Mini-Paper 2: Symbolic Regression

Louis Asanaka
Ishaan Mathur

Abstract

In this paper, we go through the optimization of a Symbolic Regression and Classification program. Through rounds of profiling and collaborative discussion using `gperftools` and `pprof`, we are able to decrease the runtime of the Diabetes dataset by 30x, Cancer by 35x, and Housing by 70x. The main optimizations, other than increasing the optimizer level, come from applying parallelism and reducing redundant operations when working with data structures.

1 Methodology

To standardize our development environment, we work in the virtual machines provided by the course, rather than through a Docker container. To profile, we modify Makefile and runner to take a constant `PROFILE` to indicate whether to profile the code or not. Building the program with `make -j PROFILE=1` results in an executable that writes the profiling results to `my_profile.prof` when executed. Using `pprof`'s web interface by running `make view-profile`, we are able to view the profile results in different forms, whether as text or a flamegraph.

We present the optimizations individually with their corresponding Git tag. Some optimizations are combined by similarity. All optimizations are benchmarked using the minimum wall-clock execution time over 5 runs.

Finally, all benchmark tests (`diabetes`, `cancer`, `housing`) use default parameters.

2 Optimizations

2.1 Baseline (Tag)

Running the baseline without any optimizations is incredibly slow, especially on the housing dataset. Looking at the graph, we see that most of the functions are not inlined, which we hypothesize to be a cause of increased execution time.

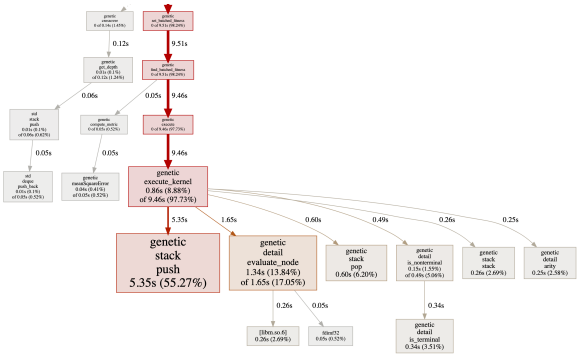

Figure 1. Baseline graph

2.2 Clang & `-O3` (Tag)

Given the lack of inlining, turning on optimizations (and switching to Clang, though this is merely a preference) help immensely, validating the hypothesis from the baseline run. However, compiler warnings still remain.

2.3 Compiler Warnings (Tag)

By fixing all the compiler warnings, we get minute performance gains while ensuring good practice.

Test	Before (s)	After (s)
Diabetes	0.953	0.912
Cancer	19.057	18.414
Housing	30.397	28.916

Table 1. Speedup before/after Warnings

2.4 Unnecessary Allocations (Tag)

After fixing the basics, we notice from the graph that `genetic::logLoss` could be a target for optimizations. Within the function, we see that an extraneous vector of size `n_samples * n_progs` is allocated, even though log loss can be computed as a running sum. We are able to cut down on memory allocation and avoid an extra iteration over the rows.

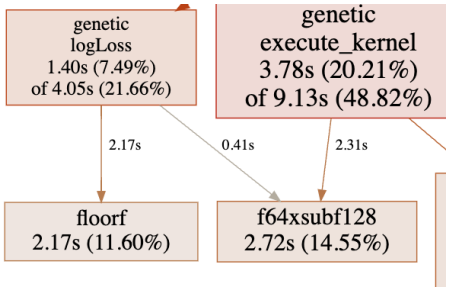

Figure 2. Before `logloss` graph

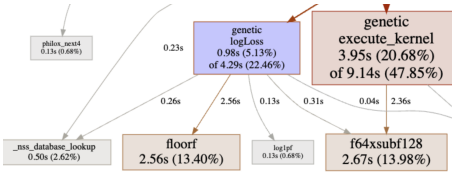

Figure 3. After `logloss` graph

2.5 `evaluate_node` Optimizations (Tag)

From the previous graph, we notice that `evaluate_node` could be another target for optimization. The function takes the absolute value of the float input regardless of whether it

is needed, so we move this into the code paths that need it rather than wasting a floating point operation for every node type. We also forcibly inline the function with `__attribute__((always_inline))` to minor benefit.

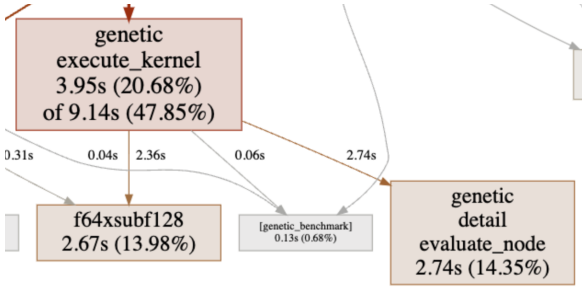

Figure 4. `evaluate_node` graph

2.6 Numerical Stability Fix (Tag)

Since there were some issues with NaNs affecting the benchmarking result, we cherry-picked the patch and decreased the optimization level to `-O2` to avoid more numerical precision issues. Performance was slightly worsened, potentially due to the selection of deeper trees, but correctness is always priority.

2.7 Remove Insertion Sort (Tag)

Since we only need the programs with the two smallest fitness value, we can avoid sorting and do a single linear scan instead. Since this is not on the hot path, performance is not affected significantly.

2.8 OpenMP Parallelization (Tag)

The main loop in `execute_kernel` over the number of programs can be trivially parallelized, so we apply an OpenMP `pragma` to do so. On the VM with 4 cores, we would expect a maximum of 4x speedup. We get slightly less due to overhead and other dataset-specific characteristics.

Test	Before (s)	After (s)	Speedup
Diabetes	1.847	0.495	3.73x
Cancer	12.899	6.789	1.90x
Housing	23.260	6.636	3.50x

Table 2. Speedup before/after Parallelization

2.9 Simplify Stack and `std::vector` Use (Tag)

Going back to the baseline graph in 1, we see that stack operations are in the hotpath, so we look to optimize those operations. `stack` in `reg_stack.h` is a stack implementation without dynamic memory allocation—however, the operations are implemented in an odd manner, likely for GPU use. Likewise, we spot `std::vector` operations, specifically `std::fill_n` during default initialization, taking up time. Since we fill most `vectors` after initialization, we do not need to default initialize the elements. We replace the initializations with `std::vector::reserve`, which allocates memory for the backing buffer without initializing the elements. This gives us a strong speedup as both data structures are used in the hotpath.

Test	Before (s)	After (s)	Speedup
Diabetes	0.495	0.339	1.46x
Cancer	6.789	6.032	1.13x
Housing	6.636	4.744	1.40x

Table 3. Speedup before/after Stack & Vector Optimizations

2.10 OpenMP Parallelization for Metrics (Tag)

Noticing the effectiveness of parallelization (as usual), we proceed to look for similar candidates for optimization. We spot that `meanSquareError` and `logloss` have similar iterations over all programs—we can parallelize loops in both functions. While we do not get as drastic of a speedup since metric calculations are not in the hotpath, we see halve the execution time for the Cancer dataset, which is configured to use the largest population / `n_progs` (16384).

Test	Before (s)	After (s)	Speedup
Diabetes	0.339	0.337	1.00x
Cancer	6.032	3.154	1.91x
Housing	4.744	4.212	1.13x

Table 4. Speedup before/after Metric Parallelization

2.11 Remove Redundant Modulus (Tag)

In the spirit of `evaluate_node` Optimizations (Tag), we look for redundant operations in the hotpath. While it does not appear clearly in the graphs, we spot a redundant modulo operation in the `tournament_kernel`. Removing it produces a minor speedup.

Test	Before (s)	After (s)	Speedup
Diabetes	0.337	0.328	1.03x
Cancer	3.154	3.098	1.02x
Housing	4.212	4.153	1.01x

Table 5. Speedup before/after Removing Modulo

3 Review

In this section, we summarize the performance improvements achieved, describe optimizations that failed to produce tangible speedups, and highlight potential avenues for further enhancements.

3.1 Summary of Optimizations

Overall, we obtained significant speed-ups by continuously examining the performance graph for which part of the program could reap the most benefit. This lead us to focus on inlining hotpath functions and adding OpenMP parallelization. We also evaluated those functions for extraneous allocations and memory-related changes (e.g., avoiding default initializations for `std::vector`, simplifying the custom stack in `reg_stack.h`) reducing overhead across many critical loops. By successively removing seemingly minor redundant operations (such as universal `abs` calls and unnecessary moduli), we further reduced execution times. Altogether, these incremental steps yielded the dramatic performance improvements—on the order of 30x for Diabetes, 35x for Cancer, and 70x for Housing.

3.2 Failed Optimizations

A few planned optimizations did not pan out. Applying the `-ffast-math` compiler flag (and similarly using `-O3`) introduced numerical issues, primarily in their treatment of `NaN` values and adding extra significant figures which would deviate from the baseline. We also attempted parallelizing smaller sections of logic, but the overhead overshadowed any benefit, so that approach was ultimately removed. Finally, prior to replacing the custom insertion sort with a single linear scan for the two smallest fitness values, we replaced it with a built-in stable sort. While it improved performance, we later realized sorting itself was unnecessary and removed that change.

3.3 Areas for Improvement

Although our optimizations yielded impressive results, there are several areas for further optimizations. We could further investigate parallelizing portions of the code (e.g., beyond the main kernel and metrics) if they show sufficient workload. Exploring vectorized instructions (such as AVX) and reorganizing data structures for cache efficiency might also yield noticeable performance improvements. Additionally, we could attempt to achieve numerical stability under aggressive optimizations and `-ffast-math` usage. Finally, we could further explore function inlining because that approach provided a lot of benefit in our optimizations.

3.4 Teamwork

We pair programmed through call, tackling an optimization each and switching roles after. Using the web visualization of the profiler results, we came up with potential optimizations and extensively tested them.

Our code can be found at https://github.com/imathur1/598APE-HW2. The overall summary of speedups is in 5 and 6.

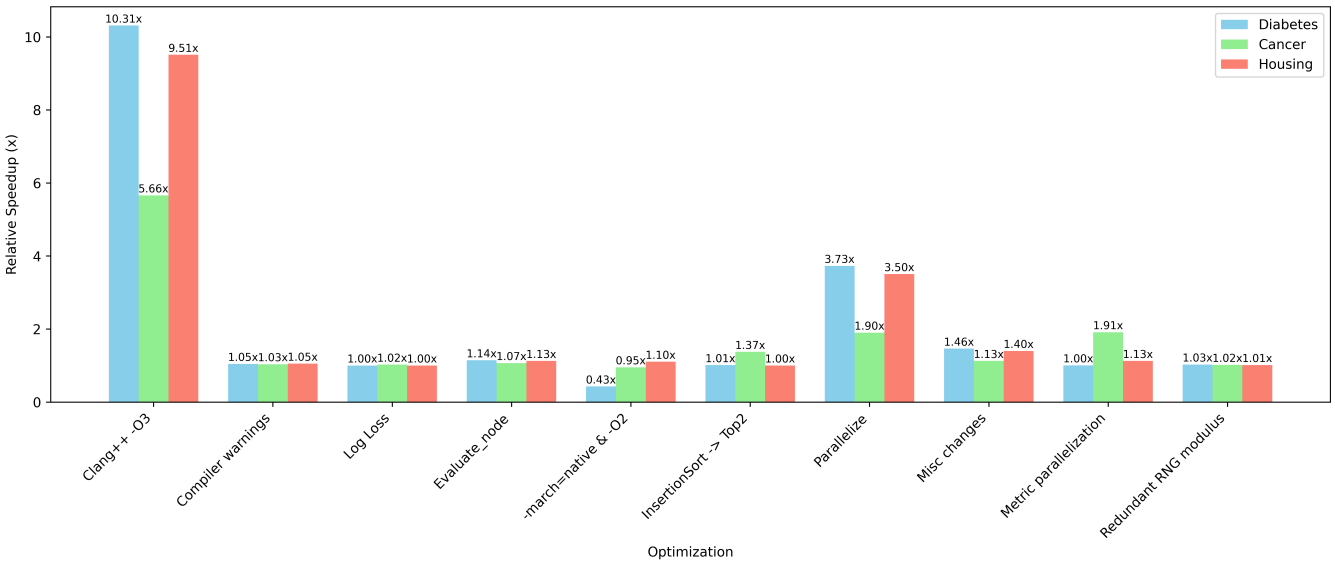

Figure 5. Symbolic Regression Optimizations

Optimization	Time (s)			Tag
	Diabetes	Cancer	Housing	
Baseline	9.832	107.841	289.125	baseline
Clang++ -O3	0.953	19.057	30.397	clang-o3
Compiler warnings	0.912	18.414	28.916	compiler-warnings
Log Loss	0.912	17.997	28.916	unnecessary-allocation
Evaluate_node	0.798	16.868	25.667	evaluate-node-optimizations
-march=native and -O2	1.869	17.735	23.251	ta-fix
InsertionSort -> Top2	1.847	12.899	23.260	remove-sort
Parallelize	0.495	6.789	6.636	parallelize
Misc changes	0.339	6.032	4.744	simple-stack-vector-alloc
Metric parallelization	0.337	3.154	4.212	metric-parallelize
Redundant RNG modulus	0.328	3.098	4.153	remove-modulus

Table 6. Minimum execution times for each optimization and dataset.