Universiteit Leiden
The Netherlands

Opleiding Informatica

Analyzing Optimization Variables in GPU Kernel Tuning

Thomas Jansen

Supervisors:
Dr. B.J.C. van Werkhoven & Dr. S.J. Heldens

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/07/2025

Abstract

Graphical Processing Units (GPUs) are used for an ever-increasing amount of computational tasks, particularly in the training of Neural Networks. To optimize performance and resource usage, GPU code must be finely tuned, which often involves a vast configuration space. Manually exploring this space is infeasible, which is why auto-tuning frameworks like Kernel Tuner have emerged. This thesis investigates the correlation and sensitivity of tuning parameters in GPU kernel tuning, and extends the Kernel Tuner dashboard with interactive graphs to visualize these effects. Experimental evaluation on a range of kernels shows that certain parameters consistently exhibit high sensitivity (e.g., block size and unroll factor), while others have negligible influence.

Contents

1 Introduction

In recent decades, computing has undergone a transformative evolution, particularly through advances in parallel processing. At the heart of this transformation is the Graphical Processing Unit (GPU). Originally designed to accelerate graphics rendering, GPUs are now widely adopted in High-Performance Computing (HPC), scientific simulations, and deep learning applications. This shift is largely driven by the GPU's architecture, which enables the concurrent execution of thousands of threads, making it ideally suited for tasks with high degrees of parallelism.

Despite their power, efficiently utilizing GPUs remains a significant challenge. When developers write GPU kernel code—typically small, performance-critical routines—they must fine-tune numerous parameters (such as block size, grid size, memory usage strategies, etc.) to achieve optimal performance. Manually navigating this large configuration space is laborious and error-prone. As a result, auto-tuning tools like Kernel Tuner have emerged to automate this process. However, while these tools find good configurations, they often function as black boxes, offering limited insight into why certain parameter settings lead to better performance.

This thesis addresses the following problem: How can we gain deeper insight into which GPU kernel parameters most significantly affect performance, both individually and in combination? Understanding these relationships not only helps optimize new kernels more efficiently, but also enhances interpretability, reproducibility, and trust in the tuning process.

To solve this problem, the Kernel Tuner Dashboard, a visualization tool that supports Kernel Tuner, is extended by incorporating analytical methods to identify and rank parameter importance. Using statistical analysis and interaction effects, the enhanced dashboard provides users with actionable insight into the sensitivity of the objective function (e.g., execution time) to various tuning parameters.

This insight is valuable for multiple reasons: It aids developers in understanding performance bottlenecks, guides manual and automated tuning strategies, and helps in developing heuristics or models for parameter selection. Ultimately, this contributes to more efficient and explainable GPU code optimization.

1.1 Thesis overview

Section 2 gives background information and discusses related work on GPU kernel tuning and performance modeling. Section 3 details the experimental setup and improvements made to the Kernel Tuner Dashboard. Section 4 presents empirical results and analyses in multiple kernel benchmarks. Section 5 discusses the resulting analyses. Section 6 concludes with directions for future work.

2 Background and related work

Before diving into the technical implementation of this thesis, it is important to provide background knowledge on GPU computing and the tools used throughout this work. This section starts by explaining the basics of GPU programming and its differences with CPU computing (Section 2.1). Then, commonly used GPU kernels relevant to this research and how they are optimized for performance are discussed (Section 2.2). Following that, the section introduces the Kernel Tuner

tool, which enables automated performance tuning of GPU kernels (Section 2.3). Finally, a brief overview of related academic work provides context for the novelty and relevance of this thesis (Section 2.4).

2.1 GPU programming

Modern GPUs are designed to accelerate data-parallel computations by executing thousands of lightweight threads concurrently. To take advantage of this capability, developers write specialized functions called *kernels*, which are executed on the GPU (the *device*) while the main application code runs on the CPU (the *host*). Kernels are launched in parallel by specifying a *grid* of *thread blocks*, where each block contains multiple threads. This hierarchical execution model enables massive parallelism, but also introduces complexity when optimizing for performance.

In GPU programming models, such as CUDA and OpenCL, developers manage memory and execution explicitly. Threads within a block can communicate through shared memory and can be synchronized using barrier operations. Thread blocks execute independently and can be scheduled in any order.

Different GPU architectures (e.g., Nvidia's Volta, Turing, Ampere) vary in their number of streaming multiprocessors (SMs), memory bandwidth, and instruction throughput. These differences influence the performance of GPU kernels and motivate the need for auto-tuning tools that can adapt configurations to specific hardware.

2.2 GPU kernels

A kernel is a piece of code that the GPU executes on many threads, in parallel, where each thread does part of the work. This differs from a function run on a CPU, where instructions are run serially.

Different GPU kernels exist for different use cases. This subsection outlines the kernels that were studied for this thesis, the description of which are outlined by Tørring's et al. in their paper on a benchmarking suite for kernel tuners known as BAT [TvWP+23]. Each subsection also lists and describes all the tunable parameters for each kernel.

It is important to note that for these kernels, there exists a set of constraints that limit the number of valid parameter value combinations. For example, the total "area" of a thread block is limited to 1024, i.e. `block_size_x` $\cdot$ `block_size_y` $<= 1024$

2.2.1 Convolution kernel

The convolution kernel [vMBS14] places a filter on a given input matrix and computes the weighted sums. Figure 1 shows an example of how this works.

Especially 2-dimensional convolution kernels have use cases in image processing, such as sharpening and edge detection. The tunable parameters for the convolution kernel are shown in Table 1.

`block_size` denotes the thread block dimensions, `tile_size` the dimensions of the output pixels by each thread. Enabling `use_padding` uses the padding scheme in shared memory to avoid shared memory bank conflicts. `read_only`, when enabled, loads the input elements from global memory through a read-only cache. Lastly, `use_shmem` enables or disables the use of shared memory.

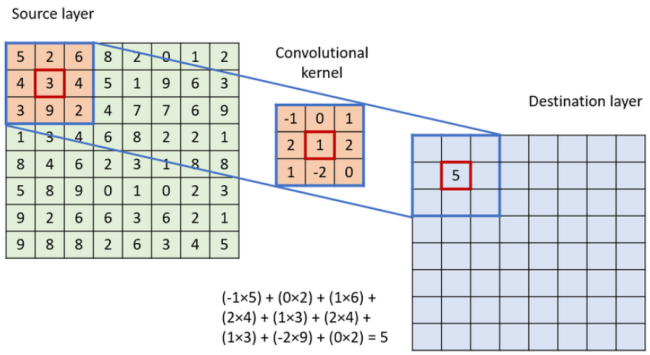


Figure 1: The workings of a convolution kernel [Jai24].

Parameter	Possible values
`block_size_x` (`bsx`)	{1,2,4,8,16,32,48, 64,80,96,112,128}
`block_size_y` (`bsy`)	{1,2,4,8,16,32}
`tile_size_x` (`tsx`)	{1,2,3,4,5,6,7,8}
`tile_size_y` (`tsy`)	{1,2,3,4,5,6,7,8}
`read_only` (`ro`)	{0,1}
`use_padding` (`pad`)	{0,1}
`use_shmem` (`shm`)	{0,1}

Table 1: Tunable parameters for convolution kernel.

2.2.2 Dedispersion kernel

The dedispersion kernel has its origins in astronomy. *Dispersion* refers to the separation of frequencies in waves (light, radio, etc.) through space. Dispersion occurs because lower frequencies are slowed down more than higher frequencies. *Dedispersion* is the reverse process; the reconstruction of the original signal. To do this, the scale of dispersion (dispersion measure) needs to be found, which depends on how far away the source of the signal is. This distance is usually unknown, so a dedispersion kernel tries out many different dispersion measures until it finds a peak.

The tunable parameters for the dedispersion kernel are shown in Table 2.

Parameter	Possible values
`block_size_x` (`bsx`)	{1,2,4,8} ∪ {16, 32, 48, ..., 512}
`block_size_y` (`bsy`)	{4n for 4n ∈ [4,128]}
`tile_size_x` (`tsx`)	{n for n ∈ [1,16]}
`tile_size_y` (`tsy`)	{n for n ∈ [1,16]}
`tile_stride_x` (`trx`)	{0,1}
`tile_stride_y` (`try`)	{0,1}
`loop_unroll_factor`	—
`blocks_per_sm`	—

Table 2: Tunable parameters for the dedispersion kernel.

The parameters `block_size_x`, `block_size_y`, `tile_size_x` and `tile_size_y` are analogous to the parameters of the same names used in the convolution kernel. The parameters `tile_stride_x` and `tile_stride_y` control the stride used to vary the amount of work per thread, in the x and y directions, respectively. That is to say, if `tile_stride_x` is equal to 1, each thread will process `tile_size_x` samples that are `block_size_x` apart. If the parameters are equal to 0, each thread will just process `tile_size_x` consecutive samples. Finally, `loop_unroll_factor` and `blocks_per_sm` are parameters that were not used for the purpose of this thesis.

2.2.3 GEMM kernel

The Generalized dense matrix-matrix multiplication (GEMM, also known as xGEMM) kernel [Nug18] does what it says on the tin; it multiplies two matrices, A and B, into an output matrix C, using scalars α and β:

$$C = \alpha AB + \beta C$$

Figure 2 shows this matrix multiplication visually, albeit without the scalars.

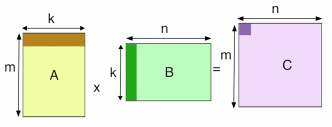


Figure 2: Matrix multiplication [War15].

GEMM is one of the most abundantly used kernels. Its tunable parameters are listed in Table 3.

Parameter	Possible values
`MWG`	{16,32,64,128}
`NWG`	{16,32,64,128}
`KWG`	{16,32}
`MDIMC`	{8,16,32}
`NDIMC`	{8,16,32}
`MDIMA`	{8,16,32}
`NDIMB`	{8,16,32}
`VWM`	{1,2,4,8}
`VWN`	{1,2,4,8}
`STRM`	{0,1}
`STRN`	{0,1}
`SA`	{0,1}
`SB`	{0,1}

Table 3: Tunable parameters for the GEMM kernel.

Here, `MWG`, `NWG` and `KWG` change how much work each thread block gets assigned, while `MDIMC` and `NDIMC` control the size of each thread block. `MDIMA` and `NDIMB` decide the amount of shared memory, and `VWM` and `VWN` describe the vector widths used for global memory. `STRM` and `STRN`

determine the use of tile strides in an analogous way to the dedispersion kernel. Lastly, `SA` and `SB` dictate whether shared memory is used for matrix A and B, respectively. The GEMM kernel has by far the most amount of tuning parameters. This makes it the most expensive to optimize. To combat the amount of parameters, they do not have as many possible values.

Additionally, the GEMM kernel may use the parameter `nvml_gr_clock`, which effectively boosts the GPU's clock speed. Clock speed is measured in Hz (s^{-1}) and represents how many operations the GPU can perform per second.

2.2.4 Hotspot kernel

The final kernel that was evaluated for this thesis is the Hotspot kernel [CBM+09]. This kernel is used to estimate processor temperatures by iteratively solving differential equations. This thesis looks at the version of Hotspot written for BAT. The tunable parameters for the Hotspot kernel can be found in Table 4.

Parameter	Possible values
`block_size_x` (`bsx`)	{1,2,4,8} ∪ {32, 64, 96, ...,1024}
`block_size_y` (`bsy`)	{1,2,4,8,16,32}
`tile_size_x` (`tsx`)	{n for n ∈ [1,10]}
`tile_size_y` (`tsy`)	{n for n ∈ [1,10]}
`temporal_tiling_factor` (`ttf`)	{n for n ∈ [1,10]}
`loop_unroll_factor_t` (`luf`)	{n for n ∈ [1,10]}
`sh_power` (`shp`)	{0,1}
`blocks_per_sm`	—

Table 4: Tunable parameters for the Hotspot kernel.

Here, the parameters `block_size_x`, `block_size_y`, `tile_size_x` and `tile_size_y` are analogous to their counterparts in the convolution and dedispersion kernels. The parameter `temporal_tiling_factor` decides the number of stencil operations performed in a single kernel launch. More information on this parameter can be found in Hijma et al's paper [HHS+23]. The parameter `loop_unroll_factor_t`, which was left unused in the dedispersion kernel, determines to what degree loop unrolling is applied. Finally, the `sh_power` parameter tells whether shared memory is used as a cache for storing the input power currents. Similarly to the dedispersion kernel, the `blocks_per_sm` parameter is not used.

2.3 Automatic tuning

It is in a developer's best interest to optimize these kernels. However, this often turns out to be difficult, mainly because of the many different methods of optimization. For this reason, automatic kernel optimization tools (auto-tuners) have been in high demand.
One such tuner is the Kernel Tuner [vW19]. Kernel Tuner was originally created by Ben van Werkhoven, with many more developers having joined the project since then. Kernel Tuner is publicly available on GitHub[1]. It is written for multiple programming models, including CUDA, OpenCL and HIP, and has an interactive dashboard, see the next section.

[1] https://github.com/KernelTuner/kernel_tuner

2.3.1 KTDashboard

KTDashboard is an interactive dashboard developed for Kernel Tuner, that, according to its README.md file, "KTdashboard allows you to monitor, analyze, and visualize an active or completed auto-tuning run of Kernel Tuner [...]" [vW]. The dashboard offers insight for developers as to what kernel configurations are the most optimal. KTDashboard is available on its public GitHub repository[2].

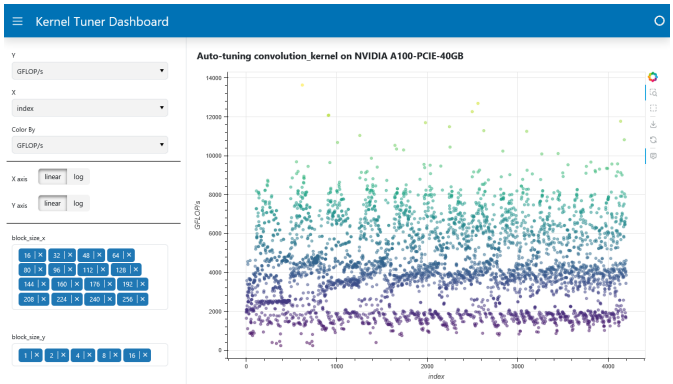


Figure 3: Screenshot of KTDashboard, January 2024.

As visible in Figure 3, the original dashboard shows all different configurations based on two tuning parameters, here `index` and `GFLOP/s`. The `index` "parameter" refers to that configuration's index in the data collected by the tuner. `GFLOP/s` means how many billion floating-point operations the GPU is able to do per second. The dashboard then colors every point according to a third parameter, usually an outcome variable, here again `GFLOP/s`. In other words, this specific screenshot does not tell a whole lot, other than which indices performed really well. Hovering over any point shows which values were used for that configuration, as well as the results for time, energy consumption, etc.

2.4 Related Work

The optimization of GPU kernels has received significant attention due to the increasing computational demands of parallel workloads. Early work by Volkov and Demmel [VD08] demonstrated that hand-tuned GPU kernels could far outperform compiler-generated ones, emphasizing the importance of tuning.

Other kernel optimisation tools, such as OpenTuner [AKV+14] from Ansel et al. and Whaley's and Dongarra's ATLAS [WD98], have explored similar avenues in CPU or linear algebra optimization, contributing strategies such as multi-armed bandits and genetic algorithms. Kernel Tuner distinguishes itself by focusing on CUDA and OpenCL kernels and integrating seamlessly with Jupyter-based workflows.

Baghsorkhi explored sensitivity and correlation analysis of tuning parameters in contexts like design space exploration [BDP+10], where they introduced adaptive performance modeling for GPU kernels, using parameter sensitivity to guide optimization.

[2]https://github.com/KernelTuner/dashboard

Meanwhile, Tørring et al. [TvWP+23] presented the BAT benchmarking suite and performed a detailed analysis of parameter sensitivity across a range of GPU kernels, highlighting which parameters most affect performance.

Jain et al. [Jai24] also conducted a comprehensive study on parameter sensitivity, employing correlation and variance-based techniques to analyze the effects of different tuning parameters on GPU kernel efficiency.

Xiang and Agrawat [LA21] experimented searching the search space with their shrinking sample strategy where they look for specific combinations of parameter values. They were able to achieve "around 99% percent of the performance from exhaustive search (on average) with orders of magnitude much less tuning time".

This thesis builds on this foundation by applying and visualizing sensitivity and correlation analyses within Kernel Tuner's framework, offering practical tools to guide kernel developers in optimizing GPU performance more effectively.

3 Methodology

This section outlines the steps taken to achieve the goals of this thesis. The first goal is to investigate which sensitivity and significance metrics work well for analyzing auto-tuning search space. The second goal is to extend Kernel Tuner's dashboard to visualize these metrics. The first goal can be broken down in calculating the correlations (Section 3.1) and the first-order sensitivities of parameters (Section 3.2). The second goal is achieved by placing interactive graphs displaying the metrics from the first goal on an empty space of the dashboard, see Section 3.3.

Kernel Tuner's and its dashboard are written in Python. Their source code can be found on their respective GitHub repositories [vW19][vW].

3.1 Parameter significance

To calculate parameter significance, Pandas' [WM10] `corr()` function is used. This function allows for a correlation method to be chosen, with the choices being Pearson's [Pea95], Spearman's [Spe04] or Kendall's [Spe38] methods. These methods each have their own uses, with Pearson's method measuring linear correlation, while Spearman's method and Kendall's method measure rank correlation. The tuning parameters are not ranked, and so, this thesis exclusively focuses on Pearson's method.

Pearson's method gives the ratio between the covariance of two variables–X and Y–and the product of their standard deviations:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

where $\operatorname{cov}(X,Y)$ is the mean of the product of X and Y's deviations from their respective means:

$$\operatorname{cov}(X,Y) = E[(X - E[X])(Y - E[Y])]$$

where $E[X]$ is the mean of X.

Setting Y as the tuning objective; one by one, each tuning parameter is filled in as X and their correlation $\rho_{X,Y}$ recorded. This always results in a value of -1 to 1, assigning a positive or negative "score" to the parameters. This immediately shows which parameters have a positive or negative impact on the objective, but suffers from the drawback of not being able to look at subsets of parameters.

3.2 Sensitivity analysis

While interesting in its own right, variable significance has one major downside: the parameters are exclusively looked at by themselves. It has, however, been proven that the interaction between specific tuning parameters can have meaningful impact on the overall kernel performance [HHS+23]. This is where sensitivity analysis comes into play. Sensitivity analysis (SA) is performed using the Python library SALib [IUH22][HU17]. SALib employs variance-based sensitivity analysis, also known as the Sobol' method [Sob01]. This method is Monte-Carlo based and decomposes a model's output variance into each variable's influence on that variance.

The sensitivity of each input is also called the sensitivity index, and can be of a specific *order*: First-order indices measure the contribution of a single input variable on the output variance; Second-order indices measure the contribution of two inputs; Total-order indices measure the contribution of a model input, including its first-order and higher-order effects.

If Y is the output variable, and $X = \{X_1, X_2, ..., X_d\}$ the set of input variables, the variance of Y – $Var(Y)$ – can be written as

$$Var(Y) = \sum_{i=1}^{d} V_i + \sum_{i<j}^{d} V_{ij} + ... + V_{12...d}$$

with

$$V_i = Var_{X_i}(E_{X_{\sim i}}(Y|X_i))$$
$$V_{ij} = Var_{X_{ij}}(E_{X_{\sim ij}}(Y|X_i, X_j)) - V_i - V_j$$

and so on. $X_{\sim i}$ denotes all variables *except* X_i, and $E(X)$. A first-order index is then measured as:

$$S_i = \frac{V_i}{Var(Y)}$$

S_i is the contribution of X_i *alone* on the ouput variance, averaged out over variations in other input parameters.

A total-order index is measured as:

$$S_{Ti} = 1 - \frac{Var_{X_{\sim i}}(E_{X_i}(Y|X_{\sim i}))}{Var(Y)}$$

S_{Ti} is then the contribution of X_i on the output variance, including all variance caused by its interactions with any other input variables.

3.3 Expanding Kernel Tuner Dashboard

To expand upon Kernel Tuner's dashboard, these analyses need to be incorporated into the already existing functionality. The dashboard has its plots constructed using Bokeh [Bok18] and Panel [Hol], two open-source Python modules that allow the creation of custom interactive plots on a local webpage.

4 Experiments and results

For the purposes of this thesis, only Nvidia and AMD GPUs are studied, and for lack of a compatible GPU, only readily available cache files are studied. In this context, a "cache file" is the output file generated by Kernel Tuner that contains the raw data of the GPU's performance on the kernel. This is also the file that KTDashboard reads to display this information. Experiments were performed on data from Milo Lurati's research on the differences between tuning Nvidia and AMD GPUs [LHSvW24] (Nvidia vs. AMD; from here on: NvA). Their work was aimed at expanding Kernel Tuner with HIP capabilities and to that end, they generated many usable cache files, mainly focused on execution time. These files are accessible on the public GitHub repository associated with the paper [LHvW]. In a later stage of the project, more cache files by Lurati were found on Floris-Jan Willemsen's GitHub repo for benchmarking auto-tuners [Wil]. The results of these experiments are found in Sections 4.1.1 and 4.1.2.

Furthermore, experiments were performed on data obtained from Schoonhoven's paper on optimizing GPU energy consumption [SVVWB22]. This data concerns the GEMM kernel and is obtained from Nvidia's A100 GPU, see Table 5. The results of this cache file are visible in Section 4.1.3.

The GPUs that were used for analyzing the cache files are listed in Table 5, which was partially obtained from Lurati's paper on bringing automatic tuning to HIP [LHSvW24].

GPU	Year	Architecture	Cores	Memory	Cache	Bandwidth (GB/s)	Peak SP (TFLOPS/s)
Nvidia A100 [Nvia]	2020	Ampere	6912	40 GB HBM2e	40 MB L2	1555	19.5
Nvidia RTX A4000 [Nvib]	2021	Ampere	6144	16 GB GDDR6	4 MB L2	448	17.8
Nvidia RTX A6000 [Nvic]	2020	Ampere	10752	48 GB GDDR6	6 MB L2	768	38.7
AMD Instinct MI250X* [AMDa]	2021	CDNA 2	7040	64 GB HBM2e	8 MB L2	1638	28.2
AMD Radeon PRO W6600 [AMDb]	2021	RDNA 2	1792	16 GB GDDR6	32 MB L3	224	10.4
AMD Radeon PRO W7800 [AMDc]	2023	RDNA 3	4480	32 GB GDDR6	64 MB L3	576	45.2

Table 5: GPUs used in experiments. *Only one out of two dies of MI250X is used.

These GPUs were chosen for being relatively new and for their equal split among Nvidia vs. AMD cards.

From here on, these cards are simply referred to by their series number (i.e. Nvidia A100 is referred to as A100).

4.1 Results from significance and sensitivity analyses

4.1.1 Significance analysis

The NvA cache files paint a difference between significant parameters between GPUs and not necessarily between different kernels. As mentioned in Section 2.2, different kernels have very

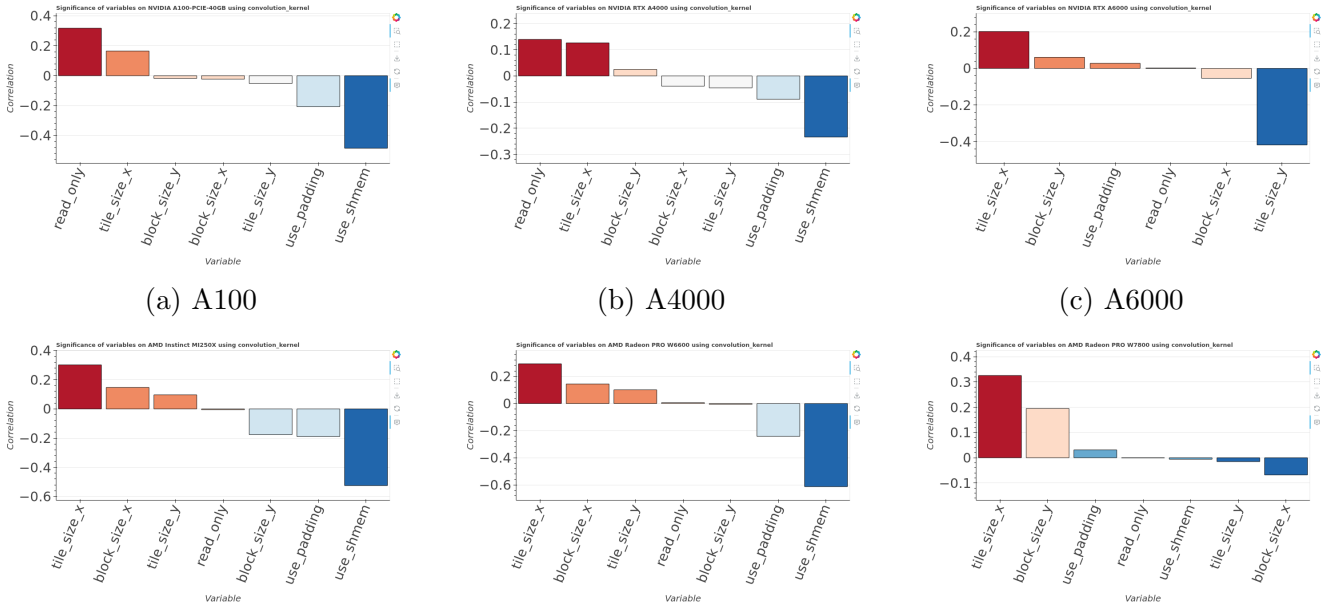
different tuning parameters associated with them and this makes it difficult to compare them.

Figures 4 through 7 show parameter significance analyses obtained from these cache files.

In Figures 4, 5 and 6, the objective is time. Positive values indicate an undesirable increase in GPU time. In Figure 7, the objective is GFLOP/s. Positive values indicate a desirable increase in performance.

Figures 4 through 7 show parameter significance for the four different kernels discussed in Section 2.2 on the GPUs mentioned above. The top two rows in these figures show Nvidia GPUs while the bottom two rows show GPUs from AMD. These figures have their contents discussed in Section 5.1.

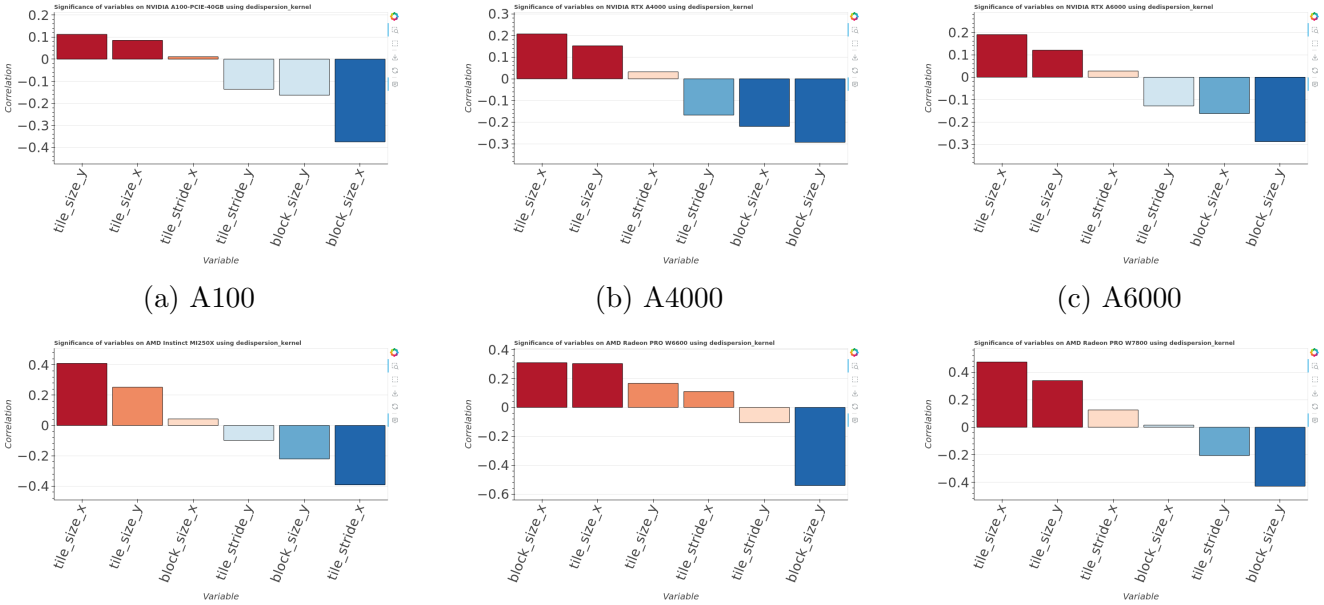
It should be noted that Figure 4c does not have a bar for the parameter `sh_mem`. This is because `sh_mem` was not present in the cache files for that specific GPU.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

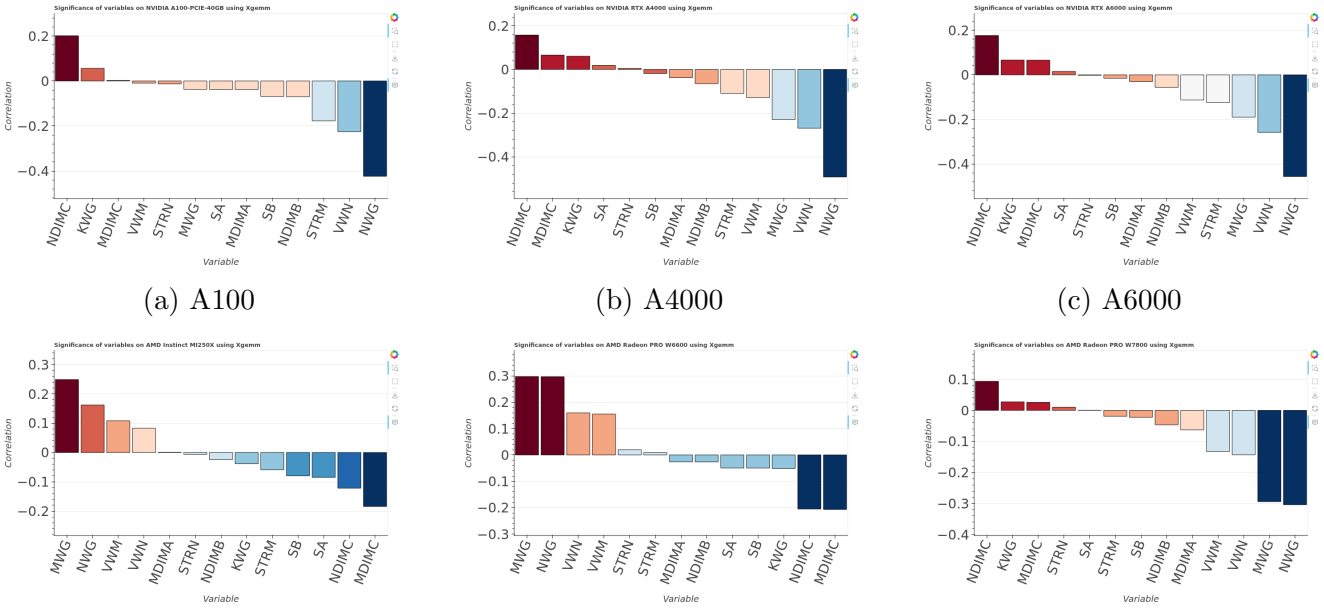
Figure 4: Parameter significance on the convolution kernel.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

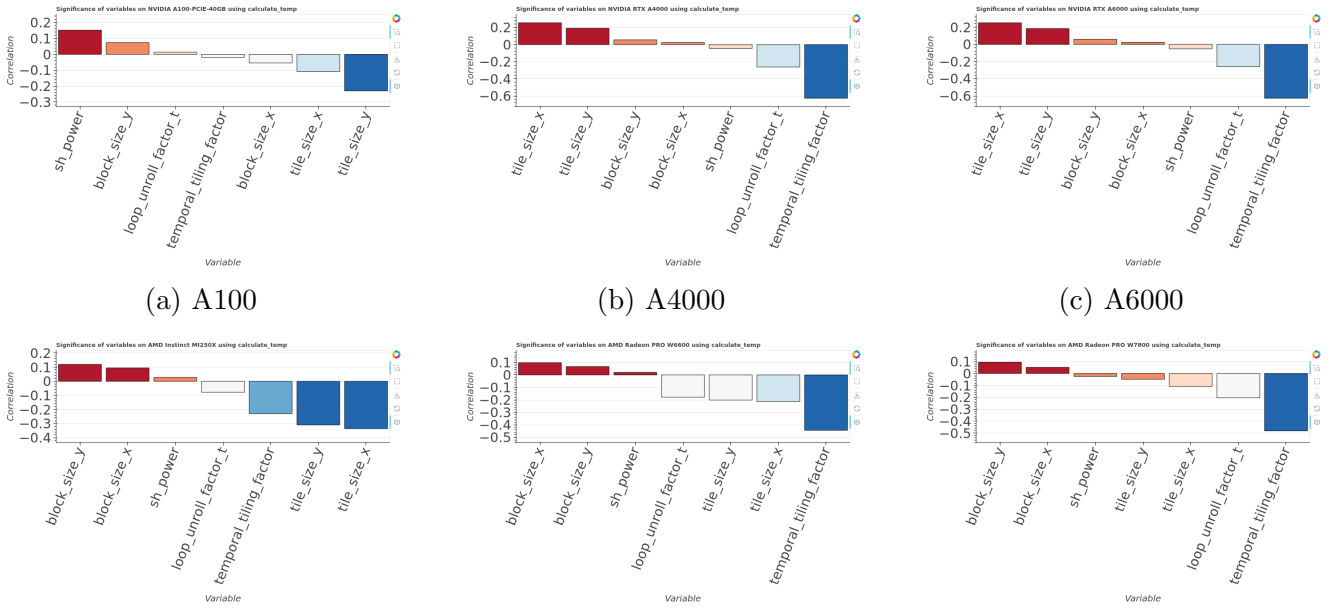
Figure 5: Parameter significance on the dedispersion kernel.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

Figure 6: Parameter significance on the GEMM kernel.



(a) A100 (b) A4000 (c) A6000

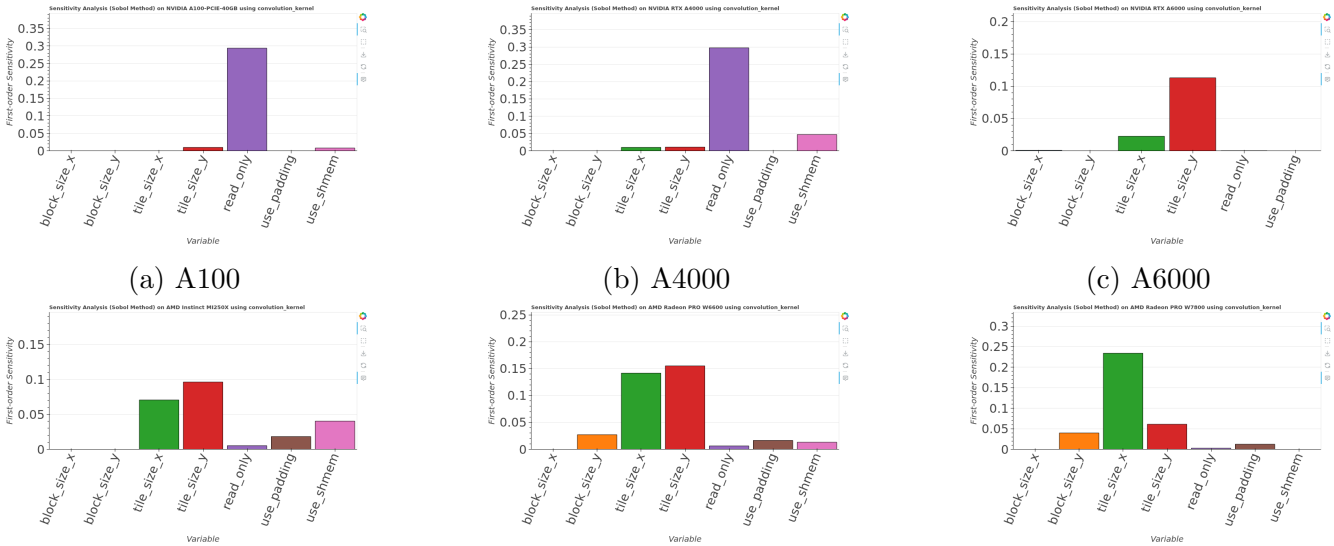
(d) MI250X (e) W6600 (f) W7800

Figure 7: Parameter significance on the Hotspot kernel.

4.1.2 Sensitivity analyses

All parameters Figures 8 through 11 show first-order sensitivity analyses performed on the four kernels discussed in Section 2.2. These analyses look at *all* parameters used for that kernel. The contents of these figures are discussed in Section 5.2.

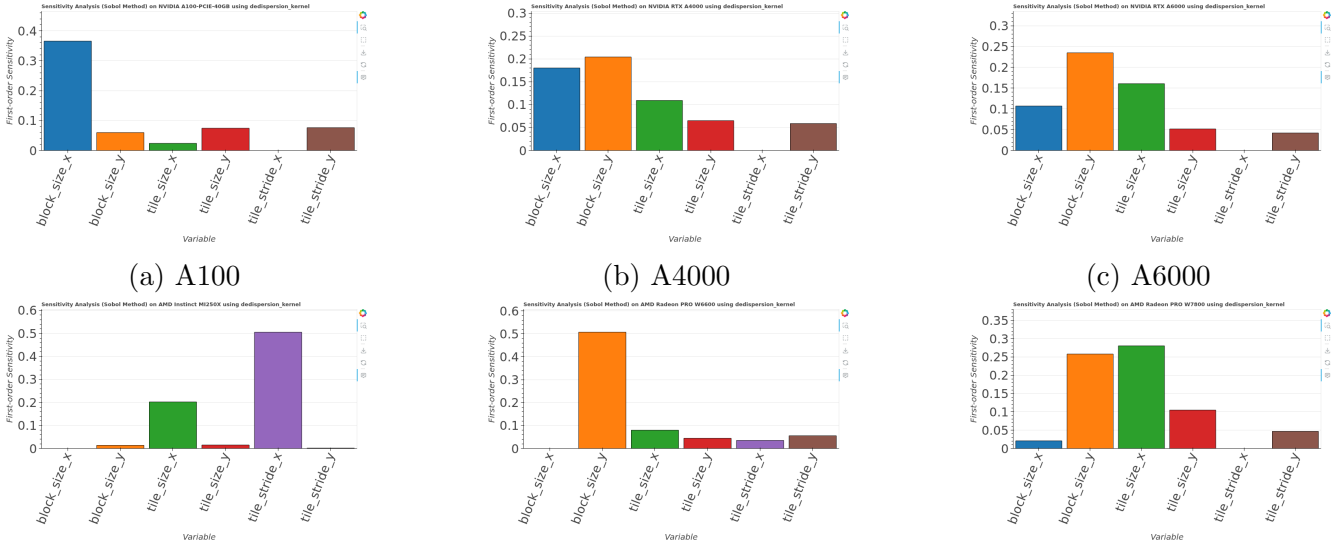
It should again be noted that Figure 8c does not have a bar for the parameter `sh_mem`.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

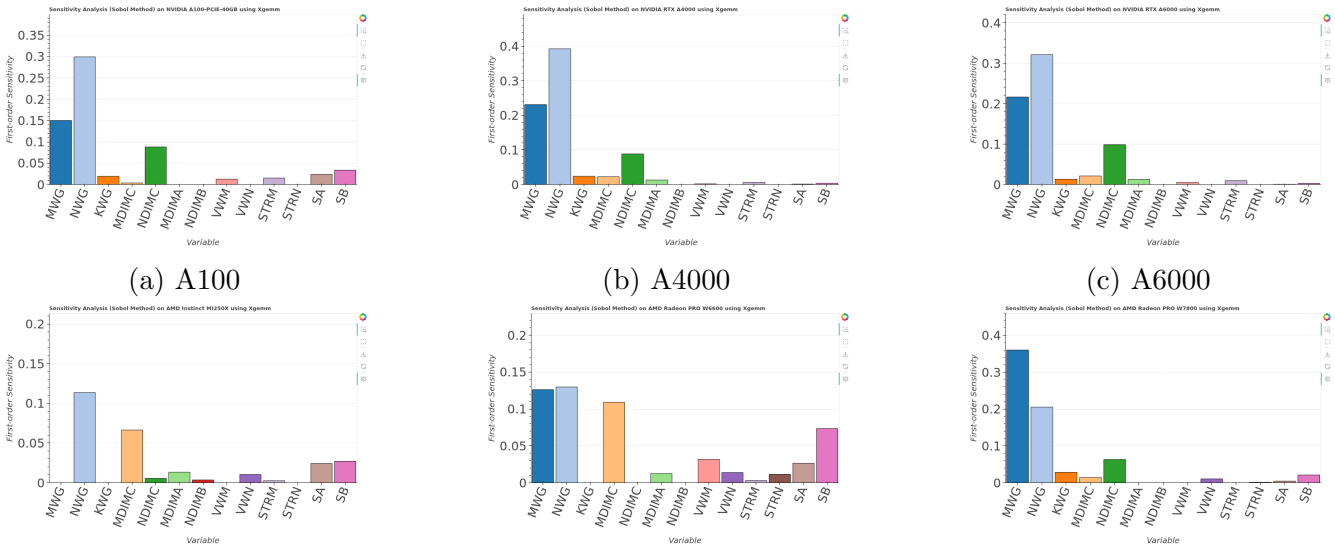
Figure 8: Sensitivity analyses on the convolution kernel.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

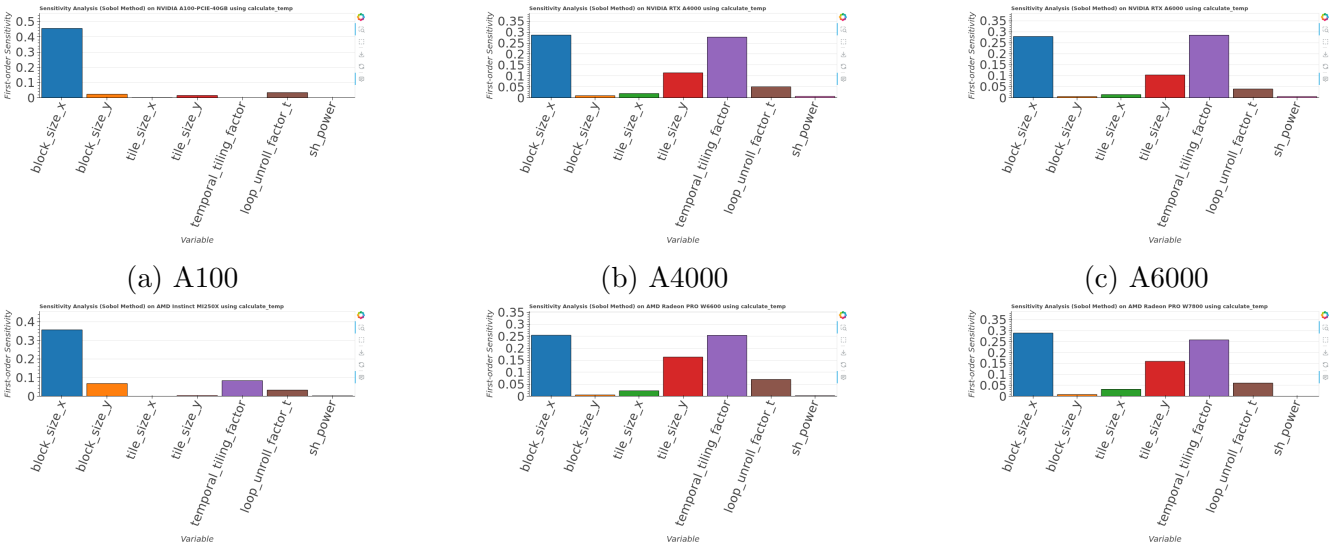
Figure 9: Sensitivity analyses on the dedispersion kernel.



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

Figure 10: Sensitivity analyses on the GEMM kernel



(a) A100 (b) A4000 (c) A6000

(d) MI250X (e) W6600 (f) W7800

Figure 11: Sensitivity analyses on the Hotspot kernel.

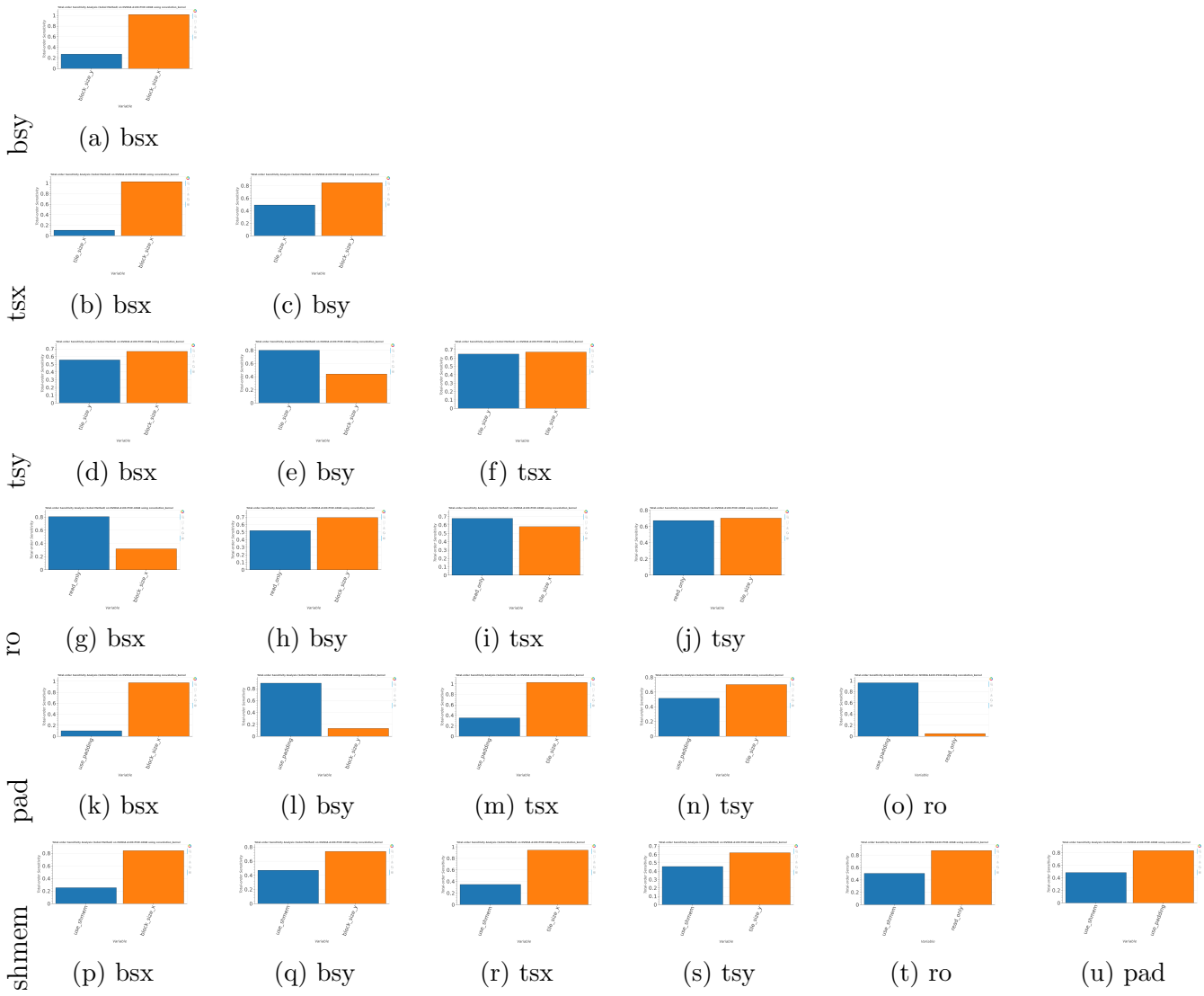
Subsets of parameters As mentioned in Section 3.2, the main highlight of sensitivity analyses is that it is possible to look at subsets of parameters. Figures 12 through 15 show total-order sensitivity analyses for pairs of parameters, with Figures 12 and 13 looking at the convolution kernel, and Figures 14 and 15 looking at the dedispersion kernel. These two kernels were chosen for this part because of their low computational complexity, which allowed the experiments to run faster, accommodating time constraints. Figures 12 and 14 look at the A100, while Figures 13 and 15 look at the A6000. These GPUs were randomly selected.

The meaning of the abbreviations in the captions of these figures can be found in Tables 1 and 2, respectively. The captions on the left indicate the blue column for each row, while the captions below each subfigure indicate the orange column. The parameters are not compared by themselves, as this always results in a sensitivity of 1.

It should again be noted that Figure 13 does not have a bar for the parameter `sh_mem`.

4.1.3 Energy cache file

In addition to the cache files between Nvidia and AMD, experiments were also performed on a cache file dedicated to measuring energy consumption, the results of which are visible in Figure 16. This "energy cache file" is only available for the GEMM kernel on the A100. This disallows for a proper comparison between GPUs or different kernels. However, since Section 4.1.1 also includes this GPU using GEMM, it is possible to compare these two cache files specifically. This cache file is also the largest that was used for this project, hosting more than 230 MB of data – a text file with 243 million characters in 239 million lines. For reference, the largest file used for Section 4.1.1 (AMD W7800, GEMM kernel) was almost 180 MB in size, but most cache files did not exceed 150 MB.



(a) bsx (b) bsx (c) bsy (d) bsx (e) bsy (f) tsx (g) bsx (h) bsy (i) tsx (j) tsy (k) bsx (l) bsy (m) tsx (n) tsy (o) ro (p) bsx (q) bsy (r) tsx (s) tsy (t) ro (u) pad

Figure 12: Sensitivity analyses, subsets on the convolution kernel, A100.

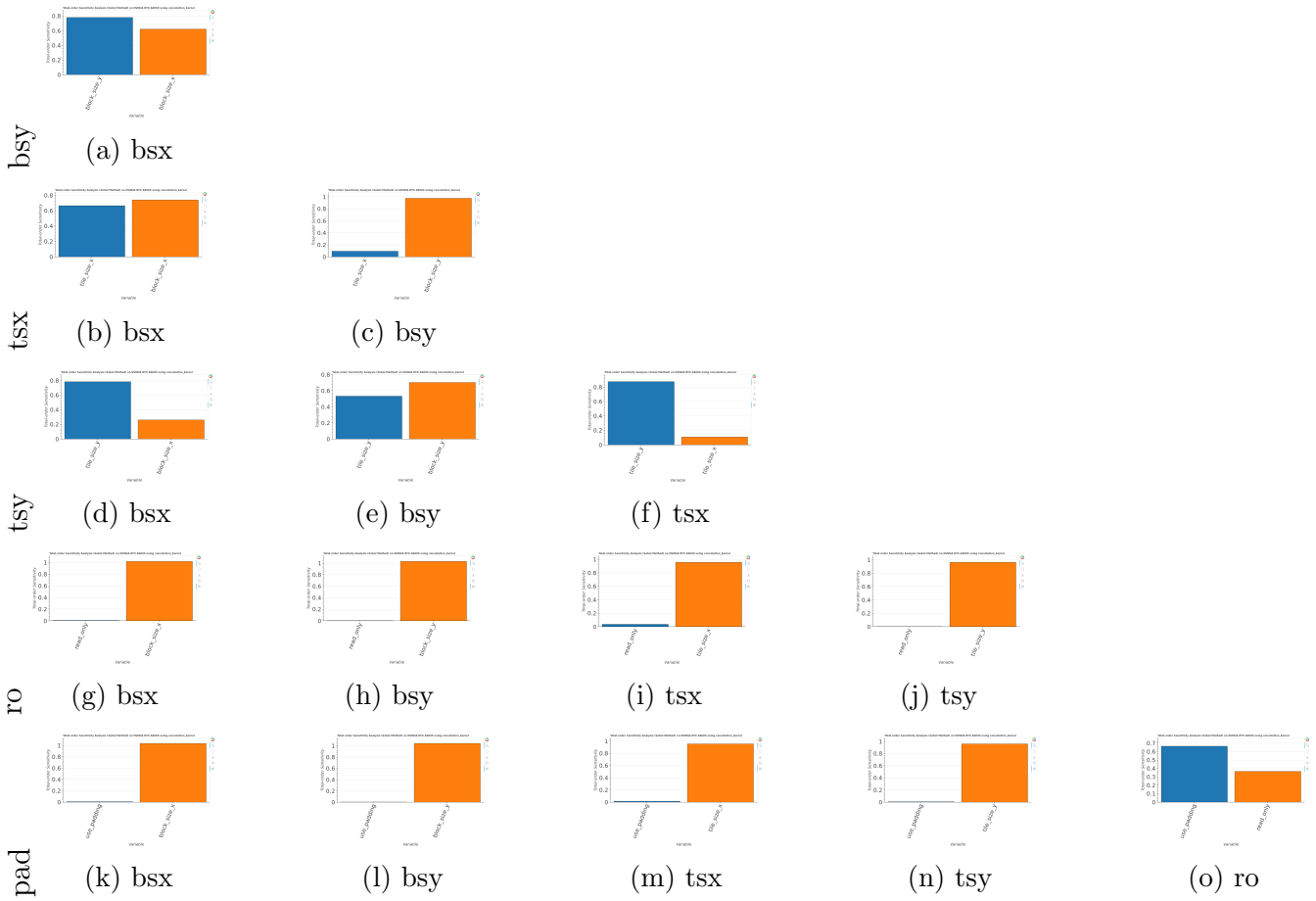
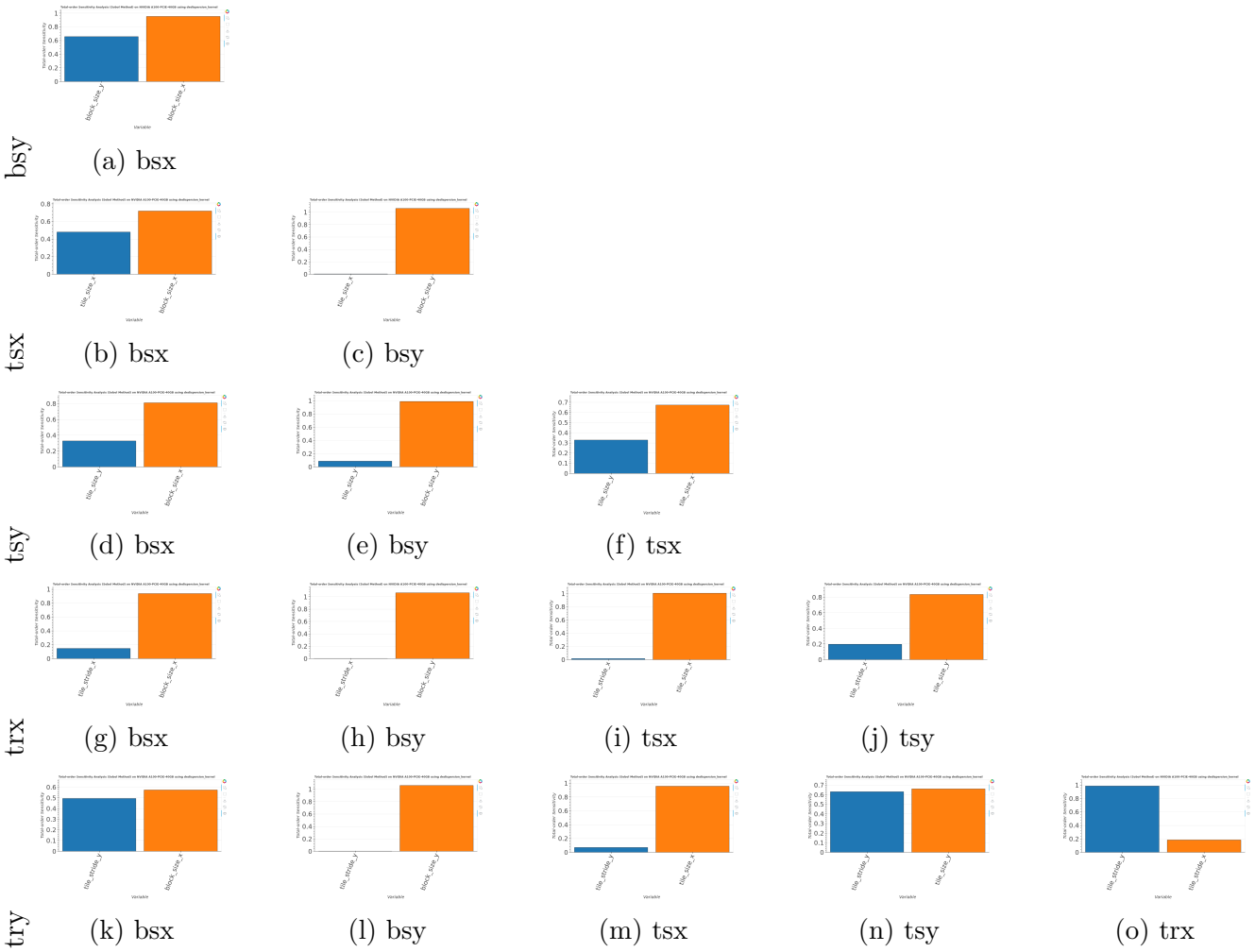
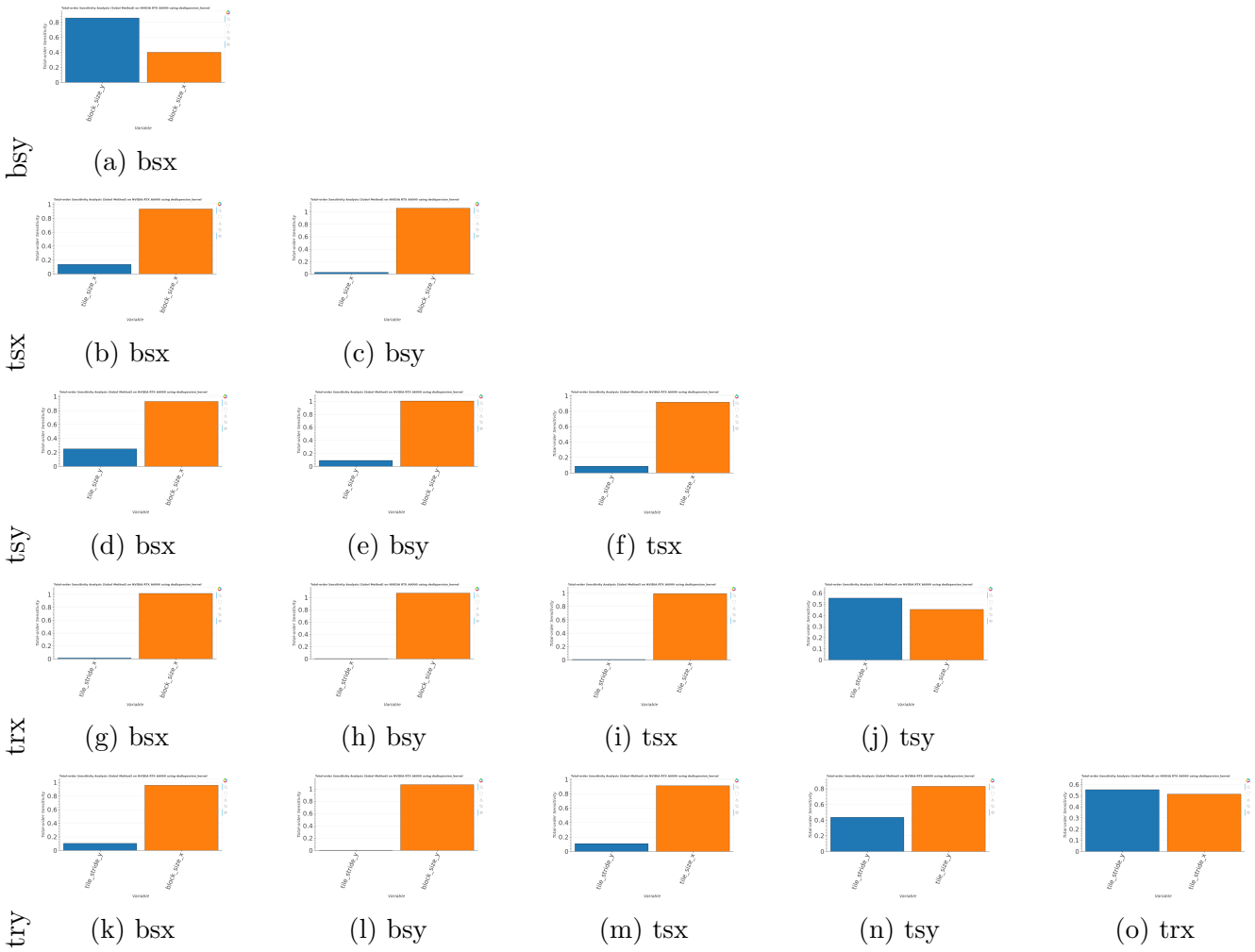


Figure 13: Sensitivity analyses, subsets on the convolution kernel, A6000.



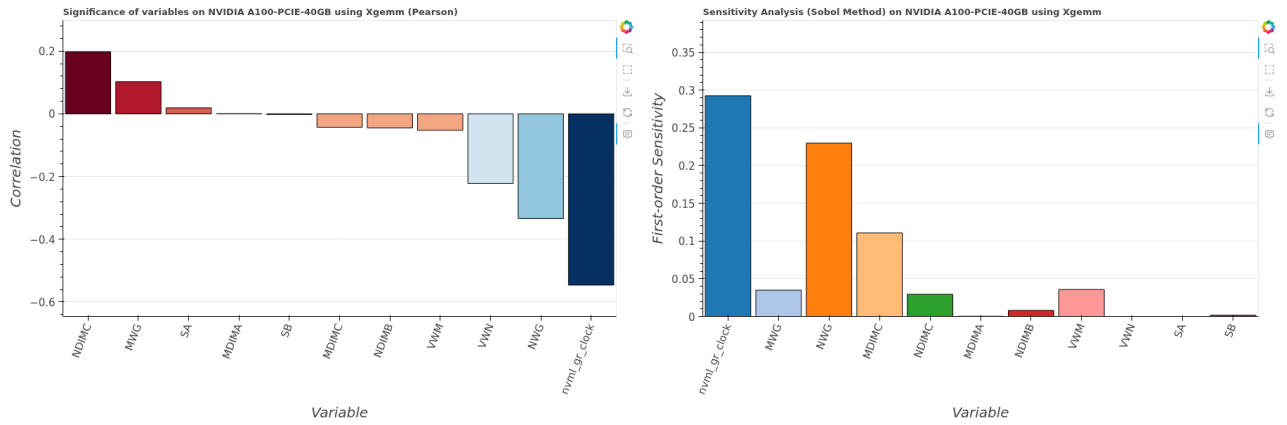
(a) bsx (b) bsx (c) bsy (d) bsx (e) bsy (f) tsx (g) bsx (h) bsy (i) tsx (j) tsy (k) bsx (l) bsy (m) tsx (n) tsy (o) trx

Figure 14: Sensitivity analyses, subsets on the dedispersion kernel, A100.



(a) bsx (b) bsx (c) bsy (d) bsx (e) bsy (f) tsx (g) bsx (h) bsy (i) tsx (j) tsy (k) bsx (l) bsy (m) tsx (n) tsy (o) trx

Figure 15: Sensitivity analyses, subsets on the dedispersion kernel, A6000.



(a) Significance analysis. (b) Sensitivity analysis.

Figure 16: Significance/sensitivity analysis on GEMM kernel, time as objective.

4.2 Dashboard improvement

The second aim of this bachelor project is to improve Kernel Tuner's dashboard. Here, it is key to find the balance between displaying useful information about tuning parameters and displaying it in a visually pleasing way. A well-crafted graph should already give the reader an idea of what it displays at first glance.

Sidebar The old and new sidebar can be seen in Figure 17. The updated sidebar includes an "objective" to calculate significance and sensitivity for and the parameters that should be included in the sensitivity analysis. At least one parameter needs to be selected for the sensitivity to be analyzed.

(a) Old sidebar.

(b) Updated sidebar.

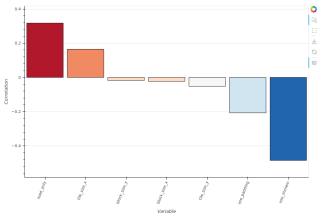
Figure 17: Sidebar difference.

Analysis plots The first versions of the plots used to visualize significance and sensitivity can be found in Figure 18, while the revised versions can be found in Figure 19. Discussion of these revisions can be found in Section 5.3.

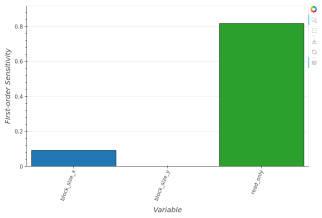
(a) Significance analysis.

(b) Sensitivity analysis.

Figure 18: First versions of analysis graphs.



(a) Significance analysis.



(b) Sensitivity analysis.

Figure 19: Revised versions of analysis graphs.

5 Discussion

This section contains discussions of the results from the previous section.

5.1 On the significance of parameters

This subsection contains a discussion of the results of the significance analyses from Section 4.1.1.

5.1.1 Convolution kernel

As visible in Figure 4, on two of the six GPUs (A100 and A4000), the parameter `read_only` appears to have a positive correlation[3] with the objective variable (time), while all other parameters are associated with shorter times. After further discussion with Professor van Werkhoven, this is a surprising result; `read_only` is a variable that is not expected to have much of an influence on the kernel's performance. As mentioned in Section 2.2, `read_only` merely enables or disables the use of a read-only cache. A possible explanation for this correlation is an architectural difference, though it is remarkable the A6000 (also using the Ampere architecture) does not show the same pattern. Another noteworthy phenomenon is that most GPUs (all but the W7800) appear to favor `use_shmem`.

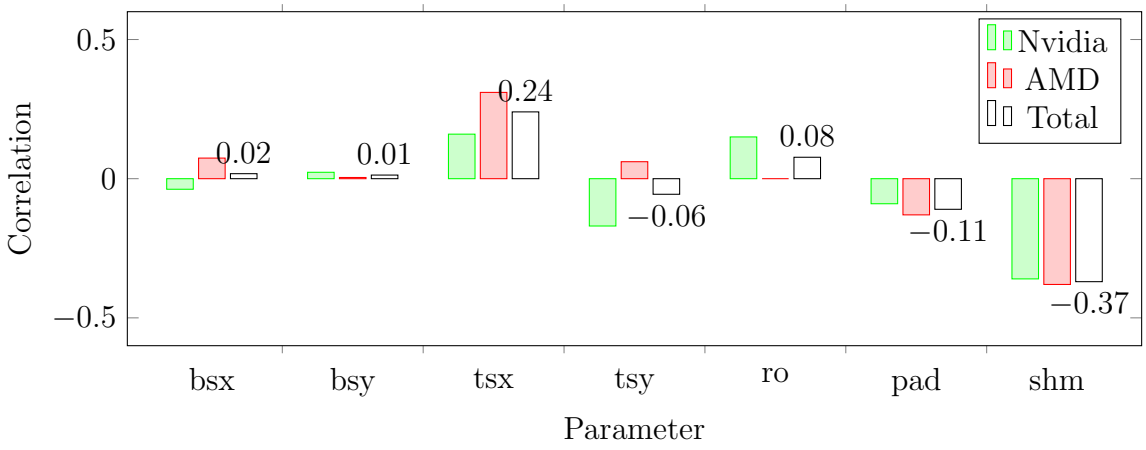


Figure 20: Average correlation on convolution kernel.

[3]To reiterate, "positive correlation" here is undesirable – the objective variable (time) should be kept as low as possible.

Figure 20 shows the average correlations of the tuning parameters of the convolution kernel; see Table 1 for the meaning of the labels. To re-iterate, negative values correspond to a decrease in time and thus a better performance. Here, it is especially noticeable how much of an impact the parameter `use_shmem` has. This is likely due to the GPU being able to communicate between threads more easily, significantly improving the performance. The highly positive correlation exhibited by `tile_size_x` may perhaps be explained by hardware misalignment. If the tiles do not fit into the GPU cache, they might hurt performance rather than boosting it.

Note that on average, the parameter `read_only` does not have a large impact on performance. This suggests that the high values found for the A100 and A4000 are outliers, and not the norm.

5.1.2 Dedispersion kernel

The significance analysis on the dedispersion kernel, visible in Figure 5, shows some extremely varying results. Not one parameter is the most significant for all GPUs, though the `block_size` parameters correlate with a lower time across most analyses; only the W6600 shows a thoroughly positive correlation to the `block_size_x` parameter. The opposite is true for the `tile_size` parameters; on all GPUs these parameters have a positive correlation to time. The `tile_stride` parameters are mostly neutral toward the objective, with the exception of `tile_stride_x` on the MI250X, where it is the most negatively correlated parameter.

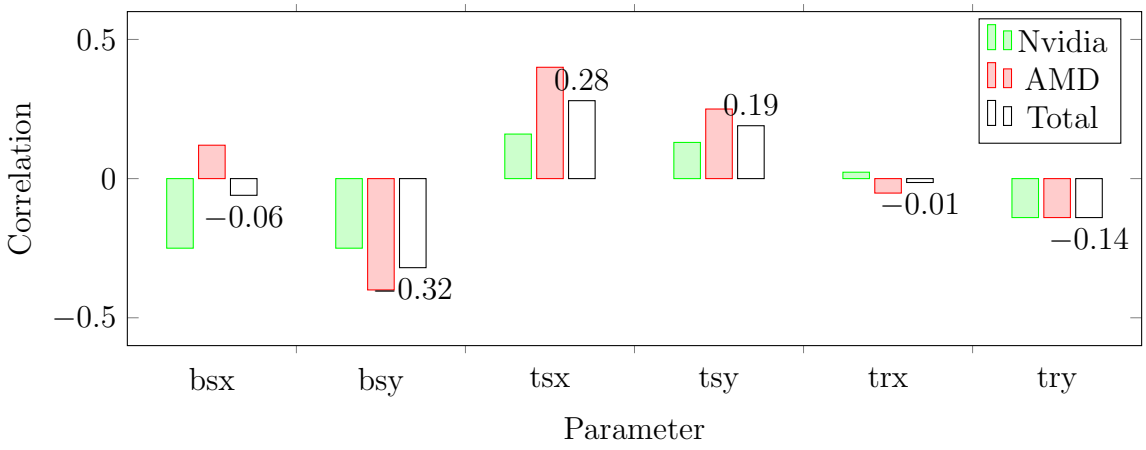


Figure 21: Average correlation on dedispersion kernel.

Figure 21 shows the average correlations of the dedispersion parameters; see Table 2 for the meaning of the labels. It is clear that, similarly to the convolution kernel, increased `tile_size` correlates to worse performance. In fact, on the dedispersion kernel, `tile_size_y` also exhibits this correlation. This may again be the result of tiles not fitting inside of the GPU cache. At the same time, increased `block_size` and/or enabling `tile_stride` correlates to better performances.

5.1.3 GEMM kernel

On the GEMM kernel, visible in Figure 6, the `NWG` parameter has the most negative correlation to time on all Nvidia GPUs and the W7800. The `MWG` takes second or third place on these same GPUs. On the other hand, the two parameters most positively correlated are `NDIMC` and `MDIMC`.

Remarkably, the A100 and W6600 exhibit almost entirely opposite behavior, with these parameters exhibiting a very positive correlation.

The significance analyses for the energy cache files, visible in Figures 16a shows mixed results. The parameters `MWG` are among the most positively correlated parameters. This aligns with the results for the A100 above. However, what is intriguing is that the parameter `NDIMC` has the most positive correlation to time, and the `NWG` has extremely negative correlation. This aligns with the results from the Nvidia GPUs. Lastly, the newly introduced `nvml_gr_clock` appears to strongly correlate to time negatively. This is entirely expected, as increasing this parameter effectively increases the GPU clock speed and thus lower the execution time.

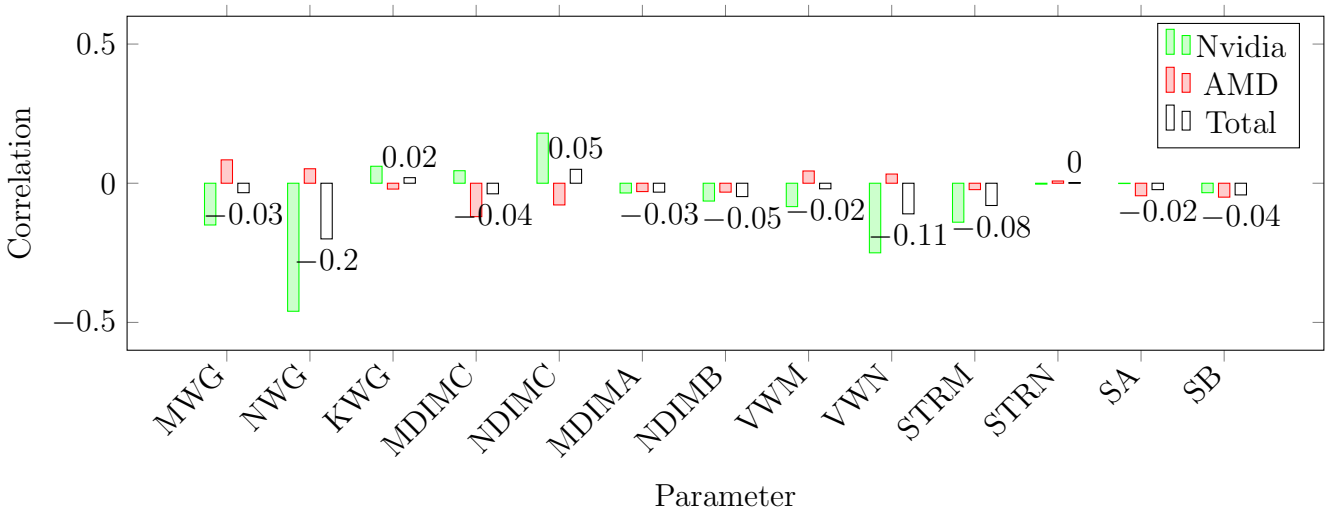


Figure 22: Average correlation on GEMM kernel.

In Figure 22, it looks as though, on average, none of the parameters reach a high significance on the AMD GPUs. This is because the W7800 has its significances flipped when compared to the MI250X and W6600, or vice versa. Since the Nvidia GPUs are largely in agreement on significance, their average values skyrocket in comparison. It is apparent that for the Nvidia GPUs, the most significant parameters here are `NWG`, `MWG` and `VWN`. Taking the AMD GPUs into consideration, `STRM` lands at third place and `MWG` is no longer as significant.

5.1.4 Hotspot kernel

The Hotspot kernel's significance analyses is visible in Figure 7. As a reminder, this analysis was done with GFLOP/s as the objective, so the expected outcome is the opposite of the previous analyses. It is immediately visible that the `temporal_tiling_factor` parameter has a negative correlation with the objective variable on most of the GPUs. The only GPU that does not show this is the A100, where this parameter shows little effect. The `block_size` and `tile_size` parameters seem to have a positive correlation with the objective, for the most part. Lastly, `loop_unroll_factor_t` shows a neutral to negative correlation.

Figure 23 (see Table 4 for the meaning of the labels) shows that the Nvidia and AMD GPUs, on average, are mostly in agreement on significance. To re-iterate, a positive correlation here indicates more floating-point operations per second. Only the `tile_size` parameters are in opposite direction

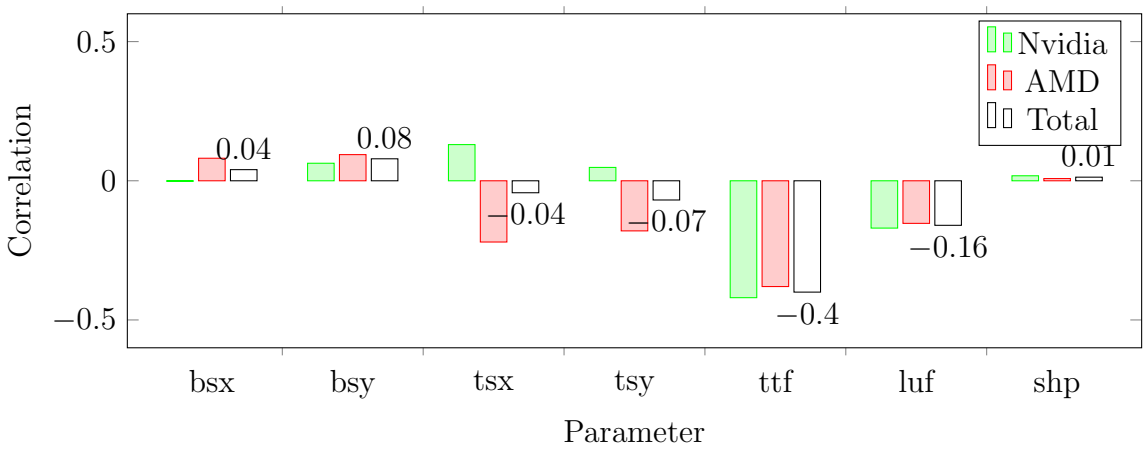


Figure 23: Average correlation on Hotspot kernel.

from one another, and even then, not far apart. It is clear that the most significant parameter here is `temporal_tiling_factor`, with `loop_unrolling_factor_t` in second place.

5.2 On the sensitivity of tuning parameters

This subsection contains a discussion of the results of the sensitivity analyses from Section 4.1.2.

5.2.1 Convolution kernel

All parameters Looking first at the convolution kernel, visible in Figure 8, it is clear that the parameter `read_only` is very sensitive on the A100 and A4000 cards. This coincides with the findings of Section 5.1. This phenomenon is especially visible on the A4000, where it is practically the only peak. Again, this behavior is not seen for the A6000, which instead exhibits a sensitive `tile_size_y`, a parameter that – for that GPU – also showed high significance in the previous section.

In stark contrast to this, the AMD cards exhibit a low sensitivity for `read_only`, and instead show relatively sensitive `tile_size_x` and `tile_size_y` parameters. Another interesting thing to note is that the W7800 in Figure 8f has a high sensitivity for the parameter `tile_size_x`, but not for `tile_size_y`.

Subsets of parameters Figure 12 provides useful insight, as the analysis for all parameters had `read_only` overshadow all other parameters. It is now visible what the relation is between those other parameters. `block_size_x` seems to be more sensitive than all parameters except `read_only`, while `use_shmem` is clearly the least sensitive parameter here. One key phenomenon to notice is that `read_only` doesn't appear that sensitive when compared to parameters other than `block_size_x`, and even displays almost no sensitivity at all when compared to `use_padding`. Also, that same `use_padding` exhibits a strong sensitivity compared to `block_size_x`. This might indicate a strong relation between the trio of parameters.

Figure 13 immediately shows that the parameters `read_only` and `use_padding` are extremely insensitive compared to all other parameters. Funnily enough, the only time `use_padding` ap-

pears sensitive is against `read_only`. What is interesting, is that none of the comparisons with `tile_size_y` indicate a high sensitivity, contrasting the observations in Figure 8c. This may arise from the difference in sensitivity index, where other parameters have their sensitivities in higher orders.

5.2.2 Dedispersion kernel

All parameters Moving on to the dedispersion kernel, visible in Figure 9, some peaks that stand out are: `block_size_x` for the A100, `tile_stride_x` for the MI250X and for `block_size_y` for the W6600. The W7800 in Figure 8f has two peaks; one in `block_size_y` and one in `tile_size_x`. The A4000 and A6000 have a more equal spread of sensitivity across parameters.

Subsets of parameters From Figure 14, it is clear that the parameters `block_size_x` and `block_size_y` enjoy a greater sensitivity than the other parameters on the A100. The former is not surprising, as this also came to light in the previous paragraph. `block_size_y` having a larger sensitivity too is a novelty, though. This may stem from a higher-order sensitivity, or perhaps because `block_size_x` is too sensitive in the general analysis. Furthermore, `tile_size_x` seems to be more sensitive than the remaining parameters, and `tile_size_y` shows more sensitivity than `tile_stride_x` but not `tile_stride_y`. At the same time, `tile_stride_y` is more sensitive when compared to `tile_stride_x`.

Looking at the A6000, `block_size_x` is more sensitive than other parameters again, but here, `block_size_y` seems to be slightly more sensitive. This behavior is replicated from the general analysis in Figure 8c. `block_size_y` is more sensitive than any other parameter and `tile_size_x` is more sensitive than most parameters, save for the `block_size` parameters.

5.2.3 GEMM kernel

The sensitivity analyses on the GEMM kernel, visible in Figure 10, show that a parameter that sticks out in almost all graphs is `NWG`. Only for the W7800 it is not the most sensitive parameter, outdone by `MWG`. The Nvidia cards follow a consistent pattern with `NWG` being the most sensitive, `MWG` being second and `NDIMC` following third.

The values for the AMD cards are all over the place. Strangely enough, the MI250X shows no sensitivity for `MWG` at all, while the W6600 shows a sensitive `MDIMC`, behaviors not found in any of the other cards, respectively.

Looking at the sensitivity analyses on the energy cache files, visible in Figure 16b, results that align with the significance analyses can be found. By far the most sensitive parameter is `nvml_gr_clock`, with `NWG` and `MDIMC` in second and third place.

5.2.4 Hotspot kernel

Lastly, the Hotspot kernel, visible in Figure 11, seems to display the most coherent pattern so far. Four out of six analyses – those for the A4000, A6000, W6600 and W7800 – show the same pattern occur: highly sensitive `block_size_x` and `temporal_tiling_factor`, with a lower sensitive `tile_size_y`. The other two GPUs show only a sensitive `block_size_x`, visible in Figures 11a and 11d. It is interesting that the GPUs between manufacturers align on this kernel. This shared feature might rise from certain similarities between the architectures of those GPUs.

The schism in the patterns between the older GPUs and the newer ones is also noteworthy. This might have something to do with the architecture changes between generations.

5.3 On the improvement of the dashboard

The old graphs, visible in Figure 18, convey the necessary information, though not very well. During the mid-terms of the bachelor project, a few points were addressed:

- For the significance analysis, it is not visible which values are desirable and which are not.
- For the sensitivity analysis, it looks as if the bars are related to each other.
- The graphs display very different information, so them having the same color scheme is confusing.

Figure 19 shows the improved versions of the graphs. These solve the problems found in their old versions. The significance analysis receives a gradient from red to blue, to indicate a spectrum of positive to negative values. The sensitivity analysis receives a wide array of colors, to distinguish between parameters. Now that the two analyses have different color schemes, it is also much easier to tell they display different information.

6 Conclusions and Further research

Section 6.1 closes with some finishing thoughts, while Section 6.2 contains suggestions for further research.

6.1 Conclusions

This thesis investigated the significance and sensitivity of tuning parameters in GPU kernel optimization. Building upon existing datasets – particularly those generated by Lurati, Schoonhoven and Willemsen – kernel tuning parameters were analyzed on how they affect performance. Additionally, the Kernel Tuner dashboard was extended by incorporating interactive visualizations that assist developers in interpreting tuning data more effectively.

From the evaluation, several key findings emerged:

- Certain parameters, such as `block_size` and `unroll_factor`, consistently demonstrated a strong influence on execution time across a range of kernels and GPU types.
- Conversely, parameters like `read_only` and `use_shmem` showed unexpected patterns of significance, occasionally appearing impactful where no strong theoretical justification existed. This may indicate underlying biases or artifacts in specific cache files or kernel implementations.
- The visualization enhancements to the Kernel Tuner dashboard have the potential to make complex sensitivity relationships more interpretable and actionable for users.

This work provides valuable empirical insight into which tuning parameters truly matter for performance, helping to narrow the search space for auto-tuning and manual optimization efforts. The improved dashboard serves as a practical tool that can reduce guesswork and guide developers toward more informed decisions when optimizing GPU code.

Several limitations must be acknowledged. Firstly, the analysis was restricted to publicly available cache files, with no new benchmarks run directly on physical hardware due to resource constraints – the GPU used on the system running the analyses did, at the time of writing, not include support for programming models supported by Kernel Tuner (CUDA, HIP, OpenCL, etc.). Secondly, the conclusions are based on a limited set of kernels and GPU models, which may not generalize to all contexts. Finally, some anomalous findings (e.g., surprising parameter impacts) suggest the need for deeper investigation into the structure and consistency of the data sources used.

In practical terms, this thesis contributes to a more efficient and informed tuning process for GPU kernels, potentially saving developers significant time and compute resources. As auto-tuning becomes more critical in high-performance computing and machine learning, tools like the enhanced Kernel Tuner dashboard will help bridge the gap between raw performance data and actionable insights.

The key takeaway from this research is that not all tuning parameters are equal, and that their importance can vary not just by kernel but by GPU architecture.

6.2 Further research

To further this work, future research could focus on:

- Conducting real-time experiments on a broader set of GPUs.
- Exploring interactions between more than two parameters but not all at the same time, as well as interactions on kernels other than those studied here.
- Study the total-order sensitivity of all parameters in the kernels that were studied here, instead of only their first-order sensitivity.

As GPU workloads become increasingly central to computing tasks – from deep learning to scientific simulations – every increment in efficiency matters. By improving our understanding of tuning parameters and enhancing the tools available to developers, this work helps pave the way toward more accessible and performant GPU programming. In this sense, the world becomes a slightly better place: where smarter tuning leads to faster computation, reduced energy consumption, and ultimately, more responsible and effective use of our computational resources.

References

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 303–316, New York, NY, USA, 2014. Association for Computing Machinery.

[AMDa] Amd instinct mi250x. https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html.

[AMDb] Amd radeon pro w6600. https://www.amd.com/en/products/graphics/workstations/radeon-pro/w6600.html.

[AMDc] Amd radeon pro w7800. https://www.amd.com/en/products/graphics/workstations/radeon-pro/w7800.html.

[BDP+10] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 105–114, New York, NY, USA, 2010. Association for Computing Machinery.

[Bok18] Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2018.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[HHS+23] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. Optimization techniques for gpu programming. *ACM Comput. Surv.*, 55(11), March 2023.

[Hol] Holoviz. Panel: The powerful data exploration & web app framework for python. https://github.com/holoviz/panel.

[HU17] Jon Herman and Will Usher. SALib: An open-source python library for sensitivity analysis. *The Journal of Open Source Software*, 2(9), jan 2017.

[IUH22] Takuya Iwanaga, William Usher, and Jonathan Herman. Toward SALib 2.0: Advancing the accessibility and interpretability of global sensitivity analyses. *Socio-Environmental Systems Modelling*, 4:18155, May 2022.

[Jai24] Abhishek Jain. All about convolutions, kernels, features in cnn. https://medium.com/@abhishekjainindore24/all-about-convolutions-kernels-features-in-cnn-c656616390a1, Feb 2024.

[LA21] Xiang Li and Gagan Agrawal. Shrinking sample search algorithm for automatic tuning of gpu kernels. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 262–271, 2021.

[LHSvW24] Milo Lurati, Stijn Heldens, Alessio Sclocco, and Ben van Werkhoven. Bringing auto-tuning to hip: Analysis of tuning impact and difficulty on amd and nvidia gpus. In *European Conference on Parallel Processing*, pages 91–106. Springer, 2024.

[LHvW] Milo Lurati, Stijn Heldens, and Ben van Werkhoven. Autotuning amd vs nvidia gpus. https://github.com/MiloLurati/AutoTuning_AMD_vs_Nvidia_GPUs/.

[Nug18] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18. ACM, May 2018.

[Nvia] Nvidia a100. https://www.nvidia.com/en-us/data-center/a100/.

[Nvib] Nvidia rtx a4000. https://www.nvidia.com/en-us/design-visualization/rtx-a4000/.

[Nvic] Nvidia rtx a6000. https://www.nvidia.com/en-us/design-visualization/rtx-a6000/.

[Pea95] Karl Pearson. Notes on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London.*, 58:240–242, 1895.

[Sob01] I.M Sobol'. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and Computers in Simulation*, 55(1):271–280, 2001. The Second IMACS Seminar on Monte Carlo Methods.

[Spe04] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15 (1):72–101, 1904.

[Spe38] C. Spearman. A new measure of rank correlation. *Biometrika*, 30 (1-2):81–91, 1938.

[SVVWB22] Richard Schoonhoven, Bram Veenboer, Ben Van Werkhoven, and K. Joost Batenburg. Going green: optimizing gpus for energy efficiency through model-steered auto-tuning. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 48–59, 2022.

[TvWP+23] Jacob Tørring, Ben van Werkhoven, Filip Petrovič, Floris-Jan Willemsen, Jiri Filipovic, and Anne Elster. Towards a benchmarking suite for kernel tuners, 03 2023.

[VD08] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.

[vMBS14] Ben van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. Optimizing convolution operations on gpus using adaptive tiling. *Future Generation Computer Systems*, 30:14–26, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[vW] Ben van Werkhoven. Ktdashboard github repo. https://github.com/KernelTuner/dashboard.

[vW19] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[War15] Pete Warden. Why gemm is at the heart of deep learning, Apr 2015.

[WD98] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, 1998.

[Wil] Floris-Jan Willemsen. Fair benchmark hub for auto-tuning. https://github.com/AutoTuningAssociation/benchmark_hub/tree/main.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.