Towards Cost-Efficient Autotuners for Machine Learning Systems: Characterizing Autotuning Costs in OpenAI's Triton

Robert Hochgraf

Abstract

Overheads from automatic performance tuning have led to hesitance in placing autotuning into Machine Learning build systems. In this work, we characterize the sources of autotuning cost in Triton tensor compiler and simulate the Pareto frontier tradeoff between tuned kernel performance and the tuning time budget. Using these results, we develop targeted optimizations to reduce autotuning cost, namely disabling or reducing cache clearing, parallel compilation, and early quitting of autotuning. We integrate the former two optimizations into Triton, resulting in up to a 2.3x speedup in end-to-end tuning time and up to a 7.9x speedup in compilation time, respectively.

1 Introduction

Machine Learning compilers and software such as Triton [26] often provide automatic tuning capabilities that adapt performance-critical computational kernels to hardware capabilities and application workloads. Such tuning can lead to significant speedups by ensuring that optimization parameters contain values suitable for the hardware and data being operated on [24, 20].

Yet, autotuning adoption into build systems and core compiler software for Machine Learning has been hesitant. As IBM puts it [25], "...the Triton autotuner is usually not used in production today." This is partly because of the overheads introduced by autotuning processes. In production applications, autotuning at runtime has latency implications that can be undesirable [20]. However, budgeting less time for autotuning reduces the performance of training and inference tasks. To partially mitigate this effect, some packages that depend on Triton, including PyTorch [1], author custom forks of the Triton autotuner specifically to implement optimizations that improve the efficiency of the Triton autotuner, however these are tightly coupled for their use cases [21].

Characterization of the individual sources that contribute to autotuning cost would be useful to develop effective tuning strategies and to target optimizations for reducing the cost. In this work, we investigate the Triton autotuner and characterize the sources of autotuning cost. Using this characterization, we develop and simulate optimizations targeted to improve tuning efficiency. The main contributions of this work are:

- We build a parallel compilation system into Triton and evaluate its performance against Triton's sequential JIT compilation. Our evaluation shows that the parallel implementation completes compilation up to 7.9x faster.
- We profile Triton's autotuner and diagnose L2 cache clearing as the largest source of non-kernel overhead. From this diagnosis, we consider strategies to reduce the overhead from cache clearing, resulting in up to 2.3x speedups to run our entire benchmark suite.
- We simulate the Pareto frontier [27] tradeoff between the tuned performance of a kernel and the tuning time budget. For practitioners, we provide insights into tuning strategy, by showing that for our benchmark, the number of configurations explored is the main determinant of the frontier. Additionally, we simulate the impact of parallel compilation and early stopping optimizations on the Pareto frontier. Our simulation shows that parallel compilation is effective in improving the frontier, while early stopping is not.

- We determine that Triton's autotuner often incorrectly short-circuits fast kernels, benchmarking them for far less time than the user requests. We diagnose this behavior as being a result of Triton estimating the runtime of a warmed-up kernel by executing a sample of non-warm kernels.

2 Background

Autotuning is a powerful technique for optimizing code to fully utilize the underlying hardware [24, 20, 2]. At a high level, autotuning explores a set of possible *configurations*, where each configuration represents a unique collection of the *optimization parameters* that can be set for the kernel, such as block sizes, thread groups, or other performance-critical code parameters. These parameters typically need to be tuned specific to each data *input* argument to the kernel (e.g. for matrix multiplication, two matrices), as different data may require different optimizations to achieve maximal performance. Thus each configuration is benchmarked to estimate the speed of the parameters for each target hardware device and input [24, 20, 21, 9].

Triton enables users to autotune their programs by wrapping Triton JIT functions (`@triton.jit`) with the `@triton.autotune` decorator. Users pass the decorator a set of configurations and set the *repetition* and *warmup* times. The repetition time is the target amount of time that a single configuration should be executed for on the GPU during benchmarking of its performance [26]. The warmup time is the target amount of time spent bringing the GPU into a steady-state prior to the repetition time. While a contributor to overhead, sufficient warmup time is critical for accurate performance benchmarking because non-warmed kernel configurations execute much less quickly than warmed ones [26].

3 Related Work

There exists a significant body of related work in this area, particularly in optimizing autotuners. Several works are directly related in that they also create their own forks of the Triton autotuner to improve performance.

Triton-Dejavu. TritonDejavu [25] is a fork of the Triton autotuner with several additional features, most notably caching autotuning results to disk. Caching to disk allows the autotuning cost for known deployments to be shifted from runtime to ahead-of-time. The project additionally provides `ConfigSpaces`, a developer-friendly way of specifying configurations. This feature is especially useful because Triton by default does not check that provided configurations are correct.

PyTorch. PyTorch [1] has a custom fork of the Triton autotuner that adds several features not found in the original. Similarly to the above, PyTorch's fork allows autotuning results to be cached on-disk in JSON or a Redis cache [22], allowing for the reuse of tuning results across executions [21]. In addition, PyTorch adds a `TuningProcessPool` that distributes one tuning process to each target device. Within each tuning process, PyTorch lazily imports Triton's `do_bench` function to perform the benchmarks on each device [23]. This implementation of distributed compilation enables the autotuner to be compatible with distributed parallelism techniques such as PyTorch's Fully Sharded Data Parallel (FSDP) that are useful for training large models [28].

Petrovic et al. Petrovic et al [20] benchmark the performance of several CUDA and OpenCL kernels using the Kernel Tuning Toolkit (KTT) and measure the total time cost posed by dynamic runtime autotuning. The authors then find the number of kernel invocations necessary to hide the autotuning cost to under 10% of total runtime and discover that this value can range from two (2) to millions of invocations, depending on the kernel. The authors also find that autotuning costs can vary widely based on the underlying hardware device.

Table 1: Compilation Times on A40 GPU

Compilation Results on NVIDIA A40 GPU, 16 Configurations, FP16				
Compilation Time (ms)	**Block Size M**	**Block Size N**	**Block Size K**	**Group Size M**
837.82	128	256	64	8
759.06	64	256	32	8
681.80	128	128	32	8
...	...	...	...	...
2638.25	256	64	128	8
560.03	64	128	64	8
419.68	128	32	64	8
Total: 15.17s				

Rasch et al. Rasch et al. [24] propose the Auto-Tuning Framework (ATF). ATF is a general-purpose autotuner that exploits inter-dependencies between parameters. By extending the traditional autotuning definition with parameter constraints, ATF enables a variety of optimizations for generating, storing, and exploring search spaces with interdependent parameters quickly. The authors find that exploiting structure from inter-dependencies can lead to more efficient autotuning.

4 Preliminary Results

In this section, we present motivating preliminary results that inspire the optimizations and simulations presented in following sections. Raw results from the autotuner on a small benchmark of 16 configurations and 16 inputs are shown in Table 1 and Table 2. Some rows are omitted for space. Each result is collected for an A40 GPU as described in Section 6.

Table 1 shows the compilation time as reported by `TRITON_PRINT_AUTOTUNING=1` for kernels with varying block sizes. Table 2 shows the autotuning time for kernels with varying block sizes as reported by setting `TRITON_PRINT_AUTOTUNING=1`, excluding compilation. A description of each of the relevant parameters is given in Table 4.

Overall, the preliminary results show that compilation times are a significant portion of the overall execution time ($\approx$ 30% of the total tuning time), and that autotuning time increases with the size of $(M \times N \times K)$. Readers may also note that kernels with larger block size dimensions tend to take longer to compile than those with smaller block sizes.

A preliminary benchmark characterizing the sources of autotuning cost within Triton is shown in Figure 1. This benchmark is collected from the benchmark data as described in Section 6. The benchmark shows several results that inform the analysis and exploration undertaken in the remainder of this manuscript. First, the benchmark shows that the amount of *overhead* from sources other than compilation and kernel executions is more than half of all tuning time for most inputs measured in our experiment. This finding hints that there may operation(s) within Triton causing significant overhead. Second, the benchmark shows that the average total tuning time (blue) is often less than the theoretical end-to-end tuning time of 125ms, as explicitly set by the user during tuning. Noticing that the repetition time for kernels was less than the 100ms that was explicitly set during the benchmark led the authors to investigate issues in Triton's estimation of repetition time.

5 Targets of Evaluation

In this section, we describe the design of (1) the optimizations implemented into Triton reported in our evaluation, and (2) the construction of simulations reported in our results. We investigate three main optimizations: reducing/disabling L2 cache clearing, parallel compilation of kernels, and early stopping of benchmarking a slow configuration ("early stopping").

Table 2: Wall-Clock Autotuning Time on A40 GPU

Autotuning Time on NVIDIA A40 GPU, GEMM FP16, Not Including Compilation				
Total Autotuning Time (s)	**M**	**N**	**K**	**# Configurations**
1.17	256	256	256	16
1.17	384	384	384	16
1.17	640	640	640	16
1.17	768	768	768	16
...	...	...	...	...
1.32	2560	2560	2560	16
1.33	2688	2688	2688	16
1.45	3840	3840	3840	16
1.47	3968	3968	3968	16
1.49	4096	4096	4096	16
Total: 38.78s				

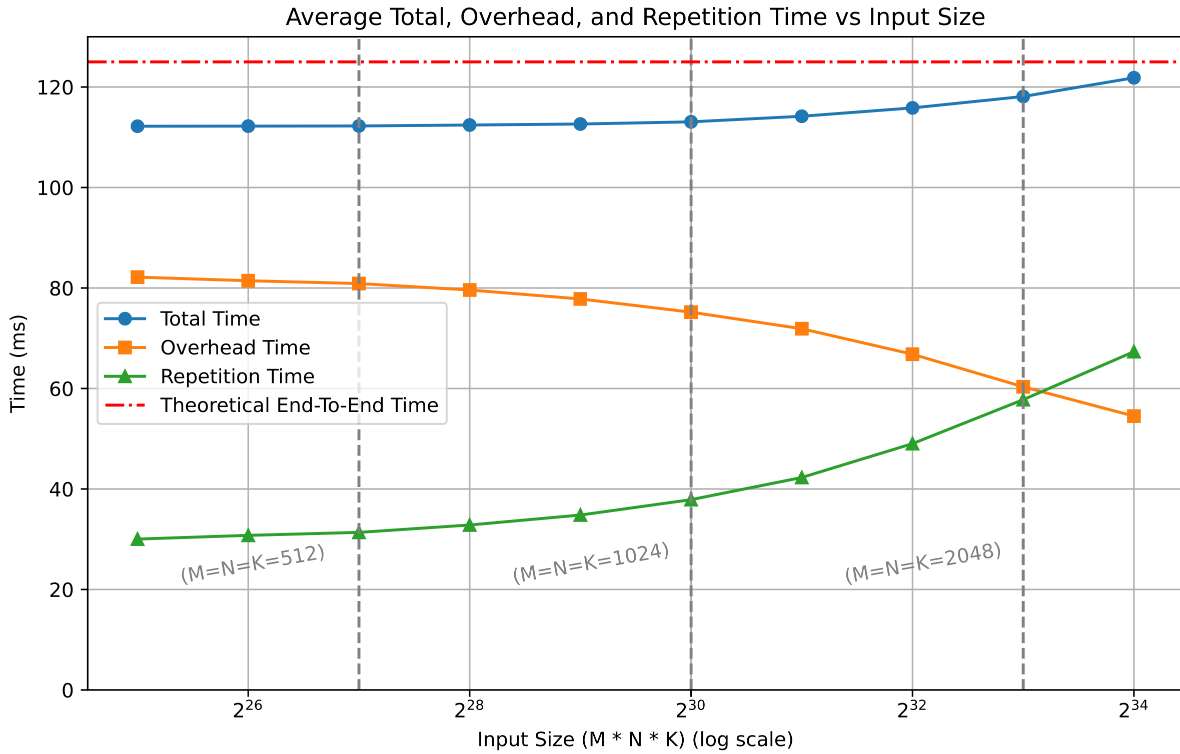

Figure 1: Total, Overhead, and Repetition Time.

Table 3: CUDA GPU Kernel Summary with and without L2 Cache Clearing Enabled

L2 Clear	Kernel Name	Total Time (s)	Avg (ns)	Min (ns)	Max (ns)	Dev (ns)
Enabled	`matmul_kernel`	4.45	91,002.3	3,359	4,068,436	222,682.0
Enabled	`elementwise`	**21.42**	394,114.2	391,650	399,105	941.5
Disabled	`matmul_kernel`	4.49	92,313.1	3,328	4,063,944	224,360.8
Disabled	`elementwise`	**0.50**	394,112.9	391,904	397,793	969.2

5.1 Reducing L2 Cache Clearing

In Section 4, we noted that most of the time spent during the autotuning run occurred due to operations other than executing the benchmarked kernels. To locate the source of the overhead, we profile the autotuner using NVIDIA Nsight Systems (`nsys`) [19]. We chose to use `nsys` over similar profilers such as `ncu`[1] [18] to capture higher-level application-wide profiling information, which was useful for diagnosing the portion of the application code the issue was arising from. We run `nsys` on a small benchmark that tasks Triton with tuning four GEMM kernel configurations across 64 data inputs [17].

The benchmark results are shown in Table 3. The profiler results reveal that the majority of the GPU time is not spent on the matrix multiplication kernels. Instead, most of the time is spent on an "vectorized elementwise" kernel. After further investigation, we discover that this elementwise kernel corresponds to an L2 cache clearing operation performed by Triton before each tuning run.

Triton clears the L2 cache prior to each tuning run to avoid having input data left over in the L2 cache after each run. With sufficiently small data, kernels could benefit from reuse of cached input data, making memory-bound kernels seem quicker than they would be in practice, impacting the benchmark quality. To perform the cache clearing, Triton allocates a `256MB` buffer on the GPU prior to benchmarking, and clears this buffer after each execution [26].

We disable the cache clearing operation during repetition time executions by removing the call to `cache.zero_()`. As shown in the table, the overhead from the cache clearing operation is quite significant. By removing this call, nearly 21 seconds is removed during our benchmark. Favorably, the matrix multiplication statistics are largely unchanged (e.g., 91,000 vs. 92,000 ns on average, minimal change in min or max), implying that the performance impact from removing the cache clearing was minor.[2]

5.2 Parallel Compilation

By default, Triton uses Just-In-Time (JIT) compilation. When a kernel is executed, Triton checks if the compiled kernel configuration is available in the `JITFunction`'s cache. If the configuration is not in the cache, Triton will compile it immediately prior to execution [26].

As noted in Section 4, Just-In-Time compilation can pose a significant runtime cost for Triton. To combat this overhead, this work implements parallel compilation within Triton.

The parallel compilation system is implemented as follows. Firstly, as threads are unsuitable for compute-bound tasks in Python due to the GIL, the parallel compilation implementation utilizes the `multiprocess` library [11, 12, 13, 5]. We use `multiprocess` instead of Python's built-in `multiprocessing` [6] so that we can change the pickling method to use `dill` [10]. This is important because `pickle` [7] cannot pickle dynamically-created Python objects, and Triton's compilation code uses many dynamically-created objects.

To generate the compilation processes, our implementation calls `spawn` from the main process. While `spawn` causes greater overhead in process creation than `fork`, it is not possible to reinitialize

[1] `ncu` is also not available on our system.

[2] `nsys` typically provides cache information directly. However, `nsys` skips collecting these results on the A40 despite our best efforts to collect this data. Further cache profiling information would be necessary to be fully confident cache behavior is minimally changed.

CUDA contexts in a forked subprocess [12]. Furthermore, to partially amortize the overhead of process creation, a `ProcessPool` is used to manage the parallel compilation tasks. Compilation tasks are enqueued using `apply_async` to allow the main python process interpreter to continue executing and enqueuing kernels during compilation. This leads the main process to receive and collect `future` objects. The main process is modified to enqueue all compilation tasks at the start of benchmarking. Once all compilation tasks are enqueued, then a `get` is issued for each `future` to wait for the process to return a compiled kernel.

Several difficulties arise in this approach. Firstly, the `JITFunction` object that represents Triton kernel code is created in the `main` scope. Because of this, when newly created processes attempt to reconstruct the `JITFunction`, they cannot find a top-level reference, and will throw an error. To avoid this issue, the parallel compilation system provides a hardcoded function to retrieve the matrix multiplication `JITFunction` object to compile. While this is not a fully satisfying solution, it satisfies the issue for the purposes of these experiments. This workaround creates an additional issue to resolve. By default, `JITFunction` objects store a cache of all the configurations that are compiled. Now that each process receives a separate `JITFunction` object copy, the caches are disparate. To resolve this, we build a cache from the precompiled results on the main process, and can utilize this cache instead during benchmarking[3].

The resulting parallel compilation system is able to schedule compilation tasks in parallel onto a process pool and build a cache of the precompiled kernels. The precompiled kernels can then be executed during benchmarking. In contrast to prior work [21], this system handles the usage of multiple CUDA contexts simultaneously *on the same device* to perform compilation.

5.3 Autotuner Modifications and Simulations

5.3.1 Autotuner Modifications

By default, Triton's autotuner provides insufficient information for the experiments sought in this work. This work modifies the Triton autotuner to collect and report compilation times, *individual*[4] kernel execution runtimes, and end-to-end tuning times, as described in Section 6.

Additionally, support for error handling is added. By default, Triton errors if any of the configurations are invalid. We find that configurations in our search space occasionally request more shared memory than is available on our system. This work's implementation simply skips any invalid configurations and prints a message to the command line. This decision impacts our results by leading to additional compilation time from invalid configurations that receive no benchmarking.

5.3.2 Pareto Frontier Simulations

To characterize the impact of parallel compilation on the Pareto frontier between tuning time and kernel performance, this work implements simulations from the results collected by the autotuner.

In the simulation results, 500 versions of the tuning hyperparameters are benchmarked with a (simulated) optimization enabled or disabled. For each set of hyperparameters, 30 simulated autotuning runs are performed. The results from the autotuning runs are averaged together.

Each tuning run is simulated with values for the following hyperparameters: `EXEC_TIME`, the simulated repetition time (a value between 10 and 100ms), and `N_CONFIG`, the number of configurations explored (a value between 1 and 80). Hyperparameters are generated using a uniform random selection of the range. Using the selected hyperparameters, the simulation code uses the following assumptions to calculate results.

[3]However, the constants used for the cache key often change between runs, as the Triton autotuner selects different non-critical parameters. We were unable to solve this issue, and thus do not report full tuning time results for parallel compilation.

[4]Without modification, Triton's autotuner reports only runtime quantiles (e.g., 20%, 50%, and 80% percentile.) The modifications made in this work collect every runtime during the repetition time.

- N_CONFIG and EXEC_TIME vary between tuning runs.
- The tuning run explores N_CONFIG randomly selected configurations from the collected data.
- All explored configurations are valid (e.g., none compiled incorrectly.)
- The repetition_time of a given configuration is calculated by adding each collected execution time until EXEC_TIME is exceeded.
- Compilation is performed once per configuration.
- Overhead time is incurred on every input.
- The total end-to-end time of a tuning run is equal to the sum of its repetition times, its compilation times, and its overhead times, across all configurations.

As stated above, after 30 separate tuning runs with the selected hyperparameters are collected, the performance values are averaged together. This reduces noise and provides more representative data about the kernel performance yielded by tuning with those parameters.

The result is 500 simulations that in total show the tradeoffs posed by varying repetition time and the search space size. For each simulation result, the available optimizations to simulate are *parallel compilation* and *early stopping*. These optimizations are simulated as follows.

Parallel Compilation. The parallel compilation optimization describes the idealized impact of parallel compilation on the Pareto frontier. To simulate parallel compilation, we add the hyperparamter compilation_factor, and the following assumptions.

- The effective compilation factor, where compilation_factor is the parallelism, is given by $\frac{N_{\text{CONFIG}}}{\lceil \frac{N_{\text{CONFIG}}}{\text{compilation_factor}} \rceil}$.
- Compilation times are divided by the effective compilation factor.

Early Stopping. The early stopping optimization describes the impact of ending repetition time early for a kernel the tuner believes to be too slow to be the optimal configuration. To simulate early stopping, we add the hyperparameters EARLY_STOP_NUM and EARLY_STOP_THRESHOLD and the following assumptions.

- After EARLY_STOP_NUM executions, the autotuner decides whether or not to skip the remainder of this configuration's execution time.
- The autotuner tracks the lowest seen execution time for each input, BEST_KNOWN.
- When deciding to skip, the autotuner skips the current configuration's remaining execution time if the average runtime for the first EARLY_STOP_NUM runtimes was greater than BEST_KNOWN $\times$ EARLY_STOP_THRESHOLD.

6 Experimental Setup

In this section, we discuss the device, software stack, kernel, and collected metrics that are used and/or collected in the evaluation.

Parameter	Description
Block Size M	Tiling parameter; size of the computation block in the M dimension.
Block Size N	Tiling parameter; size of the computation block in the N dimension.
Block Size K	Tiling parameter; size of the computation block in the K dimension.
Group Size M	The number of blocks grouped together to process rows collaboratively.
Num Stages	The number of stages the compiler uses when software-pipelining loops.
Num Warps	The number of warps (groups of 32 threads) to use per thread block.

Table 4: GEMM Configuration Parameters

6.1 Devices

A number of issues were uncovered from the original proposal during gathering preliminary results. Firstly, Triton is not intended to target Apple Silicon systems, so testing on an M1 Max is not an option. TPUs are also not an intended target [26]. Thus, two of the devices suggested in the original proposal do not suffice for this project. In this study, we collect results for an NVIDIA A40 GPU, 48GB DDR6 with a 48-core Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz (768GB RAM).

6.2 Software Stack

The main software utilized in this project is Triton. Triton is Python-based, so we utilize a Python-based stack that includes cmake [3], ninja [15], PyTorch [1], and Python 3.11 [5]. PyTorch is used for comparison and uses a traditional CUDA backend including cuBLAS [16].

The GEMM kernels are executed for 100ms of repetition time, and 25ms of warmup.

6.3 Kernels Evaluated

We collect performance benchmarks for a GEneral Matrix Multiplication (GEMM) kernel provided by Triton [26, 17]. We use a GEMM kernel because it allows us to test our Triton implementation against the reference provided by cuBLAS for correctness. The GEMM kernel also provides a variety of tunable parameters, as discussed in the following section.

6.4 Configurations Evaluated

The matrix multiplication kernel has a variety of tunable parameters that affect the performance of the kernel. A description of the parameters that are set for each configuration is shown in Table 4.

In our experiment, we allocate a search size of 100 configurations. The search space is generated randomly, following these constraints:

$$\begin{aligned}
\texttt{BLOCK_SIZE_M} &\in \{32, 64, 128, 256\},\\
\texttt{BLOCK_SIZE_N} &\in \{32, 64, 128, 256\},\\
\texttt{BLOCK_SIZE_K} &\in \{32, 64, 128, 256\},\\
\texttt{GROUP_SIZE_M} &\in \{8\},\\
\texttt{NUM_STAGES} &\in \{3, 4, 5\},\\
\texttt{NUM_WARPS} &\in \{2, 4, 8\}
\end{aligned}$$

In total, only 81 of the 100 explored configurations are valid for our A40 GPU. The most common cause for a compilation failure is a kernel requesting more shared memory than exists on the A40. The invalid configurations are caught and skipped.

6.5 Inputs Evaluated

We collect performance data for the GEMM kernel running for a variety of matrix sizes. In total, we collect 64 matrix sizes (M, N, K) such that $M, N, K \in \{256, 512, 1024, 4096\}$. This provides a diverse variety of power-of-two matrices of square and rectangular shapes.[5]

6.6 Metrics Collected

During the experiment, we collect the following direct and derived measurements.

Compilation time. The compilation time is measured using `triton.Event` (which in our environment is a reference to `torch.cuda.Event`), providing high-resolution timing results. The E profiling calls are placed directly around the call to the `compile` method that occurs within the `JITFunction`'s `run` method. To ensure compilations are not cached at the start of the benchmark, the benchmark program is run with `TRITON_ALWAYS_COMPILE=1` set [26].

Wall-clock autotuning time ("total time.") The wall-clock total time is measured using Python's `time_ns` from the `time` module [5]. The time profiling calls are placed directly around the call to the `_bench` method within the autotuner, which performs a benchmark for a given input and configuration. The total time is measured for each configuration and input and includes overhead and all execution times.

Execution Time. The individual runtimes of a GEMM kernel configuration executing for some input. Collected using `torch.cuda.Event` directly surrounding the kernel invocation call.

Repetition Time. The sum of execution times as measured above, for a given configuration and input. Corresponds to how long the GPU is occupied by our GEMM kernels.

Overhead time. A derived metric that represents the amount of time spent during benchmarking that is not due to compilation or GEMM kernel execution. Calculated by subtracting execution times and compilation time (if any) from the total time. Includes warmup time.

Kernel Performance. The TFLOPS of the kernel parameters for a given configuration and input, as returned by the autotuner. The higher the TFLOPS, the better the configuration performed.

Speedup relative to optimal. 1, if this is the best kernel for the given device and input, otherwise (autotuning time for optimal / autotuning time for this kernel). This is a purely derived metric.

7 Results and Discussion

In this section, we present the results from our study of the Triton autotuner. We discuss five main topics:

- The impact of L2 cache clearing.
- A tuning time estimation bug within Triton.
- Triton's Pareto frontier between tuning time and kernel performance.
- A comparison of parallel and sequential compilation.
- The simulated impact of early stopping.

[5]Disclaimer: The input sizes are the same as those used in an unpublished manuscript co-authored by the author of this work [8]. No experimental data or results is shared between these works.

7.1 L2 Cache Clearing

L2 Cache Clearing	Time (seconds)	Speedup
Enabled - 256MB (default)	1591.8	1.0x
Enabled - 8MB	1170.2	1.4x
Disabled - 0MB	691.2	**2.3x**

Table 5: Total Benchmark Time and Speedup by Cache Clearing Method

In Table 5, we report the time to complete our entire benchmarking process with the L2 cache clearing enabled, reduced, and disabled. With the L2 cache clearing fully disabled, the performance of executing the full benchmark described in Section 6 has a 2.3x speedup over the default cache clearing behavior – a cut of around 15 minutes. As noted in Section 5.1, removing the cache clearing does not appear to impact the execution time of the kernels, implying little impact from disabling cache clearing for GEMM kernels. However, this behavior may not hold for non-GEMM kernels. We also test the behavior of buffer targeted to the actual size of the L2 cache on our GPU. The L2 cache is 6MB on the A40 GPU. To provide confidence in the clearing, we allocate an 8MB buffer on the GPU. Performing 8MB of cache clearing as opposed to 256MB provides a 1.4x speedup while still the performing cache clearing. These results provide evidence for the claim that an option to set the cache clearing size could be a beneficial optimization (if users intend to run without cache clearing, they can merely set the parameter to zero) for Triton to provide to users.

7.2 Tuning Time Estimation Bug

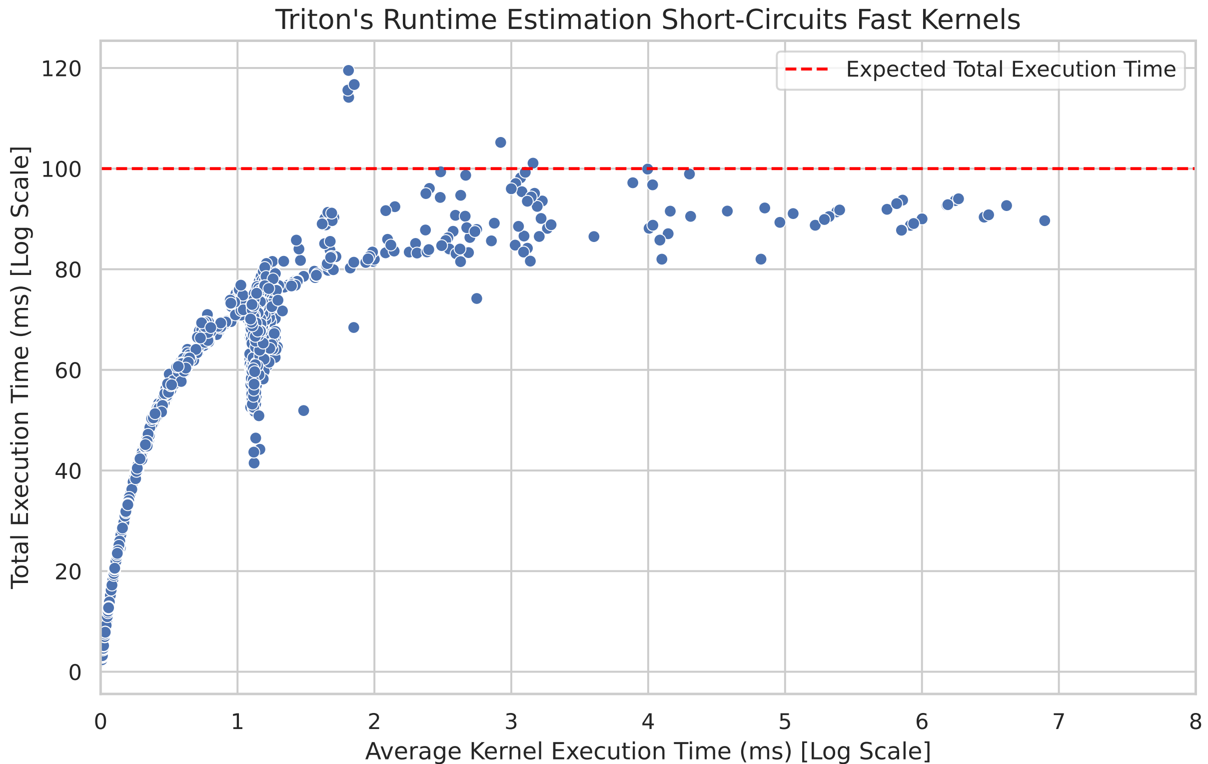

Figure 2: Triton Tuning Estimation

Triton offers users the ability to benchmark kernels for a set amount of time using the `repetition_time`

and `warmup_time` keywords to the `@triton.autotune` decorator. Our modified autotuner's benchmark finds that these estimations are often inaccurate, especially for configurations that execute quickly. Figure 2 shows the time that Triton estimates it should execute a kernel versus the average runtime of the kernel. In the figure, as the average kernel execution time decreases (e.g. the kernel performance improves), the sum execution time of the kernel decreases. In some cases, particularly when the kernel execution time is less than 0.5 miliseconds, the estimations can be over 50ms off. Further investigation reveals that the cause of this error is that Triton estimates the runtime of kernels using the first 5 executed kernels, which are not warmed up. Triton then uses this estimation to calculate the number of executions it will perform during warmup and repetition time. Because warmed-up kernels can be significantly faster than non-warm kernels, this causes the estimation to underestimate the kernel performance in many cases. Faster-executing kernels are estimated more poorly, leading to noticeable short-circuiting of faster kernels.

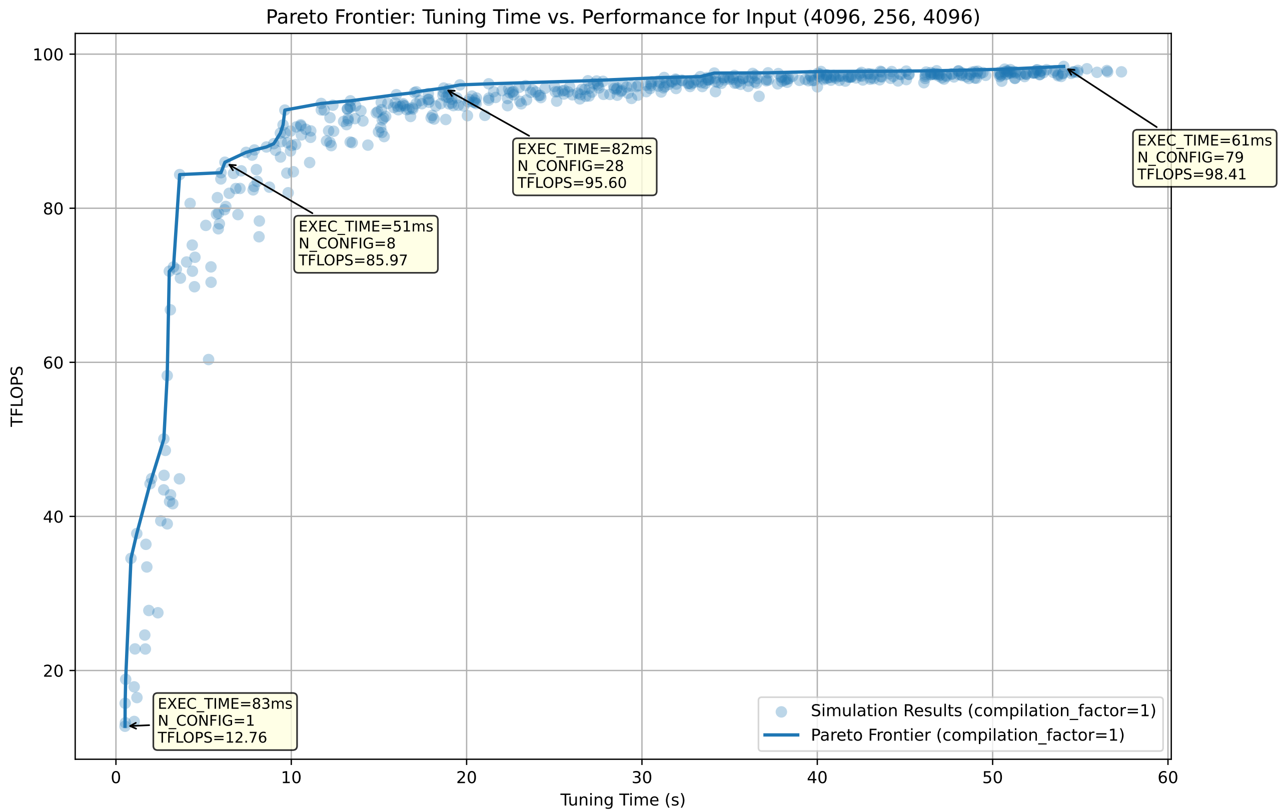

Figure 3: Pareto Frontier Simulation

7.3 Triton's Autotuning Pareto Frontier

Figure 3 contains the Pareto frontier simulation results as described in Section 5.3.2. The figure shows the tradeoff between the performance of the kernel found by the simulated autotuning run, in TFLOPS, and the tuning time. The Pareto frontier spikes initially around four seconds, where additional increase in tuning time yield large kernel performance jumps. Over time, the kernel performance gain per unit of tuning time tapers, leading to incremental gains in tuned performance after 30 seconds of tuning time. The points of the Pareto frontier are shown in Table 6. From the Pareto frontier, we note that the `N_CONFIG` appears to be the limiting factor for both TFLOPS and tuning time, increasing near-monotonically. In contrast, `EXEC_TIME` appears to be selected randomly. These results provide

Tuning Time (s)	TFLOPS	N_CONFIG	EXEC_TIME (ms)
0.52	12.76	1	83
0.53	15.74	1	26
0.55	18.85	1	59
0.86	34.55	2	86
1.19	37.77	3	28
1.94	44.24	3	31
2.03	44.91	4	47
2.73	50.06	4	88
2.92	58.27	5	66
3.04	71.82	5	75
3.28	72.38	6	51
3.63	84.37	7	92
5.99	84.61	9	31
6.22	85.97	8	51
7.42	87.25	14	35
7.90	87.56	11	63
8.58	87.95	13	40
9.00	88.38	13	58
9.38	89.75	16	22
9.53	90.81	13	61
9.63	92.74	14	97
11.71	93.60	18	52
13.37	93.94	19	71
15.99	94.74	26	13
17.02	95.00	24	51
17.13	95.12	26	38
18.70	95.60	28	82
19.62	96.01	30	13
21.26	96.15	32	73
26.92	96.54	43	28
31.27	96.97	41	36
33.32	97.08	52	70
34.13	97.55	52	81
37.19	97.58	57	12
40.21	97.75	59	87
43.60	97.77	65	22
45.07	97.80	70	32
46.81	97.84	70	95
50.68	98.05	80	17
54.05	98.41	79	61

Table 6: Pareto Frontier Points, Input (4096, 256, 4096)

solid evidence that the number of configurations explored during tuning is the greater bottleneck for the Pareto frontier given the parameters of our simulation.

7.4 Parallel Compilation

Table 7 shows the impact of implementing parallel compilation into Triton as described in Section 5.2. Five options are compared: sequential compilation, which is Triton's default, and parallel compilation

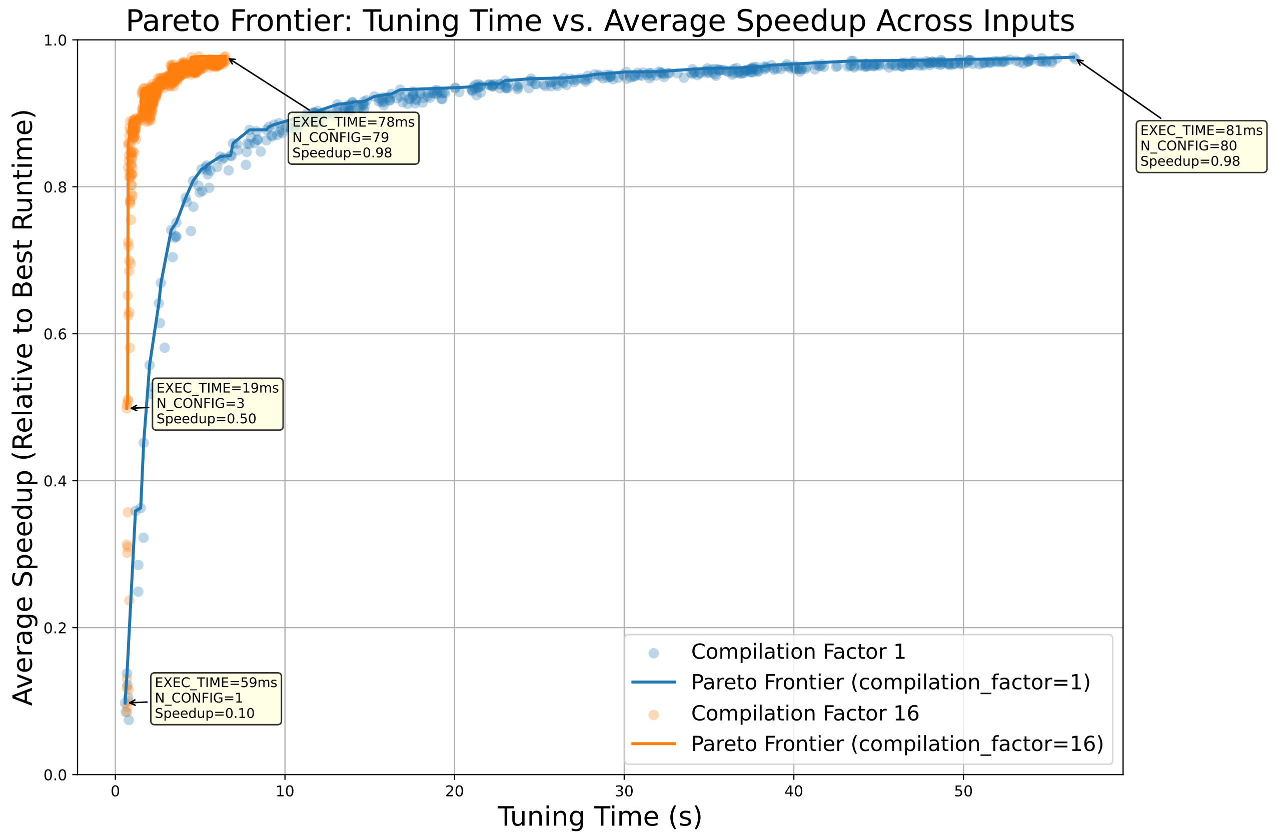

Figure 4: Parallel and Sequential Compilation Pareto Frontier Simulation

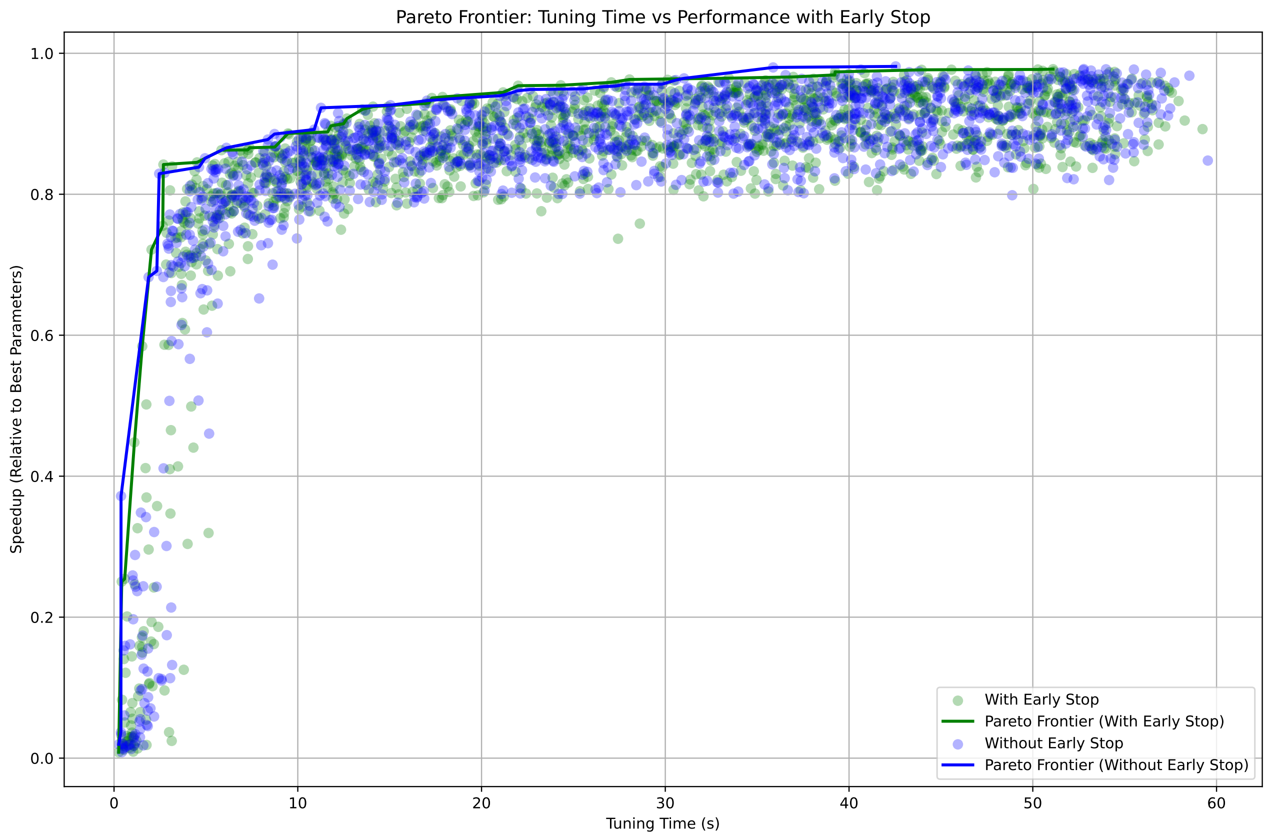

Figure 5: Pareto Frontier Simulation

Compilation Method	**Time (seconds)**	**Speedup**
Parallel Compilation (48 processes)	40.523	6.70x
Parallel Compilation (32 processes)	34.282	**7.92x**
Parallel Compilation (16 processes)	37.086	7.32x
Parallel Compilation (8 processes)	53.007	5.12x
Sequential Compilation	271.560	1.00x

Table 7: Compilation Time and Speedup by Method

with 8, 16, 32, or 48 processes. Parallel compilation in all four cases widely outperforms sequential compilation in compilation time. However, the speedups gathered by the parallel compilation implementation are significantly off of the ideal scaling. Monitoring using `htop` [14] reveals that 16 or 32 CPU cores on the machine are utilized when spawning 16 or 32 processes, respectively. However, the speedup of 32 processes over 16 processes is minor, providing just a 1.08x speedup despite using double the compute.

Figure 4 plots the average speedup relative to optimal of the tuned kernels against the tuning time utilized in each simulated run. For this simulation, all 64 inputs are included in the results, and are varied along with `N_CONFIGS` and `EXEC_TIME` (e.g., one selection per 30 simulations). The figure compares the Pareto frontiers of ideal 16x parallel compilation (shown in orange) to that of sequential compilation (shown in blue). Parallel compilation produces a majorly improved Pareto frontier. Recall that the main bottleneck to tuning is the `N_CONFIGS` hyperparameter. As the cost of compilation decreases, the autotuner is able to explore more configurations at a signifcantly reduced cost. This pushes the ideal Pareto frontier significantly to the left. Overall, the results are highly promising for parallel compilation to be an effective optimization.

7.5 Early Stopping

Figure 5 shows the speedup relative to optimal for all inputs on a tuning run. The experimental setup is comparable to that in Section 7.4, with the exception of using early stop to stop tuning suboptimal configurations early as opposed to parallel compilation. The hyperparameters used in the figure are `TUNING_TIME_THRESHOLD` uniformly randomly chosen between 1 and 1.1, and `EARLY_STOP_NUM` uniformly random chosen between 2 and 100. Early stop is shown in green, whereas default Triton is shown in blue. Overall, early stopping has no apparent effect on the Pareto frontier. We attempt several ranges of the hyperparameters but none yield any apparent speedup. Investigating this further reveals that the autotuner process often takes a significant amount of time to find a suitable "best known" kernel for comparison. This leads early stopping to trigger both uncommonly – because kernels within threshold range are common – and ineffectively – as the early stop tends to remove kernels with little time left to benchmark. In summary, these results provide an indication that early stopping of autotuning may not be an effective optimization.

8 Conclusion

This work investigates the sources of autotuning cost within Triton and finds that significant mitigable costs exist within Triton's autotuning implementation. We propose three main optimizations: parallel compilation, disabling cache clearing, and early stopping of tuning, and find supporting evidence for the first two optimizations. Evaluations show that these optimizations lead to significant performance benefits. Furthermore, our results provide insights for practitioners into tuning strategy. Firstly, our findings support the claim that for GEMM operations on the A40 GPU, exploring additional configurations is the primary bottleneck for tuning effectiveness as compared to running kernels for additional repetition time. Additionally, we show that Triton's tuning time estimation consistently underestimates repetition times for fast-executing kernel configurations.

9 Disclosures

9.1 Generative AI Disclosure.

This report was partially created from content whose creation was assisted by GPT-4o and o1 from OpenAI. The AI assisted in tasks such as generating code for graphs, creating LaTeX tables, debugging, and ideation. All information has been reviewed and edited by a human to ensure accuracy and quality [4].

References

[1] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://pytorch.org/assets/pytorch2-2.pdf.

[2] Prasanna Balaprakash et al. "Autotuning in High-Performance Computing Applications". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083. DOI: 10.1109/JPROC.2018.2841200.

[3] *CMake Documentation*. URL: https://cmake.org/documentation/.

[4] *Disclosing the Use of AI at Princeton University*. Oct. 2024. URL: https://libguides.princeton.edu/generativeAI/disclosure.

[5] Python Software Foundation. URL: https://docs.python.org/3.11/reference/index.html.

[6] Python Software Foundation. URL: https://docs.python.org/3/library/multiprocessing.html.

[7] Python Software Foundation. *pickle — Python object serialization — Python 3.X documentation*. URL: https://docs.python.org/3/library/pickle.html.

[8] Robert Hochgraf and Sreepathi Pai. "Portability Tuning of Linear Algebra Kernels". unpublished. N.D.

[9] Junhong Liu et al. "An Autotuning Protocol to Rapidly Build Autotuners". In: *ACM Trans. Parallel Comput.* 5.2 (Jan. 2019). ISSN: 2329-4949. DOI: 10.1145/3291527. URL: https://doi-org.ezproxy.rit.edu/10.1145/3291527.

[10] Michael McKerns. *dill*. URL: https://pypi.org/project/dill/.

[11] Michael McKerns. *multiprocess*. URL: https://pypi.org/project/multiprocess/.

[12] Michael McKerns and Michael Aivazis. *pathos: a framework for heterogeneous computing*. 2010. URL: https://uqfoundation.github.io/project/pathos.

[13] Michael M. McKerns et al. "Building a Framework for Predictive Science". In: *CoRR* abs/1202.1056 (2012). arXiv: 1202.1056. URL: http://arxiv.org/abs/1202.1056.

[14] Hisham Muhammad. *GitHub htop-dev/htop*. Jan. 2004. URL: https://github.com/htop-dev/htop.

[15] ninja-build. *GitHub ninja-build/ninja: a small build system with a focus on speed*. May 2024. URL: https://github.com/ninja-build/ninja.

[16] NVIDIA. *Basic Linear Algebra on NVIDIA GPUs*. July 2013. URL: https://developer.nvidia.com/cublas.

[17] NVIDIA. *Matrix Multiplication Background User's Guide*. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[18] NVIDIA. *Nsight Compute Documentation*. 2024. URL: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[19] NVIDIA. *NVIDIA Nsight Systems*. URL: https://developer.nvidia.com/nsight-systems.

[20] Filip Petrovič et al. "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit". In: *Future Generation Computer Systems* 108 (2020), pp. 161–177. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.02.069. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19327360.

[21] pytorch. *Github - pytorch/pytorch*. 2016. URL: https://github.com/pytorch/pytorch/blob/main/torch/_inductor/runtime/triton_heuristics.py#L175.

[22] pytorch. *pytorch/torch/_inductor/remote_cache.py at main · pytorch/pytorch*. 2016. URL: https://github.com/pytorch/pytorch/blob/main/torch/%5C_inductor/remote%5C_cache.py#L325.

[23] pytorch. *pytorch/torch/_inductor/runtime/benchmarking.py · pytorch/pytorch*. 2016. URL: https://github.com/pytorch/pytorch/blob/2cc01cc6d3ad2aff47e8460667ba654b2e4c9f21/torch/%5C_inductor/runtime/benchmarking.py#L172.

[24] Ari Rasch et al. "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)". In: *ACM Trans. Archit. Code Optim.* 18.1 (Jan. 2021). ISSN: 1544-3566. DOI: 10.1145/3427093. URL: https://doi-org.ezproxy.rit.edu/10.1145/3427093.

[25] Burkhard Ringlein and Caleb Thomas. *GitHub - IBM/triton-dejavu: Framework to reduce autotune overhead to zero for well known deployments.* 2024. URL: https://github.com/IBM/triton-dejavu.

[26] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi-org.ezproxy.rit.edu/10.1145/3315508.3329973.

[27] *Topics - Pareto Front*. URL: https://www.sciencedirect.com/topics/engineering/pareto-front.

[28] Yanli Zhao et al. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel". In: *Proc. VLDB Endow.* 16.12 (Aug. 2023), pp. 3848–3860. ISSN: 2150-8097. DOI: 10.14778/3611540.3611569. URL: https://doi-org.ezproxy.rit.edu/10.14778/3611540.3611569.