Towards a Benchmarking Suite for Kernel Tuners

Jacob O. Tørring[*], Ben van Werkhoven[†], Filip Petrovič[‡], Floris-Jan Willemsen[†], Jiří Filipovič[‡] and Anne C. Elster[*]

[*] *Norwegian University of Science and Technology (NTNU), Trondheim, Norway*
[†] *Netherlands eScience Center, Amsterdam, Netherlands*
[‡] *Masaryk University, Brno, Czech Republic*
{jacob.torring, elster}@ntnu.no, {b.vanwerkhoven, f.j.willemsen}@esciencecenter.nl, {fillo, fila}@mail.muni.cz

Abstract—As computing system become more complex combining CPUs and GPUs, it is becoming harder and harder for programmers to keep their codes optimized as the hardware gets updated. Autotuners try to alleviate this by hiding as many architecture-based optimization details as possible from the end-user, so that the code can be used efficiently across different generations of systems. Several autotuning frameworks have emerged, but a comparative analysis between these related works is scarce, owing to the significant manual effort required to port a tunable kernel from one tuner another.

In this article we introduce a new benchmark suite for evaluating the performance of optimization algorithms used by modern autotuners targeting GPUs. The suite contains tunable GPU kernels that are representative of real-world applications, allowing for comparisons between optimization algorithms and the examination of code optimization, search space difficulty, and performance portability. Our framework facilitates easy integration of new autotuners and benchmarks by defining a shared problem interface.

Our benchmark suite is evaluated based on five characteristics: convergence rate, local minima centrality, optimal speedup, Permutation Feature Importance (PFI), and performance portability. The results show that optimization parameters greatly impact performance and the need for global optimization. The importance of each parameter is consistent across GPU architectures, however, the specific values need to be optimized for each architecture.

Our portability study highlights the crucial importance of autotuning each application for a specific target architecture. The results reveal that simply transferring the optimal configuration from one architecture to another can result in a performance ranging from 58.5% to 99.9% of the optimal performance, depending on the GPU architecture. This highlights the importance of autotuning in modern computing systems and the value of our benchmark suite in facilitating the study of optimization algorithms and their effectiveness in achieving optimal performance for specific target architectures.

Keywords-autotuning, benchmarking

I. INTRODUCTION

As computers have become more advanced in recent decades, there has been a significant increase in their complexity. Central Processing Units (CPUs) still form the core of modern computers, but we have seen a growing use of accelerators like Graphics Processing Units (GPUs) and co-processors to improve efficiency and performance. These accelerators can be highly effective, but they can also make the task of optimizing programs for performance increasingly difficult.

New data centers and many of the world's top supercomputers, such as the Top500 systems, have increasingly relied on High-Performance Computing (HPC) systems that use a combination of CPUs and GPUs [1]. These heterogeneous systems can make code optimization a challenging task, as the architectures of the different components change rapidly.

To make programs run efficiently on these systems, programmers in computer science and related fields spend considerable efforts in optimization [2]. However, as the complexity of these systems grows, the task of understanding how all the system components interact becomes increasingly complex. To address this, analytical methods such as compiler optimization are used to understand the systems and turn that knowledge into rules for optimizing programs. These rules can modify large chunks of code without changing the semantics of the program, but they can also contain heuristics that may not always lead to the optimal solution.

As architectures are constantly evolving and different, a heuristic-based rule cannot always generalize well to the entire system. In these cases, empirical methods which search for the optimal solution through trial and error can be used, this is called *autotuning*.

One of the main consideration when developing such solutions is to hide as many architecture-based optimization details as possible from the end-user, so that the code can be used efficiently across different generations of systems. The goal is to provide easy-to-use libraries and APIs that enable developers to write code that runs well on different systems without having to understand the intricacies of each system's architecture.

In recent years, several studies have been conducted that present advancements in optimization algorithms for autotuning [3]. Despite this, comparative analysis between related works is scarce, owing to the significant manual effort required to porting a tunable kernel from one tuner another. In order to effectively study the performance of optimization algorithms for autotuning, as well as to facilitate comparisons between optimization algorithms implemented in different tuners, it is necessary to have a benchmark suite that is compatible with all of these tuners.

Existing benchmark suites, such as Rodinia [4], SHOC [5], PolyBenchGPU [6], are not suitable for this

purpose as they are not tunable. They typically have hard-coded thread block dimensions, but even worse, the code is often written with assumptions that the block dimensions, parallelizations, and amount of work per thread will never change. Furthermore, in many cases, the hardcoded numbers are directly dependent on the input problems, which are often unrealistically small for modern GPUs. As a result, tuning these codes requires extensive modifications.

Earlier attempts at creating tunable benchmark suites [7], [8] are limited regarding studying the effectiveness of optimization algorithms. The benchmark suite from Petrovič et al. [7] only supports one tuning framework, while the optimization parameters in Sund et al. [8] have limited performance impact.

The "BAT 2.0" benchmark suite aims to stimulate autotuning research by offering a set of tunable kernels that are representative of those used in various real-world applications, such as machine learning, image processing, astrophysics, thermal modeling, microscopy, geospatial information systems, and radio astronomy.

This benchmark suite facilitates for comparisons between optimization algorithms from different autotuners by providing a standardized problem interface for both the autotuners and benchmarks. The benchmark suite provides general configuration space and kernel handler classes providing for easy integration towards Optuna [9], SMAC3 [10], Kernel-Tuner [11], KTT[7] as well as our own basic reference tuner. This enables the study of code optimization effectiveness, search space difficulty, performance portability, and more. With the creation of this benchmark suite, researchers can now investigate key questions about code optimization, search space, and performance portability.

II. BACKGROUND

GPUs were originally designed for fast processing of graphics, but they have also been found to be effective accelerators for parallelizable AI and computational science workloads. After their programming environments facilitated using them as General Purpose GPUs (GPGPUs) to speed up a variety of scientific computations, they are now a key component in many of the world's largest computing clusters. The optimization workload running on GPU architectures is thus important for a range of fields beyond video game graphics.

A. Performance Portability

Relative performance* can vary greatly across different hardware platforms, even if a program functionally behaves the same on all of them. A problem configuration that runs well on one GPU may perform poorly on another GPU. When writing high-performance software, it is essential to utilize the hardware as efficiently as possible. One solution to this is to use libraries like ATLAS [12], which

*performance of a code compared to peak performance of hardware

automatically tunes its configuration based on the executing hardware. This can avoid the manual effort of finding the optimal configuration for every hardware platform. FFTW [13] also used a similar self-tuning approach to optimize its fast Fourier transforms.

We can measure how portable these configurations are by finding the optimal configurations for our target platforms and then examine the relative performance of these optimal configurations on other systems. We can thus analyze how sensitive the configurations are to platform changes and how large the relative performance differences are between different architectures.

B. Analysing characteristics of benchmarks

1) Feature importance: Feature importance in ML is a technique to identify influential features of a dataset on a model's outcome [14]. Different methods compute feature importance, e.g. permutation importance, feature importances from tree-based models, LASSO, etc. These methods quantify feature importance to understand a dataset's characteristics, identify redundant/irrelevant features, and guide feature selection. The feature importance scores express the dataset's characteristics and the contribution of each feature to the model's performance.

In this study, we use Permutation Feature Importance (PFI) to evaluate feature importance. PFI measures a model's performance decrease when a feature's values are shuffled to understand the feature's importance. We calculate PFI by training a Catboost [15] Regression model on the original dataset, shuffling each feature's values, retraining the model, and comparing the original and shuffled dataset's performance metric. The PFI score for each feature is the difference between the two. PFI helps identify important features for a model's performance and can detect multicollinearity among features to prevent overfitting and unstable models.

2) Proportion of Centrality metric: The proportion of centrality metric introduced by Schoonhoven et al. [3] is a way to quantify the difficulty of GPU tuning. It is based on the concept of the fitness flow graph (FFG), which contains all points in the search space and creates a directed edge to a neighboring point if the neighbor has lower fitness. This means that a random walk across the FFG mimics the behavior of a randomized first-improvement local search algorithm. The expected proportion of arrivals of each minimum then gives a metric for weighting reachability of each minimum. The likelihood of arrival per local minima is computed using the PageRank node centrality, which was originally used to determine the relevance of a webpage. The PageRank values are the values of the dominant right eigenvector of the adjacency matrix of a directed graph G, rescaled such that each column adds up to 1. The metric is a measure of difficulty, it considers how likely a certain subset of "suitably good" local minima are to be visited by a local search algorithm relative to the rest. This subset is

defined by the optimal fitness and the proportion p, taking the set of nodes consisting of local minima with fitness less than $(1 + p)f_{opt}$ for minimization problems, otherwise $(1 - p)f_{opt}$.

III. RELATED WORK

In the field of autotuning, the most relevant prior work includes the benchmark suite developed by Petrovič et al. [7], Polybench-GPU [6], and Sund et al. [8]. However, these benchmark suites have issues that limit their usefulness. The benchmark in PolyBench-GPU has small search spaces, ranging from 116 to 725 different possible configurations, which would fall below our threshold for an interesting autotuning study as many real-world applications have much larger search spaces. The benchmark suite from Petrovič et al. only supports a single autotuning framework, while the optimization parameters in the benchmark suite by Sund et al. have limited performance impact. To address this issue, we have selected benchmarks with larger and more interesting search spaces for this new version of BAT, and also added several new benchmarks that meet our criteria. These benchmarks all give significant speedups, with the performance of the optimal configurations significantly varying between different systems.

For other related works, the Collective Knowledge framework (CK) developed by Fursin et al. [16] offers a more generalized approach to benchmarking and reproducibility. In the area of Hyperparameter Optimization (HPO) several benchmark suites have been developed, such as the high-dimensional HPO benchmark suite [17], LassoBench [18], and HPOBench [19].

IV. BENCHMARKS

A. GEMM

Generalized dense matrix-matrix multiplication (GEMM) is part of the BLAS linear algebra specification, and is one of the most widely-used GPU kernels. The GEMM kernel included in BAT is from CLBlast [20], a tunable OpenCL BLAS library. GEMM implements the multiplication of two matrices, A and B:

$$C = \alpha A \cdot B + \beta C$$

where α and β are scalars and C is the output matrix. The CLBlast GEMM kernel is tunable with the parameters shown in Table I. MWG and NWG control the amount of work assigned to each thread block. MDIMC and NDIMC describe the size of thread block, while MDIMA and MDIMB control shared memory usage, VWM and VWN are the vector widths used for loading from and storing to global memory, and SA and SB enables or disables the use of shared memory for elements in A and B.

Table I: Tunable parameters – GEMM kernel in BAT.

Parameter	Values	#
MWG	{16, 32, 64, 128}	4
NWG	{16, 32, 64, 128}	4
MDIMC	{8, 16, 32}	3
NDIMC	{8, 16, 32}	3
MDIMA	{8, 16, 32}	3
NDIMB	{8, 16, 32}	3
VWM	{1, 2, 4, 8}	4
VWN	{1, 2, 4, 8}	4
SA	{0, 1}	2
SB	{0, 1}	2

B. N-body

The N-body kernel computes gravitational forces between N bodies, typically applied in astrophysical simulations. The N-body kernel in BAT was created by Petrovič et al. for use in KTT [7], as a tunable implementation of the code sample from the CUDA SDK. The N-body kernel follows a simple quadratic scheme where the forces between all pairs of bodies are computed every iteration. As such, the kernel is very compute intensive.

The tunable parameters for the N-body kernel in BAT are shown in Table II. The inner loop unroll factor parameters determine the degree to which partial loop unrolling is applied for various loops in the kernel. The outer_unroll_factor controls the amount of work allocated to each thread. The use_soa parameter specifies whether the input bodies are stored in an array of structures or a structure of arrays. local_mem enables or disables the use of shared memory as a software managed cache. vector_type is to control the number of elements loaded from memory in one instruction.

Table II: Tunable parameters – Nbody kernel in BAT.

Parameter	Values	#
block_size	{64, 128, 256, 512}	4
outer_unroll_factor	{1, 2, 4, 8}	4
inner_unroll_factor1	{0, 1, 2, 4, 8, 16, 32}	7
inner_unroll_factor2	{0, 1, 2, 4, 8, 16, 32}	7
use_soa	{0, 1}	2
local_mem	{0, 1}	2
vector_type	{1, 2, 4}	3

C. Hotspot

The Hotspot kernel included in BAT is based on the Hotspot kernel in the Rodinia Benchmark suite [4]. The kernel is part of a thermal simulation application used to estimate processor temperature based on processor architecture and simulated power currents. The kernel iteratively solves a series of differential equations. The kernel inputs are the power and initial temperatures, the output is a grid of average temperature values spanning the chip.

To simplify the indexing scheme and increase the tunability of the kernel, we have re-implemented the Hotspot kernel in Rodinia from scratch. The main difference of our implementation with that of Rodinia is that our kernel can be used with any thread block dimension, can arbitrarily vary the amount of work per thread, and vary the extent to which temporal tiling is applied.

The tunable parameters for the Hotspot kernel in BAT are shown in Table III. `block_size_x` and `block_size_y` describe the thread block dimensions in x and y, the kernel uses at least 32 and at most 1024 threads. `tile_size_x` and `tile_size_y` control the number of output elements computed by each thread in the x and y dimensions. `temporal_tiling_factor` is the number of iterations of the stencil operation performed by a single kernel launch, for more details on the temporal tiling optimization see Hijma et al. [2]. `sh_power` enables or disables the use of shared memory as a cache for storing the input power currents. `blocks_per_sm` is used in the `__launch__bounds()` directive in CUDA to hint the compiler to aim for a certain occupancy when running the kernel, effectively this optimization encourages the compiler to decrease register usage in the kernel.

Table III: Tunable parameters – Hotspot kernel in BAT.

Parameter	Values	#
`block_size_x`	$\{1,2,4,8,32n \mid 32n \in [32,1024]\}$	37
`block_size_y`	$\{1, 2, 4, 8, 16, 32\}$	6
`tile_size_x`	$\{n \mid n \in [1, 10]\}$	10
`tile_size_y`	$\{n \mid n \in [1, 10]\}$	10
`temporal_tiling_factor`	$\{n \mid n \in [1, 10]\}$	10
`loop_unroll_factor_t`	$\{n \mid n \in [1, 10]\}$	10
`sh_power`	$\{0, 1\}$	2
`blocks_per_sm`	$\{0, 1, 2, 3, 4\}$	5

D. Pnpoly

Pnpoly (Point-in-polygon) kernel is used by Goncalves et al. [21] as part of a geospatial database system for massive point clouds obtained through airborne LiDAR. The kernel is used to query all points within a certain outline, for example points on highways or all points within a city. Pnpoly has been used as a benchmark kernel for autotuning in several studies [3], [22]. However, the Pnpoly kernel in BAT includes only the GPU kernel of the full GPU-enabled database operator.

The tunable parameters of the Pnpoly kernel in BAT are listed in Table IV. `block_size_x` is simply the number of threads per block. `tile_size` the amount of points processed by each thread. `between_method` selects the algorithm to use to see if a point lies between two other points. Similarly, `use_method` selects the algorithm that is used to keep track of whether the evaluated point is inside or outside of the polygon.

Table IV: Tunable parameters – Pnpoly kernel in BAT.

Parameter	Values	#
`block_size_x`	$\{32n \mid 32n \in [32, 1024]\}$	31
`tile_size`	$\{1, 2n \mid 2n \in [2, 20]\}$	11
`between_method`	$\{0, 1, 2, 3\}$	4
`use_method`	$\{0, 1, 2\}$	3

E. Convolution

Van Werkhoven et al. [23] have implemented an optimized and highly-tunable GPU-accelerated library for 2D Convolution operations, which has become a commonly used benchmark in autotuning [3], [7], [24], [25].

A convolution operation computes a linear combination of weights and a range of the input image for each output pixel. A 2D convolution of an input image I of size $(w \times h)$ and a convolution filter F of size $(F_w \times F_h)$ computes an output image O of size $((w - F_w) \times (h - F_h))$:

$$O(x,y) = \sum_{j=0}^{F_h} \sum_{i=0}^{F_w} I(x+i, y+j) \times F(i,j)$$

The tunable parameters of the Convolution kernel in BAT are listed in Table V. `block_size_x` and `block_size_y` describe the thread block dimensions, `tile_size_x` and `tile_size_y` the number of output pixels processed by each thread in the x and y dimensions. `use_padding` controls whether or not to use the padding scheme in shared memory that is used to avoid shared memory bank conflicts as described in Van Werkhoven et al. [23]. Padding is only significant when the `block_size_x` is not a multiple of number of memory banks in shared memory. Finally, `read_only` controls whether or not to load input elements from global memory through read-only cache.

Table V: Tunable parameters – Convolution kernel in BAT.

Parameter	Values	#
`block_size_x`	$\{1, 2, 4, 8, 16, 32, 48, 64,$ $80, 96, 112, 128\}$	12
`block_size_y`	$\{1, 2, 4, 8, 16, 32\}$	6
`tile_size_x`	$\{1, 2, 3, 4, 5, 6, 7, 8\}$	8
`tile_size_y`	$\{1, 2, 3, 4, 5, 6, 7, 8\}$	8
`use_padding`	$\{0, 1\}$	2
`read_only`	$\{0, 1\}$	2

F. Expdist

The Expdist kernel is part of a localization microscopy applications that implements a template-free particle fusion algorithm by combining many different observations into a single super-resolution reconstruction [26]. The expdist kernel is used as part of the registration process where the kernel is called repeatedly to quantify the registration of two particles. The distance between two particles t and m, given

registration M, is computed as follows:

$$D = \sum_{i=1}^{K_t} \sum_{j=1}^{K_m} \exp\left(-\frac{\|\vec{x}_{t,i} - M(\vec{x}_{m,j})\|^2}{2\sigma^2}\right)$$

The kernel operates directly on the individual localizations ($\vec{x}_t$ and $\vec{x}_m$) in each particle rather than pixelated images and takes the uncertainties in the localizations (σ) into account. The algorithm is quadratic in the number of localizations per particle and is as such very compute intensive.

The tunable parameters used in the ExpDist kernel are shown in Table VI. The kernel supports two main implementations that are controlled by the `use_column` parameter. When `use_column` is set to 1, the kernel reduces the number of thread blocks used to perform the computation by using a fixed number of thread blocks in the y dimension, set by `n_y_blocks`. `use_shared_mem` use shared memory selects the way in which shared memory is used.

Table VI: Tunable parameters – ExpDist kernel in BAT.

Parameter	Values	#
`block_size_x`	{32, 64, 128, 256, 512, 1024}	6
`block_size_y`	{1, 2, 4, 8, 16, 32}	6
`tile_size_x`	{1, 2, 3, 4, 5, 6, 7, 8}	8
`tile_size_y`	{1, 2, 3, 4, 5, 6, 7, 8}	8
`use_shared_mem`	{0, 1, 2}	3
`loop_unroll_factor_x`	{1, 2, 3, 4, 5, 6, 7, 8}	8
`loop_unroll_factor_y`	{1, 2, 3, 4, 5, 6, 7, 8}	8
`use_column`	{0, 1}	2
`n_y_blocks`	{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}	11

G. Dedispersion

The Dedispersion kernel in BAT originates from the AMBER pipeline for the detection of single pulse astronomical transients [27]. Dedispersion is the process of reverting the dispersion of a radio signal transmitted over many frequencies through space. The signal component with the highest frequency f_h is received at time t_x, while simultaneously emitted components with lower frequency arrive at $t_x + k$, where k is the delay in seconds as by the dispersion equation:

$$k \approx 4150 \times DM \times \left(\frac{1}{f_i^2} \times \frac{1}{f_h^2}\right)$$

The kernel takes samples in time across many frequency bands (channels) as input and outputs the dedispersed samples for many different dispersion measure DM values. The kernel is parallelized such that each thread can work on multiple samples and dispersion measures, while iterating over the frequency bands. As input for the BAT Dedispersion kernel, we are using the parameters from the ARTS survey on the Apertif telescope [28], which uses a sampling rate of 24.4 KHz, 2048 DMs, and 1536 channels.

The tunable parameters of the dedispersion kernel are shown in Table VII. The `loop_unroll_factor_channel` parameter depends on the input, as any divisor of the number of channels can be used as a partial loop unrolling factor for the inner loop in the kernel. When the loop unroll factor is 0, it is left to the CUDA compiler to decide whether or not to apply loop unrolling. `tile_stride_x` controls the stride used to vary the amount of work per threads. When `tile_stride_x` is 0 and `tile_size_x` is larger than 1, threads will process `tile_size_x` consecutive samples, when `tile_stride_x` is 1 threads will process `tile_size_x` samples that are each `block_size_x` apart in the input. `tile_stride_y` works similarly but for dispersion measures in the y-dimension.

Table VII: Tunable parameters – Dedispersion kernel in BAT.

Parameter	Values	#
`block_size_y`	$\{1, 2, 4, 8, 16n \mid 16n \in [16, 512]\}$	36
`block_size_y`	$\{4n \mid 4n \in [4, 128]\}$	32
`tile_size_x`	$\{n \mid n \in [1, 16]\}$	16
`tile_size_y`	$\{n \mid n \in [1, 16]\}$	16
`tile_stride_x`	{0, 1}	2
`tile_stride_y`	{0, 1}	2
`loop_unroll_factor_channel`	{0, 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1536}	21
`blocks_per_sm`	{0, 1, 2, 3, 4}	5

V. EXPERIMENTAL DESIGN

A. Benchmarks, Hardware and Runtime Environment

We ran our benchmarks on four different systems with four different Nvidia GPUs. These GPUs include the RTX 2080Ti, RTX 3060, RTX 3090 and RTX Titan. For the Pnpoly, Nbody, GEMM and Convolution benchmarks we performed an exhaustive search of the entire search space. For the Hotspot, Dedisp and Expdist benchmarks our results are based on 10 000 random configurations from the search space for each architecture.

VI. RESULTS AND DISCUSSION

A. Distribution of configurations

Fig 1 shows the distribution of configurations centered around the median performing configuration. The plot extends from the worst to the best configuration. The first thing to observe is that the distribution shapes are significantly different between the different benchmarks, but similar in shape across GPUs (this agrees with results observed in other benchmark sets [29]). Most of the benchmarks have a high density of configurations around the median and then exponential decay toward the best-performing configurations. The Hotspot benchmark in Fig. 1b is an outlier among the benchmarks, with a high density around the median
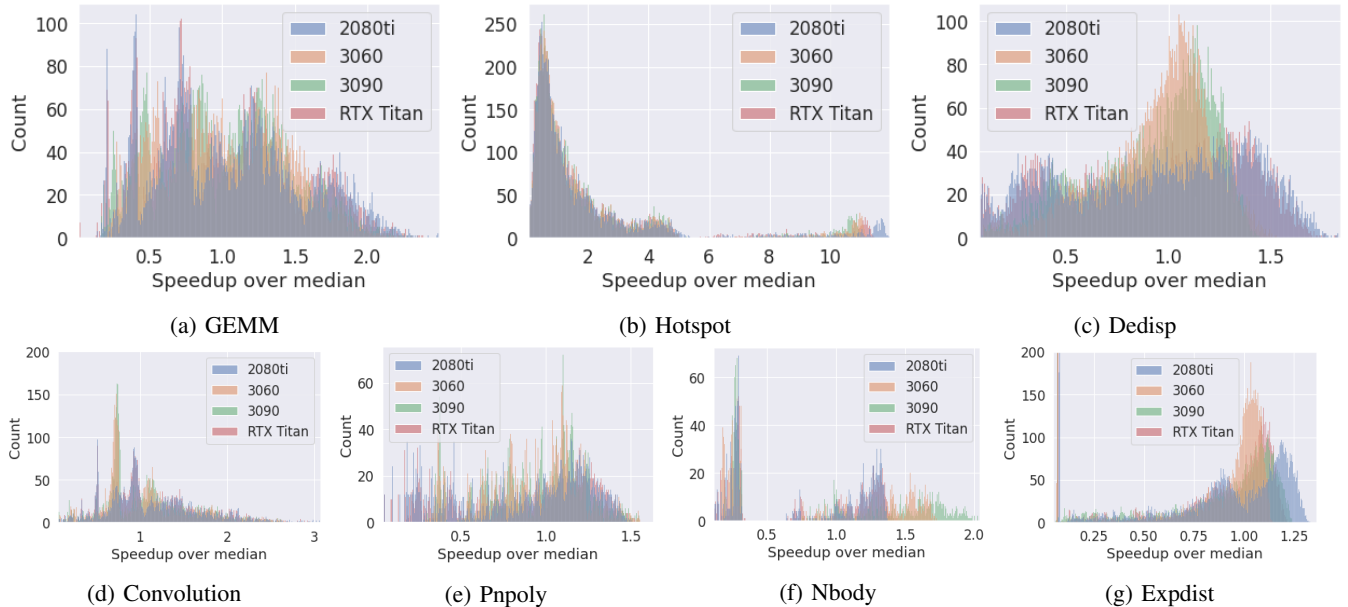
Figure 1: Performance distribution of configuration for all benchmarks on all architectures

configurations, but a cluster of very highly performing configurations giving more than 10x speedup. The Nbody distribution in Fig. 1f also shows a distinct high-density cluster of configurations that perform very poorly.

B. Convergence towards optimum using Random Search

Fig. 2 shows the convergence to the optimum configuration with relative performance (y-axis) plotted against the number of function evaluations on a Symmetric Log scale (linear scale from 0 to 1). Results are from random sampling 100 times from exhaustive or partial runs, with the median of the best evaluation plotted.

We can observe that there is a significant variance in the convergence between benchmarks, while there is less difference between GPUs. Again the Hotspot benchmark in Fig. 2b is a clear outlier, with Random Search quickly approaching a performance that is close to optimal. We stipulate that this is the due to the size of the high performing cluster shown in Section VI-A. This cluster is likely of a sufficient size such that random search can quickly find a solution in this cluster, which is then close to optimal.

There are also significant differences in how quickly the other benchmarks converge towards the optimal. Expdist in Fig. 2g and Nbody in Fig. 2f achieve a 90% optimum performance after just 10 function evaluations. For Dedisp (Fig. 2c) and PnPoly (Fig. 2e) it takes around 100 evaluations to reach the same level. We can also see here how for the RTX Titan the PnPoly benchmark shows how a single highly performing configuration can be the source of the final jump in relative performance.

Lastly Convolution (Fig. 2d) and GEMM (Fig. 2a) require hundreds of configurations to exceed 90%.

C. Proportion of centrality

We are using the proportion of centrality metric proposed by Schoonhoven et al. [3] to calculate the search difficulty of the benchmarks. We did not have sufficient resources to calculate the metric for the benchmarks with the largest search spaces, incl. Hotspot, Dedisp and Expdist. The results are shown in Figure 3. The results indicate that local search algorithms will generally find better performing configurations on the Convolution benchmark compared with GEMM and Pnpoly, which are comparatively more difficult benchmarks under this metric. This is in contrast to the results from our Random Search results, where all three benchmarks have similar trajectories towards the optimum.

D. Max speedup over Median

Fig. 4 shows the speedup between the Median performance configuration of the search space and the best possible configuration found. While most of the benchmarks have speedups between 1.5 - 3.06x, outliers like the Hotspot benchmark have very significant speedups from 11.12 - 11.97x.

E. Performance Portability

We can analyze the performance portability of configurations to measure the degree to which configurations are specifically optimized for each architecture. In Fig. 5 we show the relative performance compared with the optimal configuration for each architecture as the optimal configurations are transferred to the other architectures. The direction for this transfer is described by the optimal configuration for the GPU labeled in each row, being transferred to the different GPUs labeled on each of the columns.
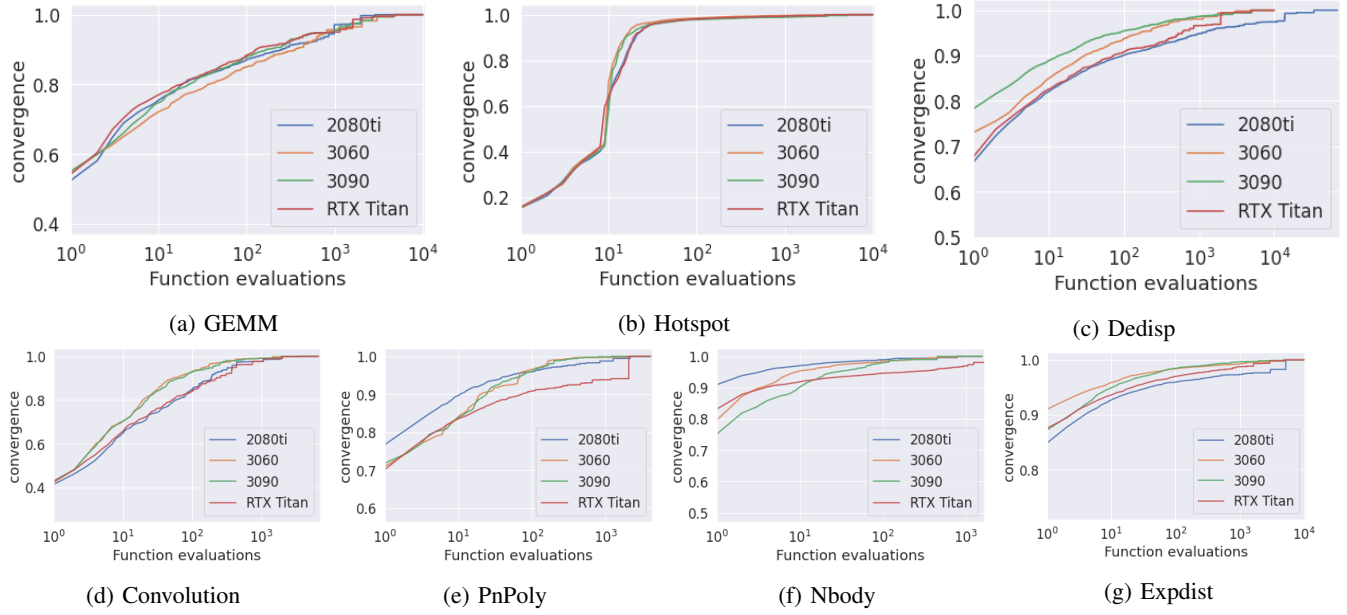
6

Figure 2: Convergence towards optimum for all benchmarks on all architectures
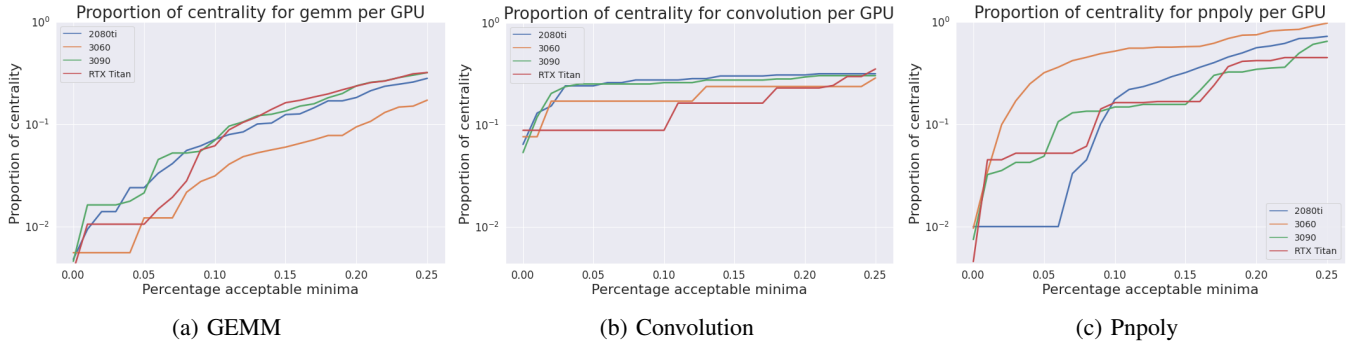

Figure 3: Proportion of centrality for all benchmarks on all architectures

Figure 4: Max speedup over Median configuration

In Fig. 5b we plot the portability for PnPoly. This shows that configurations are very portable between the RTX 3060 and RTX 3090, however configurations optimized for the

RTX 3090 transfer poorly to the RTX Titan (58.5% of optimal) and the 2080Ti (67.1%). Similarly for the Convolution benchmark in Fig. 5a the optimal configuration for the RTX 3060 transfers poorly to the RTX 2080Ti (73.3%) and RTX Titan (75.0%).

F. Feature importance

To assess the importance of the different search parameters for the benchmark's objective, we train a Catboost Regression model over the dataset and analyze this model to investigate which features it finds useful for prediction. We use Permutation Feature Importance to then analyze the impact of each individual parameter on the model's predictive power.

Training the model on the datasets the majority of the benchmarks and GPUs that use our CatBoost model is able to predict the performance of different configurations very precisely, with an R-squared score from 0.992 and upwards
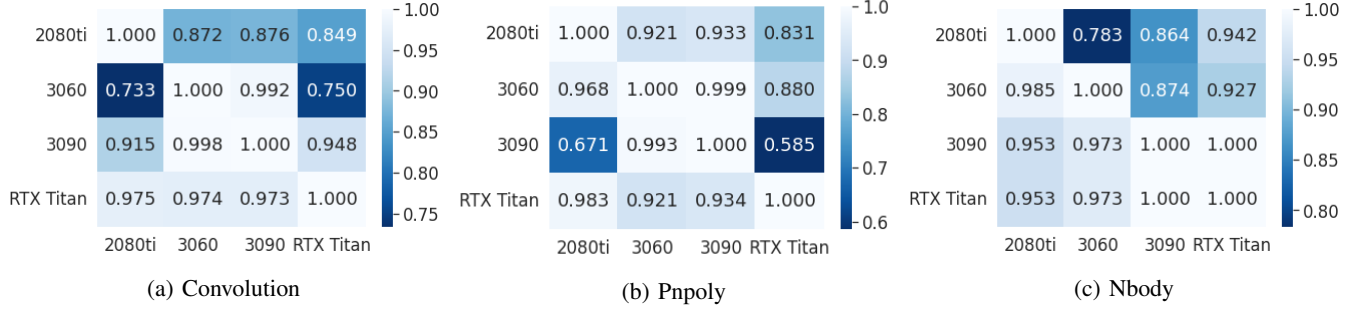
(a) Convolution

(b) Pnpoly

(c) Nbody

Figure 5: Performance portability for exhaustively searched benchmarks on all architectures

(a) GEMM

(b) Nbody

(c) Hotspot

(d) Convolution

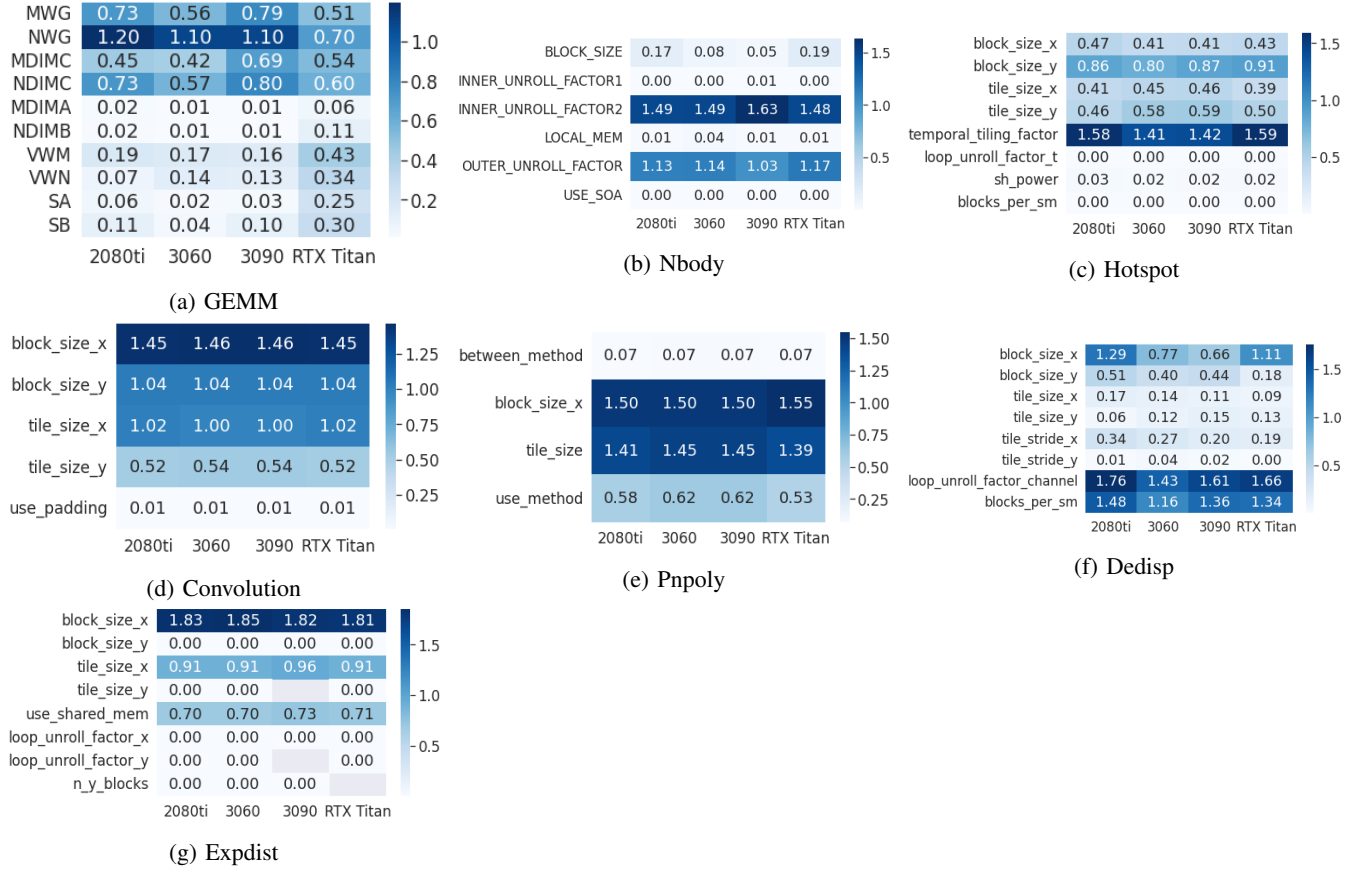(e) Pnpoly

(f) Dedisp

(g) Expdist

Figure 6: Feature importance for all benchmarks on all architectures

for all benchmarks except Convolution, where it ranges from 0.9268 to 0.9361.

Using this model we can then generate the feature importances in Fig. 6. We can observe that for many of the benchmarks, especially GEMM in Fig. 6a and Nbody in Fig. 6b, many of the parameters do not appear to have any meaningful impact on the model's predictive performance. Although the significance of these features may not extend to GPUs that are vastly different from those tested in this study, our findings are generally consistent across the various GPUs examined.

G. Feature importance impact on relevant search space

Given our previous analysis we can reduce the search space of the benchmarks to only include those parameters that has at least 0.05 feature importance on any of the architectures. The results for the cardinality of the search spaces can be seen in Table VIII. This gives an indication towards the size of the most interesting parts of the search space for the tested GPUs. Researchers using this benchmark can search over the full search space, but use this information to get better insight into how their models are able to

Table VIII: Search space sizes of benchmarks in BAT

Benchmark	Cardinality	Constrained	Valid	Reduced	Reduce-Constrained
PnPoly	4 092	4 092	3 734 - 3 774	4 092	3 734 - 3 774
Nbody	9 408	1 568	1 568	112	70
Convolution	18 432	9 400	5 220 - 5 256	4 700	4 700
GEMM	82 944	17 956	17 956	17 956	17 956
Expdist	9 732 096	540 000	N/A	144	96
Hotspot	22 200 000	21 850 147	N/A	220 000	202 582
Dedisp	123 863 040	107 011 905	N/A	3 870 720	3 327 135

concentrate on the most interesting parts of the search space.

H. Discussion

The results indicate the optimization parameters in our benchmarks have a significant impact on performance. While some parameters have more impact than others, the act of optimizing some parameters interact with other parameters. We observe this behavior through the Permutation Feature Importance summing up to a value much greater than 1 for many of the benchmarks. This behavior only occurs when there are significant dependencies between features. Thus this provides evidence towards the need for global optimization as opposed to orthogonal search algorithms. While the importance of each parameter is generally consistent across GPU architectures, the specific values for these parameters need to be optimized for the target architecture. Our portability study shows that simply transferring the optimal configuration from one architecture to another can give as low as 58.5% of the optimal performance, while other configurations can be ported at 99.9% of the optimal performance. Generally this is the case for GPUs of the same family of architectures like our RTX 3060 and RTX 3090.

VII. Conclusion and Future work

HPC systems with GPU are becoming increasingly more complex and challenging to hand-tune codes for. Autotuning framworks provide means that parameterizes kernels for a range of system parameters. BAT 2.0, the new benchmarking suite introduced in this study, provides a comprehensive framework for evaluating the performance of optimization algorithms in modern computing systems utilizing GPUs. The results of our analysis reveal that the optimization parameters have a significant impact on performance and the need for global optimization. The importance of autotuning is highlighted in the portability study, which shows that optimal performance can only be achieved by optimizing each application for a specific target architecture. The benchmarking suite facilitates the study of optimization algorithms and their effectiveness in achieving optimal performance, positioning it as a valuable tool in modern autotuning research.

A C++-based interface towards BAT is currently under development to support tuners like KTT [30] and CLTune [24].

Future work should thus include a comparison between C++-based and Python-based kernel tuners.

References

[1] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," *ACM Computing Surveys (CSUR)*, 2020.

[2] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "Optimization techniques for gpu programming," *ACM Comput. Surv.*, 2022.

[3] R. Schoonhoven, B. van Werkhoven, and K. J. Batenburg, "Benchmarking optimization algorithms for auto-tuning gpu kernels," *IEEE Transactions on Evolutionary Computation*, 2022.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[5] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[7] F. Petrovič, D. Střelák, J. Hozzová, J. Oľha, R. Trembecký, S. Benkner, and J. Filipovič, "A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit," *arXiv:1910.08498 [cs]*, 2019.

[8] I. Sund, K. A. Kirkhorn, J. O. Tørring, and A. C. Elster, "BAT: A Benchmark suite for AutoTuners," *Norsk IKT-konferanse for forskning og utdanning*, 2021.

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential Model-Based Optimization for General Algorithm Configuration," in *Learning and Intelligent Optimization*, 2011.

[11] B. van Werkhoven, "Kernel Tuner: A search-optimizing GPU code auto-tuner," *Future Generation Computer Systems*, 2019.

[12] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, 2001.

[13] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, 1998.

[14] A. Zien, N. Krämer, S. Sonnenburg, and G. Rätsch, "The Feature Importance Ranking Measure," in *Machine Learning and Knowledge Discovery in Databases*, 2009.

[15] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: Unbiased boosting with categorical features," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018.

[16] G. Fursin, "Collective knowledge: Organizing research projects as a database of reusable components and portable workflows with common interfaces," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 2021.

[17] Z. Wang, W. Kuang, C. Zhang, B. Ding, and Y. Li, *FedHPO-B: A Benchmark Suite for Federated Hyperparameter Optimization*, 2022.

[18] K. Šehić, A. Gramfort, J. Salmon, and L. Nardi, *LassoBench: A High-Dimensional Hyperparameter Optimization Benchmark Suite for Lasso*, 2022.

[19] K. Eggensperger, P. Müller, N. Mallik, M. Feurer, R. Sass, A. Klein, N. Awad, M. Lindauer, and F. Hutter, *HPOBench: A Collection of Reproducible Multi-Fidelity Benchmark Problems for HPO*, 2022.

[20] C. Nugteren, "CLBlast: A tuned OpenCL BLAS library," in *Proceedings of the International Workshop on OpenCL*, 2018.

[21] R. Goncalves, T. van Tilburg, K. Kyzirakos, F. Alvanaki, P. Koutsourakis, B. van Werkhoven, and W. van Hage, "A spatial column-store to triangulate the netherlands on the fly.," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[22] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven, "Bayesian Optimization for auto-tuning GPU kernels," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021.

[23] B. van Werkhoven, J. Maassen, H. E. Bal, *et al.*, "Optimizing convolution operations on gpus using adaptive tiling," *Future Gener. Comput. Syst.*, 2014.

[24] C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-Tuner for OpenCL Kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2015.

[25] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, 2019.

[26] H. Heydarian, F. Schueder, M. T. Strauss, B. van Werkhoven, M. Fazel, K. A. Lidke, R. Jungmann, S. Stallinga, and B. Rieger, "Template-free 2d particle fusion in localization microscopy," *Nature methods*, 2018.

[27] A. Sclocco, S. Heldens, and B. van Werkhoven, "Amber: A real-time pipeline for the detection of single pulse astronomical transients," *SoftwareX*, 2020.

[28] W. van Cappellen, T. Oosterloo, M. Verheijen, E. Adams, B. Adebahr, R. Braun, K. Hess, H. Holties, J. van der Hulst, B. Hut, *et al.*, "Apertif: Phased array feeds for the westerbork synthesis radio telescope-system overview and performance characteristics," *Astronomy & Astrophysics*, 2022.

[29] J. Ol'ha, J. Hozzová, J. Fousek, and J. Filipovič, "Exploiting historical data: Pruning autotuning spaces and estimating the number of tuning steps," *Concurrency and Computation: Practice and Experience*, 2020.

[30] J. Filipovič, F. Petrovič, and S. Benkner, "Autotuning of OpenCL Kernels with Global Optimizations," in *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, 2017.