Final Project - Accelerated Symbolic Regression

JINHA KIM, MIT CSAIL, USA
YOUNGHUN ROH, MIT CSAIL, USA

Symbolic Regression (SR) is a widely used tool for scientific discovery and AI interpretability, valued for distilling data into transparent mathematical models [7]. Recent advancements have introduced hardware-accelerated frameworks, including end-to-end CUDA systems like EvoGP [10] and Julia-based environments [2]. However, despite order-of-magnitudes speedups on evaluation, GPU acceleration is not yet widely adopted in state-of-the-art SR systems due to optimization challenges. Current implementations leave significant room for improvement, particularly regarding semantic-aware work efficiency and compilation overhead.

In this paper, we present a high-performance GPU evaluation library that addresses these bottlenecks through subtree caching and Just-In-Time compilation. We achieved 30% overall speedup in our benchmark, and 3x kernel speedup with JIT compilation. Our code is published in https://github.com/phi-jkim/symbolic_regression_gpu

1 Introduction

1.1 Problem Definition and Motivation

The central challenge of Symbolic Regression (SR) lies in discovering the appropriate symbolic expression, often referred to as the *structure*, which captures the underlying physical law of a dataset. Unlike linear regression where the form is fixed, SR involves a combinatorial search over a space of mathematical operators. Once a candidate structure is identified, finding the optimal constant coefficients can be achieved efficiently using standard continuous optimizers such as gradient descent or BFGS [1]. Therefore, the primary bottleneck in SR research focuses on efficiently navigating the discrete space of possible functional forms before the fitting coefficients are even known.

In the big picture, this project aims to improve the overall performance of SR algorithms by drastically increasing the throughput of their evaluation phase. Most current implementations, whether based on Neural Networks (NN) or Tree-based Genetic Programming (TGP), rely on generic evaluation libraries such as SymPy [4], PyTorch, or TensorFlow. While flexible, these libraries are not optimized for the highly irregular and random nature of SR workloads, where thousands of heterogeneous small trees must be evaluated against massive datasets. As expression complexity increases, these approaches suffer from linearly growing evaluation spans, limiting the ability to discover complex laws.

1.2 Historical Context and Existing Gaps

Historically, TGP has been the dominant method for generating structures [3]. Recent advancements have leveraged Deep Learning to guide this search, with works like *AI Feynman* [9], *Deep Symbolic Regression* [6], and *Phy-SO* [8] establishing new benchmarks. To address the computational demands of these approaches, hardware-aware frameworks like *EvoGP* [10] have emerged. EvoGP accelerates TGP by implementing the entire evolutionary pipeline on the GPU using a tensorized tree encoding and "hybrid parallelism"—parallelizing across both data and population dimensions. However, even with these advances, significant inefficiencies remain in the raw evaluation kernel. Regression candidates in a population often resemble previous candidates or share significant substructures, yet current systems re-evaluate every tree from scratch, ignoring this redundancy.

1.3 Proposed Approach/Objectives: Heterogeneous Optimization

To bridge this gap, we propose a high-performance evaluation library that focuses on three key objectives: subtree recycling, Just-In-Time Compilation and efficient heterogeneous engineering.

Authors' Contact Information: Jinha Kim, MIT CSAIL, USA, jinhakim@mit.edu; Younghun Roh, MIT CSAIL, USA, yhunroh@mit.edu.

Subtree Caching and CPU Coordination: Our most significant objective is to exploit the redundancy in SR populations. Since regression candidates consist of various primitive functions that are highly likely to resemble their parents or peers, we implement a caching mechanism for common sub-trees. Given that expression trees are highly irregular, we rely on the CPU for the preprocessing step: identifying, normalizing, and hashing sub-components. The CPU, with its superior branch prediction and low latency, is well-suited for this irregular "deduplication" task. It marks cached components so that the GPU kernel, which is optimized for bandwidth, can simply look up pre-computed results rather than re-calculating them. This heterogeneous split allows us to process each candidate structure once on the CPU, saving massive redundant computation over the dataset dimension on the GPU.

Dynamic Created Kernels: As expressions are represented as prefix encodings of tokens and values, EvoGP uses a stack based kernel that pushes and pops from the stack interleaved with compute. This results in dynamic indexing of the stack and also sequential dependencies between having to load and compute. We explore dynamically creating then compiling custom kernels during runtime to mitigate the use of iterating over a stack but hardcode the operations in custom CUDA kernel or alternatively as a PTX string that avoids the memory loading of tokens, branch instruction comparisons of token value against the operation constants. By having the CUDA kernel, and the PTX string first load all of the X datapoints then perform computes on the expressions we potentially promote ILP across compute within a single expression and across independent expressions, which further analysis is provided.

1.4 Contributions

In this paper, we present the following contributions to the field of accelerated symbolic regression:

1. **Semantic-Aware Subtree Caching:** We introduce a mechanism to detect and reuse common sub-expressions across the population, reducing redundant floating-point operations.
2. **JIT Compilation Analysis:** We provide a comparative analysis of runtime compilation (PTX to SASS, or C++ to SASS) versus manual stack-based evaluation, illustrating the trade-offs between compilation latency and execution throughput.
3. **Comprehensive Benchmarking:** We validate our system on the AI Feynman benchmark dataset, demonstrating performance gains over both CPU and GPU baseline implementations.

2 Methodology and Algorithm Design

In this section, we describe the evolution of our evaluation engine, starting from the standard CPU baseline to our optimized GPU architecture. We analyze the specific bottlenecks of stack-based interpretation on parallel hardware and propose two major strategies: heterogeneous subtree caching and runtime compilation.

2.1 Baseline CPU Implementation

To evaluate a symbolic expression, we utilize a standard prefix-order encoding. Each expression tree is flattened into two arrays: `tokens` (integers representing operators, variables, or constants) and `values` (floating-point constants or variable indices).

The evaluation follows a linear pass using a stack. When the iterator encounters a value producer (constant or variable), it pushes the value onto the stack. When it encounters an operator, it pops the required number of operands, computes the result, and pushes it back. Algorithm 1 illustrates this logic, which serves as the correctness reference for all subsequent implementations.

Algorithm 1 CPU Prefix Evaluation

Require: Integer array $tokens[0 \ldots N-1]$, Real array $values[0 \ldots N-1]$, Data row x
Ensure: Result of expression

```
 1: sp ← 0                                          ▷ Stack pointer
 2: for i ← N − 1 downto 0 do
 3:     tok ← tokens[i]
 4:     if tok > 0 then                             ▷ Operator
 5:         v1 ← POP(stk)
 6:         if tok < 10 then                        ▷ Binary Op
 7:             v2 ← POP(stk)
 8:             tmp ← APPLY(tok, v1, v2)
 9:         else                                    ▷ Unary Op
10:             tmp ← APPLY(tok, v1)
11:         PUSH(stk, tmp)
12:     else if tok = 0 then                        ▷ Constant
13:         PUSH(stk, values[i])
14:     else                                        ▷ Variable
15:         idx ← int(values[i])
16:         PUSH(stk, x[idx])
17: return stk[0]
```

2.2 Baseline GPU Implementation

Our GPU baseline adapts the CPU logic using a "Hybrid Parallelism" strategy similar to EvoGP. We map the workload as follows:

- **Grid Dimension (Blocks):** Each thread block is assigned a subset of the population (candidate expressions).
- **Block Dimension (Threads):** Each thread within a block evaluates its assigned expression on a specific subset of data points.

Crucially, to avoid branch divergence, all threads in a warp evaluate the *same* expression structure simultaneously, differing only in the data point index (x) they process. The `tokens` and `values` arrays are stored in global memory and accessed via coalesced loads.

2.3 Bottleneck Analysis: The Local Memory Stack

Despite the parallelism, microbenchmarks revealed that the evaluation kernel was bounded by memory latency rather than arithmetic throughput.

Analysis using `ptxas` showed that the compiler generated a 272-byte stack frame for the evaluation kernel. Because the stack pointer (sp) is dynamically indexed based on the token stream, the GPU compiler cannot map the stack array to registers. Instead, it places the stack in **Local Memory**, which is backed by the L1/L2 cache and DRAM. This results in high-latency loads and stores for every intermediate value.

We attempted to force register promotion by implementing a "static stack" where dynamic indexing is replaced by manual unrolling (e.g., shifting all elements $stk[i] \leftarrow stk[i+1]$). While this successfully mapped the stack to registers, it significantly increased register pressure and instruction count, leading to brittle performance. This motivated us to pursue higher-level optimizations that reduce the work itself.

2.4 Optimization 1: Heterogeneous Subtree Caching

Evolutionary populations inherently contain redundancy; child trees often share identical sub-structures with their parents. We exploit this by caching the results of common subtrees.

We implement a heterogeneous workflow where the CPU performs the irregular "detection" phase, and the GPU performs the "evaluation" phase.

(1) **Detection (Host):** The CPU hashes subtrees across the population. If a subtree appears frequently, it is marked as "cached," and a slot in GPU Shared Memory is allocated.
(2) **Evaluation (Device):** During kernel execution, if a node is marked as cached, the thread simply reads the pre-computed result from shared memory instead of executing the tokens.

Algorithm 2 Host-Side Subtree Detection

1: $Map \leftarrow$ Empty Hash Map
2: **for** each $tree$ in $Population$ **do**
3:     **for** each $node$ in $tree$ **do**
4:         $h \leftarrow \text{ComputeHash}(subtree(node))$
5:         $Map[h].count \leftarrow Map[h].count + 1$
6: **for** each $node$ in $Population$ **do**
7:     **if** $Map[node.hash].count >$ THRESHOLD **then**
8:         $node.is_cached \leftarrow$ TRUE
9:         Assign shared memory index to $node$

The objective was to first have a persistent cache stored in the GPU (gpu_subtree_batch_state.cu:L31) which holds an unordered map called hash_to_idx that maps the specific hash computed for that expression using Fnv-1, a non cryptographic hash, to a cache slot index. At a high level d_results is a 2 dimensional array of floats with size [MAX_CAPACITY][number of datapoints] and each index d_results[slot] stores for a specific subtree expression (e.g. 3x + 5). Then d_results[slot][datapoint idx] store the specific evaluation of the expression on the ith datapoint. This allows coalesced loads and writes to a specific cache slot of d_results.

Given the current population of expressions, the CPU detects common new subtrees across the population of expressions based on parameters of minimum frequency and minimum size of subtree (gpu_subtree_batch_state.cu:L466). We found that since this does not require computation over the entire dataset but just detection over the population, the CPU was sufficiently fast for this task (and does not require explicit synchronization on GPU to count repeated subtree expressions) Note that these subtrees must explicitly meet the parameter requirements (due to this there is a small set of detected subtrees that is repeated across expressions).

This is first passed to a designated kernel for evaluating these (gpu_subtree_batch_state.cu:L114) which is identical to the baseline kernel in implementation and uses a prefix encoding based implementation with a local memory stack except that subtree expressions are a much smaller set of expressions and it avoids redundant computation. This kernel notably writes the results for each subexpression to the appropriate d_results[slot][datapoint_idx] where the slot for each subexpression is provided by the CPU.

Here, there was one design choice of whether we should keep a subtree reuse cache persistent (i.e. cache that keeps old cached entries of previous generations for the current once) or if we should make it stateless (although the cache is allocated once at the very start to avoid memory allocation overhead, in each generation, we consider the cache to be “empty” and overwrite old slots and only have the kernel use the slots written by the current generation).

The main benefit of a stateless cache that is considered empty per generation is that there is less worry of exceeding the maximum stack capacity size but empirically we found that the max capacity was never reached

and the performance of the persistent cache was better as it allowed for reuse of subtrees across generations (avoiding computing one subtree that was already fully cached again in another generation) and also allow detecting and using subtrees within the current generation which does not meet the min frequency requirements but have already been computed (and can reduce total computation and also expression length for those that include that subexpression, reducing multiple reads and writes to local memory with just one memory load to the cache stored in global DRAM).

The src/eval directory contains both versions of gpu_subtree_batch_state.cuwhich uses a persistent cache and gpu_subtree.cu which is stateless. Since this is a pretty intuitive engineering decision, we don't provide further performance analysis between the two but these can be further evaluated using the make commands within the directory.

Assuming we are using a persistent cache, the CPU marks the total set of subtrees that either already existing within the cache or now exists after the first kernel stored the new candidates into the cache. The final kernel (gpu_subtree.cu:L180) is also almost identical to the baseline except that the CPU before passing the expressions to the GPU, removes entire subtrees that have been cached and only leave its root with a special marked token. In particular, the prefix-ordering encoded tokens and values array is modified (reducing the length of the total reads and writes to the stack as well as computation) and instead writes a special OP_REF token for the kernel to instead directly load the result from the cache for that token. The corresponding value for that token holds the cache slot index. Note that threads within a warp are assigned consecutive datapoints and also execute the same set of expressions at a time; loads to the cache are always coalesced at the warp level.

2.5 Optimization 2: Just-In-Time (JIT) Compilation

To eliminate the overhead of the interpreter loop and local memory stack entirely, we explored generating PTX in runtime.

NVRTC (C++ Source): We generate a CUDA C++ kernel string with the expression hard-coded (and also inlined operations). (cl_gpu_nvrtc.cu). The C++ kernel string corresponds to the baseline kernel but instead uses static indexing (as each token value and specific stack index is known in runtime) The NVIDIA Runtime Compiler (NVRTC) compiles this into PTX. This allows the compiler to perform Constant Subexpression Elimination (CSE) and register allocation automatically.

Direct PTX Generation: Alternatively, we manually assemble PTX instructions. This bypasses the heavy C++ to PTX compilation time (cl_gpu_ptx.cu), reducing compilation latency, but sacrifices the advanced optimizations of the standard compiler (-O3). The core operations of apply op is compiled to PTX prior to runtime and linked to the generated PTX string using cuLink.

The PTX string allocates MAX_EVAL_STACK numbers registers and follows the logic of pushing and popping to a "stack" but instead by maintaining a pointer for the top of the stack (which register index holds the last element) and directly compiling a PTX string to store and load intermediate outputs in specific registers (versus dynamic indexing over a local memory stack).

2.6 Additional Engineering Explorations

Beyond the core algorithms, we explored several engineering optimizations to maximize throughput:

- **Parallel Hashing:** Parallelizing the subtree hashing step on the CPU using OpenMP to ensure the preprocessing phase does not become a bottleneck.
- **Double Buffering:** Overlapping the CPU hashing phase of the *next* generation with the GPU evaluation of the *current* generation.
- **Static Stack:** Limiting stack size by reordering encoding order, to set statically access the stack and avoid L1 cache access by using registers.

- **Memory Pinning:** Using pinned host memory to accelerate the transfer of random batches of data to the GPU.

3 Experiments

In this section, we validate our proposed optimizations against standard baselines. We compare the broken-down and overall per-round processing latency in different benchmark settings, to illustrate the scalability of our algorithm and the trade-off between compilation overhead and execution throughput.

3.1 Experimental Setup

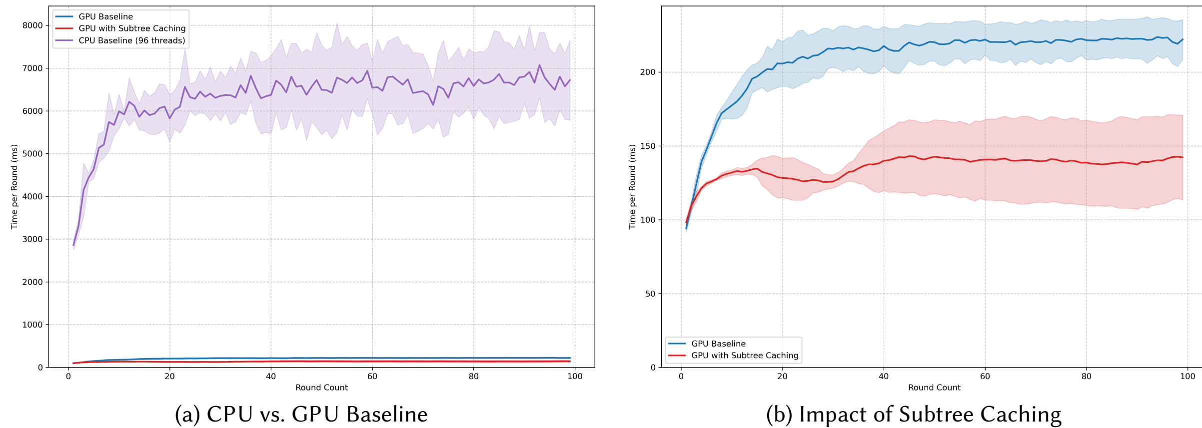
All experiments were conducted on a MIT's Engaging Cluster, equipped with an AMD EPYC 9654 96-Core Processor (running with 2-way hyperthreading) and a single NVIDIA L40S GPU (48GB VRAM).

We utilized the AI Feynman benchmark dataset [9], a standard suite for symbolic regression consisting of 120 scientific equations. For our stress tests, we generated random populations of candidate trees with varying parameters:

- **Population Size:** 50 to 2,000 candidates.
- **Data Points:** 40,000 to 1,000,000 rows.
- **Generations:** 30 to 100 rounds.

3.2 Evaluation Latency and Caching Efficiency

First, we establish the baseline performance gap between CPU and GPU baseline and our GPU implementation, to illust the speedup of CPU to GPU, and speedup of the Subtree Caching strategy.



(a) CPU vs. GPU Baseline

(b) Impact of Subtree Caching

Fig. 1. End-to-End Evaluation Latency. (a) The CPU baseline (purple) averages ≈ 7000ms, while the GPU baseline (blue) is under 250ms, demonstrating an order-of-magnitude speedup. (b) Zoomed view showing that heterogeneous caching (red) further reduces latency from ≈ 220ms to ≈ 140ms by eliminating redundant work.

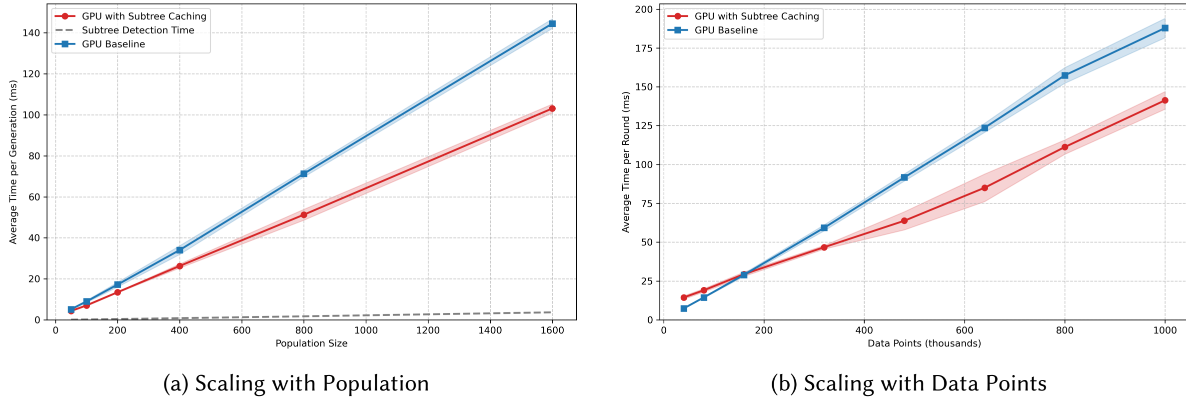
As shown in Figure 1(a), the CPU implementation hovers around 7,000ms per generation. In contrast, the GPU implementation completes the same workload in under 250ms. This confirms that for large-scale symbolic regression, GPU acceleration is a necessity.

Figure 1(b) highlights the specific contribution of our Subtree Caching strategy. The caching implementation (red) consistently outperforms the GPU baseline (blue) across all generations, stabilizing at ≈ 140ms per round.

This gap represents the "saved work"—floating-point operations skipped because sub-expressions were detected as redundant and loaded from shared memory.

3.3 Scalability Analysis

We further analyzed how our caching strategy scales with respect to the two primary dimensions of Symbolic Regression workloads: population size (N) and dataset size (M).



(a) Scaling with Population

(b) Scaling with Data Points

Fig. 2. Scalability Analysis. (a) Both methods scale linearly with population, but caching has a lower slope. The CPU detection overhead (grey dashed) is negligible ($<$ 5ms). (b) Caching benefits widen as the dataset grows, as each cached subtree saves M operations.

Figure 2(a) demonstrates that while both methods scale linearly with population size, the caching strategy has a significantly lower slope. Crucially, the "Subtree Detection Time" (grey dashed line) remains negligible even at 1,600 candidates. This confirms our hypothesis that the CPU is highly efficient at the irregular task of hashing, adding virtually no overhead to the pipeline.

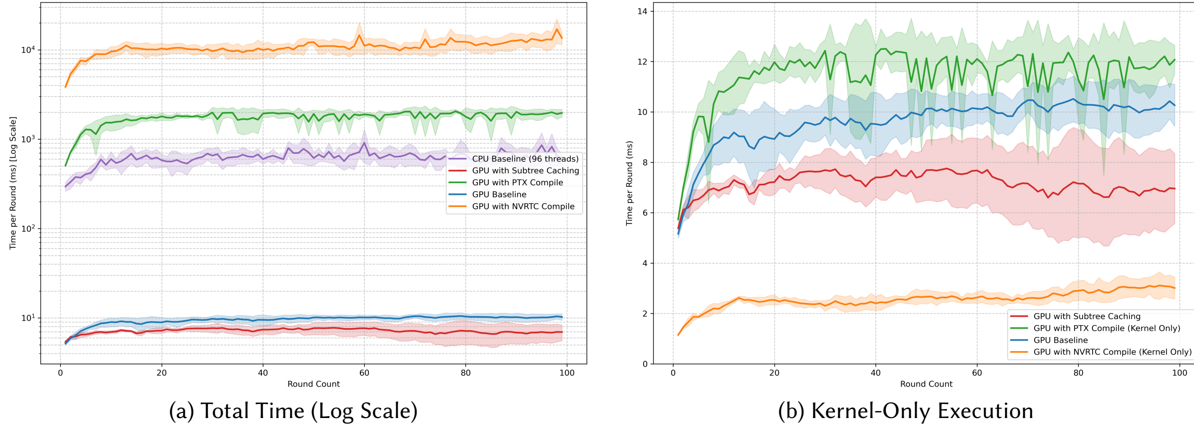
Similarly, Figure 2(b) shows that as the dataset size increases to 1 million, the performance gap widens. Since every cached subtree saves N operations (where N is the number of data points), the return on investment for caching increases with larger datasets.

3.4 Trade-offs in Runtime Compilation (JIT)

Finally, we analyze the performance of our Just-In-Time (JIT) compilation strategies: NVRTC (C++ to PTX) and direct PTX generation. We examine both the total turnaround time and the pure kernel execution speed.

Table 1. Average Evaluation Latency (ms) across Generation Buckets. Comparisons show the massive gap between Kernel execution and Overall time (including compilation) for JIT approaches.

Gen	GPU Subtree	GPU Base	CPU Base	PTX (Krn)	NVRTC (Krn)	PTX (All)	NVRTC (All)
0-19	11.07	12.16	507.17	10.01	2.12	1347.69	8558.76
20-39	7.46	9.44	616.15	11.78	2.42	1794.86	10054.57
40-59	7.57	9.91	666.81	11.90	2.58	1841.63	10958.82
60-79	7.13	10.19	655.31	11.78	2.65	1890.26	11230.68
80-99	6.89	10.20	665.44	11.73	2.98	1886.93	12570.58



(a) Total Time (Log Scale)

(b) Kernel-Only Execution

Fig. 3. Compilation vs. Execution Analysis. (a) Total time per round (generation) including compilation overhead. NVRTC (orange) is orders of magnitude slower due to the C++ compiler. (b) Pure kernel execution time. NVRTC generates the fastest code (≈ 2.5ms), significantly outperforming the baseline and caching methods.

Figure 3(a) reveals the high cost of runtime compilation. The NVRTC approach (orange) is approximately 100× slower than the baseline when including compilation time, making it viable only for extremely long-running evolutionary jobs.

However, Figure 3(b) shows the theoretical ceiling of performance. When ignoring compilation latency, the NVRTC kernel is the clear winner, achieving ≈ 2.5ms per round. This is nearly 4× faster than even our cached implementation (≈ 7ms). This illustrates the massive benefit of inlining expression trees to enable aggressive register allocation, provided the compilation latency can be reduced or amortized.

3.5 Inspection on Compilation

To eliminate the overhead of the interpreter loop and local memory stack entirely, we explored generating machine code at runtime. We compared two approaches: compiling a generated C++ kernel via NVRTC versus directly generating PTX assembly.

3.5.1 Approach A: NVRTC with Inlining. We generate a CUDA C++ kernel string with the expression hard-coded and operations inlined (cl_gpu_nvrtc.cu). This allows the NVIDIA Runtime Compiler (NVRTC) to perform Constant Subexpression Elimination (CSE) and register allocation automatically.

Analysis of the generated SASS (machine code) shows that the compiler efficiently interleaves instructions, although it occasionally fails to reorder independent global stores. The kernel uses 44 registers and a minimal 32-byte stack frame.

```
/*0a50*/ IMAD.WIDE R36, R24, R37, c[0x0][0x178] ;
/*0a60*/ STG.E [R36.64], R0 ;     // store one result
...
/*10f0*/ IMAD.WIDE R36, R3, 0x4, R36 ;
/*1110*/ STG.E [R36.64], R0 ;
```

Performance: With force-inlining, NVRTC achieves the highest throughput (0.48ms execution time). However, the compilation cost is prohibitive for dynamic workloads: 1532ms for C++⟶PTX and 1021ms for PTX⟶SASS.

3.5.2 Approach B: Direct PTX Generation. To bypass the expensive C++ frontend, we directly generate PTX assembly. We define 64 general-purpose registers to simulate a "register stack," where the i-th register corresponds conceptually to the stack index i.

```
/*0100*/  LDG.E R5, [R26.64+0x4] ;
/*0110*/  MOV R6, R4 ;
/*0120*/  MOV R4, 0x3 ;
/*0130*/  MOV R20, 32@lo((kernel + .L_x_1@srel)) ;
/*0140*/  MOV R21, 32@hi((kernel + .L_x_1@srel)) ;
/*0150*/  CALL.ABS.NOINC `(apply_op) ;
```

As seen in the SASS above, direct PTX generation creates sequential dependencies (reusing R4 and R6), which limits Instruction Level Parallelism (ILP). The execution time is slower (3.88ms) compared to NVRTC.

Trade-off: Direct PTX generation is significantly faster to compile (38ms total) compared to NVRTC ($\approx$ 2500ms). Thus, Direct PTX is the viable path for evolving populations where the structure changes every generation, whereas NVRTC is superior for static, long-running workloads.

4 Discussion and Conclusion

4.1 Performance Analysis: The Bandwidth Bottleneck

While our GPU implementation achieves orders of magnitude speedup over the CPU, a theoretical analysis reveals significant room for improvement. Consider a typical generation with 2,000 expressions evaluated over 1 million data points, assuming an average of 20 operations per tree. This workload represents approximately 4×10^{10} floating-point operations. With an execution time of 150ms, our effective throughput is:

$$\frac{4 \times 10^{10} \text{ ops}}{0.15 \text{ s}} \approx 266 \text{ GFLOPS} \tag{1}$$

Given that the Nvidia L40S has a theoretical peak performance of $\approx$ 90 TFLOPS [5], our system is utilizing less than 0.5% of the hardware's compute potential .

To investigate this discrepancy, we analyzed the kernel using the CUDA compiler's verbose output (`ptxas`). The analysis reveals that the compiler allocates a significant amount of Local Memory for the evaluation stack because the stack index is dynamically determined at runtime.

```
ptxas info : Compiling entry function '_Z24eval...' for 'sm_89'
ptxas info : Function properties for _Z24eval...
    272 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 22 registers, 272 bytes cumulative stack size
```

The 272-byte stack frame corresponds to our `MAX_EVAL_STACK` of 60 floats. Unlike registers, Local Memory is physically located in the L1/L2 cache and DRAM hierarchy. Since dynamic indexing prevents register mapping, every push and pop operation becomes a high-latency memory transaction.

We isolated this bottleneck through a series of microbenchmarks on a controlled workload (100 candidates, 100k data points). As shown in Table 2, we selectively enabled or disabled specific parts of the kernel to measure their latency cost.

The results confirm that the system is memory-bound, specifically by stack manipulation:

- Data Loading is Free: The "X-only" test takes only 0.11ms, proving that loading the input features from global memory is negligible.
- Stack Overhead Dominates: The "Stack R/W" test, which performs all push/pop logic but skips the actual math, takes 8.44ms—accounting for 84% of the total execution time.

Table 2. Microbenchmarks of evaluation kernel (100 candidates, 100k datapoints).

Benchmark	Stack R/W	Arithmetic	Time (ms)	Rel. to Baseline
Baseline (Full Eval)	Yes	Yes	10.03	1.00× (100%)
X-only (Load only)	No	No	0.11	87.49× (1.1%)
Stack R/W (No Compute)	Yes	Minimal	8.44	1.19× (84.1%)
Fixed Op (All Adds)	Yes	Yes (Add)	9.34	1.07× (93.1%)
Compute (No Stack)	No	Yes (Sim)	7.91	1.27× (78.8%)

- Compute is Cheap: Conversely, the "Compute (No Stack)" test, which performs the math using registers without touching the local stack, is significantly faster (7.91ms) than the stack-only test.

This explains the low GFLOPS utilization: the GPU cores are stalled waiting for Local Memory latencies to service the stack. With thousands of threads each maintaining a private 240-byte stack in L1 cache, the cache bandwidth is saturated, confirming that future optimizations must focus on register promotion (like our JIT approach) or minimizing stack depth (like our Subtree Caching) rather than raw arithmetic throughput.

4.2 Summary of Contributions

In this paper, we presented a high-performance GPU library for Symbolic Regression. We tackled the evaluation bottleneck by introducing two novel strategies:

1. **Heterogeneous Subtree Caching:** We demonstrated that offloading semantic hashing to the CPU allows for effective reuse of sub-expressions, reducing evaluation time by $\approx$ 30% without incurring synchronization penalties.
2. **JIT Compilation Analysis:** We showed that while runtime compilation (NVRTC) offers the highest theoretical kernel speed (2.5ms), its compilation overhead makes it prohibitive for dynamic short-term evolution.

5 Future Work

Our analysis opens several promising directions for future research:

Lightweight GPU JIT Compiler: The primary drawback of our NVRTC approach is the heavy overhead of the full C++ compiler. A dedicated, lightweight JIT that translates algebraic strings directly to GPU machine code (SASS/PTX) could bridge the gap. By supporting only simple mathematical primitives, such a compiler could achieve near-instant compilation times, making the "inlined" kernel approach viable for dynamic populations.

Library Integration: Currently, our system exists as a standalone prototype. We plan to integrate this evaluation engine into popular frameworks like PySR [2], SymPy [4], or PyTorch. Providing a drop-in replacement for their evaluation backends would allow the broader scientific community to leverage these speedups immediately.

Advanced Parallelism & Speculation: We aim to explore tighter integration with recent SR algorithms by utilizing semantic information for speculative execution. For instance, if a mutation is minor, the system could speculatively compute only the changed branch while reusing the rest of the tree from the previous generation's cache.

6 Breakdown on Contributions

- **Jinha Kim:** Jinha worked on the dynamically created kernels during runtime with two designs of creating a PTX string to be JIT compiled and a CUDA kernel to be dynamically compiled with NVRTC. Jinha helped implement baseline kernel of EvoGP and their mutation, crossover, random generation kernels and also the kernel that uses subtree reuse with a persistent cache across generations.

- **Younghun Roh:** Younghun worked on setting up the AI Feynman benchmark and multiple micro-benchmarks throughout, to evaluate the performance of different kernels. He also implemented the kernels that use subtree reuse with a stateless cache and implemented the CPU multicore baseline as well as optimization of heterogeneous data movement along with GPU profiling and analysis.

- Jinha Kim and Younghun Roh

References

[1] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. 2021. Neural Symbolic Regression that Scales. arXiv:2106.06427 [cs.LG] https://arxiv.org/abs/2106.06427

[2] Miles Cranmer. 2023. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl. arXiv:2305.01582 [astro-ph.IM] https://arxiv.org/abs/2305.01582

[3] John R. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing* 4, 2 (June 1994), 87–112. doi:doi:10.1007/BF00175355

[4] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. doi:10.7717/peerj-cs.103

[5] NVIDIA Corporation. 2023. NVIDIA L40S Datasheet. Online. https://resources.nvidia.com/en-us-l40s/l40s-datasheet-28413

[6] Brenden K. Petersen, Mikel Landajuela, T. Nathan Mundhenk, Claudio P. Santiago, Soo K. Kim, and Joanne T. Kim. 2021. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. arXiv:1912.04871 [cs.LG] https://arxiv.org/abs/1912.04871

[7] Michael Schmidt and Hod Lipson. 2009. Distilling free-form natural laws from experimental data. *Science* 324, 5923 (April 2009), 81–85.

[8] Wassim Tenachi, Rodrigo Ibata, and Foivos I. Diakogiannis. 2023. Deep Symbolic Regression for Physics Guided by Units Constraints: Toward the Automated Discovery of Physical Laws. *The Astrophysical Journal* 959, 2 (Dec. 2023), 99. doi:10.3847/1538-4357/ad014c

[9] Silviu-Marian Udrescu and Max Tegmark. 2020. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances* 6, 16 (2020), eaay2631. arXiv:https://www.science.org/doi/pdf/10.1126/sciadv.aay2631 doi:10.1126/sciadv.aay2631

[10] Zhihong Wu, Lishuang Wang, Kebin Sun, Zhuozhao Li, and Ran Cheng. 2025. Enabling Population-Level Parallelism in Tree-Based Genetic Programming for GPU Acceleration. arXiv:2501.17168 [cs.NE] https://arxiv.org/abs/2501.17168