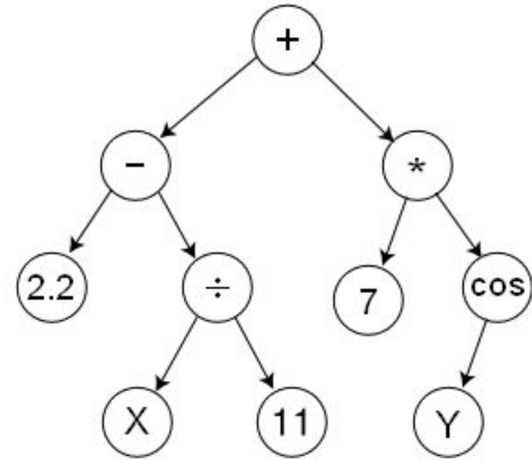
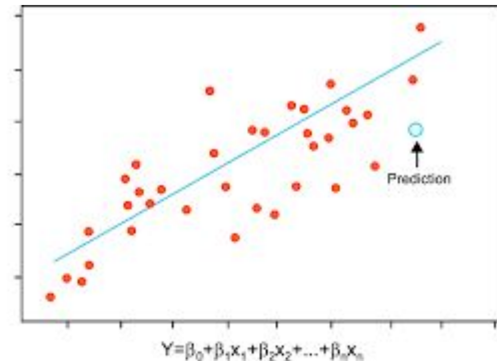
Accelerated Symbolic Regression

Jinha Kim, Younghun Roh

What is Symbolic Regression

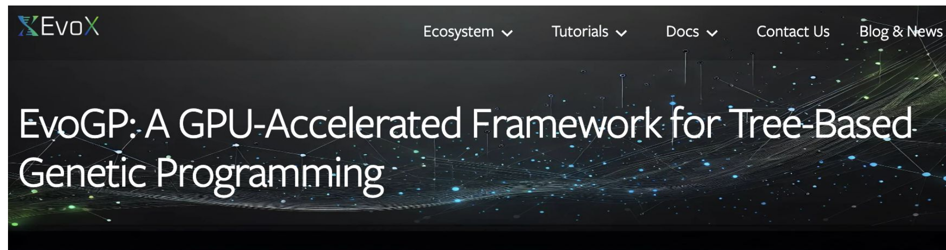
- Given data set {(x,y,z,....f), …}
- Find out a fitting expression,
  with only allowed operations
  - Arithmetic, sin, cos, pow, log, etc
- For better interpretability!
  (than neural nets)





What is Symbolic Regression

- Used in scientific discoveries!
    - Physics and engineering applications
    - Biological applications of growth rate, survival curves
    - Explaining how AI works
- Recent works
    - End-to-end GPU SR (EvoGP, 2024)
    - LM + RAG (RAG-SR, ICLR 2025)



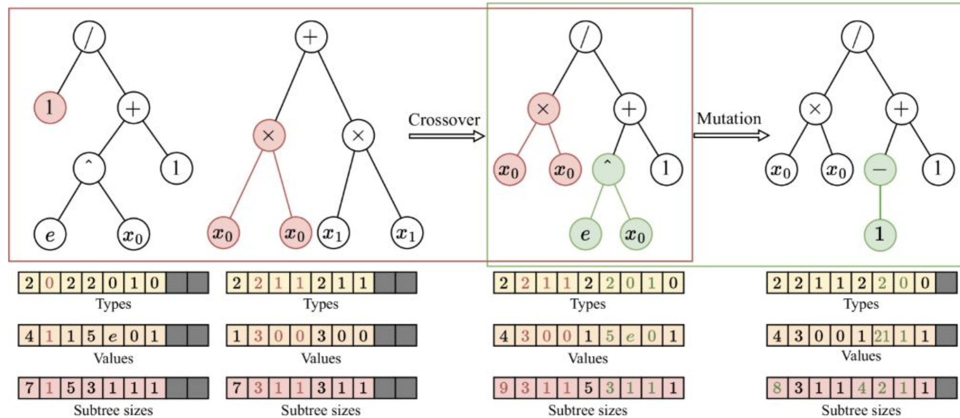
Published as a conference paper at ICLR 2025

RAG-SR: RETRIEVAL-AUGMENTED GENERATION FOR NEURAL SYMBOLIC REGRESSION

Hengzhe Zhang[1,3], **Qi Chen**[1,3], **Bing Xue**[1,3], **Wolfgang Banzhaf**[2], **Mengjie Zhang**[1,3]*
[1]School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
[2]Department of Computer Science and Engineering, Michigan State University, USA
[3]Centre for Data Science and Artificial Intelligence, Victoria University of Wellington, New Zealand

Tree-Based Genetic Programming

- Generate random ‘organisms’ (expression) , and compute their fitness (MSE)
- Only the fit ones can pass on their ‘genes’ (subtree) to the next ‘generation’
- Each expression is represented as a tree of nodes (constant, variable, function)



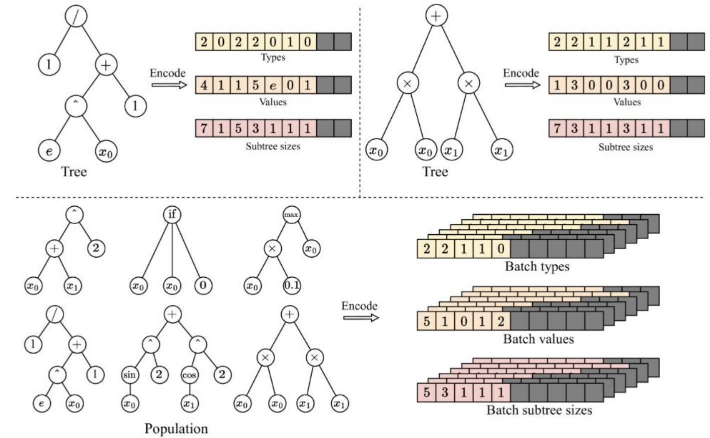


Figure 2: Illustration of the tree encoding process. In tensorized encoding, a tree is encoded into three tensors: types, values, and subtree sizes. We use NaN padding to ensure that these tensors reach a uniform maximum size, allowing trees of different structures to be encoded into tensors of the same shape. This enables the encoding of an entire population of trees into three batched tensors.

Source: https://arxiv.org/html/2501.17168v2

What are we solving

- Faster 'evaluation' in SR workload
  - For each generation, strings of exprs are given
  - Evaluate expressions on the data points
  - Compute the fitness for each expression
- How?
  - Reuse previous computation (less work)
  - Better parsing (Just-in-Time compilation)

How to choose expressions is out of our scope

Algorithm 1 Main Process of TGP

Require: Population size N; Target fitness f_{target}; Max generations count G;
  $P \leftarrow$ Randomly generate N trees;
  for $g = 1$ to G **do**
    $Fit \leftarrow$ Evaluate fitness values of P;
    if $\max(Fit) \geq f_{\text{target}}$ **then**
      break
    end if
    $C \leftarrow$ The empty population.
    while Not enough trees in C **do**
      $Parent1, Parent2 \leftarrow$ Select parents;
      $Child \leftarrow$ Crossover($Parent1, Parent2$);
      $Child \leftarrow$ Mutation($Child$);
      $C \leftarrow C \cup \{Child\}$
    end while
  end for
  return $P[\arg\max(Fit)]$

Source: https://arxiv.org/html/2501.17168v2

CPU Baseline

Simple parallel stack evaluation

- Parallelize on each expression
- Parse expression into ops & vals
- Iterate them with a computation stack
- Switch on op value, and load vals
- Do for all data points

Algorithm 2 Evaluate Prefix-Encoded Expression Tree on CPU

```
Require: integer array tokens[0 ... numTokens − 1]
Require: real array values[0 ... numTokens − 1]
Require: real array x[0 ... numVars − 1]
Ensure: real value of the expression
 1: sp ← 0                                          ▷ reset stack
 2: tmp, v1, v2 ← 0
 3: for i ← numTokens − 1 downto 0 do
 4:     tok ← tokens[i]
 5:     if tok > 0 then                             ▷ operation token
 6:         v1 ← Pop(stk)
 7:         if tok < 10 then                        ▷ binary operation (1–9)
 8:             v2 ← Pop(stk)
 9:         else
10:             v2 ← 0                              ▷ unary op case (if any)
11:         tmp ← EvalOp(tok, v1, v2)
12:         Push(stk, tmp)
13:     else if tok = 0 then                        ▷ constant
14:         Push(stk, values[i])
15:     else if tok = −1 then                       ▷ variable
16:         idx ← int(values[i])                    ▷ integer cast
17:         Push(stk, x[idx])
18:     else
19:         error: "Invalid token"                  ▷ handle invalid token
20: return stk[0]
```

GPU Baseline

Same stack evaluation, but more parallel

- Task: (data points, expressions)

Algorithm 4 GPU Kernel: Batched Prefix Evaluation

Require: integer arrays d_tokens_batch, d_token_offsets, d_num_tokens
Require: real arrays d_values_batch, d_vars_flat, d_pred_batch
Require: integers num_vars, num_dps, num_exprs, exprs_per_block

```
1:  block_expr_start ← blockIdx.x · exprs_per_block
2:  block_expr_end ← min(block_expr_start + exprs_per_block, num_exprs)
3:  dp_start ← blockIdx.y · blockDim.x
4:  dp_idx ← dp_start + threadIdx.x
5:  if dp_idx ≥ num_dps then
6:      return
7:  for expr_idx ← block_expr_start to block_expr_end −1 do
8:      token_offset ← d_token_offsets[expr_idx]
9:      d_tokens ← d_tokens_batch + token_offset
10:     d_values ← d_values_batch + token_offset
11:     num_tokens ← d_num_tokens[expr_idx]
12:     d_pred ← d_pred_batch + (expr_idx · num_dps)
13:     d_pred[dp_idx] ← EvalTreeDevice(
         d_tokens, d_values, d_vars_flat,
         num_tokens, num_vars, num_dps, dp_idx )
```

What is bottleneck?

```
ptxas info    : Compiling entry function '_Z24eval_prefix_kernel_batchPKiPKfS2_iiiPfS3_' for 'sm_89'
ptxas info    : Function properties for _Z24eval_prefix_kernel_batchPKiPKfS2_iiiPfS3_
    272 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 22 registers, 272 bytes cumulative stack size, 408 bytes cmem[0], 184 bytes
cmem[2]
```

Using 272 bytes of local memory (Stack) as stack is dynamic indexed (can't be stored in registers)

Table 1: Microbenchmarks of EvoGP evaluation kernel on 100 candidates (max length 60 tokens) over 100k datapoints.

Benchmark	Stack R/W	Arithmetic	Time (ms)	Rel. to baseline
Baseline (full eval)	Yes	Yes	10.0342	1.00× (100%)
X-only (load only)	No	No	0.114688	87.49× faster (1.14%)
Stack R/W, no compute	Yes	Minimal (dummy branch)	8.43981	1.19× faster (84.11%)
Fixed operator (all adds)	Yes	Yes (add only)	9.34195	1.07× faster (93.10%)
Compute in tmp, no stack R/W	No	Yes (simulated)	7.90534	1.27× faster (78.78%)

Local stack reads and writes then compute are bottlenecks

Strategy 1: Cache Reuse

- There will be a lot of common ‘parts’ of expressions
    - Most expressions are ‘mutated’ from few trees
    - Cache the common parts, and skip computation!
- CPU scans the expressions, detect and mark common subtrees by hashing
- Pre-compute the common subtrees, and keep them in GPU memory
- When evaluating, fetch the computed data when we see it
- Saved work!

Strategy 2: JIT compilation C++ to PTX vs manual PTX

- Fixed set of available registers (like the local stack)
- Runtime: write C++ kernel string for a custom kernel and dynamically compile using NVRTC
    - Avoids use of local stack memory and just keep in registers
    - Avoid having to parse expression following prefix encoding based on token values
    - Room for ILP and cross-subexpression reuse is promoted by reducing complex dependencies (to be further explored)

Strategy 2: JIT compilation C++ to PTX vs manual PTX

Table 3: Compilation and execution timings for inlined NVRTC (C++) and PTX JIT on a synthetic workload consisting of 20 generations on 500k data-points with 1000 expressions per generation, truncated to 100 expressions per generation for these measurements.

Variant	Source	Build src/PTX (ms)	C++→PTX (ms)	PTX→SASS (ms)	Launch+sync (ms)
NVRTC (inlined, example gen)	C++	3.40112	1532.97	1021.16	0.487152
PTX JIT (no inlining, example gen)	PTX	4.71014	–	38.6642	3.88451

Other Engineering Efforts

- Parallel tree hashing & aggregating
- Static stack, depth
- Double Buffering, cache pattern
- Memory pinning
- Parameter tuning (Tiling, detection threshold)
- (Lots of benchmark setups)

Experiments

All experiments done in Engaging Cluster

- AMD EPYC 9654 96-Core CPU (2-way hyperthreading) (~4 TFLOPS)
- Nvidia L40S (VRAM 48GB, ~90 TFLOPS)

30 - 100 Generations (rounds)

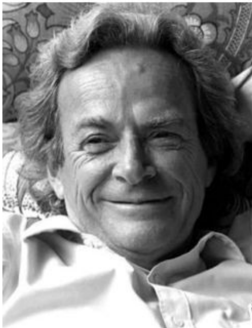
50 - 2000 Population (expressions)

40k - 1M Data points (float)

Measured with 5 iterations, basic SR algorithm

AI-Feynman Dataset

- 120 scientific equations from his ‘red book’
- Less than 10 variables, 64 ‘tokens’
- Random generated 1M data points
- Widely used for SR benchmark

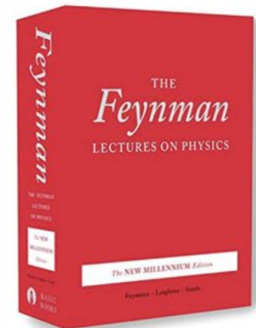


$$L = \frac{\hbar\omega^3}{\pi^2 c^2 (e^{\hbar\omega/k_b T} - 1)}$$

$$F = \frac{G m_1 m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

$$r = \frac{a(1 - e^2)}{1 + e\cos(\theta_1 - \theta_2)}$$

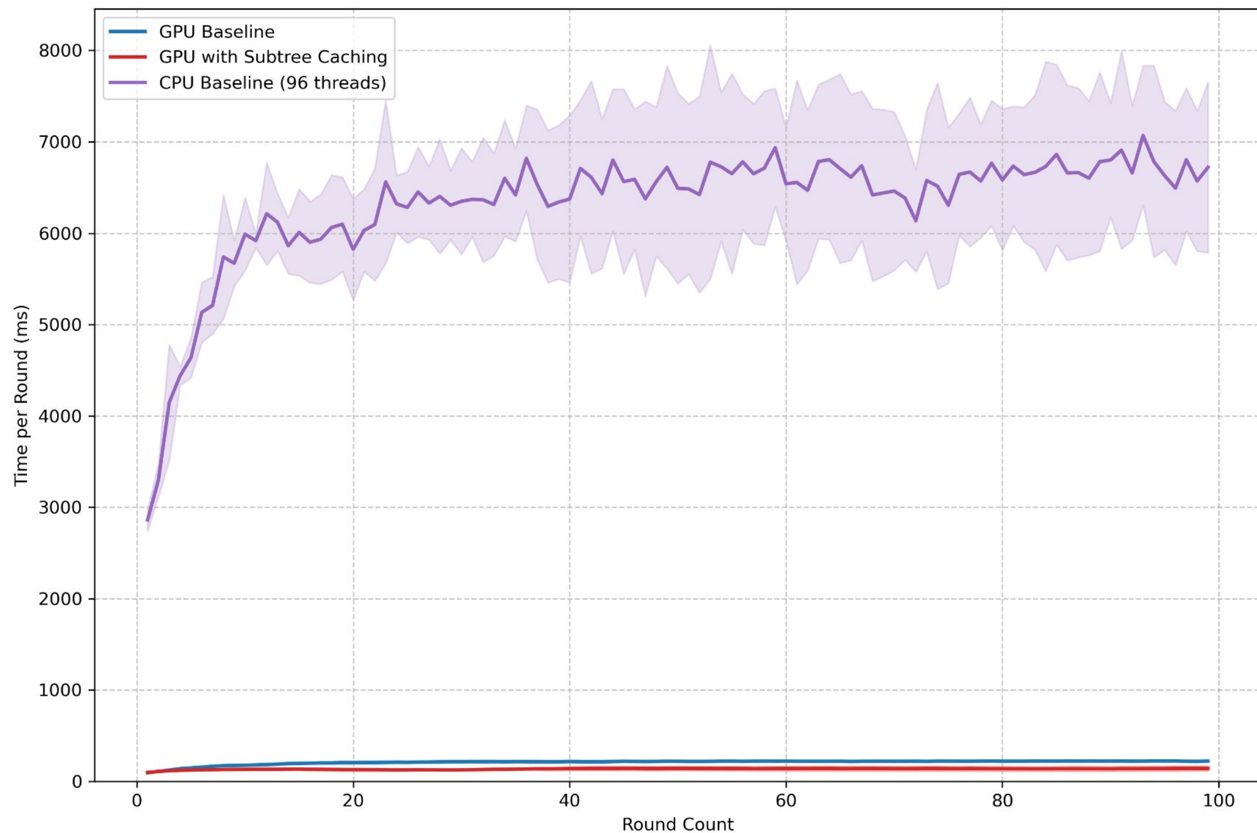
$$\frac{d\sigma}{d\cos\theta} = \frac{\pi\alpha^2\hbar^2}{m^2 c^2}\left(\frac{\omega'}{\omega}\right)^2\left(\frac{\omega'}{\omega} + \frac{\omega}{\omega'} - \sin^2\theta\right)$$



Performance

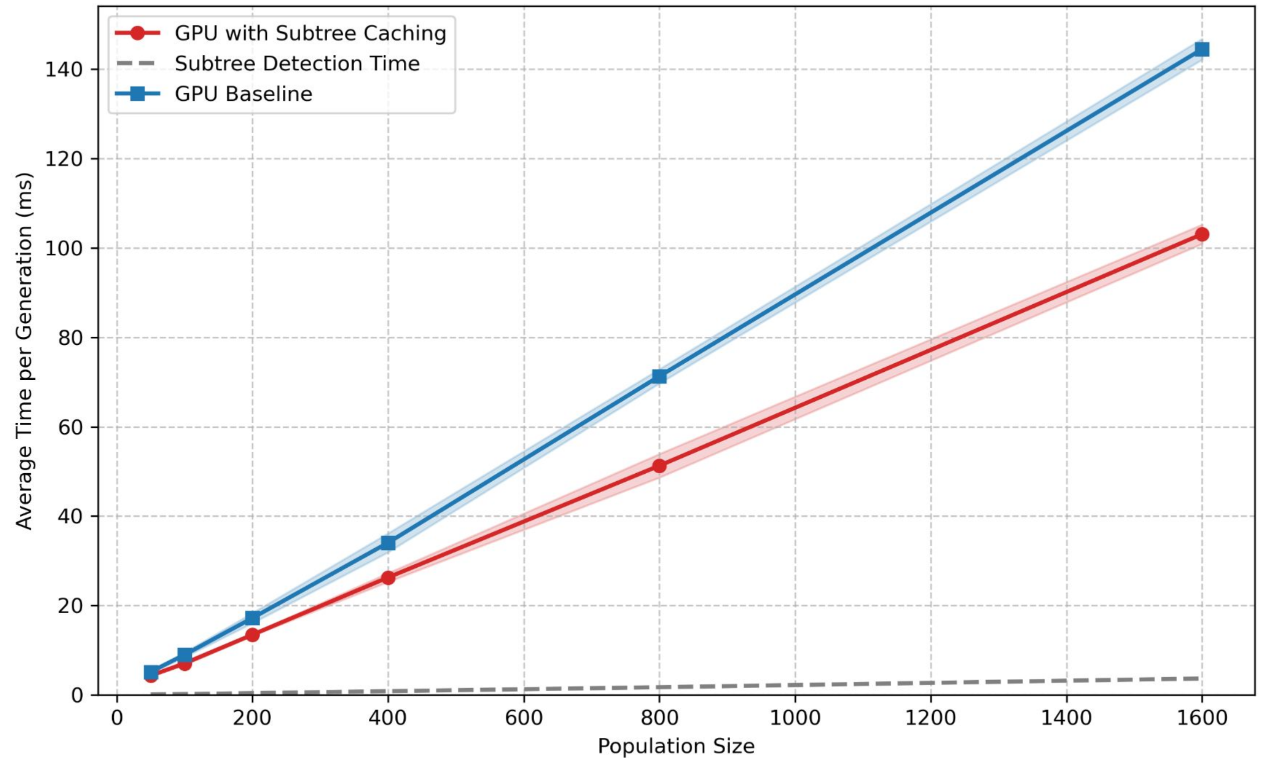
Time per round (GPU)

Less work by reusing!



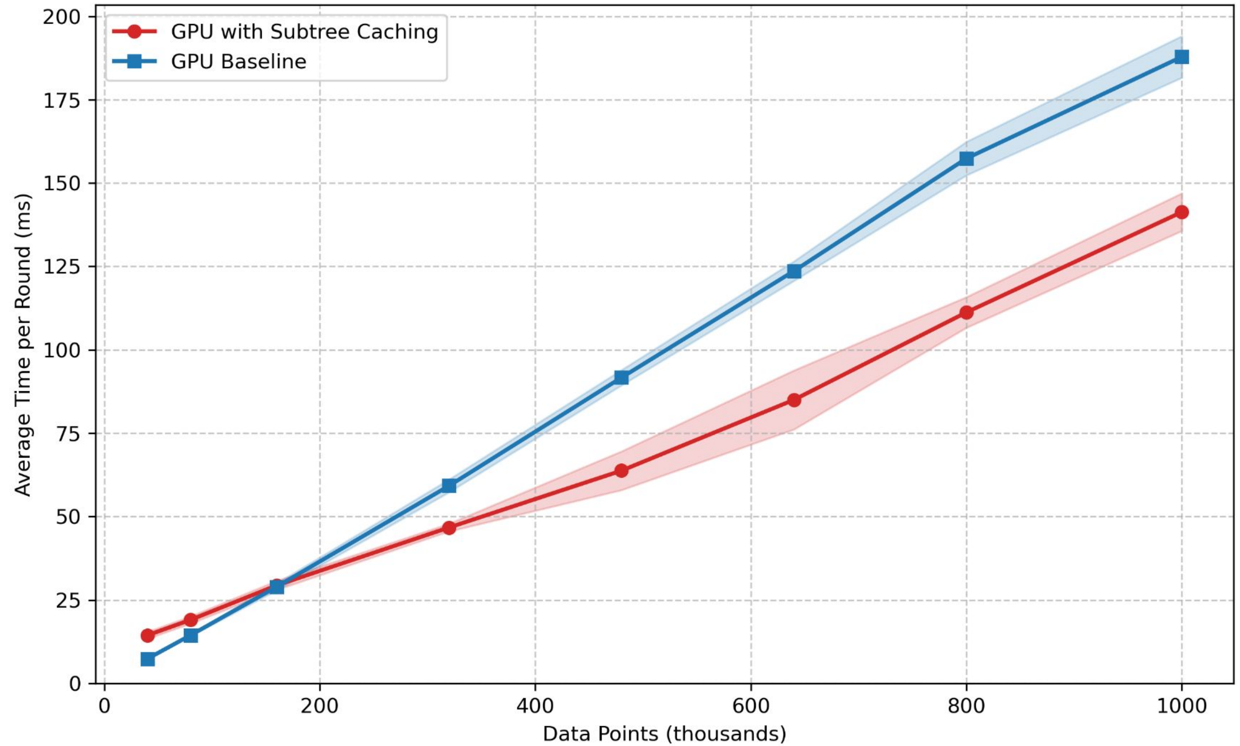
Increasing Population

Negligible detection time



Increasing Data Size

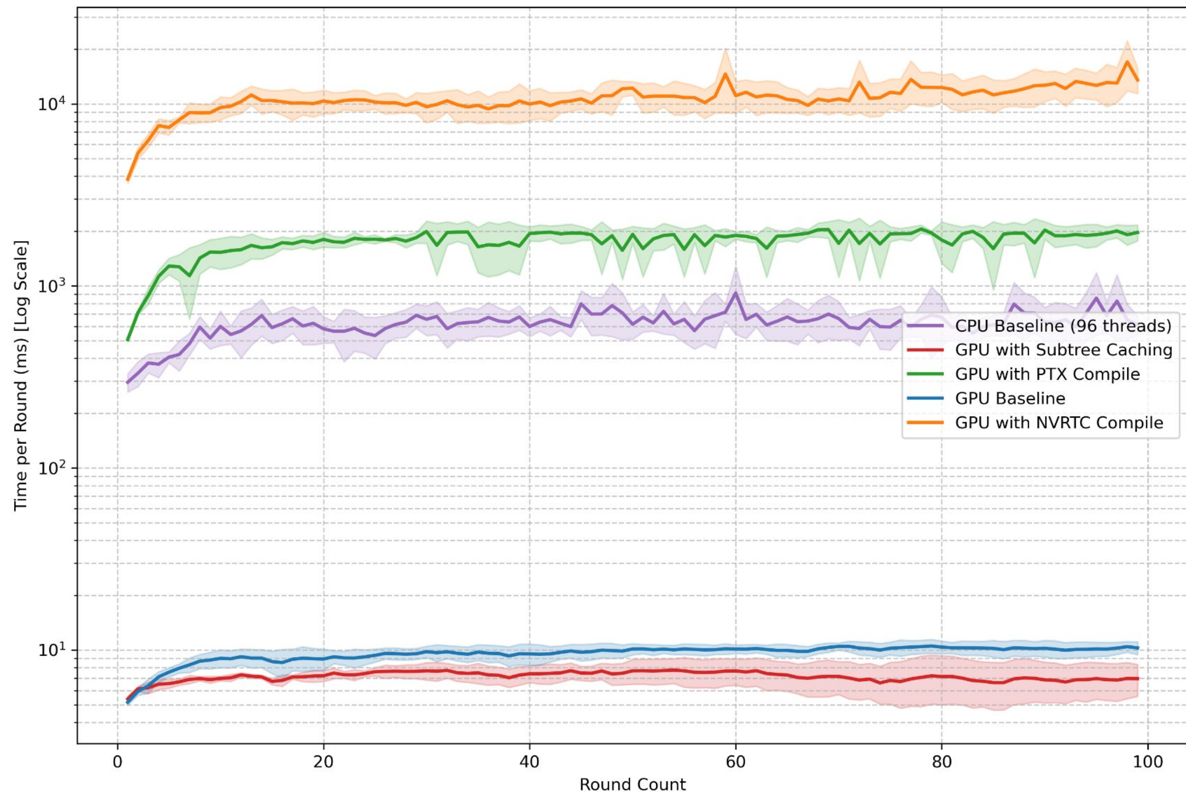
Tradeoff for detection



Runtime Compilation

Compiling expression
… in runtime

Much faster kernel!



Analysis & Notes

- How much FLOPS? (GPU Baseline)
    - (1M dps * 2000 exprs * 20 ops / gen) / (150 ms / gen) ~= 300 GFLOPS
    - HW cap is ~90 TFLOPS. Why is this slow? – L1 cache bandwidth!
- Plenty of room for generic GPU (and CPU) engineering
    - Parameter tuning, better parallelism split
    - Load/store latency hiding, cache/register efficiency, SIMD
    - How to create a better ‘assembly’ to reduce work & increase utilization
- Efficiency will vary on how we do mutations & expression complexity
        - Less common subtrees, less work saved

Future works

- Integrate to popular libraries (PySR, SymPy, Pytorch, …)
  - Or create a new one!
- Tighter integration with recent SR systems
  - Utilizing semantic info for reusing / compiling
- Better lightweight GPU JIT compiler (string -> GPU machine code)
  - Support only simple expression, with much faster compilation and evaluation
  - Can support subtree detect & reuse
- Application to actual challenges
  - Astrophysics, particle accelerator data, etc