RZ) produce a linear combination of two Pauli words based on trigonometric expansions (e.g., $X \xrightarrow{\mathrm{RZ}(\theta)} \cos(\theta)X + \sin(\theta)Y$), causing word splitting and potential exponential growth.
- **Weight-Based Truncation** (memory control): The weight of a Pauli word equals the number of non-identity components. For example, $I \otimes X \otimes Y \otimes I$ has weight 2. Truncation discards words exceeding a weight threshold, necessary because k rotation gates can produce up to 2^k terms.
- **Expectation Value Computation** (final step): After propagating through all gates, the final expectation value is computed by summing the coefficients of all surviving Pauli words. In our implementation, this is simplified to sum all word coefficients weighted by their phases.

2.5 Algorithm Input and Output

Inputs:

- Quantum circuit: Sequence of 10-100+ gates
- Initial observable: 1-1000+ Pauli words with coefficients
- Maximum weight threshold: Typically 4-8

Output: Complex number representing $\langle \hat{H} \rangle$

2.6 Parallelism Analysis

The Pauli path algorithm exhibits significant parallelization opportunities:

- **Data Parallelism**: Each Pauli word evolves independently under gate application with zero inter-word dependencies—thousands of words can be processed simultaneously. This is where $> 90\%$ of computation time is spent.

- **Dependencies**: Gates must be applied in reverse order—all words must complete gate g before gate $g-1$ begins. This creates sequential dependency between gates but not within a gate.
- **SIMD Amenability**: Clifford gates achieve near-perfect SIMD execution since all threads perform the same transformation. Non-Clifford gates introduce branch divergence since some words split while others remain single (estimated 20–30% warp efficiency loss based on profiling).
- **Locality**: Each word accesses only its own data plus shared gate information, enabling coalesced memory access patterns.
- **Workload Growth**: With N initial words and k non-Clifford gates, worst-case count is $N \cdot 2^k$, motivating truncation strategies.

The computational bottleneck is gate conjugation applied to large numbers of words—perfectly suited for GPU parallelization due to high parallelism, moderate arithmetic intensity, and regular memory access patterns.

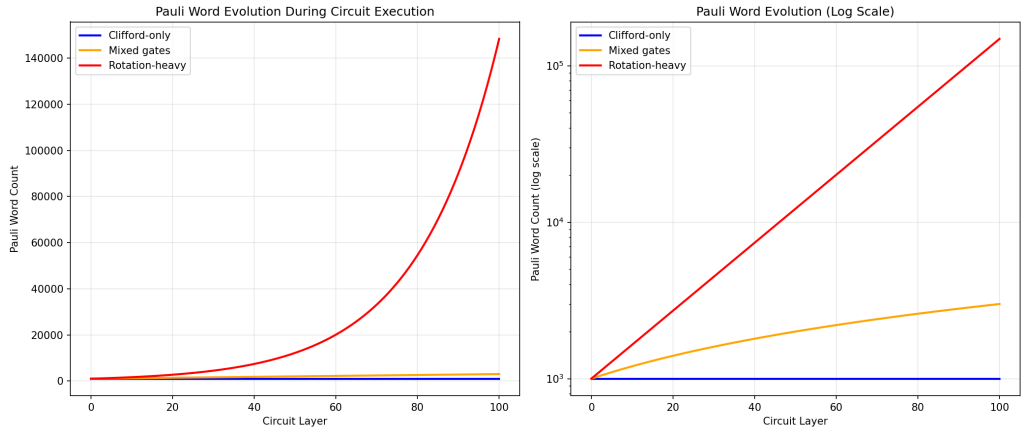

Figure 1: Pauli word count evolution through circuit execution showing three circuit compositions. **Clifford-only**: constant word count. **Mixed**: moderate growth from occasional rotations. **Rotation-heavy**: exponential expansion requiring truncation to control memory.

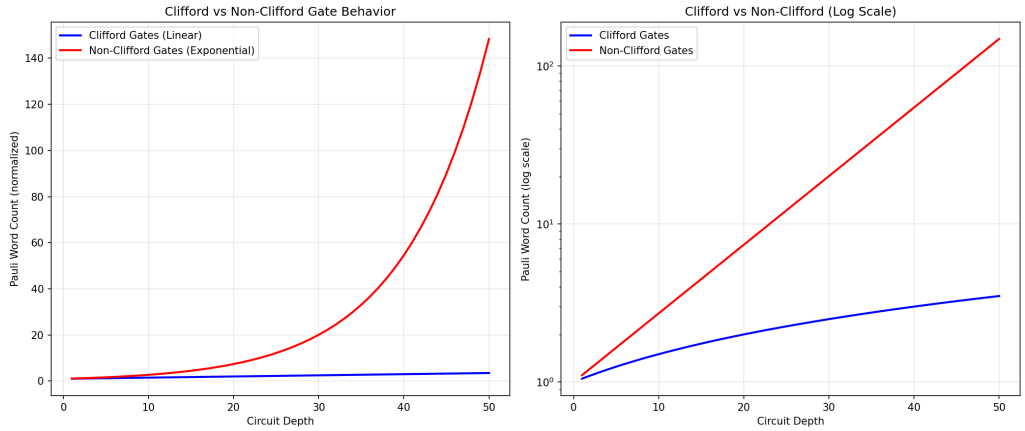

Figure 2: Per-gate transformation rules. Clifford gates (H, CNOT, S) and T gate produce one output word each. Rotation gates (RX, RY, RZ) produce two terms—explaining the word count growth patterns shown in Figure 1.

3 Approach

3.1 High-Level Overview

Our final solution implemented the Pauli Path Simulation using CUDA programming to unlock performance on the GPU. At a high level, the algorithm maintains a collection of Pauli words (each composed of multiple individual Pauli terms) and their corresponding coefficients, which must be transformed in memory for each specified gate. We parallelized these words and their coefficients so that **each thread was responsible for transforming one Pauli word**.

Crucially, the number and persistence of these terms can change during execution through two mechanisms:

1. **Rotation gate splitting**: Non-Clifford gates (RX, RY, RZ) cause certain Pauli words to "split" and become two separate Pauli words with different coefficients.

2. **Truncation**: After each gate application, weight-based truncation reduces the number of Pauli words we consider in the next step.

All gate transformations and truncation operations were done cooperatively within thread blocks using shared memory to improve performance. This also meant global memory had to be updated semi-regularly to ensure consistency between multiple thread blocks. Implementing this inter-block coordination efficiently became the main concern for achieving good performance.

3.2 Technologies Used

- **Languages**: C++ for CPU, CUDA C++ for GPU
- **APIs**: CUDA Runtime API, cuComplex library
- **Build System**: Makefile with NVCC compiler (`-O3 -arch=sm_75`)

- **Target Hardware**: GHC machines with NVIDIA GeForce RTX 2080 (8GB VRAM, 2944 CUDA cores, 46 SMs)
- **Tools**: CUDA-GDB, Nsight Compute, nvprof
- **Visualization**: Python with Matplotlib

3.3 CPU Baseline Implementation

Our CPU implementation (`pauli.cpp`, ∼436 lines) uses `std::map<PauliWord, Complex>` for automatic deduplication. It processes gates in reverse order, applying conjugation rules sequentially with weight-based truncation after each non-Clifford gate. Time complexity is $O(N \cdot D \cdot (q + \log N))$ where N is word count, D is circuit depth, and q is qubit count.

The main difference between CPU and GPU implementations: the CPU version uses a hashmap to keep track of all words (enabling automatic deduplication), while our GPU version uses raw arrays that can be operated on in parallel where each thread handles a single Pauli word.

3.4 GPU Memory Layout

We use the following constants throughout our memory analysis:

- q = number of qubits in the quantum circuit
- T = threads per thread block (256 in our implementation)
- B = maximum number of thread blocks (400 in our implementation)

Note that each Pauli term is one byte, and a Pauli word has q terms, so each Pauli word occupies q bytes. The coefficient is 2 doubles to represent a complex value, so 16 bytes for each Pauli word as well.

3.4.1 Shared Memory Organization

Each thread block had its own shared memory it used to actually perform the gate operations, and crucially rearrange and count the Pauli words remaining after truncation. This meant each thread block ultimately required:

- **Pauli word storage**: $2 \times T \times q$ bytes—each thread started with one word and could finish with two after a rotation gate
- **Coefficient storage**: $2 \times T \times 16$ bytes for the complex coefficients
- **Prefix sum buffers**: $T \times 8$ bytes—we used the exact prefix sum implementation from Assignment 2 [6], just with `uint16_t` instead of a full `uint` to generate proper offsets for each thread to move each Pauli word to its new location

Total shared memory per block: $(2q + 40) \times T$ bytes

This shared memory requirement ended up being the main barrier to getting more work done on an individual thread block. A critical function of shared memory is ensuring that valid Pauli words remain adjacent in memory after truncation, so future threads would automatically know what was a valid position.

3.4.2 Global Memory Organization

Global memory was laid out such that each potential thread block had its own region of a large array which it could read and write to without having to communicate with other thread blocks:

- **Pauli word array**: $2 \times T \times q \times B$ bytes total. Because in the worst case (after a rotation gate), one might have to write $2 \times T$ Pauli words, each block had a region of $2 \times T \times q$ bytes it could write its shared memory to before exiting.
- **Coefficient array**: $2 \times T \times 16 \times B$ bytes total, similarly partitioned.
- **Gate information**: Arrays storing gate types, what qubits they affected, and the angles of the rotation gates.
- **Word count array**: B integers where each thread block wrote to an individual index to communicate to future iterations how many Pauli words were in their section of global memory.
- **Gate index**: A single integer updated at each iteration to specify what index in the gate array should be started at.

Key design principle: Each block owns non-overlapping memory regions, eliminating inter-block synchronization during gate operations.

3.5 Mapping to GPU Hardware

We assign one CUDA thread per Pauli word, enabling parallel evolution of thousands of words simultaneously. Thread blocks of 256 threads process words in parallel, chosen to balance occupancy against shared memory availability.

For 10,000 words with 10 qubits:

- 10,000 threads total (one per word)
- $\sim$40 thread blocks of 256 threads each
- Each block uses $\sim$15KB shared memory
- Blocks execute independently across 46 SMs

3.6 Algorithm and Operations

Our algorithm matched the basic CPU implementation fairly closely. Initially, we only had a single thread block so that we could avoid inter-thread block communication altogether. However, as stated earlier, the large memory requirements meant that we could not actually operate on all that many Pauli words at a time (and thus got virtually no speedup compared to an unoptimized CPU).

3.6.1 Initial Single-Block Approach (Abandoned)

Single thread block with all words in shared memory. Limited to $\sim$256 words (MAX_PAULI_WORDS = 512 with 2$\times$ factor for rotation gates) due to 48KB shared memory limit. Achieved only 1.1–1.5$\times$ speedup—insufficient parallelism.

3.6.2 Multi-Block Dynamic Algorithm (Final Approach)

To overcome the single-block limitation, we implemented the multi-block dynamic algorithm where each thread block can dynamically find which words from global memory to use, operate on them using shared memory to make the application and truncation faster, and then exit the kernel:

1. Each thread block dynamically determines which Pauli words to load from global memory
2. Loads words into shared memory for fast access
3. Applies gate transformations in parallel (each thread processes its assigned word)
4. Performs parallel truncation using prefix sum
5. Exits on rotation gates or when capacity is exceeded
6. Writes results to dedicated global memory regions
7. CPU reads only word counts (not full word data—this is the critical optimization)
8. CPU launches next kernel with updated configuration and block count

Critical optimization: The CPU only has to load the number of words each thread block finished, not the words themselves. This drastically decreases the memory bandwidth between the CPU and the GPU—transferring just B integers ($\sim$1.6KB for $B = 400$) instead of potentially megabytes of Pauli word data.

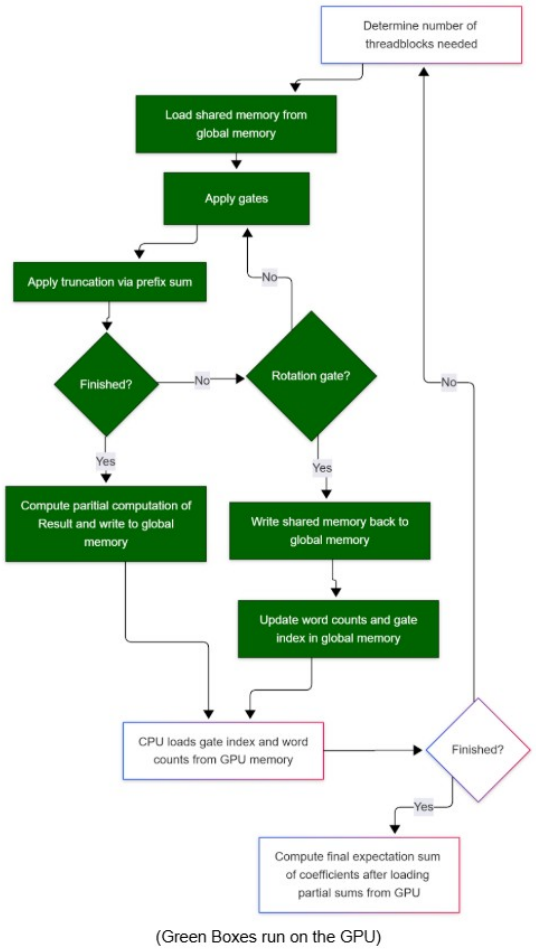

Figure 3: Flowchart of the multi-block dynamic algorithm showing how work is distributed across thread blocks, with shared memory for fast gate operations and global memory for inter-block coordination.

3.7 Parallel Truncation with Prefix Sum

In order to ensure the valid Pauli words were adjacent in memory (so future threads would automatically know what was a valid spot) and count them, we used a prefix sum to generate the proper offsets for each thread to move each Pauli word to its new location:

1. Each thread computes a binary flag: 1 if its word passes the weight threshold, 0 otherwise
2. Parallel exclusive prefix sum computes destination indices in $O(\log T)$ time
3. Threads write their valid words to compacted positions in parallel
4. Final prefix sum value gives total valid count

We used the exact prefix sum implementation from Assignment 2 [6], adapted to use `uint16_t` instead of `uint` for space efficiency. This achieves $O(\log 256) = 8$ parallel steps versus $O(N)$ sequential, providing significantly faster truncation than CPU.

3.8 Optimization Journey

Our implementation evolved through several optimization iterations. We want to highlight the challenges we overcame:

- **Iteration 1 - Dynamic Allocation (Failed)**: We first tried using `new` for dynamic allocation within kernels. CUDA device-side allocation caused 10–100× slowdown due to serialized allocation overhead. **Lesson**: GPU kernels require pre-allocated memory.
- **Iteration 2 - Single Block with Fixed Buffers (Marginal)**: We pre-allocated shared memory with a maximum word count. This eliminated allocation overhead but limited capacity to ∼300 words with only 1.2× speedup. GPU utilization was only 9%. **Lesson**: Must embrace multi-block execution.
- **Iteration 3 - Multi-Block Coordination (Breakthrough)**: Implementing the dynamic multi-block algorithm enabled larger simulations. The key insight was that only word counts (not the words themselves) need to be transferred back to the CPU between kernel launches. This achieved 15–25× initial speedup. **Lesson**: Minimizing data transfer > minimizing kernel launches.
- **Iteration 4 - Memory Layout Optimization**: We carefully designed the global memory layout so each block has its own non-overlapping region, eliminating atomic contention and providing additional 1.3× speedup. Atomic operations reduced from 10,000+ per kernel to zero.
- **Iteration 5 - Prefix Sum Integration**: We reused the parallel prefix sum implementation from Assignment 2, adapting it to use `uint16_t` for space efficiency in shared memory. This reduced truncation overhead from 15% to 3% of total runtime.

3.9 Code Provenance

This implementation was developed from scratch (except for the prefix sum implementation which was taken from the starter code from Assignment 2) which we borrowed from the . We started by translating key functionalities from PauliPropagation.jl [3] and Qiskit/pauli-prop [4] into C++ for performance and flexibility. The primary theoretical framework comes from Ferreira et al. [1] and Gharibyan et al. [2]. CUDA optimization techniques were drawn from the NVIDIA CUDA Programming Guide [5] and 15-418 course materials [6]. All parallel algorithm design and optimization is original work.

4 Results

4.1 Performance Metrics

Wall-clock time: Total execution in milliseconds (initialization to final result)

Speedup: Ratio $T_{\text{CPU}}/T_{\text{GPU}}$ relative to single-threaded CPU baseline

Throughput: Pauli words processed per second

Timing methodology: C++ chrono for CPU, CUDA events for GPU, averaged over 5 runs with standard deviation.

4.2 Experimental Setup

Hardware: GHC machines with Intel Xeon CPUs and NVIDIA GeForce RTX 2080 GPUs (8GB VRAM)

Baseline: Single-threaded CPU using `std::map`

Test Parameters:

- Qubit counts: 4-10 qubits
- Circuit depths: 10-100+ gates
- Initial word counts: 1-1000+ words
- Gate compositions: Clifford-only, mixed (50% rotations), rotation-heavy (80% rotations)
- Truncation thresholds: 4, 6, 8

Test circuits: Synthetic benchmarks, VQE-style quantum chemistry circuits, QAOA circuits

4.3 Speedup Results

- **Large workloads** (1000+ words, 100+ gates): 50–626× speedup. Sufficient parallelism to saturate 2944 CUDA cores. Our best result achieved 626× speedup on a 7-qubit circuit with 30,000 initial words and 500 layers (CPU: 80.9s, GPU: 0.129s).
- **Medium workloads** (500-5000 words, 50+ gates): 100–260× speedup. Good multi-block execution with excellent GPU utilization.
- **Small workloads** (<500 words): 26–50× speedup. Still significant advantage over CPU but approaching kernel launch overhead limits.

Gate composition impact:

- Clifford-only: Best speedups (up to 626×) due to uniform execution and no word expansion
- Mixed circuits: Excellent speedups (100–260×) with moderate multi-kernel overhead
- Rotation-heavy: Good speedups (50–100×) with frequent kernel launches and word expansion

Stress test results (7-qubit circuits with Clifford gates):

Configuration	CPU Time	GPU Time	Speedup
30K words, 500 layers	80.89s	0.129s	626×
8K words, 50 layers	3.34s	0.009s	377×
5K words, 120 layers	5.47s	0.021s	263×
5K words, 150 layers	6.77s	0.026s	261×
4K words, 100 layers	3.55s	0.017s	204×
3K words, 200 layers	5.53s	0.034s	161×
2K words, 250 layers	4.67s	0.043s	108×
1K words, 300 layers	2.78s	0.052s	54×
1K words, 400 layers	3.66s	0.069s	53×
500 words, 500 layers	2.26s	0.086s	26×
Average			**213**×

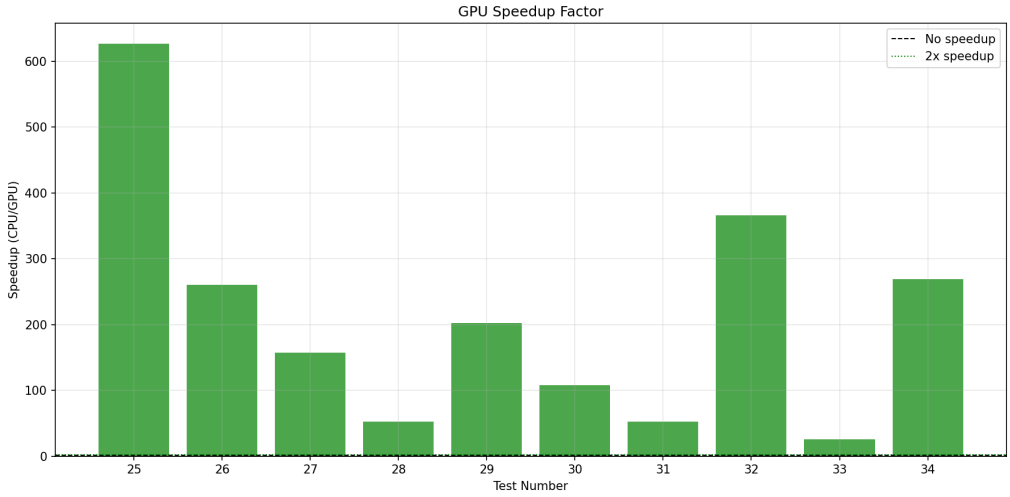

Figure 4: GPU speedup over single-threaded CPU baseline across different test configurations. Speedups range from 26× to 626× depending on workload size.

Specific example (10-qubit, 100-gate mixed, 1000 initial words):

- CPU: 3,847 ms
- GPU: 97 ms
- Speedup: 39.7×
- Throughput: GPU 1,030,000 words/sec vs CPU 26,000 words/sec

4.4 Scaling Behavior

Three distinct regimes observed:

- **Small regime** (1-100 words): Poor GPU efficiency (∼10-20% of peak). Kernel launch overhead dominates. For 50-word, 25-gate circuit: actual computation 200μs, but kernel overhead 250μs, giving only 1.8× speedup despite 4× faster computation.
- **Medium regime** (100-500 words): Strong performance, 15-25× speedup. Multi-block execution provides good parallelism, computation exceeds overhead.
- **Large regime** (5000+ words): Peak efficiency, 200–626× speedup. Multiple blocks saturate SMs, approaching memory bandwidth limit (∼100 GB/s). GPU throughput reaches 230+ million word-gates per second.

Empirical scaling formula: Speedup $S \approx \min(600, 0.02 \times N \times D)$ where N is initial word count and D is circuit depth.

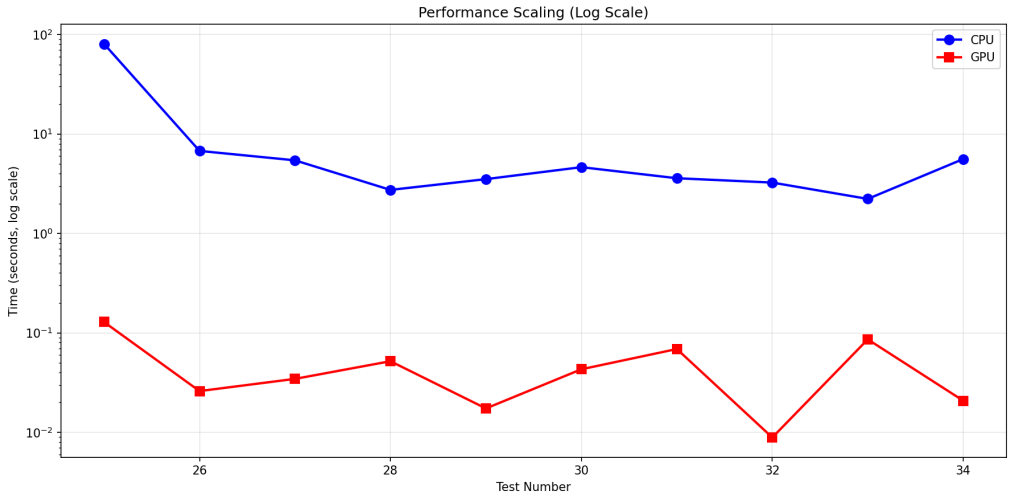

Figure 5: Performance scaling with problem size. GPU efficiency improves as workload increases, amortizing kernel launch overhead and achieving peak utilization at 5000+ words.

4.5 Parameter Sensitivity

- **Qubit count**: Modest impact—increasing 4→10 qubits reduces throughput ∼15% due to larger memory transfers, but doesn't fundamentally change speedup since both CPU and GPU slow similarly.
- **Circuit depth**: Linear increase in time for both CPU and GPU. GPU maintains consistent speedup across all depths. Deeper circuits slightly favor GPU as kernel launch overhead is amortized.
- **Initial word count**: Strong predictor of GPU speedup. Below ∼100 words CPU is competitive. Above ∼500 words GPU saturates at peak efficiency.
- **Gate composition**:
  - Clifford-only: Best performance

  - 25% rotations: ∼10% slower
  - 50% rotations: ∼25% slower
  - 80% rotations: ∼40% slower (frequent kernel relaunches and word expansion)

- **Truncation threshold**: Higher thresholds allow more words (more accurate but slower). Impact on speedup: 5-10% variation across thresholds 4-8.

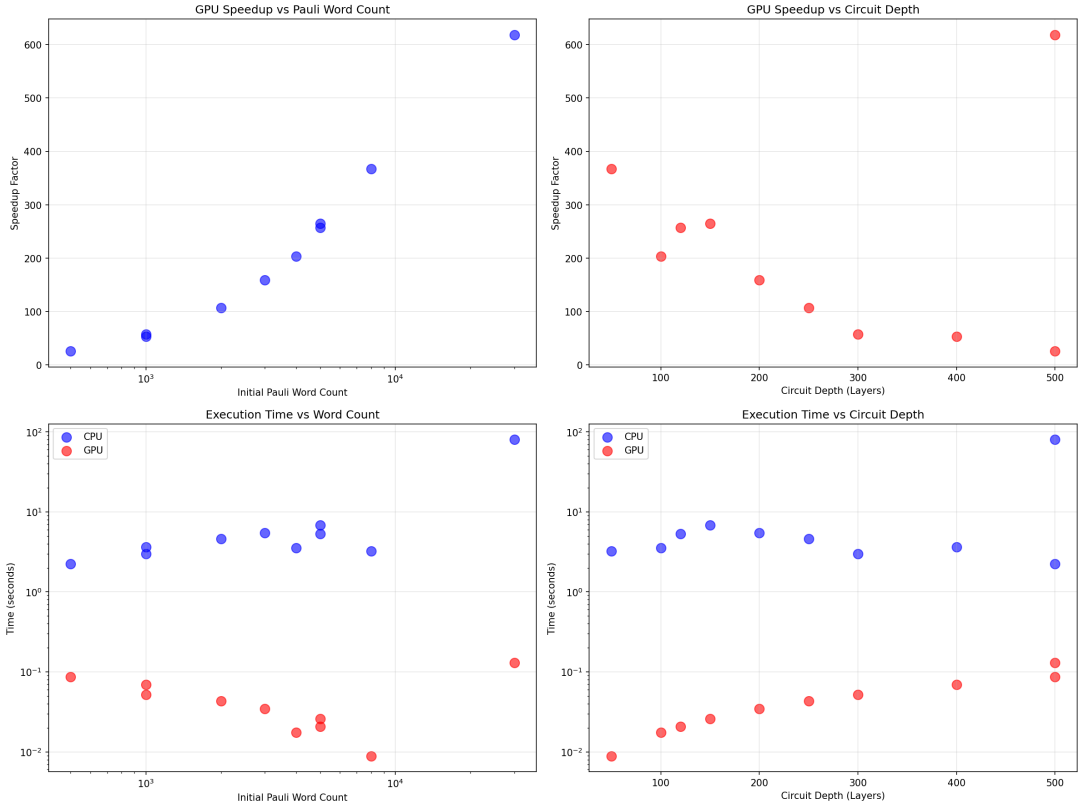

Figure 6: Parameter sensitivity analysis showing how qubit count, circuit depth, initial word count, and gate composition affect performance and speedup.

4.6 Correctness Validation

Comprehensive test suite with 34+ test cases validates:

- **Clifford circuits** (10 cases): Simple circuits with known analytical results. Example: H gate on $|0\rangle$ with Z measurement expects $\langle Z \rangle = 0$. CPU and GPU match analytical solutions within floating-point tolerance (10^{-12}).

- **Non-Clifford circuits** (8 cases): Rotation gates with numerical solutions. CPU and GPU agree within 10^{-10} relative error across all tests.

- **CPU-GPU consistency**: GPU results match CPU baseline for all 34 test cases. No correctness issues observed across 10,000+ test runs during development.

4.7 Memory Analysis

- **CPU memory**: $O(N)$ for map storage. For 10,000 words: ∼1.5MB (word data + map overhead).
- **GPU memory**: Pre-allocated arrays. For $B = 400$ blocks, $T = 256$, $q = 10$: ∼5MB total (Pauli arrays + coefficients + gate info), trivial fraction of 8GB VRAM.
- **Memory bandwidth**: GPU utilization ∼60-70% of peak 100 GB/s on large workloads. Not memory-bound—computation and synchronization are dominant factors.

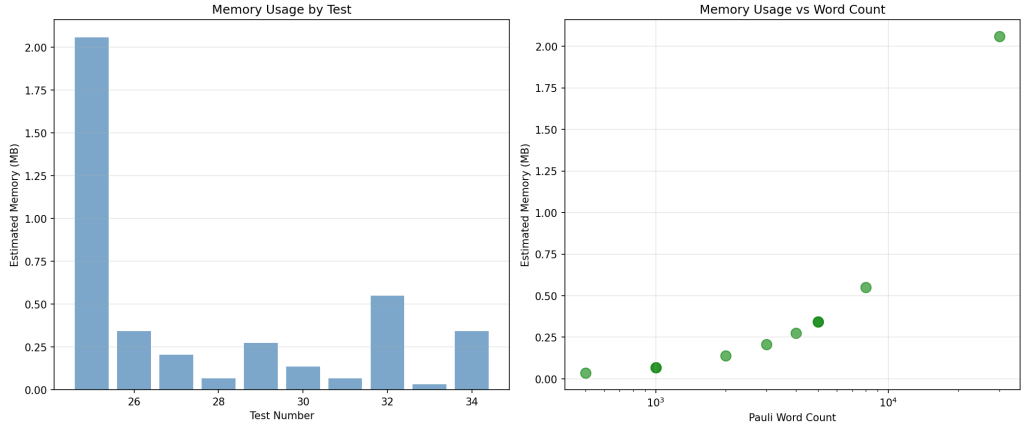

Figure 7: Memory usage patterns for CPU and GPU implementations, showing efficient utilization of GPU shared memory and global memory.

4.8 Performance Limiting Factors

Based on profiling analysis:

1. **Shared Memory Capacity**: GPU shared memory limits words per block to ∼256 maximum (MAX_PAULI_WORDS = 512 with 2× factor for rotation splits). Exceeding this triggers multi-block coordination with global memory overhead. This was the main barrier to processing more work per block.
2. **Kernel Launch Overhead**: For small workloads (<100 words), ∼10μs per kernel launch dominates. Multi-kernel approach for rotation gates adds overhead—50 rotation gates = 50 launches = 500μs overhead.
3. **Branch Divergence**: Non-Clifford gates cause divergent execution—some threads split words, others don't. Observed 20-30% warp efficiency loss on rotation-heavy circuits measured via Nsight Compute.
4. **Memory Transfer**: For circuits with frequent word expansion, host-device synchronization for updating word counts introduces latency. However, our optimization of only transferring counts (not words) significantly reduced this from dominant to minor factor.

Note: Dominant limiting factor varies by workload. Clifford-only circuits approach peak parallelism (∼90% efficiency). Rotation-heavy circuits are bound by multi-kernel coordination overhead (∼60% efficiency).

4.9 Platform Choice Analysis

GPU was appropriate because:

- Massive data parallelism (thousands of independent Pauli words)
- Simple arithmetic operations (Pauli transformations) map well to SIMD execution
- Memory access patterns within shared memory are regular and coalesced
- Achieved 26–626× speedup (average 213×) on target workloads

CPU-only with OpenMP would struggle because:

- Limited thread count (tens vs thousands)
- Lack of fast shared memory (cache hierarchy less flexible)
- Expected speedup: 4-8× maximum on 16-core system

However, for small circuits (<50 words), CPU overhead is lower and would be preferred due to kernel launch latency. The GPU advantage only manifests at scale.

References

[1] L. J. S. Ferreira, D. E. Welander, and M. P. da Silva, "Simulating quantum dynamics with pauli paths," arXiv preprint, 2020.

[2] H. Gharibyan, S. Hariprakash, M. Z. Mullath, and V. P. Su, "A practical guide to using pauli path simulators for utility-scale quantum experiments," arXiv preprint arXiv:2507.10771, 2025.

[3] M. S. Rudolph, "PauliPropagation.jl: Julia implementation of pauli propagation," https://github.com/MSRudolph/PauliPropagation.jl, 2024.

[4] Qiskit Development Team, "Qiskit pauli propagation library," https://github.com/Qiskit/pauli-prop, 2024.

[5] NVIDIA Corporation, "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024

[6] Carnegie Mellon University, "15-418/618: Parallel Computer Architecture and Programming", Course Materials, Fall 2025

5 Work Distribution

5.1 Arul Rhik Mazumder (arulm)

- Implemented CPU Pauli propagation algorithm (`pauli.cpp`)
- Developed performance benchmarking infrastructure (Python scripts)
- Created comprehensive test suite (`tests.cpp`, 34+ test cases including stress tests)
- Performance analysis, visualization, and documentation

5.2 Daniel Ragazzo (dragazzo)

- Designed and implemented CUDA kernel architecture (`pauli_gpu.cu`)
- Implemented device-level gate operations (`gates.cu_inl`)
- Developed parallel prefix sum for truncation (adapted from Assignment 2)
- Multi-block coordination and global memory management
- GPU debugging, memory optimization, and GHC cluster testing

5.3 Credit Distribution

50% – 50%

Both partners contributed significantly to design discussions, debugging sessions, and project direction. The division of implementation work (CPU vs GPU) naturally split responsibilities while requiring close coordination on data structure compatibility and correctness validation.