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

Research Report

December 16, 2025

Abstract

This report presents an architectural analysis and implementation strategy for accelerating Symbolic Regression, a subfield of Genetic Programming (GP), using Graphics Processing Units (GPUs). The objective was to design a high-throughput evaluation engine to minimize the Mean Square Error (MSE) of candidate mathematical expressions against a large-scale dataset ($N = 100,000$). We implemented a hybrid Python-CUDA approach using CuPy's `RawKernel` interface, achieving massive parallelism across both data rows and candidate functions. Our "Mega-Kernel" design incorporates dynamic runtime compilation, automated batch size tuning, and high-precision benchmarking. Results demonstrate that the GPU implementation significantly outperforms sequential CPU execution, with detailed analysis provided on the impact of dataset size (N) and feature dimensionality (D).

1 Introduction

1.1 The Evaluation Bottleneck

Genetic Programming (GP) distinguishes itself from other machine learning paradigms by evolving the model structure itself. In Symbolic Regression, the fitness of individuals is determined by their predictive accuracy against a target dataset. The computational cost is $O(P \times G \times N \times \bar{S})$, where P is population size, G generations, N rows, and $\bar{S}$ average tree size. With modern datasets often exceeding 10^5 rows and populations requiring 10^3 individuals, a single run can demand trillions of floating-point operations. Sequential CPU execution fails to scale, severely limiting search space exploration.

1.2 GPU Architecture Suitability

GPUs are theoretically optimal due to two dimensions of parallelism:

- **Data Parallelism:** A single candidate function must be evaluated across all N rows.
- **Task Parallelism:** Multiple distinct candidate functions (P) must be evaluated simultaneously.

However, the irregular nature of GP trees causes warp divergence, challenging Single Instruction, Multiple Threads (SIMT) architectures.

2 Methodology

2.1 Technology Selection

We compared three potential pathways: Numba, PyTorch, and CuPy.

2.1.1 Numba: The "Bad Stride" Pitfall

Numba offers convenient JIT compilation but suffers from memory access patterns if not managed manually. If a kernel is written naively where thread id accesses index id of a row-major array, consecutive threads access non-consecutive memory addresses. This forces the memory controller to service separate transactions, reducing effective bandwidth by up to 80%.

2.1.2 PyTorch: Kernel Launch Latency

PyTorch evaluates expressions like `sin(x) + cos(x)` by launching separate kernels for each operation. For a GP tree with 50 nodes, this results in 50 kernel launches per individual. With $P = 1000$, this triggers 50,000 launches per generation, causing massive driver overhead.

2.1.3 CuPy & RawKernel (Selected)

We selected CuPy with `RawKernel` as it allows dynamic compilation of C++ CUDA code strings. This enables "Kernel Fusion," where the entire expression tree is compiled into a single kernel, and provides direct control over memory layout.

2.2 Data Layout Optimization

To resolve "bad stride" issues, input data was transposed from Row-Major to **Column-Major (Structure of Arrays)**. In this layout, all values for feature A are contiguous. When a warp of 32 threads executes a grid-stride loop, Thread k reads Row k and Thread $k+1$ reads Row $k+1$, which are adjacent in memory. This results in coalesced memory transactions, maximizing bandwidth.

2.3 Reproducibility and Data Persistence

To ensure experiments are reproducible across sessions (e.g., Google Colab) while avoiding unnecessary regeneration of inputs, the implementation:

- Mounts Google Drive on Colab and stores artifacts under `MyDrive/Genetic_programming_GPU/data`; locally, files live under the repository's `data/`.
- Generates `data.csv` and `functions.txt` only once and records the parameters in `gen_meta.json` (`N`, `FEATURES`, `SEED`).
- Regenerates inputs only when these parameters change, otherwise reuses persisted files.
- Persists benchmark timings to `results.json` for later inclusion in reports.

3 Parallel Architecture Design

3.1 Hybrid "Mega-Kernel" Mapping

We adopted a 2D Grid/Block mapping to exploit both data and task parallelism:

- **Grid (Y-dim):** Maps to Population (Functions). `blockIdx.y` corresponds to the function index.
- **Blocks/Threads (X-dim):** Map to Data Rows. Threads utilize a Grid-Stride Loop to iterate over the dataset.

3.2 Dynamic Kernel Generation

A Python transpiler converts `functions.txt` strings into CUDA C++ code. The pipeline involves:

1. **Parsing:** Regex replaces mathematical tokens (e.g., `sinf`) and variable placeholders. Variables like `_j_` are mapped to specific offsets in the column-major input array (e.g., `data[idx + 9 * n_rows]`).
2. **Injection:** The parsed code is injected into a C++ `switch` statement inside the kernel.
3. **Dispatcher:** A main kernel switches between functions based on `blockIdx.y`.

3.3 Automated Batch Size Tuning

A critical challenge in the Mega-Kernel approach is balancing compilation cost against execution speed.

- **Too Large Batch:** Compiling a kernel with thousands of `case` statements takes excessive time and may hit compiler resource limits (register pressure).
- **Too Small Batch:** Increases the number of kernel launches, re-introducing overhead.

Our implementation includes an automated tuning loop that tests batch sizes (e.g., 32, 64, ..., 512) to find the optimal balance. It measures both `compile_time` and `execution_time` to select the configuration that minimizes total time-to-solution.

4 Implementation Details

4.1 Transpiler Implementation

The Python transpiler ensures that high-level symbolic expressions are converted into efficient CUDA C code.

```python
def transpile_to_cuda(line, features):
    # Replace variables _a_ -> data[idx + offset]
    for i, c in enumerate(features):
        # Map column char to array offset
        offset = f"{i} * n_rows"
        line = line.replace(f"_{c}_",
            f"data[idx + {offset}]")

    # Replace functions with CUDA intrinsics
    # Custom protected sqrt to handle negatives
    line = line.replace("sqrtf", "p_sqrtf")
    return line
```

Listing 1: Python Transpiler Function

4.2 CUDA Mega-Kernel

The core of the implementation is the Mega-Kernel, which evaluates multiple functions in a single launch. It uses Shared Memory for efficient parallel reduction of the Mean Squared Error.

```
__device__ inline float p_sqrtf(float x) {
    return sqrtf(fabsf(x));
}

extern "C" __global__
void evaluate_population(const float* data,
    const float* y_true, float* fitness_scores,
    int n_rows) {

    int func_idx = blockIdx.y; // Function ID
    int tid = threadIdx.x;
    float error_sum = 0.0f;

    // Grid-stride loop over data rows
    for (int idx = tid; idx < n_rows;
         idx += blockDim.x) {
        float y_pred = 0.0f;

        // Dynamic Switch-Case Dispatch
        switch(func_idx) {
            // Cases injected by Python
            case 0: y_pred = ...; break;
            case 1: y_pred = ...; break;
            // ...
        }

        float diff = y_pred - y_true[idx];
        error_sum += diff * diff;
    }

    // Block Reduction in Shared Memory
    __shared__ float sdata[256];
    sdata[tid] = error_sum;
    __syncthreads();

    // Binary Tree Reduction
    for (unsigned int s = blockDim.x / 2;
         s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // Write final MSE for this function
    if (tid == 0) {
        fitness_scores[func_idx + BLOCK_OFFSET] =
            sdata[0] / n_rows;
    }
}
```

Listing 2: CUDA Mega-Kernel Template

4.3 High-Precision Timing

To accurately measure the performance of sub-millisecond kernel executions, we replaced standard `time.time()` with `time.perf_counter()`. This ensures that the benchmarking results capture the true execution cost without the noise of system clock granularity.

5 Experimental Analysis

We conducted a comprehensive performance analysis covering three key dimensions.

5.1 Batch Size Optimization

We swept batch sizes from 32 to 512 functions per kernel.

- **Observation:** Larger batches reduce execution time by amortizing launch overhead but increase compilation time significantly.
- **Result:** The system automatically selects the batch size that offers the best trade-off (typically around 64-128 functions) for the final benchmark.

5.2 Crossover Point Analysis (Scaling N)

We analyzed the "Crossover Point"—the dataset size (N) at which the GPU outperforms the CPU.

- **Small N ($< 10^3$):** CPU is faster due to GPU data transfer and kernel launch latency.
- **Large N ($> 10^4$):** GPU throughput dominates.
- **Conclusion:** For modern datasets ($N = 10^5+$), GPU acceleration is essential.

5.3 Feature Scaling Analysis (Scaling D)

We investigated how the number of input features ($D \in \{5, 10, 15, 20\}$) impacts performance.

- **CPU:** Performance degrades linearly as expression complexity increases with more variables.
- **GPU:** Performance remains relatively stable due to high memory bandwidth and the column-major layout, which ensures efficient memory access regardless of feature count.

6 Results Summary

We benchmarked the implementations on a dataset with $N = 100,000$ rows.

Method	Time (s)	Speedup
Sequential CPU	≈ 145.0	1.0x
Naive CuPy	≈ 4.5	32.2x
CUDA Mega-Kernel	≈ 0.12	1210.0x

Table 1: Performance Comparison ($N = 10^5$)

As shown in Table **??**, the Mega-Kernel approach provides a dramatic speedup. The Naive CuPy implementation is faster than CPU but suffers from kernel launch overhead for each individual in the population. The Mega-Kernel eliminates this, reducing the cost to effectively a single kernel launch.

7 Future Work

7.1 Constant Optimization

Currently, the system evolves structure but not constants (e.g., $3.14 \times x$). Future work will integrate a local search (e.g., Gradient Descent or Hill Climbing) to optimize constants for each skeleton.

7.2 Multi-GPU Scaling

The "Mega-Kernel" design is naturally parallelizable across multiple GPUs. We plan to implement `cupy.cuda.nccl` to distribute the population across devices, allowing for larger populations ($P > 10,000$).

7.3 Advanced Genetic Operators

Moving Crossover and Mutation operations to the GPU would eliminate the remaining D2H transfers, keeping the entire evolutionary loop on the device.

8 Conclusion

The hybrid Python/CuPy/RawKernel architecture successfully accelerates Symbolic Regression. By optimizing memory layout (Column-Major) and using dynamic compilation (Mega-Kernel), we eliminate common bottlenecks like "bad stride" and kernel launch latency. This enables real-time evolution of complex models on consumer-grade GPUs, fulfilling the project's high-performance computing objectives.