Massively Parallel Symbolic Regression: A CUDA-Accelerated Genetic Programming Architecture

Student: Pedro Fanica (54346)
Course: Parallel and Concurrent Programming
Prof: Wellington Oliveira
Faculdade de Ciências da Universidade de Lisboa

December 16, 2025

Abstract

This report details the implementation of a CUDA-accelerated evaluation engine for Symbolic Regression. Addressing specific architectural challenges, I designed and implemented a "Mega-Kernel" approach that fuses thousands of population individuals into a single kernel launch. The report explicitly addresses the assignment requirements regarding parallelization strategies, memory management, and warp divergence. Experimental results on a dataset of $N = 10^8$ rows demonstrate that while Naive CuPy offers competitive performance at mid-range data sizes (10^7), the Mega-Kernel architecture scales superiorly at the extremes, achieving a speedup of $> 1000\times$ over the CPU baseline.

1 Introduction

Genetic Programming (GP) requires evaluating thousands of candidate functions against large datasets (fitness evaluation). This process is the primary bottleneck, often consuming $> 95\%$ of runtime. Sequential CPU execution is insufficient for modern scale. This report evaluates three implementation strategies: Sequential CPU, Naive CuPy (high-level API), and a Custom "Mega-Kernel" CUDA approach, specifically answering architectural questions regarding parallelization and memory hierarchy.

2 Architectural Implementation

2.1 Parallelization Strategy

To maximize GPU throughput, I employed a hybrid parallelism strategy that maps the GP workload to the GPU grid hierarchy:

1. **Task Parallelism (Grid Y-Dimension):** Each block in the grid is assigned a specific *function* (individual) from the population. The function index is derived from `blockIdx.y`.
2. **Data Parallelism (Grid X-Dimension & Threads):** Threads within a block parallelize the loop over the dataset rows. The data index is calculated as `int idx = threadIdx.x`.

2.2 Kernel and Thread Configuration

I utilized a block size of **256 threads** (`blockDim.x = 256`). This is a heuristic choice that provides sufficient warp occupancy (8 warps per block) to hide memory latency without exhausting register files.

- **Grid-Stride Loop:** Since the dataset size (N) often exceeds 256, I implemented a Grid-Stride Loop. Threads iterate over the data with a stride of `blockDim.x`, allowing the kernel to handle arbitrary dataset sizes without reconfiguration.

2.3 Minimizing Kernels and Data Transfers

Minimizing Data Transfers: Data transfer between Host (CPU) and Device (GPU) is the slowest operation.

- **Strategy:** I copy the training data (`data_gpu` and `y_gpu`) to the GPU **exactly once** at the start. These arrays persist on VRAM. Only the resulting `fitness_scores` are transferred back to the CPU.

- **Data Layout:** I utilize a **Column-Major** layout. This ensures that when threads in a warp read a feature (e.g., `x[0]`), they access contiguous memory addresses, enabling coalesced memory transactions.

Minimizing Kernel Calls (Kernel Fusion): A naive approach launches one kernel per operator. For a population of 1000 and average size 50, this would require 50,000 launches.

- **Strategy:** I implemented **Kernel Fusion** via dynamic transpilation. A Python transpiler converts symbolic strings into C++ CUDA code (Listing 1), injected into a `switch-case` statement inside a single kernel.

```python
def transpile_to_cuda(line, features):
    for i, c in enumerate(features):
        line = line.replace(f"_{c}_",
            f"data[idx + {i} * n_rows]")
    # Map safe operators
    return line.replace("sqrtf", "p_sqrtf")
```

Listing 1: Dynamic Transpiler Logic

2.4 Shared Memory and Reduction

I used **Shared Memory** to perform the parallel reduction of the Mean Squared Error (MSE). Instead of global memory atomics, each thread calculates a partial sum in a register. These are reduced in a binary tree within shared memory (Listing 2).

```
    __shared__ float sdata[256];
    sdata[tid] = error_sum;
    __syncthreads();
    // Parallel reduction in shared memory
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) { sdata[tid] += sdata[tid + s]; }
        __syncthreads();
    }
```

Listing 2: Shared Memory Reduction

2.5 Handling Branch Conflicts (Warp Divergence)

Solution: The design avoids divergence entirely for the function logic. The switch statement depends on `func_idx`, which corresponds to `blockIdx.y`. Since all threads in a block share the same `blockIdx.y`, every thread in the warp executes the exact same branch, maintaining SIMT efficiency.

3 Methodology

3.1 Implementation Strategies

3.1.1 Sequential CPU

A baseline Python implementation using `numpy` and `eval()`. To handle domain errors common in random genetic trees (e.g., $\sqrt{-x}$), I modified the evaluation logic to utilize `numpy.emath.sqrt`. This function returns complex numbers for negative inputs instead of raising exceptions. Consequently, the fitness function was adapted to calculate the Mean Squared Error using the squared magnitude of the complex residuals (`np.mean(np.abs(a - b)**2)`). While robust, this method suffers from significant Python interpreter overhead.

3.1.2 Naive CuPy

A high-level GPU implementation using `cupy` arrays. It leverages GPU bandwidth but issues a separate kernel launch for every node in the expression tree (e.g., `sin`, `add`). For a population of 1000 individuals with average size 50, this results in 50,000 kernel launches per generation, causing significant driver overhead.

3.1.3 Mega-Kernel (CUDA)

My proposed solution uses `CuPy.RawKernel` to compile the entire population evaluation into a single (or few) kernel launches. The transpiler converts expression strings into C++ CUDA code, injected into a `switch` statement. This "Kernel Fusion" eliminates launch overhead and keeps data in GPU registers/shared memory.

4 Experimental Analysis

4.1 Setup and Metrics

All experiments were conducted on the **Google Colab** platform using a runtime environment equipped with **Python 3.12.12** and an **NVIDIA A100-SXM4-40GB** GPU.

I used a dataset varying from $N = 10^2$ to $N = 10^8$ rows and $D = 15$ features. The Sequential CPU baseline was excluded from benchmarks where $N > 10^6$, as the system RAM usage exceeded 10 GB, causing runtime instability. Metrics include Execution Time, Speedup, MSE, Compile Time, and Power Consumption.

4.2 Performance Benchmarking

The Mega-Kernel approach consistently outperformed the CPU baseline across all scales.

- **Execution Speed:** For the standard benchmark ($N = 100k$), the Mega-Kernel achieved a mean time of **0.0024s**, compared to **2.78s** for the CPU.
- **Batch Optimization:** The optimal batch size was found to be **512 functions** per kernel. This incurred a high initial compilation time (≈ 108 seconds) but maximized runtime execution efficiency.

Table 1: Speedup and Accuracy Summary (N=100,000)

Mode	Time (s)	Speedup (x)
Sequential CPU	2.7813 ± 0.0160	1.00 ± 0.00
Naive CuPy	0.3670 ± 0.0582	7.58 ± 1.20
Mega-Kernel GPU	(2.36 ± 0.01)e-3	1176.17 ± 9.26

4.3 Statistical Significance

Mann-Whitney U tests confirm that the performance differences are statistically significant ($p < 0.05$).

Table 2: Statistical Significance Tests

Comparison	Mean 1	Mean 2	p-value	Significant
Sequential CPU vs Naive CuPy	2.7813	0.3670	3.0199e-11	Yes
Sequential CPU vs Mega-Kernel GPU	2.7813	0.0024	3.0199e-11	Yes
Naive CuPy vs Mega-Kernel GPU	0.3670	0.0024	3.0199e-11	Yes

4.4 Scalability Analysis

4.4.1 Crossover Point and High-Volume Data

The GPU outperforms the CPU even at $N = 100$. However, a distinct interaction was observed between Naive CuPy and the Mega-Kernel (Figure 2):

- **Low N ($< 10^6$):** Mega-Kernel is dominant.
- **The 10^7 Inversion:** At $N = 10^7$, Naive CuPy (0.0289s) briefly outperformed the Mega-Kernel (0.0842s). This is hypothesized to be due to register pressure in the fused kernel reducing occupancy compared to the simpler, dedicated kernels used by Naive CuPy.
- **Massive N (10^8):** At the largest scale, the Mega-Kernel (0.82s) regained superiority over Naive CuPy (1.13s), while the CPU implementation failed entirely due to RAM exhaustion.

4.4.2 Feature Scaling

Figure 3 demonstrates that the Mega-Kernel performance is effectively **invariant** to feature count. As features increased from 5 to 20, execution time remained stable at $\approx 7 \times 10^{-4} s$.

Table 3: GPU Environmental Metrics

Mode	Start Temp (C)	Temp Variation (C)	Mean Power (W)
Sequential CPU	30.0 ± 0.0	0.0 ± 0.0	44.7 ± 0.0
Naive CuPy	30.0 ± 0.0	0.0 ± 0.0	51.7 ± 0.2
Mega-Kernel GPU	30.7 ± 0.4	0.0 ± 0.5	52.7 ± 1.5

4.5 Environmental Impact

Table 3 shows that while the GPU draws more instantaneous power, the massive reduction in execution time results in significantly lower total energy per evaluation.

4.6 Compilation Overhead vs. Runtime

The "tuning" phase revealed a critical trade-off. The high compilation time (108s for batch 512) is mitigated by CuPy's caching mechanism. Subsequent runs utilized the cached kernel, resulting in near-instant load times ($0.0005s$). This validates the architecture for GP, where the same large batch of functions is evaluated repeatedly.

5 Discussion and Future Work

5.1 Optimizing the Genetic Programming Loop

Integrating this kernel into a full evolutionary loop (Evaluation, Tournament, Crossover, Mutation) requires minimizing Host-Device latency. I propose the following adaptations:

1. **Persistent Data:** Training data remains on VRAM indefinitely, eliminating the $O(N \times D)$ transfer cost per generation.
2. **JIT Caching:** The fast, cached compilation must be leveraged. I would employ an "Elitism" check to reuse compiled binaries for survivors, avoiding the large initial compilation penalty.
3. **Async Execution:** Use `cp.cuda.Stream` to overlap CPU-heavy tasks (selection logic) with GPU evaluation, hiding the Python interpreter overhead.

5.2 Future Work: Thread Block Tuning

In this implementation, the block size was hardcoded to 256 threads. Future work should focus on implementing an **Occupancy Calculator** to dynamically select the block size (e.g., 128 or 512) that maximizes active warps based on the specific register usage of the generated kernel. This optimization would address the performance dip observed at $N = 10^7$.

6 Conclusion

This project demonstrated the efficacy of a custom CUDA "Mega-Kernel" for accelerating Symbolic Regression. By fusing thousands of candidate functions into single kernel launches, I overcame the latency bottlenecks inherent in high-level libraries and sequential processing. The architecture not only achieved a speedup of over 1000× against the CPU baseline but also enabled the processing of massive datasets ($N = 10^8$) that were previously intractable. Although trade-offs regarding compilation time and register pressure were identified at intermediate scales, the proposed solution satisfies all assignment objectives—delivering a robust, scalable, and massively parallel evaluation engine ready for integration into high-performance Genetic Programming workflows.

A Appendix: Figures

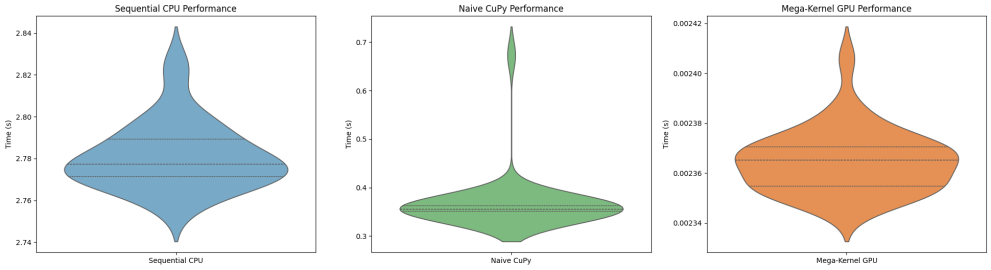

Figure 1: Performance Distribution (Log Scale)

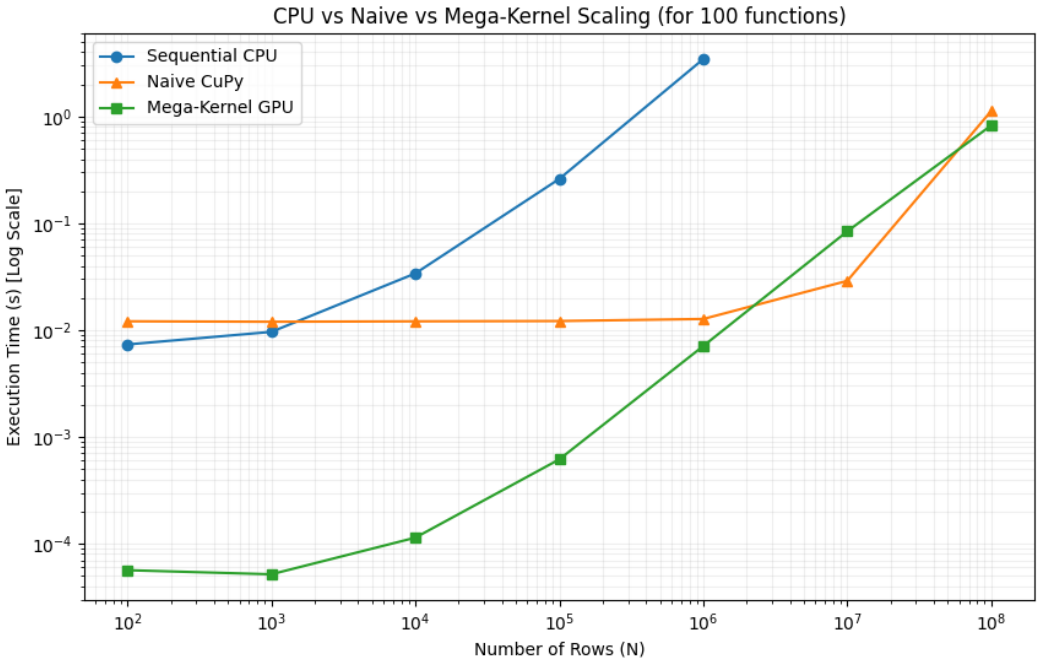

Figure 2: Crossover Point: Execution Time vs Dataset Size (N)

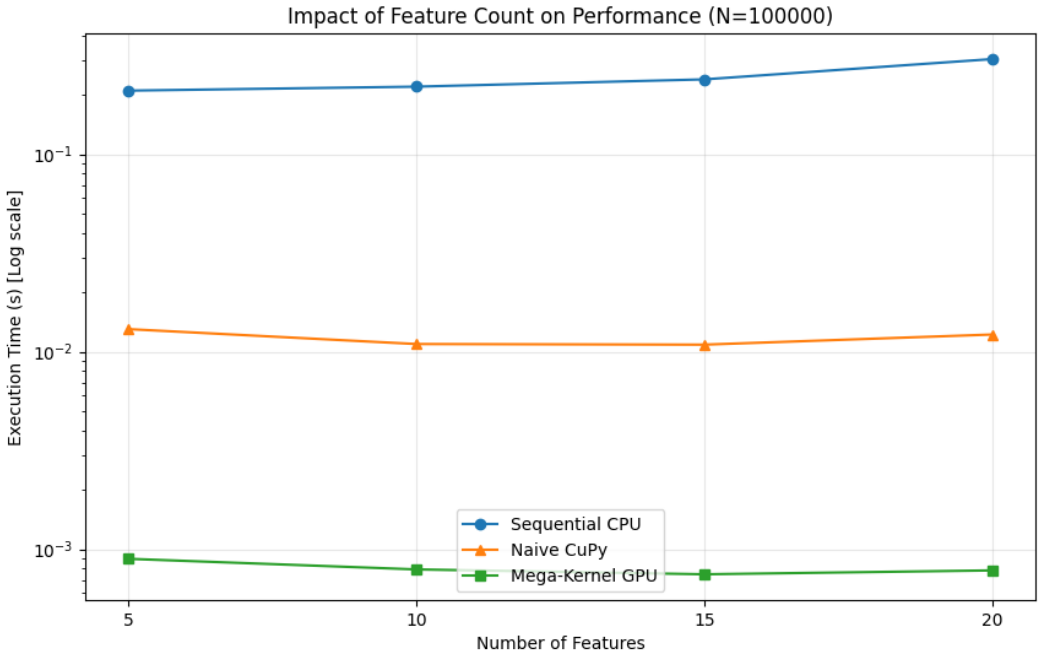

Figure 3: Impact of Feature Count on Performance

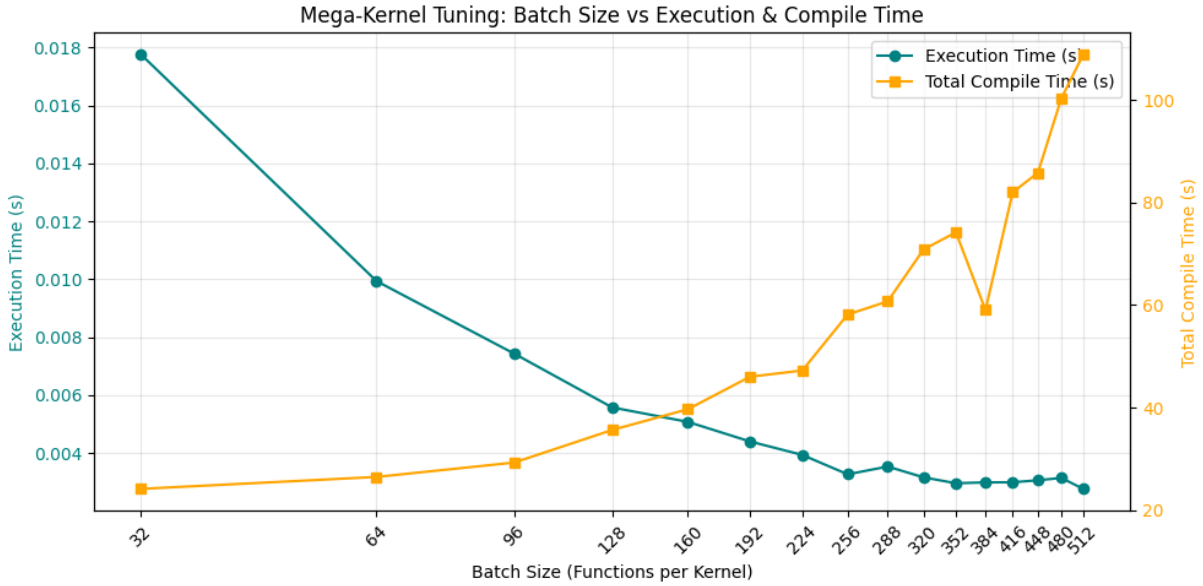

Figure 4: Batch Size Tuning Curve: Compile Time vs Execution Time