Benchmarking optimization algorithms for auto-tuning GPU kernels

Richard Schoonhoven[1,3] Ben van Werkhoven[1,2] K. Joost Batenburg[1,3]
[1]Computational Imaging Group, Centrum Wiskunde & Informatica, Amsterdam, Netherlands
[2]Netherlands eScience Center, Amsterdam, Netherlands
[3]Leiden Institute of Advanced Computer Science, Leiden, Netherlands
{richard.schoonhoven, ben.van.werkhoven, k.j.batenburg}@cwi.nl

***Abstract*—Recent years have witnessed phenomenal growth in the application, and capabilities of Graphical Processing Units (GPUs) due to their high parallel computation power at relatively low cost. However, writing a computationally efficient GPU program (kernel) is challenging, and generally only certain specific kernel configurations lead to significant increases in performance. Auto-tuning is the process of automatically optimizing software for highly-efficient execution on a target hardware platform. Auto-tuning is particularly useful for GPU programming, as a single kernel requires re-tuning after code changes, for different input data, and for different architectures. However, the discrete, and non-convex nature of the search space creates a challenging optimization problem. In this work, we investigate which algorithm produces the fastest kernels if the time-budget for the tuning task is varied. We conduct a survey by performing experiments on 26 different kernel spaces, from 9 different GPUs, for 16 different evolutionary black-box optimization algorithms. We then analyze these results and introduce a novel metric based on the PageRank centrality concept as a tool for gaining insight into the difficulty of the optimization problem. We demonstrate that our metric correlates strongly with observed tuning performance.**

***Index Terms*—GPU computing, Auto-tuning, Performance optimization, Evolutionary computing, Fitness landscape analysis**

I. Introduction

GRAPHICS Processing Units (GPUs) have revolutionized the HPC landscape in the past decade [13], and are seen as one of enabling factors in recent breakthroughs in Artificial Intelligence (AI) [23]. GPUs originated as processors for gaming and then adapted to more general workloads as co-processors in many HPC systems. Over the past decade, GPUs have started to again penetrate new markets such as IoT devices [35] and autonomous vehicles [28]. The range of applications of GPUs as such continues to expand. Because of their relatively low cost with respect to their parallel processing power, more and more supercomputers come equipped with GPUs, and in 2020, the majority of modern supercomputers use GPUs [59] as the major source of compute power.

The sections of code that run on a GPU, called *kernels*, can be challenging to configure such that they run efficiently for a varying combinations of datasets and GPU architectures [63]. The kernel parameters can be split into those defined by the program, and those that are a consequence of the underlying architecture and models behind the GPU. The hardware-specific parameters define how the thousands of threads in a GPU are grouped. An ineffective layout can cause under-utilization of GPU resources. In general, the computational efficiency can drop by an order of magnitude depending on certain implementation choices. Typically, only a small subset of the possible configurations lead to a large increase in performance [53]. Therefore, it is vital to be able to select an efficient kernel configuration.

The search space for this problem is formed by all feasible combinations of GPU kernel parameters. This space is discrete and non-convex [45], making it hard to carry out the optimization. For most GPU kernels used in practice, the size of this search space is such that traversing the options by hand or brute-force is infeasible. An additional complication in optimizing kernel parameters is that evaluating the performance of each configuration requires costly recompilation and test runs. Furthermore, the same GPU kernel often requires re-tuning for different input data, hardware, or after changes to the code [18], [24], [40], [41]. Large throughput pipelines often rely on computationally expensive GPU kernels that consume large amount of resources [54], [55], and cannot be tuned exhaustively due to the aforementioned reasons.

Automatic performance tuning (auto-tuning) techniques rely on empirical results and feedback to optimize the kernel parameters with respect to desired performance metrics. These techniques aim to be widely applicable across architectures. For this reason, auto-tuning can be used to find configurations with increased performance for GPU programs. As the search space for the auto-tuning task depends on various aspects (kernel source, code layout, input data, GPU-architecture), the optimization framework must deal with a broad variety of search spaces and constraints. We, therefore, treat the problem as a black-box optimization task. This raises the question of which optimization algorithm is best suited to find highly efficient settings for GPU kernels, and how these optimization algorithms need to be configured to tune GPU kernels.

The main contribution of this work is to determine which optimization algorithms produce the fastest GPU kernels for different tuning-time ranges. To do so, we conduct a survey of 16 evolutionary optimization algorithms for 9 different NVidia and AMD GPUs, and run 3 real-world applicable benchmark kernels. We select our benchmark problems such that we are able (given ample time) to compute the entire search space,

Manuscript received September 13, 2021; revised June 10, 2022.

and make these spaces publicly available. To benchmark the GPU kernels we use the Kernel Tuner package [61]. We use the wide range of optimization algorithms present in Kernel Tuner for a large-scale comparison, and provide favourable default hyperparameters for GPU tuning for each algorithm. In addition, we extend Kernel Tuner with several highly-efficient optimization algorithms, including iterative local search (ILS) and dual annealing that cannot be found in any other generic auto-tuning framework.

Secondly, we aim to quantify tuning difficulty for these seemingly challenging and capricious search spaces. To do so, we introduce the *fitness flow graph* (FFG), which is a network of the points in the search space, with directed edges between neighbours with a better fitness. By computing the likelihood of local search walks terminating in good local minima, we use FFGs to better understand the discrepancies between optimization algorithms, and subsequently tailor them to better suit GPU tuning. In addition, FFGs can help explain the differences across GPU manufactures, architectures, and kernel programs, and such knowledge can help steer future development. To quantify tuning difficulty per kernel, we introduce a novel metric based on Google's PageRank algorithm [3], [43].

This work is structured as follows. In section II we discuss existing GPU kernel tuning approaches. In section III we introduce the preliminaries on GPU kernels, and describe the optimization algorithms that are considered in this survey. In section IV, we describe certain implementation details of Kernel Tuner and our Python optimization package BlooPy. We discuss the setup of our experiments in section V. In section VI we tune the hyperparameters of the algorithms, and present our findings on optimization algorithm performance. In section VII we introduce fitness flow graphs (FFGs) and quantify tuning difficulty for kernel search spaces. Finally, we present our conclusions in section VIII.

II. Related work

A. Automated performance tuning

It is well-known that GPU tuning can yield considerable gains in computational efficiency and utilization for large-scale, high-throughput pipelines that run on compute clusters. As an example, we mention the AMBER pipeline [54], [55], which is used to detect Fast Radio Bursts (FRBs) and other single pulse radio transients in astronomy. The pipeline has a throughput of 2 TB/s, and uses a large amount of resources. Benchmarking a single configuration is expensive, and the search space consists of millions of configurations, meaning that sophisticated tuning approaches have to be developed.

Research in automated performance tuning (auto-tuning) can be grouped into two main categories: (1) auto-tuning compiler-generated code optimizations [20], [44], [48], [57], and (2) software auto-tuning [24], [68]. Ashouri et al. [2] wrote an excellent survey on machine-learning methods for compiler-based auto-tuning. In this paper, we limit our scope to (2), i.e., optimizations methods for software auto-tuning, which is sometimes referred to as automated design space exploration [36]. Software auto-tuning allows developers to automatically optimize individual functions and allows, for example, to tune for entirely different implementations and parallelizations that solve the same problem.

As such, auto-tuning techniques are often employed to optimize the source code of high-performance libraries and applications for the CPU, e.g. ATLAS [66] or FFTW [8], as well as for GPUs [12], [24], [33], [53], [58], [62], [68].

A number of generic auto-tuning frameworks have been introduced in recent years. OpenTuner [1] was one of the first generic software auto-tuning frameworks, supporting a number of different search optimization algorithms, but with no support for tuning individual GPU kernels. GPTune [29] and HyperMapper [36] are recently proposed frameworks that both use Bayesian Optimization for auto-tuning on different platforms, but do not target GPUs.

Grauer-Gray et al. [11] have applied auto-tuning to the high-level directive-based HMPP framework, which can compile to CUDA or OpenCL code. They demonstrate significant performance improvements using auto-tuning over unoptimized HMPP kernels in the PolyBench benchmark suite. Wang et al. [65] take a similar compiler-based approach to automatically convert shared memory OpenMP applications into OpenCL code for GPUs. They use a machine-learning approach, which based on the number of compute operations and memory accesses in the kernels, predicts the best performing hardware platform to execute the kernels either the multi-core CPU using OpenMP, or on the GPU using OpenCL. Hou et al. [16] proposed a data-sensitive auto-tuning framework for sparse matrix vector (SpMV) multiplication that automatically finds the best parallelization strategy. They use a two-step machine learning approach in which they first determine the optimal way to group data into bins and then select the most suitable kernel to process the rows in each bin.

CLTune [40] was the first generic auto-tuning framework with specific support for directly tuning GPU kernels written in the OpenCL programming language. CLTune supports several optimization algorithms, including simulated annealing and particle swarm optimization, but these do not outperform random search [40]. Kernel Tuning Toolkit (KTT) [7] is developed specifically to support online auto-tuning and pipeline tuning, which allows for the exploration of combinations of tunable parameters over multiple kernels. An interesting feature of KTT is the support to keep track of hardware performance counters, such as L2 cache utilization, during benchmarking, which can also be used in advanced search strategies [6]. Auto-Tuning Framework (ATF) [49] implements an innovative way to generate auto-tuning search spaces, for efficient storage and fast exploration of constrained search spaces, but does not focus on introducing new optimization algorithms.

In earlier work, we have introduced Kernel Tuner [61], a generic auto-tuning framework specifically designed to be an easy-to-use and easy to extend tool for researching auto-tuning optimization algorithms. Kernel Tuner is a state-of-the-art framework that implements the largest range of search optimization strategies of all generic auto-tuning frameworks, and was the first generic framework to implement multiple search strategies that consistently outperformed random search [61].

B. Analyzing auto-tuning search spaces

In this paper, we do not only compare the performance of different optimization algorithms on the GPU auto-tuning problem, but we also investigate the properties of the search spaces to understand why certain optimization algorithms outperform others, and gain insight into the difficulty of the optimization problem.

Ryoo et al. [50] were one of the first to study the properties of optimization spaces for GPU applications. They defined two performance metrics to model the efficiency and utilization of a CUDA kernel and used these to find kernel configurations on the pareto curve that maximizes the two metrics. A downside of this approach is that the performance metrics have to be constructed for each kernel individually and require manual inspection and counting instructions in assembly code. Lim et al. [25] also look into search space properties to preselect certain parameter values using static code analysis in order to try to limit exploration of the search space by an auto-tuner.

In [42], Ochoa et al introduce the concept of *Local Optima Networks* (LONs). When constructing LONs the search space is partitioned into basins of attraction, i.e., sets of points where a local search algorithm will terminate in the same local minimum. A LON is a graph with the local optima as vertices, and a directed edge between nodes if a local search step transforms a solution from one basin of attraction to another. We build upon this idea to define *fitness flow graphs* (FFGs), which as opposed to LONs contain all the points in the search space. Due to the large number of points with “failure fitnesses” (when a configuration fails to compile) in GPU kernel spaces, defining the basin of attraction is difficult. Instead, we simplify the ideas behind LONs to the entire search space, and quantify how likely local search algorithms terminate in good local optima. To do so, we look at PageRank centrality of local optima. In [15] the idea of using PageRank centrality for LONs as predictor of performance for local-seach based heuristics was proposed, and in [14] the PageRank was used to rank space difficulty. We extend this idea to FFGs to determine GPU tuning difficulty.

III. Method: Optimization problem

In this section, we define the performance optimization of GPU kernels as a mathematical optimization problem, and we present the optimization algorithms that are part of our experiments.

A. GPU kernels

GPU kernels are executed by millions of threads in parallel to perform data-parallel computations on the GPU. However, the compute performance of a GPU kernel depends on how the software has been optimized for the hardware.

There are various different design choices that have an impact on the performance of GPU kernels, and this impact is challenging to accurately predict. For example, the way that a computation is parallelized and mapped on the thread blocks and individual threads affects the utilization of the GPU cores. Other design choices include what data types and data layouts to use in the various memory spaces available to GPU applications. There may also be entirely different algorithms to choose from to implement certain parts of the computation.

Other tunable parameters are introduced through code optimizations that can be enabled and may in turn introduce new parameters, such as tiling factors, vector data types, or partial loop unrolling factors. GPU kernels also have a number inherent parameters in terms of the number of thread blocks and the number of threads per block that are used to execute the kernel. The multitude of implementation choices for GPU kernels result in sizeable, non-convex, and discontinuous kernel design spaces.

To automate the kernel design space exploration process, GPU code can be parameterized, either using a kernel template or a code generator. An auto-tuner can take such a kernel template or code generator and empirically benchmark different kernel configurations, until it has found an efficient implementation. The search performed by the auto-tuner can be treated as a mathematical optimization problem of the form:

$$x^* = \underset{x \in X}{\arg\min}\, f(x) \tag{1}$$

where $f(x)$ is the performance metric to be minimized, for kernel configuration x for a combination of kernel, GPU device, and input settings. In this work the performance metric to be minimized will be the runtime of the kernel.

B. GPU kernel search spaces

The search space of possible settings for a GPU kernel can be characterized as a finite subset $X \subset \mathbb{Z}^n$ for n different parameters. Specifically, for each dimension $1 \leq i \leq n$, every entry x_i of a point $x \in X$ takes values from a finite set $S_i \subset \mathbb{Z}$. For example, the block dimension might allow values in $\{16, 32, 64, 128\}$. The total search space is the Cartesian product of these finite sets

$$X = S_1 \times S_2 \times \cdots \times S_n.$$

The local structure of a search space depends on the definition of *neighbouring points*. A common definition of the neighbours of a point are the points which differ only in one dimension, and are equal for all other dimensions. Mathematically, according to this definition the set of neighbours $N(x)$ of a point x is

$$N(x) := \bigcup_{i=1}^{n} \left\{ y \in X \setminus \{x\} \;\middle|\; y_j = x_j,\ \forall j \neq i \right\}. \tag{2}$$

Here, we will consider a more restrictive type of neighbourhood concept where we place the additional requirement that the parameter that differs from x_i should have a value adjacent to x_i in the list S_i. For example, if the block dimension is allowed to be $[16, 32, 64, 128]$, then neighbours of $x_i = 64$ would be 32 and 128, and the neighbour of 128 would only be 64. We consider this restriction because this definition gives information on whether closely related parameter values are related in performance.

Points of special interest in the search space are local minima. A point is a local minimum if all neighbouring points have a worse fitness. In other words, there are no improvements to be found in the local neighbourhood. Algorithms that scan local neighbourhoods can get stuck in local minima as there are no close points with better fitness.

C. Black-box optimization algorithms

As the search space is typically too large to iterate over all feasible points, a range of more sophisticated optimization algorithms are used in practice. In this section we describe the optimization algorithms that are considered in the experiments. As a categorization of these algorithms, we distinguish continuous, and discrete algorithms, and algorithms that learn about the stochasticity of the problem.

1) Optimization - discrete algorithms: Since tuning GPU kernels involves choosing the best from a finite set of possibilities, it makes sense to consider *discrete optimization algorithms*. These algorithms are considered in this work:

- **Random sampling** randomly generates solutions and records the highest scoring one. This strategy serves as a baseline comparison to determine if optimization algorithms offer significant benefits.

We consider several local search (or hill climb) algorithms which iteratively check for a neighbouring solution of lower fitness to visit, until a local minimum is reached. For all local search algorithms we distinguish between *best-improvement* search, where we move to the best neighbour next, and *first-improvement*, where we examine neighbours in a random order and move to one when we encounter an improvement.

Local search algorithms can vary the neighbourhood function that they use to generate new candidates. The algorithms can use both the version outlined in equation 2 (called *Hamming*), and the more restrictive neighbourhood definition in section III-B (called *adjacent*). First-improvement variants can decide whether they continue checking the remaining variables first after finding an improvement, or if they restart the search (hyperparameter *restart search*).

- **Multi-start local search (MLS)** repeatedly generates random starting solutions, and hill climbs them until a local minimum is reached.
  Hyperparameters: neighbourhood, restart.
- **Iterative local search (ILS)** [31] is similar to MLS but inherits part of the original local minimum when generating a new starting solution. After reaching a minimum, ILS performs several random permutations to generate a new starting solution. This *perturbation size* is a tunable parameter. In addition, a tunable *exit after no improve* hyperparameter randomly restarts if no improvement is found after a that many iterations, which helps to escape basins of attraction for small perturbation sizes.
  Hyperparameters: perturbation size, exit after no improve, neighbourhood, restart.
- **Tabu search** [9] maintains a queue of previously visited solutions which the algorithm is not allowed to visit. The tunable hyperparameter *tabu size* defines the queue size, and ensures that the new solution has not been visited for *tabu size* iterations. Tabu search always picks a new solution, whether it is an improvement or not.
  Hyperparameters: tabu size, neighbourhood.
- **Simulated annealing (SA)** [21] maintains a temperature parameter that, together with the fitness values, determines the probability that we move to a (potentially worse) neighbouring solution. The temperature parameter is decreased each iteration to mimic the behaviour of cooling processes in material physics. A tunable *exploration* parameter determines the size of the mutation of the current solution that is performed at each iteration. A *hill climber* subsequently optimizes the new solution, which is accepted with a certain probability.
  Hyperparameters: exploration, hill climber, neighbourhood

We also consider two discrete population-based algorithms, which require a *population size* parameter to determine the number of solutions they maintain. Population-based methods iteratively create a next generation of solutions by mixing solutions from the previous generation. A *reproduction* operator creates new initial solutions from existing ones, e.g., with two-point crossover a section of the solution vector is swapped between two solutions. After new solutions have been created, and their fitnesses determined, a *selection* mechanism determines which solutions are kept for this generation. For example, in tournament selection a number of randomly picked solutions compete for a spot in the next generation.

- **Genetic local search (GLS)** [17] (or memetic algorithm) is a population-based method where every solution in a generation is subsequently *hill climbed*. The initial population is made up of randomly generated solutions, which after hill climbing are all local minima. Next, a number of children is created by reproduction, and a new initial starting population is selected from the batch. These solutions are subsequently hill climbed and the procedure is repeated.
  Hyperparameters: hill climber, population size, reproduction, selection.
- **Genetic algorithm (GA)** [34] is similar to GLS, but instead of hill climbing each solution, it performs a single mutation only, e.g., permuting a single parameter. Instead of a hill climbing algorithm, GA has a tunable hyperparamter *mutation* that determines the fraction of variables of a solution that are mutated each generation.
  Hyperparameters: mutation, population size, reproduction, selection.

2) Optimization - continuous algorithms: As an alternative to discrete algorithms, we can consider *continuous optimization algorithms* which operate on real-valued solutions. In order to apply algorithms which assume continuous variables to a discrete problem such as GPU kernel tuning, we need to define a mapping between a real-valued vector, and the discrete values in the search space. Suppose the search space allows values $x_1, x_2, \ldots, x_n$ for a particular variable, then a continuous variable $y \in [0, 1]$ gets mapped to the closest grid

	GPU Specifications					
		CUDA cores/	Device		Bandwidth	Peak compute
GPU model	Year	Stream processors	memory	Boost clock	(GB/s)	(GFLOP/s)
NVidia Tesla K20	2012	2496	5 GB	0.76 GHz	208	3524
NVidia GTX Titan X	2015	3072	12 GB	1.08 GHz	336	6605
NVidia Tesla P100 PCIe	2016	3584	12 GB	1.30 GHz	549	9340
NVidia GTX 1080 Ti	2017	3584	11 GB	1.58 GHz	484	11340
NVidia Tesla V100 PCIe	2017	5120	32 GB	1.37 GHz	900	14899
AMD Radeon Instinct MI50	2018	3840	16 GB	1.73 GHz	1024	13300
NVidia Titan RTX	2018	4608	24 GB	1.77 GHz	672	16312
NVidia RTX 2070 Super	2019	2560	8 GB	1.77 GHz	448	9060
NVidia A100 PCIe	2020	6912	40GB	1.41 GHz	1555	19500

TABLE I: Specifications of graphical processing unints (GPUs) used to create experimental data.

point $\bar{y}$:

$$\begin{aligned} B &= \{\frac{1}{2n}, \frac{3}{2n}, \ldots, \frac{2n-1}{2n}\} \\ j^* &= \arg\min_{i=1,\ldots,n}\{|B_i - y|\} \\ \bar{y} &= x_{j^*}. \end{aligned}$$

Effectively, this ensures that all possible discrete values are equally spaced across the interval $[0, 1]$, and the continuous variable is mapped to the closest one. The continuous optimization algorithms operate on real-valued vectors with dimensions equal to the number of parameters that are to be optimized. Each entry is bounded to the unit interval.

The mapping ensures that points close together in real-valued space can get mapped to the same point in the GPU tuning space. While this can negatively impact the performance of continuous algorithms, it does not automatically lead to poor performance, as illustrated by the strong performance of continuous algorithms in Kernel Tuner [61]. Furthermore, this mapping allows us to explore a new class of algorithms. Here, we consider two local search algorithms.

- **Basin hopping** [64] is a global stepping algorithm that chooses new starting positions for local minimization. It requires the local minimizer *method* and a *temperature* parameter to be chosen. The temperature parameter determines the accept–reject criterion. Currently supported minimization methods are the nonlinear conjugate gradient (CG) [38], simplex (Nelder-Mead) [37], conjugate direction (Powell) [46], L-BFGS-B [4], Constrained Optimization BY Linear Approximation (COBYLA) [47], and Sequential Least Squares Programming (SLSQP) [22] methods.
  Hyperparameters: minimizer method, temperature.
- **Dual annealing** [60] is an extension of generalized simulated annealing, paired with a local minimization method. It combines global and local search procedures, and it requires users to choose a local minimization *method.*
  Hyperparameters: minimizer method.

Lastly, we consider two population-based algorithms.

- **Particle swarm optimization (PSO)** [19] initializes a *number of particles* at random in the search space. Each iteration, these particles update their position and velocity. Particles transmit information to a certain *number of neighbours*, thereby influencing the movement of the other particles.
  Hyperparameters: #particles, neighbours evaluated.
- **Differential evolution** [56] is similar to a genetic algorithm, but mixing strategies are based on real-valued solutions. Typically they involve mixing the best solution with a random candidate, and accepting the result with a certain probability.
  Hyperparameters: mixing method, population size, mutation size, recombination probability.

3) Optimization - tuning algorithms for stochastic optimization:

In this survey we consider two state-of-the-art parameter tuning algorithms;

- **Sequential Model Algorithm Configuration (SMAC)** [27] is a random forest-based Bayesian optimization method that is designed for optimization of stochastic problems. However, it can also be used to optimize deterministic problems. SMAC requires the *model type* of its Bayesian optimizer to be chosen. The *gp-mcmc* model was significantly slower and worse than *gp* in the preliminary experiments. We therefore use the *gp* model type in this work. The *acquisition function* of the BO is another tunable hyperparameter.
  Hyperparameters: acquisition function.
- **Iterated racing (irace)** [30] is a statistical approach for selecting the best configuration out of a set of candidates for stochastic optimization problems. After consulting the authors [30], we set *firstTest* and *nbConfigurations* as tunable hyperparameters for irace.
  Hyperparameters: firstTest, nbConfigurations.

IV. Implementation

In this section we comment on certain implementation details for the software developed for this work. The algorithms and analysis tools are implemented in the BlooPy Python package, and the tuning of the GPU kernels is performed by Kernel Tuner.

A. BlooPy and SOTA packages

The algorithms evaluated in this work are implemented in the discrete optimization package **BlooPy** (BLackbOx Optimization Python) [51]. The package implements the algorithms by encoding solutions as bitstrings. BlooPy implements several

		#local	#variables	#failed
Kernel	#points	minima	to tune	points
Convolution*	18432	89	6	12656*
GEMM	82944	64	5	64988
Point-in-polygon	8184	220	10	335

TABLE II: Statistics (averaged across GPU models) of kernel spaces. Number of failed points refers to the average number of configurations in the kernel space that failed to compile.
*Convolution has 864 points for AMD MI50, and 450 fail points.

functions for converting discrete solutions that are encoded as lists or arrays to bitstrings. Similarly, continuous solutions are mapped to discrete solution vectors using the mapping outlined in section III-C2. BlooPy requires the search space to be finite. Using the bitstring encodings, BlooPy's algorithms can make use of the computationally efficient Python module `bitarray` which implements fast low-level bitstrings in C. In addition, the algorithms are automatically applicable to benchmark bitstring-based optimization problems such as randomized Nk-landscapes [67].

Optimization algorithms in BlooPy maintain a cache of previously visited solutions. This means that a solution that has been visited before does not count as a function evaluation, and instead the cached value is returned. In addition to a variety of optimization algorithms, BlooPy implements several search space analysis tools. For example, it implements functions to determine the type of points in the search space, e.g., local minima and saddle points. Furthermore, BlooPy implements functions to compute the fitness flow graphs outlined in section VII-B. BlooPy can be installed from the GitHub source repository [51], or by package manager. To perform experiments with SMAC we used the Python package [26], and for irace we used the R package [32].

B. Kernel Tuner

Kernel Tuner [61] implements a wide range of optimization algorithms, and builds on top of various backends (e.g. PyOpenCL, PyCUDA, Cupy, GCC) that take care of the compilation process.

Kernel Tuner runs Python code, provided by the user, which calls the tuner function. In addition, the user needs to provide a code generator or parameterized template for the kernel they wish to optimize. An optimization algorithm then selects different kernel configurations for benchmarking.

V. Experimental Setup

To analyze the structure of different kernel spaces, and find the optimization algorithm best suited to finding strong kernel settings, we run experiments on 9 different GPUs, for 3 real-world applicable kernel programs (26 kernels in total).

- **Convolution** [62] operations are an essential tool in image processing, and are often used for tasks such as edge detection, blurring, or sharpening. They also feature prominently in deep learning methods for image processing as they form the backbone of the convolutional neural network (CNN).
- **GEMM** (Generalized dense matrix–matrix multiplication) [39] is one of the most widely-used kernels across many application domains, including neural networks. Here we perform the calculation $C = \alpha A \cdot B + \beta C$ for 4096×4096 matrices A, B, C, and constants α and β.
- **PnPoly** (Point-in-Polygon) kernel is used by Goncalves et al. [10] as part of a geospatial database management system to, for example, return all objects within the outline of a specific area.

Some statistics on the kernel spaces is given in Table II. For convolution and GEMM the majority of the points in the kernel space fail to compile, 68% and 78% respectively. In the case of a failed compilation we attribute a "fail" fitness of 10^{10} to this point. The exact kernel spaces can be found in the table of tunable parameters (Table V) in Appendix IX-A.

We selected these kernels programs since they are tunable common subroutines in real-world applications, but also have a compact parameter space which can be fully explored, given ample computation time. Note that this is not feasible for many other kernels used in practice (see section II-A). We have generated cache files of the entire search space for each kernel by brute-force calculation. This allows us to know the optimal settings for each problem, and therefore score solutions returned by algorithms. It also allows us to develop analysis metrics on the entire search space, which could at a later stage be adapted to work when sampling only small parts of the space. We supply our cache files for benchmarking optimization algorithms [52], similar to other computationally expensive applications such as neural architecture search [5].

The 9 GPUs that are used for testing are given in Table I. The convolution kernel is implemented in CUDA, GEMM in OpenCL, and PnPoly is a heterogeneous kernel that runs partly on the CPU, and partly on the GPU using CUDA. The PnPoly kernel uses CUDA-specific features that are not available on AMD GPUs. We have used CUDA Version 11.2, OpenCL 1.2, Python 3.8.5, PyCUDA v2021.1, PyOpenCL v2020.3.1, and BlooPy version 0.4.2.

Experimental setup: The GPU kernels are tuned with respect to runtime (ms). The runtime of a GPU kernel is stochastic, and can vary slightly per execution. Kernel Tuner automatically benchmarks a given configuration 32 times to acquire a mean runtime per configuration. In most cases, the compilation time for a given kernel configuration significantly exceeds the time needed to benchmark 32 runs. Therefore, most of our experiments are performed in a deterministic setting where the fitness of a configuration is the mean runtime. However, we also perform a stochastic experiment where a single kernel runtime is returned for every evaluation. This means that the fitness for the same point in the search space can vary, and algorithms that learn stochastic information, such as irace and SMAC, can potentially benefit.

After discussion with the authors [30] we decided to benchmark irace only for the stochastic experiment. This was decided as it was deemed inappropriate for the deterministic setting since the point of using irace is to dynamically handle stochasticity in expensive problems.

The algorithms are evaluated based on the fraction of the optimal runtime they can find within a limited budget of evaluations. For each algorithm, we run experiments with a

	Maximum number of function evaluations (budget)						
Basin hopping	25	50	100	200	400	800	1600
method	Powell	COBYLA				SLSQP	
temperature	0.1					1.0	
Dual annealing	25	50	100	200	400	800	1600
method	COBYLA	Powell					
Differential evolution	25	50	100	200	400	800	1600
population size	1	2		4	8	16	32
method	best1bin				best2bin	best1exp	
recombination	0.5	0.7					
mutation	(0.2, 0.7)						
Particle swarm optimization	25	50	100	200	400	800	1600
Number of particles	25		10	20	40	80	160
neighbours evaluated	5			10	20	26	32
FirstILS	25	50	100	200	400	800	1600
perturbation size	1.0					0.05	
Exit after no improve	25					10	
neighbour method	Hamming					adjacent	
restart search	False				True		
BestILS	25	50	100	200	400	800	1600
perturbation size	1.0					0.05	
Exit after no improve	25						
neighbour method	adjacent		Hamming			adjacent	
FirstTabu	25	50	100	200	400	800	1600
tabu size	4		2000				
neighbour method	Hamming						
BestTabu	25	50	100	200	400	800	1600
tabu size	2000						
neighbour method	Hamming						
FirstMLS	25	50	100	200	400	800	1600
restart search	True		False		True		
neighbour method	Hamming						
BestMLS	25	50	100	200	400	800	1600
neighbour method	adjacent			Hamming			
Simulated annealing	25	50	100	200	400	800	1600
explore (p)	1.0				0.7	0.1	
hill climber	None	RandomFirst					
neighbour method	Hamming						
Genetic local search	25	50	100	200	400	800	1600
hill climber	RandomFirst						
population size	2	2	4	16	20	40	80
reproductor	uniform		2point		uniform		
selector	RTS						
neighbour method	Hamming						
Genetic algorithm	25	50	100	200	400	800	1600
mutation	0.02			0.05		0.02	
population size	8	10	20	40	80	128	320
reproductor	1point					2point	
selector	tour8					tour4	
SMAC	25	50	100	200	400	800	1600
model type	gp					NA	
acquisition function	LCB					NA	
irace	25	50	100	200	400	800	1600
firstTest	NA			2			
nbConfigurations	NA			0			

TABLE III: Selected hyperparameters across different budgets (50 to 2000) for the optimization algorithms used in this work. The budgets columns are given in red. These hyperparameters were optimized using the convolution, GEMM, and PnPoly kernels on the NVidia P100 GPU.

maximum function evaluation limit (budget) of 25, 50, 100, 200, 400, 800, 1600. The goal of our experiments is to benchmark algorithms when traversing only a fraction of the search space. Therefore, we set the highest budget limit at 1600 since it is already approximately 20% of the Point-in-polygon search space. Every run is performed 50 times in order to get an indication of the spread. Due to computational demand, SMAC and irace experiments are ran 20 times. SMAC is only run up to a budget of 400 evaluations due to the high tuning time. Data and scripts for the experiments and figures can be found in the GitHub repository [52].

VI. Results: Benchmarking optimization algorithms on runtime

In this section, we first discuss how to initialize each algorithm with favourable hyperparameters. Next, we discuss which algorithms are best suited to tuning GPU kernels.

A. Setting hyperparameters

In order to compare the optimization algorithms fairly for the GPU tuning problem, we need to choose sensible hyperparameters. Which hyperparameters can be varied per algorithm is outlined in italics in section III-C. We test different combinations of hyperparameters on the P100, RTX

2070 Super, and GTX 1080Ti, for all three kernels (9 out of 26 kernels). This way we select reasonable hyperparameters across various architectures and kernels, which users can use as defaults for new GPU tuning problems. We performed a bruteforce search over all combinations of parameter values, and ran each set 20 times for each algorithm. For PSO we kept the w, c_1, c_2, p parameters constant as these appeared to have little effect on algorithm performance. Note that the time required to tune hyperparameters varies greatly between algorithms due to this exhaustive search. The hyperparameters chosen are in Table III.

To choose hyperparameters, we first group settings which perform similarly statistically, and attempt to find one set of hyperparameters that performs well across all 9 kernels. For a budget p, let $f_{p,best}$ be the lowest average fitness achieved for a set of hyperparameters, and $\sigma_{p,best}$ the standard deviation. We perform the following selection approach:

1) For every kernel, we create a set of hyperparameter settings whose average found fitness is within $k \cdot \sigma_{p,best}$ of $f_{p,best}$.
2) For each budget, intersect the acceptable settings for convolution, GEMM, and PnPoly, across the 3 GPUs.
3) For each budget, if this intersection is non-empty, reduce k and repeat. If the intersection is empty, increase k and repeat. Repeat until only one set of hyperparameters remains in the intersection.

B. Kernel tuning algorithm comparison

To quantify which kernel tuning algorithms perform best for certain budgets, we can check whether an algorithm provided statistically significantly better results than others for a certain experiment. To do so, we use a *two-sample independent t-test* with $\alpha = 0.05$. For each GPU, kernel, and budget combination, we perform the t-test to see if an algorithm A performed significantly better than algorithm B. We subsequently combine the total number of "wins" for algorithm A across all GPUs (excluding those that were used for tuning).

We split the competitions into low-range (200 evaluations or fewer), and medium-range budgets. The competition tables at different splits can be found in the Appendix IX-B. The full plots per GPU and algorithm can be found in Appendix IX-C. The results of these inter-algorithm competitions are given in the heatmaps displayed in Figure 1. The competition heatmaps display how often the column algorithm found a statistically better solution than the row algorithm in that budget range.

1) Kernel tuning: deterministic fitness: In this section we present the results of our deterministic experiments, i.e., the algorithms have to minimize the runtime of a GPU kernel where the runtime is fixed at the mean of 32 runs.

Low-range budget: For 200 function evaluations or fewer, for convolution dual annealing was statistically better than all other algorithms for all GPU models (see column with "DualAnnealing" for convolution ≤ 200 function evaluations), with simulated annealing as second. For GEMM, basin hopping and dual annealing perform equally with dual annealing beating basin hopping 5 times, and basin hopping beating dual annealing 6 times. For PnPoly dual annealing was again best, followed by SMAC. We show the total number of wins and losses across all kernels and GPUs in Table IV. Here we see that for low budgets dual annealing has significantly more wins, and fewer losses than all other algorithms.

	Low budget		Medium budget	
	total wins	total losses	total wins	total losses
Basin hopping	403	204	131	393
Dual annealing	**680**	**65**	227	233
Differential evolution	426	192	150	347
PSO	290	327	136	368
FirstILS	352	233	**361**	77
BestILS	125	472	257	126
FirstTabu	148	430	255	183
BestTabu	35	677	220	185
FirstMLS	317	205	341	64
BestMLS	143	466	226	174
Simulated annealing	412	150	360	**59**
Genetic local search	320	216	329	**59**
Genetic algorithm	376	162	201	207
SMAC	356	344	48	145
Random sampling	222	463	7	629

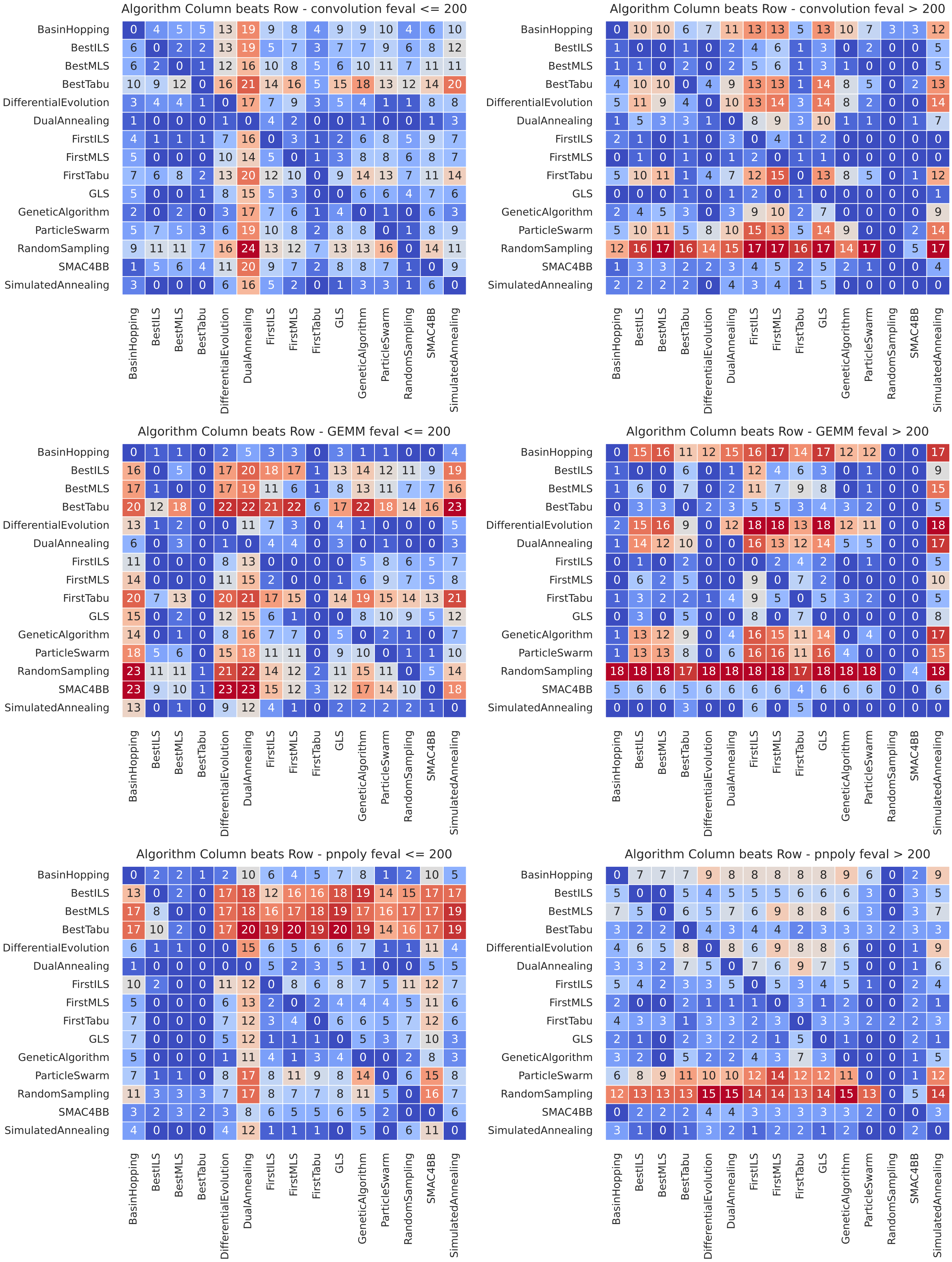
TABLE IV: Total number of wins: sum of occurrences when the algorithm found statistically better solutions than other algorithms (summed over all kernels). A win (and corresponding loss for the other algorithm) is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). Low budget is for $\leq$ 200 budgets, i.e., 25, 50, 100, and 200. Medium budget is for $>$ 200 budgets, i.e., 400, 800, 1600. The top 3 cells are coloured green.

Interestingly, SMAC was second for PnPoly, in the best half of algorithms for convolution, but did not even beat random sampling for GEMM. We hypothesize this is either due to the number of variables to optimize for each of the three kernels (see Table II), or the increasing fraction of fail fitnesses for these kernels. We hypothesize that the Bayesian optimizer could not fit a proper surrogate for GEMM with a low budget, and many failed compilations.

Medium budget: For more than 200 function evaluations, FirstMLS and GLS performed best for convolution, followed by FirstILS. For GEMM, FirstILS and simulated annealing performed best, followed by GLS. For PnPoly, FirstMLS is the strongest algorithm, followed by simulated annealing and GLS. As can be seen from Table IV, FirstILS and simulated annealing have the most number of total wins, and simulated annealing and genetic local search have the least number of losses.

Additional remarks: In general, best-improvement local search algorithms performed significantly worse than the first-improvement variants. In fact, for the low range, they proved statistically worse than random sampling for PnPoly, and in general have fewer wins and more losses. This can be explained due to the fact that exploring all the neighbours before taking a step costs many evaluations, and leads to only exploring a single neighbourhood for low budgets. For the population-based methods, GLS is the best performing algorithm for medium budgets, but does worse than differential evolution and GA for low budgets. PSO performed significantly worse.

Interestingly, dual annealing, which works on real-valued solution vectors, performs well for low budgets. It seems that the mapping from $[0, 1]^n$ to discrete space does not prevent



Algorithm Column beats Row - convolution feval <= 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	4	5	5	13	19	9	8	4	9	9	10	4	6	10
BestILS	6	0	2	2	13	19	5	7	3	7	7	9	6	8	12
BestMLS	6	2	0	1	12	16	10	8	5	6	10	11	7	11	11
BestTabu	10	9	12	0	16	21	14	16	5	15	18	13	12	14	20
DifferentialEvolution	3	4	4	1	0	17	7	9	3	5	4	1	1	8	8
DualAnnealing	1	0	0	0	1	0	4	2	0	0	1	0	0	1	3
FirstILS	4	1	1	1	7	16	0	3	1	2	6	8	5	9	7
FirstMLS	5	0	0	0	10	14	5	0	1	3	8	8	6	8	7
FirstTabu	7	6	8	2	13	20	12	10	0	9	14	13	7	11	14
GLS	5	0	0	1	8	15	5	3	0	0	6	6	4	7	6
GeneticAlgorithm	2	0	2	0	3	17	7	6	1	4	0	1	0	6	3
ParticleSwarm	5	7	5	3	6	19	10	8	2	8	8	0	1	8	9
RandomSampling	9	11	11	7	16	24	13	12	7	13	13	16	0	14	11
SMAC4BB	1	5	6	4	11	20	9	7	2	8	8	7	1	0	9
SimulatedAnnealing	3	0	0	0	6	16	5	2	0	1	3	3	1	6	0

Algorithm Column beats Row - convolution feval > 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	10	10	6	7	11	13	13	5	13	10	7	3	3	12
BestILS	1	0	0	1	0	2	4	6	1	3	0	2	0	0	5
BestMLS	1	1	0	2	0	2	5	6	1	3	1	0	0	0	5
BestTabu	4	10	10	0	4	9	13	13	1	14	8	5	0	2	13
DifferentialEvolution	5	11	9	4	0	10	13	14	3	14	8	2	0	0	14
DualAnnealing	1	5	3	3	1	0	8	9	3	10	1	1	0	1	7
FirstILS	2	1	0	1	0	3	0	4	1	2	0	0	0	0	0
FirstMLS	0	1	0	1	0	1	2	0	1	1	0	0	0	0	0
FirstTabu	5	10	11	1	4	7	12	15	0	13	8	5	0	1	12
GLS	0	0	0	1	0	1	2	0	1	0	0	0	0	0	1
GeneticAlgorithm	2	4	5	3	0	3	9	10	2	7	0	0	0	0	9
ParticleSwarm	5	10	11	5	8	10	15	13	5	14	9	0	0	2	14
RandomSampling	12	16	17	16	14	15	17	17	16	17	14	17	0	5	17
SMAC4BB	1	3	3	2	2	3	4	5	2	5	2	1	0	0	4
SimulatedAnnealing	2	2	2	2	0	4	3	4	1	5	0	0	0	0	0

Algorithm Column beats Row - GEMM feval <= 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	1	1	0	2	5	3	3	0	3	1	1	0	0	4
BestILS	16	0	5	0	17	20	18	17	1	13	14	12	11	9	19
BestMLS	17	1	0	0	17	19	11	6	1	8	13	11	7	7	16
BestTabu	20	12	18	0	22	22	21	22	6	17	22	18	14	16	23
DifferentialEvolution	13	1	2	0	0	11	7	3	0	4	1	0	0	0	5
DualAnnealing	6	0	3	0	1	0	4	4	0	3	0	1	0	0	3
FirstILS	11	0	0	0	8	13	0	0	0	0	5	8	6	5	7
FirstMLS	14	0	0	0	11	15	2	0	0	1	6	9	7	5	8
FirstTabu	20	7	13	0	20	21	17	15	0	14	19	15	14	13	21
GLS	15	0	2	0	12	15	6	1	0	0	8	10	9	5	12
GeneticAlgorithm	14	0	1	0	8	16	7	7	0	5	0	2	1	0	7
ParticleSwarm	18	5	6	0	15	18	11	11	0	9	10	0	1	1	10
RandomSampling	23	11	11	1	21	22	14	12	2	11	15	11	0	5	14
SMAC4BB	23	9	10	1	23	23	15	12	3	12	17	14	10	0	18
SimulatedAnnealing	13	0	1	0	9	12	4	1	0	2	2	2	2	1	0

Algorithm Column beats Row - GEMM feval > 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	15	16	11	12	15	16	17	14	17	12	12	0	0	17
BestILS	1	0	0	6	0	1	12	4	6	3	0	1	0	0	9
BestMLS	1	6	0	7	0	2	11	7	9	8	0	1	0	0	15
BestTabu	0	3	3	0	2	3	5	5	3	4	3	2	0	0	5
DifferentialEvolution	2	15	16	9	0	12	18	18	13	18	12	11	0	0	18
DualAnnealing	1	14	12	10	0	0	16	13	12	14	5	5	0	0	17
FirstILS	0	1	0	2	0	0	0	2	4	2	0	1	0	0	5
FirstMLS	0	6	2	5	0	0	9	0	7	2	0	0	0	0	10
FirstTabu	1	3	2	2	1	4	9	5	0	5	3	2	0	0	5
GLS	0	3	0	5	0	0	8	0	7	0	0	0	0	0	8
GeneticAlgorithm	1	13	12	9	0	4	16	15	11	14	0	4	0	0	17
ParticleSwarm	1	13	13	8	0	6	16	16	11	16	4	0	0	0	15
RandomSampling	18	18	18	17	18	18	18	18	17	18	18	18	0	4	18
SMAC4BB	5	6	6	5	6	6	6	6	4	6	6	6	0	0	6
SimulatedAnnealing	0	0	0	3	0	0	6	0	5	0	0	0	0	0	0

Algorithm Column beats Row - pnpoly feval <= 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	2	2	1	2	10	6	4	5	7	8	1	2	10	5
BestILS	13	0	2	0	17	18	12	16	16	18	19	14	15	17	17
BestMLS	17	8	0	0	17	18	16	17	18	19	17	16	17	17	19
BestTabu	17	10	2	0	17	20	19	20	19	20	19	14	16	17	19
DifferentialEvolution	6	1	1	0	0	15	6	5	6	6	7	1	1	11	4
DualAnnealing	1	0	0	0	0	0	5	2	3	5	1	0	0	5	5
FirstILS	10	2	0	0	11	12	0	8	6	8	7	5	11	12	7
FirstMLS	5	0	0	0	6	13	2	0	2	4	4	4	5	11	6
FirstTabu	7	0	0	0	7	12	3	4	0	6	6	5	7	12	6
GLS	7	0	0	0	5	12	1	1	1	0	5	3	7	10	3
GeneticAlgorithm	5	0	0	0	1	11	4	1	3	4	0	0	2	8	3
ParticleSwarm	7	1	1	0	8	17	8	11	9	8	14	0	6	15	8
RandomSampling	11	3	3	3	7	17	8	7	7	8	11	5	0	16	7
SMAC4BB	3	2	3	2	3	8	6	5	5	6	5	2	0	0	6
SimulatedAnnealing	4	0	0	0	4	12	1	1	1	0	5	0	6	11	0

Algorithm Column beats Row - pnpoly feval > 200

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	7	7	7	9	8	8	8	8	8	9	6	0	2	9
BestILS	5	0	0	5	4	5	5	5	6	6	6	3	0	3	5
BestMLS	7	5	0	6	5	7	6	9	8	8	6	3	0	3	7
BestTabu	3	2	2	0	4	3	4	4	2	3	3	3	2	3	3
DifferentialEvolution	4	6	5	8	0	8	6	9	8	8	6	0	0	1	9
DualAnnealing	3	3	2	7	5	0	7	6	9	7	5	0	0	1	6
FirstILS	5	4	2	3	3	5	0	5	3	4	5	1	0	2	4
FirstMLS	2	0	0	2	1	1	1	0	3	1	2	0	0	2	1
FirstTabu	4	3	3	1	3	3	2	3	0	3	3	2	2	2	3
GLS	2	1	0	2	3	2	2	1	5	0	1	0	0	2	1
GeneticAlgorithm	3	2	0	5	2	2	4	3	7	3	0	0	0	1	5
ParticleSwarm	6	8	9	11	10	10	12	14	12	12	11	0	0	1	12
RandomSampling	12	13	13	13	15	15	14	14	13	14	15	13	0	5	14
SMAC4BB	0	2	2	2	4	4	3	3	3	3	3	2	0	0	3
SimulatedAnnealing	3	1	0	1	3	2	1	2	2	1	2	0	0	2	0

Fig. 1: Heatmaps counting the occurrences when the column algorithm found statistically better solutions than the row algorithm for the (top) convolution, (middle) GEMM, and (botttom) PnPoly kernels. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Left): Heatmap for low ≤ 200 budgets, i.e., 25, 50, 100, and 200. (Right): Heatmap for mid and high > 200 budgets, i.e., 400, 800, 1600. Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

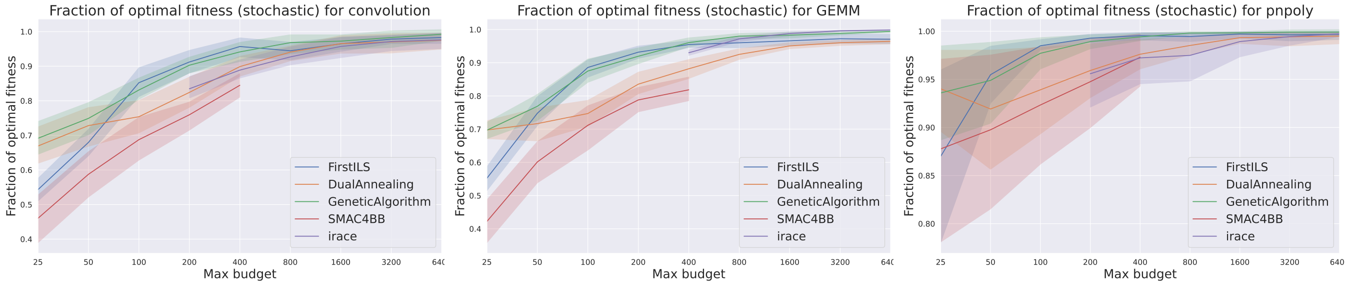


Fig. 2: Fraction of optimal runtime for max budget supplied over all GPUs. Each point is the mean fraction of optimal runtime found (y-axis) for each budget limit (x-axis) over all GPUs, with the shaded region indicating 95% confidence interval. Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

dual annealing from finding strong solutions quickly. One of the main drawbacks of using continuous algorithms is that if a continuous algorithm updates its real-valued solution vector, it could mean that it does not actually update the discrete solution vector it is mapped to. However, since our algorithms cache previously visited solutions (only for the deterministic experiments), such redundant optimization steps do not cost any budget. We think this may negatively impact gradient-based algorithms as the subroutines Powell and COBYLA, which do not require derivatives to be known, are the selected solvers for dual annealing during hyperparameter tuning.

As a final remark, we notice that SMAC performs poorly in the medium budgets. Note that SMAC only has $1/3$ as many data points in the medium budget since we do not perform the 800 and 1600 budget experiments for SMAC. Nevertheless the algorithm performs poorly on the 400 budget compared to other methods. It seems that SMAC is unsuccessful in fitting a meaningful surrogate model for kernel tuning. This could be due to the deterministic setup of this experiment, or due to the high number of fail configurations with “infinite” fitness.

2) Kernel tuning: stochastic fitness: For the stochastic experiments the algorithms have to minimize the runtime of a GPU kernel where the runtime is a random draw from the 32 timings. In addition to running SMAC and irace, we also run FirstILS, GA and dual annealing which did well for deterministic fitnesses. We remark that irace throws an error if the budget is too small with respect to the number of variables, and therefore starts at a budget of 200 for convolution and PnPoly, and 400 for GEMM.

The experimental results are shown in Figure 2. Here we aggregate the results per kernel for all GPU models by showing the mean fraction of optimum (and 95% confidence interval) for a given max budget. We see that GA and dual annealing are best for low budgets in the stochastic experiments. FirstILS does well for budgets ≥ 100. Irace is the best method for GEMM with budgets ≥ 800, but for convolution and pnpoly irace is not as good as GA, dual annealing, and FirstILS.

SMAC consistently achieves a lower fraction of optimality than the competing algorithms across kernels and budgets. Again, we hypothesize that this is because of the high number of fail fitnesses in the search spaces (see Table II). This makes it hard for the Bayesian optimizer to fit a meaningful surrogate.

Stochastic or deterministic: Overall, we notice that higher budgets are necessary to find good solutions for the stochastic experiments than in the deterministic case. This leads to a higher overall tuning time. We therefore recommend to treat GPU kernel tuning as a deterministic optimization problem, with the mean runtime as fitness. The added stochastic information does not appear to allow SMAC or irace to consistently outperform conventional black-box algorithms. This could be because the runtime does not vary much; the average (normalized) runtime and standard deviation is 1.000 ± 0.011. Second, the high number of failure configurations could confuse models that try to learn stochastic information.

VII. Quantifying GPU tuning difficulty

In this section we want to gain insight into the difficulty of the GPU kernel tuning optimization problem, and quantify kernel spaces according to tuning difficulty. When attempting to understand why certain GPU kernel spaces appear difficult to optimize we found that relatively simple metrics do not coincide with our experimental results. We outline discrepancies between an intuitive simple metric and our experimental results, and introduce a novel refined approach that does correlate with our results.

A. Naive metric: fraction of optimal fitness of local minima

Method: As an example of a simple metric that intuitively could explain the results, we consider the fraction of optimal fitness of local minima. For a minimum x_i, and optimal fitness f_{opt}, we can consider the *fraction of optimal fitness* of the minimum $f_{opt}/f(x_i)$. In this case, we divide the global minimal runtime by the runtime of the minima.

Results: In Figure 3 we show a scatter plot of the fraction of optimal fitness for the local minima for the convolution kernel (per GPU model). According to this distribution, the V100 and A100 GPUs have the closest to optimal median fitness for local minima. This means that an algorithm that randomly explores local minima with equal probability will obtain the closest to optimal runtime for these kernels.

Analysis: To empirically check how difficult the GPU kernels are to tune, we can plot the fraction of optimal fitness that optimization algorithms managed to achieve for certain budgets. If f_j is the lowest fitness found for a single run for some budget p, a point in the plot is the average over 50 runs computed as $\tilde{f}_p := (1/50) \cdot \sum_{j=1}^{50} (f_{opt}/f_{j,p})$. In Figures 4

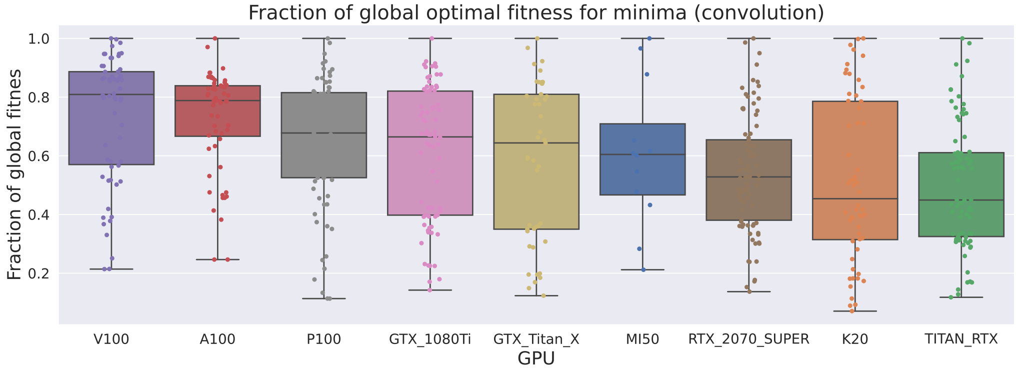


Fig. 3: Fraction of optimal fitness of local minima for each GPU model, for the convolution kernel. The box plots shows the median line, the box designates the quartiles, and the whiskers the full extend of the distribution. Additionally, a scatter plot of the fitness for each local minimum is shown. The GPUs are ordered in descending median fraction of optimal fitness from left to right.

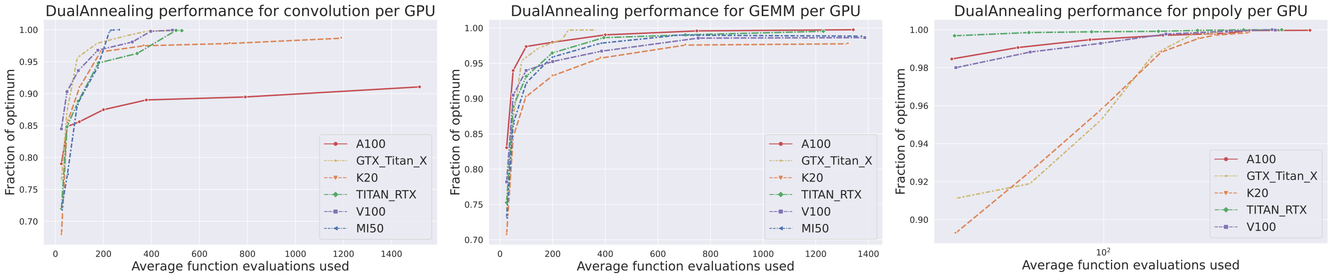


Fig. 4: **Dual annealing:** Fraction of optimal runtime for different budgets (per GPU). Each point is the average fraction of optimal runtime found (y-axis) for each budget, with respect to the average number of evaluations actually used (x-axis) for that budget. Function evaluations used counts only the unique settings that were visited. Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

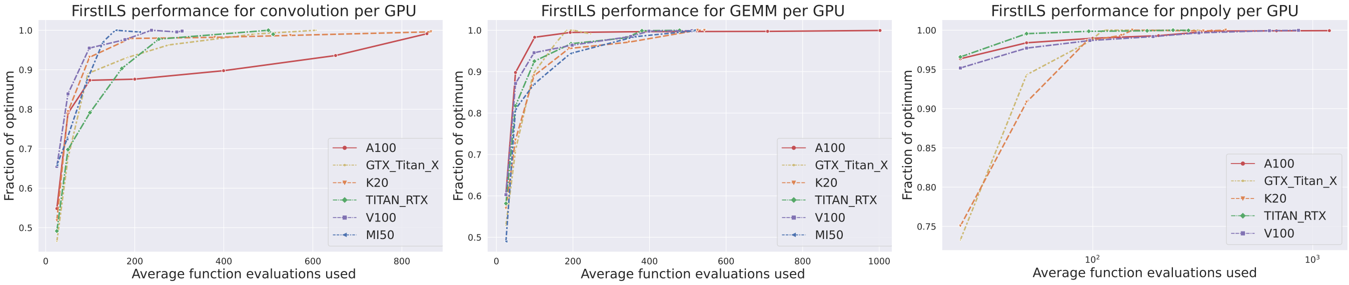


Fig. 5: **FirstILS:** Fraction of optimal runtime for different budgets (per GPU). Each point is the average fraction of optimal runtime found (y-axis) for each budget, with respect to the average number of evaluations actually used (x-axis) for that budget. Function evaluations used counts only the unique settings that were visited. Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

and 5 we plot $\tilde{f}_p$ for dual annealing and FirstILS. We chose dual annealing and FirstILS as they represent the strongest algorithms for low, and medium budgets respectively.

We see that for convolution on the A100 GPU both algorithms returned solutions which were furthest away from the optimum, while for the V100 both optimizers return close to optimal solutions for few function evaluations. These observations are opposite to what would be expected on the basis of Figure 3. Hence, the distribution of fitness of the local optima does not properly explain tuning difficulty.

B. Refined approach: Fitness flow graphs and PageRank

Method: A more refined metric to quantify GPU tuning difficulty may be to compute how likely local search algorithms terminate in local minima. For this purpose, we introduce the *fitness flow graph* (FFG), which contains all points in the search space, and creates a directed edge to a neighbouring point if the neighbour has lower fitness. This means that a random walk across the FFG mimics the behaviour 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. We show two example FFGs in Figure 6.

To compute the likelihood of arrival per local minima, we compute the PageRank *node centrality*, which was originally used to determine the relevance of a webpage [3], [43]. Let A_G be the adjacency matrix of a directed graph G, rescaled such that each column adds up to 1. Essentially, this means that for every node, the column is a probability vector of visiting

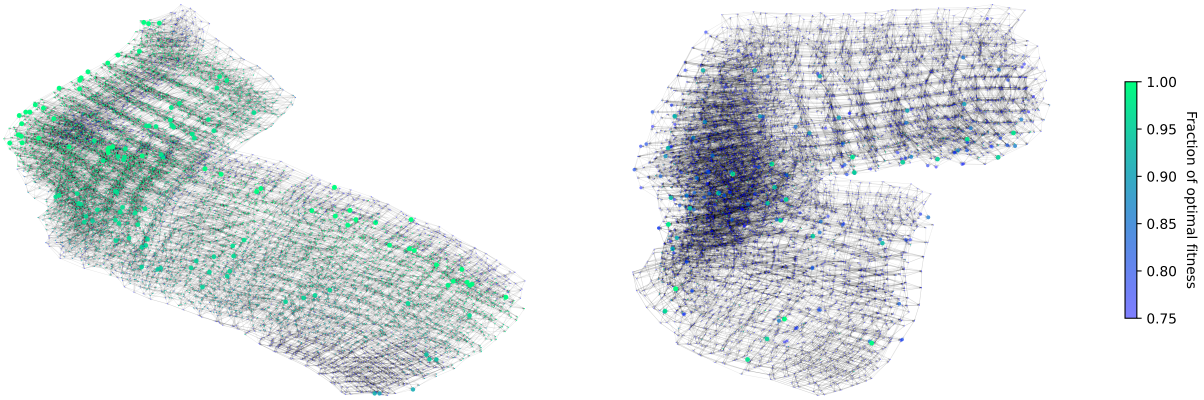


Fig. 6: Fitness flow graphs of PnPoly kernel search spaces of the (left) NVidia Titan RTX, and (right) NVidia GTX 1080Ti. Each node is a point in the search space. There is a directed edge between neighbouring points from higher to lower fitness. Points are coloured within a fitness range of $+25\%$ with respect to the global minimal fitness (global minimum in green), i.e., each point is coloured by its fraction of optimal fitness, and points with a fraction below 0.75 are given the same colour. Local minima are represented as larger nodes.

adjacent nodes with equal likelihood. The PageRank values are then the values of the dominant right eigenvector of A_G. For an FFG, this means that the PageRank value of a local minimum is the probability of arriving in that minimum after a long random walk through the graph.

As a measure of difficulty we consider how likely a certain subset of "suitably good" local minima are to be visited by a local search algorithm relative to the rest. Suppose that f_{opt} is the optimal fitness, and let $L(X)$ be the set of local minima of X. Given a proportion p, we take the set of nodes $L_p(X)$ consisting of local minima with fitness less than $(1+p)f_{opt}$ (for minimization problems, otherwise $(1-p)f_{opt}$). For a centrality function c_G, we define the *p-proportion of centrality*

$$C_p(G, X) = \frac{\sum_{x \in L_p(X)} c_G(x)}{\sum_{x \in L(X)} c_G(x)}. \tag{3}$$

Results: The proportion of centrality for strong local minima for each FFG is shown in Figure 7. We calculate the proportion of centrality for different acceptance percentages with respect to the global minimum of $p = 0, 1, 2, \ldots, 15\%$.

Revisiting the A100 and V100 convolution kernel comparison, we see that the proportion of centrality matches the experimental observations for dual annealing and FirstILS. Figure 7 shows that the NVidia V100 has the most central local minima, whereas the A100 has the least central local minima. For the GEMM and PnPoly kernels, Figures 4 and 5 align with the expectations based on the proportion of centrality. For example, the group of PnPoly kernels with lowest proportion of centrality (P100, GTX Titan X, K20, GTX 1080Ti) are indeed the hardest to tune for both algorithms.

One exception is the K20 GEMM kernel, where proportion of centrality does not entirely reflect the perceived difficulty for dual annealing. This suggests that the proportion of centrality may correlate with GPU tuning difficulty better for certain optimization algorithms. This is to be expected since the PageRank centrality on the FFG in expectation mimics the performance of randomized first-improvement local search. Algorithms that are substantially different than first-improvement local search will therefore also correlate less with the expected difficulty on the basis of proportion of centrality.

Analysis: Overall, the experimental results suggest that the proportion of centrality is a suitable metric for estimating tuning difficulty for GPU kernels. By using FFGs and the PageRank algorithm, we are able to observe kernel differences that were otherwise unknown. For example, both the A100 and V100 convolution kernels have few outlier minima with a close to optimal fitness. In fact, the existence of only a few kernel configurations that lead to large increases in performance is a general property of certain GPU kernels [61]. Crucially however, the likelihood of local search algorithms arriving in such minima differs greatly between the A100 and V100. The proportion of centrality of an FFG gives us a tool to quantify this likelihood. However, further research is necessary to quantitatively determine how well our proposed metric correlates with GPU tuning difficulty.

As a final remark on kernel differences, the experimental results shows that the difficulty of tuning a particular kernel can greatly differ from one GPU to the next, and that these changes do not appear to be correlated with release time of the models. The A100 is the most recent GPU in our set, while the K20 is the oldest. For GEMM and PnPoly, we can say that it has become easier to tune these kernels with more recent GPUs, but the convolution kernel has become more difficult to tune, except on the V100.

VIII. Conclusion

In this paper, we have investigated which optimization algorithms produce the fastest GPU kernel configurations across different tuning-time ranges. To do so, we analyzed 26 GPU kernel spaces for 9 GPUs. We computed sets of optimal hyperparameters for GPU tuning for each optimization algorithm. From among the tested algorithms in this set of experiments, we conclude that dual annealing performs best as GPU kernel tuner when a limited amount of function evaluations is desirable. When more evaluations are possible, first-improvement local searchers such as FirstILS proved the best GPU kernel tuners. Using these algorithms, we are convinced that GPU programmers can reliably auto-tune GPU kernels to close to optimal runtime while requiring relatively few re-compilations of the code. Furthermore, we conclude that

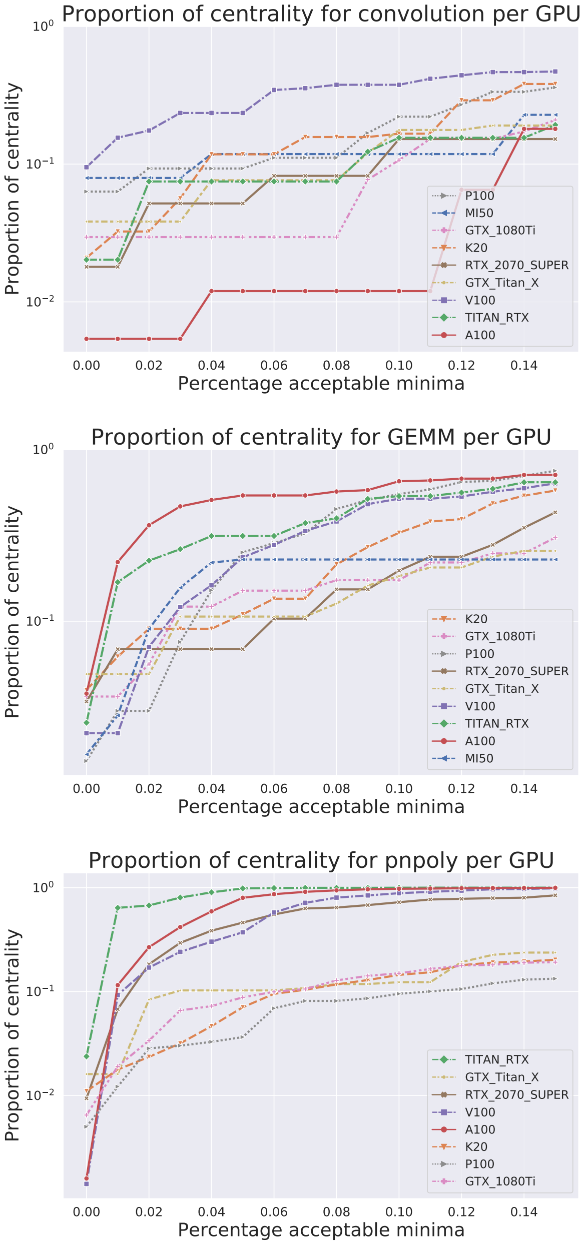


Fig. 7: Proportion of centrality for fitness flow graphs for each GPU. The proportion of centrality is computed by taking the sum of PageRank centrality for local minima within $p\%$ of the optimal fitness, divided by the total PageRank centrality of all local minima. From top to bottom are convolution, GEMM, and PnPoly kernel.

treating GPU tuning as a deterministic optimization problem is preferred over treating the runtime as a stochastic variable.

We showed that the basic metric of fraction of optimality of local minima is not suitable for explaining the results observed in the experimental benchmarks. To make steps towards a metric for tuning difficulty, we introduced the concept of fitness flow graphs, and proportion of centrality. Our results suggest that the proportion of centrality can be used to quantify tuning difficulty. For future work, in cases where exhaustive exploration is infeasible, perhaps a procedure to dynamically update the proportion of centrality of FFGs can be used. Such dynamic estimates of tuning difficulty could be used for automatic algorithm selection within frameworks such as Kernel Tuner. Furthermore, the pagerank centrality of strong local minima within FFGs can be used to investigate why certain minima are unlikely to be visited, for example because neighbouring configurations fail to compile. Lastly, in this work we fully computed 26 kernel spaces, and made these publicly available. We aim to extend this to a benchmark dataset for evolutionary computation algorithms.

Acknowledgements

This work has made use of the experimental systems on the Dutch national e-infrastructure with the support of the SURF Cooperative. The CORTEX project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call (file number NWA.1160.18.316). This work is also financially supported by the Netherlands Organization for Scientific Research (NWO), project number 639.073.506.

References

[1] J. Ansel, S. Kamil, K. Veeramachaneni, et al. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014.

[2] A. H. Ashouri, W. Killian, J. Cavazos, et al. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), 2018.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[4] R. H. Byrd, P. Lu, J. Nocedal, et al. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

[5] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. Natsbench: Benchmarking nas algorithms for architecture topology and size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3634–3646, 2022.

[6] J. Filipovič, J. Hozzová, A. Nezarat, et al. Using hardware performance counters to speed up autotuning convergence on GPUs. *arXiv preprint arXiv:2102.05297*, 2021.

[7] 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.

[8] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[9] F. Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[10] R. Goncalves, T. van Tilburg, K. Kyzirakos, et al. 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*, GIS '16, pages 80:1–80:4, New York, NY, USA, 2016. ACM.

[11] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee, 2012.

[12] D. Grewe and A. Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 12. ACM, 2011.

[13] S. Heldens, P. Hijma, B. van Werkhoven, et al. The landscape of exascale research: a data-driven literature analysis. *ACM Computing Surveys (CSUR)*, 53(2):1–43, 2020.

[14] S. Herrmann. Determining the difficulty of landscapes by PageRank centrality in local optima networks. In *Evolutionary Computation in Combinatorial Optimization*, pages 74–87. Springer, 2016.

[15] S. Herrmann and F. Rothlauf. Predicting heuristic search performance with PageRank centrality in local optima networks. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 401–408, 2015.

[16] Kaixi Hou, Wu-chun Feng, and Shuai Che. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi- and many-core processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 713–722. IEEE, 2017.

[17] H. Ishibuchi and T. Murata. A multi-objective genetic local search algorithm and its application to flowshop scheduling. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 28(3):392–403, 1998.

[18] S. Kamil, C. Chan, L. Oliker, et al. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE international symposium on parallel & distributed processing (IPDPS)*, pages 1–12. IEEE, 2010.

[19] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[20] Nesrine Khouzami, Friedrich Michel, Pietro Incardona, Jeronimo Castrillon, and Ivo F. Sbalzarini. Model-based autotuning of discretization methods in numerical simulations of partial differential equations. *Journal of Computational Science*, 57:101489, 2022.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[22] D. Kraft. *A software package for sequential quadratic programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

[23] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[24] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science*, pages 884–892. Springer, 2009.

[25] R. Lim, B. Norris, and A. Malony. Autotuning GPU kernels via static and predictive analysis. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 523–532. IEEE, 2017.

[26] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, René Sass, and Frank Hutter. Sequential Model Algorithm Configuration (SMAC). https://github.com/automl/SMAC3, 2021.

[27] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization, 2021.

[28] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[29] Y. Liu, W. M. Sid-Lakhdar, O. Marques, et al. GPTune: Multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 234–246, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[31] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[32] Manuel López-Ibáñez, Leslie Pérez Cáceres, and Jérémie Dubois-Lacoste. irace: Iterated Racing for Automatic Algorithm Configuration. https://github.com/MLopez-Ibanez/irace, 2021.

[33] A. Mametjanov, D. Lowell, C. Ma, et al. Autotuning stencil-based computations on GPUs. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 266–274. IEEE, 2012.

[34] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[35] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 97:428–442, 2019.

[36] L. Nardi, A. Souza, D. Koeplinger, et al. Hypermapper: a practical design space exploration framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019.

[37] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[38] J. Nocedal and S. J. Wright. Conjugate gradient methods. *Numerical optimization*, pages 101–134, 2006.

[39] C. Nugteren. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, pages 5:1–5:10. ACM, 2018.

[40] 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*, pages 195–202. IEEE, 2015.

[41] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 30. ACM, 2009.

[42] G. Ochoa, M. Tomassini, S. Vérel, et al. A study of NK landscapes' basins and local optima networks. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 555–562, 2008.

[43] L. Page, S. Brin, R. Motwani, et al. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[44] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. Srtuner: Effective compiler optimization customization by exposing synergistic relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 118–130, 2022.

[45] F. Petrovič, D. Střelák, J. Hozzová, et al. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Gener. Comput. Syst.*, 108:161–177, 2020.

[46] M. J. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

[47] M. J. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*, pages 51–67. Springer, 1994.

[48] M. Puschel, J. M. Moura, J. R. Johnson, et al. SPIRAL: Code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005.

[49] A. Rasch, R. Schulze, M. Steuwer, et al. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.*, 18(1), 2021.

[50] S. Ryoo, C. I. Rodrigues, S. S. Stone, et al. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[51] Richard A. Schoonhoven. BlooPy: Black-box optimization Python for bitstring, categorical, and numerical discrete problems with local, and population-based algorithms. https://github.com/schoonhovenrichard/BlooPy, 2021.

[52] Richard A. Schoonhoven. Data and plotting scripts for GPU benchmarking 2021 paper. https://github.com/schoonhovenrichard/GPU_benchmarking_paper, 2021.

[53] A. Sclocco, H. E. Bal, J. Hessels, et al. Auto-tuning dedispersion for many-core accelerators. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 952–961. IEEE, 2014.

[54] A. Sclocco, S. Heldens, and B. van Werkhoven. AMBER: A real-time pipeline for the detection of single pulse astronomical transients. *SoftwareX*, 12:100549, 2020.

[55] A. Sclocco, J. Van Leeuwen, H. E. Bal, et al. A real-time radio transient pipeline for arts. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 468–472. IEEE, 2015.

[56] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[57] A. Tiwari, C. Chen, J. Chame, et al. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[58] S. Tomov, R. Nath, H. Ltaief, et al. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[59] Top500. List–november 2020. https://top500.org/lists/top500/2020/11/, (accessed November 27, 2020).

[60] C. Tsallis and D. A. Stariolo. Generalized simulated annealing. *Physica A*, 233(1-2):395–406, 1996.

[61] B. van Werkhoven. Kernel Tuner: A search-optimizing GPU code autotuner. *Future Gener. Comput. Syst.*, 90, 2018.

[62] B. Van Werkhoven, J. Maassen, H. E. Bal, et al. Optimizing convolution operations on gpus using adaptive tiling. *Future Gener. Comput. Syst.*, 30:14–26, 2014.

[63] B. van Werkhoven, W. J. Palenstijn, and A. Sclocco. Lessons learned in a decade of research software engineering gpu applications. In *International Conference on Computational Science*, pages 399–412. Springer, 2020.

[64] D. J. Wales and J. P. Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

[65] Zheng Wang, Dominik Grewe, and Michael FP O'boyle. Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2014.

[66] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[67] A. H. Wright, R. K. Thompson, and J. Zhang. The computational complexity of N-K fitness functions. *Trans. Evol. Comp*, 4(4):373–379, 2000.

[68] Y. Zhang and F. Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164, 2012.

Kernel	parameter to tune	list of values	number of possible values
Convolution (except MI50)	`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
Convolution (MI50)	`block_size_x`	$\{16, 32, 48, 64, 80, 96, 112, 128\}$	8
	`block_size_y`	$\{1, 2, 4, 8, 16, 32\}$	6
	`tile_size_x`	$\{1, 2, 4\}$	3
	`tile_size_y`	$\{1, 2, 4\}$	3
	`use_padding`	$\{0, 1\}$	2
GEMM	`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
Point-in-polygon	`block_size_x`	$\{32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448, 480, 512, 544, 576, 608, 640, 672, 704, 736, 768, 800, 832, 864, 896, 928, 960, 992\}$	31
	`tile_size`	$\{1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$	11
	`between_method`	$\{0, 1, 2, 3\}$	4
	`use_precomputed_slopes`	$\{0, 1\}$	2
	`use_method`	$\{0, 1, 2\}$	3

TABLE V: Tunable parameters per kernel, and list of possible values for each parameter.

IX. Appendix

A. Appendix: Tunable parameters per GPU kernel

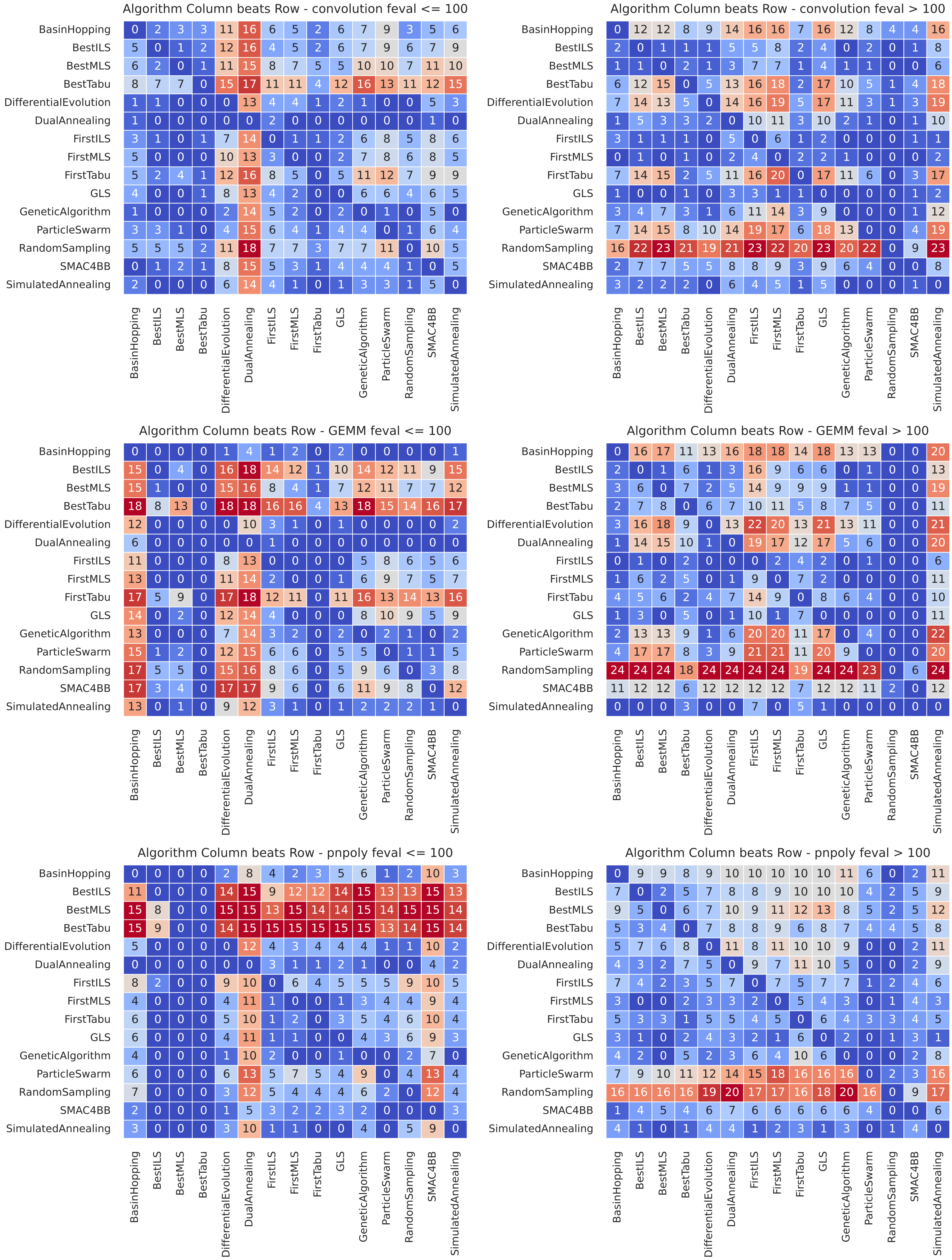
In Table V we show the tunable parameters per kernel, and the values each parameter could take. For the convolution kernel, the MI50 GPU (the only AMD model) required a different problem setup due to hardware constraints.

B. Appendix: Alternative splits for competition heatmaps

In Figures 8 and 9 we show the algorithm competition heatmaps such as in Figure 1 but when split at 100 and 400 budgets instead of 200.

C. Appendix: Per kernel graphs of experimental results

In Figures 10 to 18 we show plots of algorithm performance in terms of fraction of optimal fitness found for certain budget used (per GPU).



Algorithm Column beats Row - convolution feval <= 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	2	3	3	11	16	6	5	2	6	7	9	3	5	6
BestILS	5	0	1	2	12	16	4	5	2	6	7	9	6	7	9
BestMLS	6	2	0	1	11	15	8	7	5	5	10	10	7	11	10
BestTabu	8	7	7	0	15	17	11	11	4	12	16	13	11	12	15
DifferentialEvolution	1	1	0	0	0	13	4	4	1	2	1	0	0	5	3
DualAnnealing	1	0	0	0	0	0	2	0	0	0	0	0	0	1	0
FirstILS	3	1	0	1	7	14	0	1	1	2	6	8	5	8	6
FirstMLS	5	0	0	0	10	13	3	0	0	2	7	8	6	8	5
FirstTabu	5	2	4	1	12	16	8	5	0	5	11	12	7	9	9
GLS	4	0	0	1	8	13	4	2	0	0	6	6	4	6	5
GeneticAlgorithm	1	0	0	0	2	14	5	2	0	2	0	1	0	5	0
ParticleSwarm	3	3	1	0	4	15	6	4	1	4	4	0	1	6	4
RandomSampling	5	5	5	2	11	18	7	7	3	7	7	11	0	10	5
SMAC4BB	0	1	2	1	8	15	5	3	1	4	4	4	1	0	5
SimulatedAnnealing	2	0	0	0	6	14	4	1	0	1	3	3	1	5	0

Algorithm Column beats Row - convolution feval > 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	12	12	8	9	14	16	16	7	16	12	8	4	4	16
BestILS	2	0	1	1	1	5	5	8	2	4	0	2	0	1	8
BestMLS	1	1	0	2	1	3	7	7	1	4	1	1	0	0	6
BestTabu	6	12	15	0	5	13	16	18	2	17	10	5	1	4	18
DifferentialEvolution	7	14	13	5	0	14	16	19	5	17	11	3	1	3	19
DualAnnealing	1	5	3	3	2	0	10	11	3	10	2	1	0	1	10
FirstILS	3	1	1	1	0	5	0	6	1	2	0	0	0	1	1
FirstMLS	0	1	0	1	0	2	4	0	2	2	1	0	0	0	2
FirstTabu	7	14	15	2	5	11	16	20	0	17	11	6	0	3	17
GLS	1	0	0	1	0	3	3	1	1	0	0	0	0	1	2
GeneticAlgorithm	3	4	7	3	1	6	11	14	3	9	0	0	0	1	12
ParticleSwarm	7	14	15	8	10	14	19	17	6	18	13	0	0	4	19
RandomSampling	16	22	23	21	19	21	23	22	20	23	20	22	0	9	23
SMAC4BB	2	7	7	5	5	8	8	9	3	9	6	4	0	0	8
SimulatedAnnealing	3	2	2	2	0	6	4	5	1	5	0	0	0	1	0

Algorithm Column beats Row - GEMM feval <= 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	0	0	0	1	4	1	2	0	2	0	0	0	0	1
BestILS	15	0	4	0	16	18	14	12	1	10	14	12	11	9	15
BestMLS	15	1	0	0	15	16	8	4	1	7	12	11	7	7	12
BestTabu	18	8	13	0	18	18	16	16	4	13	18	15	14	16	17
DifferentialEvolution	12	0	0	0	0	10	3	1	0	1	0	0	0	0	2
DualAnnealing	6	0	0	0	0	0	1	0	0	0	0	0	0	0	0
FirstILS	11	0	0	0	8	13	0	0	0	0	5	8	6	5	6
FirstMLS	13	0	0	0	11	14	2	0	0	1	6	9	7	5	7
FirstTabu	17	5	9	0	17	18	12	11	0	11	16	13	14	13	16
GLS	14	0	2	0	12	14	4	0	0	0	8	10	9	5	9
GeneticAlgorithm	13	0	0	0	7	14	3	2	0	2	0	2	1	0	2
ParticleSwarm	15	1	2	0	12	15	6	6	0	5	5	0	1	1	5
RandomSampling	17	5	5	0	15	16	8	6	0	5	9	6	0	3	8
SMAC4BB	17	3	4	0	17	17	9	6	0	6	11	9	8	0	12
SimulatedAnnealing	13	0	1	0	9	12	3	1	0	1	2	2	2	1	0

Algorithm Column beats Row - GEMM feval > 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	16	17	11	13	16	18	18	14	18	13	13	0	0	20
BestILS	2	0	1	6	1	3	16	9	6	6	0	1	0	0	13
BestMLS	3	6	0	7	2	5	14	9	9	9	1	1	0	0	19
BestTabu	2	7	8	0	6	7	10	11	5	8	7	5	0	0	11
DifferentialEvolution	3	16	18	9	0	13	22	20	13	21	13	11	0	0	21
DualAnnealing	1	14	15	10	1	0	19	17	12	17	5	6	0	0	20
FirstILS	0	1	0	2	0	0	0	2	4	2	0	1	0	0	6
FirstMLS	1	6	2	5	0	1	9	0	7	2	0	0	0	0	11
FirstTabu	4	5	6	2	4	7	14	9	0	8	6	4	0	0	10
GLS	1	3	0	5	0	1	10	1	7	0	0	0	0	0	11
GeneticAlgorithm	2	13	13	9	1	6	20	20	11	17	0	4	0	0	22
ParticleSwarm	4	17	17	8	3	9	21	21	11	20	9	0	0	0	20
RandomSampling	24	24	24	18	24	24	24	24	19	24	24	23	0	6	24
SMAC4BB	11	12	12	6	12	12	12	12	7	12	12	11	2	0	12
SimulatedAnnealing	0	0	0	3	0	0	7	0	5	1	0	0	0	0	0

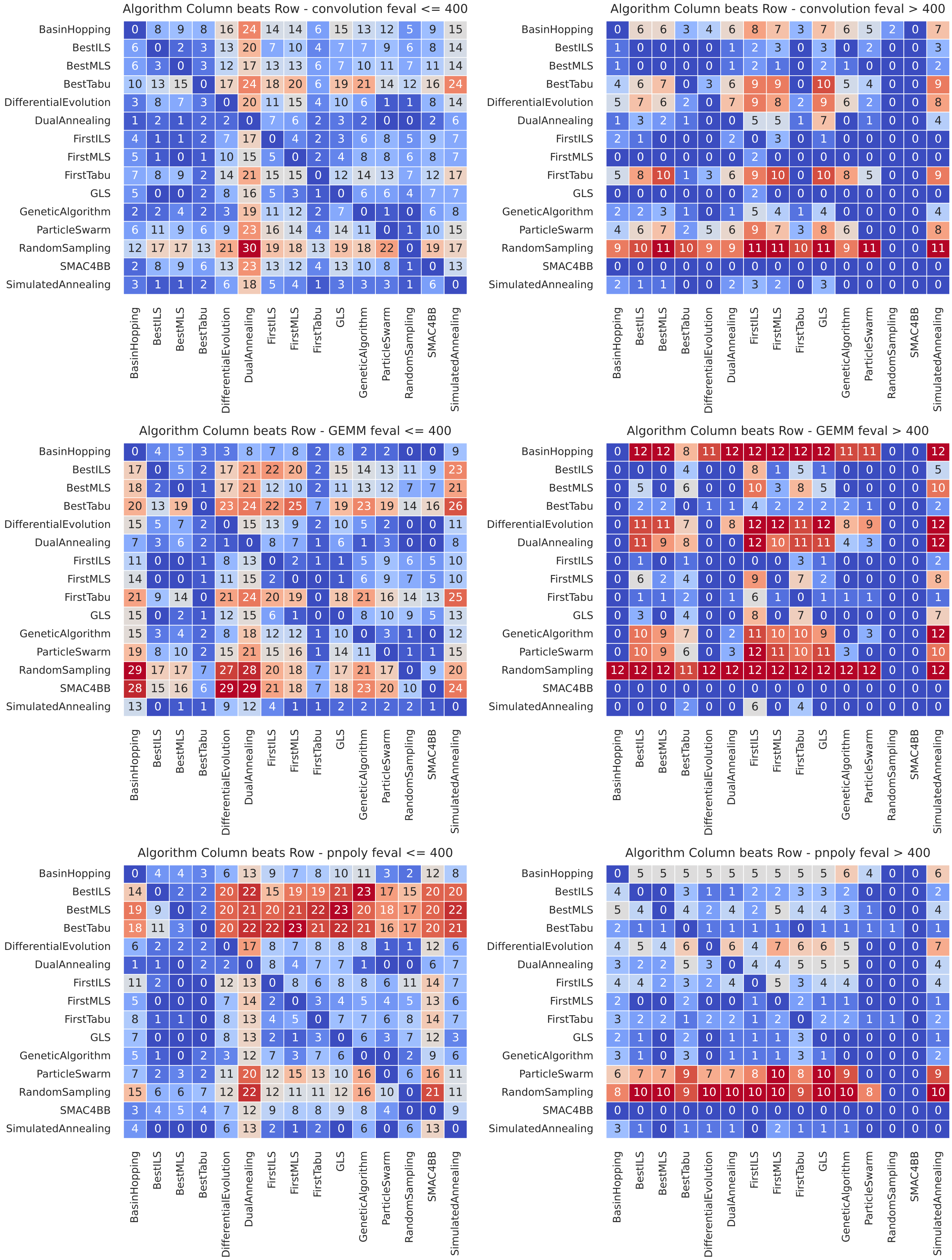
Algorithm Column beats Row - pnpoly feval <= 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	0	0	0	2	8	4	2	3	5	6	1	2	10	3
BestILS	11	0	0	0	14	15	9	12	12	14	15	13	13	15	13
BestMLS	15	8	0	0	15	15	13	15	14	14	15	14	15	15	14
BestTabu	15	9	0	0	14	15	15	15	15	15	15	13	14	15	14
DifferentialEvolution	5	0	0	0	0	12	4	3	4	4	4	1	1	10	2
DualAnnealing	0	0	0	0	0	0	3	1	1	2	1	0	0	4	2
FirstILS	8	2	0	0	9	10	0	6	4	5	5	5	9	10	5
FirstMLS	4	0	0	0	4	11	1	0	0	1	3	4	4	9	4
FirstTabu	6	0	0	0	5	10	1	2	0	3	5	4	6	10	4
GLS	6	0	0	0	4	11	1	1	0	0	4	3	6	9	3
GeneticAlgorithm	4	0	0	0	1	10	2	0	0	1	0	0	2	7	0
ParticleSwarm	6	0	0	0	6	13	5	7	5	4	9	0	4	13	4
RandomSampling	7	0	0	0	3	12	5	4	4	4	6	2	0	12	4
SMAC4BB	2	0	0	0	1	5	3	2	2	3	2	0	0	0	3
SimulatedAnnealing	3	0	0	0	3	10	1	1	0	0	4	0	5	9	0

Algorithm Column beats Row - pnpoly feval > 100

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	9	9	8	9	10	10	10	10	10	11	6	0	2	11
BestILS	7	0	2	5	7	8	8	9	10	10	10	4	2	5	9
BestMLS	9	5	0	6	7	10	9	11	12	13	8	5	2	5	12
BestTabu	5	3	4	0	7	8	8	9	6	8	7	4	4	5	8
DifferentialEvolution	5	7	6	8	0	11	8	11	10	10	9	0	0	2	11
DualAnnealing	4	3	2	7	5	0	9	7	11	10	5	0	0	2	9
FirstILS	7	4	2	3	5	7	0	7	5	7	7	1	2	4	6
FirstMLS	3	0	0	2	3	3	2	0	5	4	3	0	1	4	3
FirstTabu	5	3	3	1	5	5	4	5	0	6	4	3	3	4	5
GLS	3	1	0	2	4	3	2	1	6	0	2	0	1	3	1
GeneticAlgorithm	4	2	0	5	2	3	6	4	10	6	0	0	0	2	8
ParticleSwarm	7	9	10	11	12	14	15	18	16	16	16	0	2	3	16
RandomSampling	16	16	16	16	19	20	17	17	16	18	20	16	0	9	17
SMAC4BB	1	4	5	4	6	7	6	6	6	6	6	4	0	0	6
SimulatedAnnealing	4	1	0	1	4	4	1	2	3	1	3	0	1	4	0

Fig. 8: Heatmaps counting the occurrences when the column algorithm found statistically better solutions than the row algorithm for the (top) convolution, (middle) GEMM, and (botttom) PnPoly kernels. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Left): Heatmap for low ≤ 100 budgets, i.e., 25, 50, and 100. (Right): Heatmap for mid and high > 100 budgets, i.e., 200, 400, 800, 1600. Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.



Algorithm Column beats Row - convolution feval <= 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	8	9	8	16	24	14	14	6	15	13	12	5	9	15
BestILS	6	0	2	3	13	20	7	10	4	7	7	9	6	8	14
BestMLS	6	3	0	3	12	17	13	13	6	7	10	11	7	11	14
BestTabu	10	13	15	0	17	24	18	20	6	19	21	14	12	16	24
DifferentialEvolution	3	8	7	3	0	20	11	15	4	10	6	1	1	8	14
DualAnnealing	1	2	1	2	2	0	7	6	2	3	2	0	0	2	6
FirstILS	4	1	1	2	7	17	0	4	2	3	6	8	5	9	7
FirstMLS	5	1	0	1	10	15	5	0	2	4	8	8	6	8	7
FirstTabu	7	8	9	2	14	21	15	15	0	12	14	13	7	12	17
GLS	5	0	0	2	8	16	5	3	1	0	6	6	4	7	7
GeneticAlgorithm	2	2	4	2	3	19	11	12	2	7	0	1	0	6	8
ParticleSwarm	6	11	9	6	9	23	16	14	4	14	11	0	1	10	15
RandomSampling	12	17	17	13	21	30	19	18	13	19	18	22	0	19	17
SMAC4BB	2	8	9	6	13	23	13	12	4	13	10	8	1	0	13
SimulatedAnnealing	3	1	1	2	6	18	5	4	1	3	3	3	1	6	0

Algorithm Column beats Row - convolution feval > 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	6	6	3	4	6	8	7	3	7	6	5	2	0	7
BestILS	1	0	0	0	0	1	2	3	0	3	0	2	0	0	3
BestMLS	1	0	0	0	0	1	2	1	0	2	1	0	0	0	2
BestTabu	4	6	7	0	3	6	9	9	0	10	5	4	0	0	9
DifferentialEvolution	5	7	6	2	0	7	9	8	2	9	6	2	0	0	8
DualAnnealing	1	3	2	1	0	0	5	5	1	7	0	1	0	0	4
FirstILS	2	1	0	0	0	2	0	3	0	1	0	0	0	0	0
FirstMLS	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
FirstTabu	5	8	10	1	3	6	9	10	0	10	8	5	0	0	9
GLS	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
GeneticAlgorithm	2	2	3	1	0	1	5	4	1	4	0	0	0	0	4
ParticleSwarm	4	6	7	2	5	6	9	7	3	8	6	0	0	0	8
RandomSampling	9	10	11	10	9	9	11	11	10	11	9	11	0	0	11
SMAC4BB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SimulatedAnnealing	2	1	1	0	0	2	3	2	0	3	0	0	0	0	0

Algorithm Column beats Row - GEMM feval <= 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	4	5	3	3	8	7	8	2	8	2	2	0	0	9
BestILS	17	0	5	2	17	21	22	20	2	15	14	13	11	9	23
BestMLS	18	2	0	1	17	21	12	10	2	11	13	12	7	7	21
BestTabu	20	13	19	0	23	24	22	25	7	19	23	19	14	16	26
DifferentialEvolution	15	5	7	2	0	15	13	9	2	10	5	2	0	0	11
DualAnnealing	7	3	6	2	1	0	8	7	1	6	1	3	0	0	8
FirstILS	11	0	0	1	8	13	0	2	1	1	5	9	6	5	10
FirstMLS	14	0	0	1	11	15	2	0	0	1	6	9	7	5	10
FirstTabu	21	9	14	0	21	24	20	19	0	18	21	16	14	13	25
GLS	15	0	2	1	12	15	6	1	0	0	8	10	9	5	13
GeneticAlgorithm	15	3	4	2	8	18	12	12	1	10	0	3	1	0	12
ParticleSwarm	19	8	10	2	15	21	15	16	1	14	11	0	1	1	15
RandomSampling	29	17	17	7	27	28	20	18	7	17	21	17	0	9	20
SMAC4BB	28	15	16	6	29	29	21	18	7	18	23	20	10	0	24
SimulatedAnnealing	13	0	1	1	9	12	4	1	1	2	2	2	2	1	0

Algorithm Column beats Row - GEMM feval > 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	12	12	8	11	12	12	12	12	12	11	11	0	0	12
BestILS	0	0	0	4	0	0	8	1	5	1	0	0	0	0	5
BestMLS	0	5	0	6	0	0	10	3	8	5	0	0	0	0	10
BestTabu	0	2	2	0	1	1	4	2	2	2	2	1	0	0	2
DifferentialEvolution	0	11	11	7	0	8	12	12	11	12	8	9	0	0	12
DualAnnealing	0	11	9	8	0	0	12	10	11	11	4	3	0	0	12
FirstILS	0	1	0	1	0	0	0	0	3	1	0	0	0	0	2
FirstMLS	0	6	2	4	0	0	9	0	7	2	0	0	0	0	8
FirstTabu	0	1	1	2	0	1	6	1	0	1	1	1	0	0	1
GLS	0	3	0	4	0	0	8	0	7	0	0	0	0	0	7
GeneticAlgorithm	0	10	9	7	0	2	11	10	10	9	0	3	0	0	12
ParticleSwarm	0	10	9	6	0	3	12	11	10	11	3	0	0	0	10
RandomSampling	12	12	12	11	12	12	12	12	12	12	12	12	0	0	12
SMAC4BB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SimulatedAnnealing	0	0	0	2	0	0	6	0	4	0	0	0	0	0	0

Algorithm Column beats Row - pnpoly feval <= 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	4	4	3	6	13	9	7	8	10	11	3	2	12	8
BestILS	14	0	2	2	20	22	15	19	19	21	23	17	15	20	20
BestMLS	19	9	0	2	20	21	20	21	22	23	20	18	17	20	22
BestTabu	18	11	3	0	20	22	22	23	21	22	21	16	17	20	21
DifferentialEvolution	6	2	2	2	0	17	8	7	8	8	8	1	1	12	6
DualAnnealing	1	1	0	2	2	0	8	4	7	7	1	0	0	6	7
FirstILS	11	2	0	0	12	13	0	8	6	8	8	6	11	14	7
FirstMLS	5	0	0	0	7	14	2	0	3	4	5	4	5	13	6
FirstTabu	8	1	1	0	8	13	4	5	0	7	7	6	8	14	7
GLS	7	0	0	0	8	13	2	1	3	0	6	3	7	12	3
GeneticAlgorithm	5	1	0	2	3	12	7	3	7	6	0	0	2	9	6
ParticleSwarm	7	2	3	2	11	20	12	15	13	10	16	0	6	16	11
RandomSampling	15	6	6	7	12	22	12	11	11	12	16	10	0	21	11
SMAC4BB	3	4	5	4	7	12	9	8	8	9	8	4	0	0	9
SimulatedAnnealing	4	0	0	0	6	13	2	1	2	0	6	0	6	13	0

Algorithm Column beats Row - pnpoly feval > 400

	BasinHopping	BestILS	BestMLS	BestTabu	DifferentialEvolution	DualAnnealing	FirstILS	FirstMLS	FirstTabu	GLS	GeneticAlgorithm	ParticleSwarm	RandomSampling	SMAC4BB	SimulatedAnnealing
BasinHopping	0	5	5	5	5	5	5	5	5	5	6	4	0	0	6
BestILS	4	0	0	3	1	1	2	2	3	3	2	0	0	0	2
BestMLS	5	4	0	4	2	4	2	5	4	4	3	1	0	0	4
BestTabu	2	1	1	0	1	1	1	1	0	1	1	1	1	0	1
DifferentialEvolution	4	5	4	6	0	6	4	7	6	6	5	0	0	0	7
DualAnnealing	3	2	2	5	3	0	4	4	5	5	5	0	0	0	4
FirstILS	4	4	2	3	2	4	0	5	3	4	4	0	0	0	4
FirstMLS	2	0	0	2	0	0	1	0	2	1	1	0	0	0	1
FirstTabu	3	2	2	1	2	2	1	2	0	2	2	1	1	0	2
GLS	2	1	0	2	0	1	1	1	3	0	0	0	0	0	1
GeneticAlgorithm	3	1	0	3	0	1	1	1	3	1	0	0	0	0	2
ParticleSwarm	6	7	7	9	7	7	8	10	8	10	9	0	0	0	9
RandomSampling	8	10	10	9	10	10	10	10	9	10	10	8	0	0	10
SMAC4BB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SimulatedAnnealing	3	1	0	1	1	1	0	2	1	1	1	0	0	0	0

Fig. 9: Heatmaps counting the occurrences when the column algorithm found statistically better solutions than the row algorithm for the (top) convolution, (middle) GEMM, and (botttom) PnPoly kernels. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Left): Heatmap for low ≤ 400 budgets, i.e., 25, 50, 100, 200, and 400. (Right): Heatmap for mid and high > 400 budgets, i.e., 800, 1600. Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

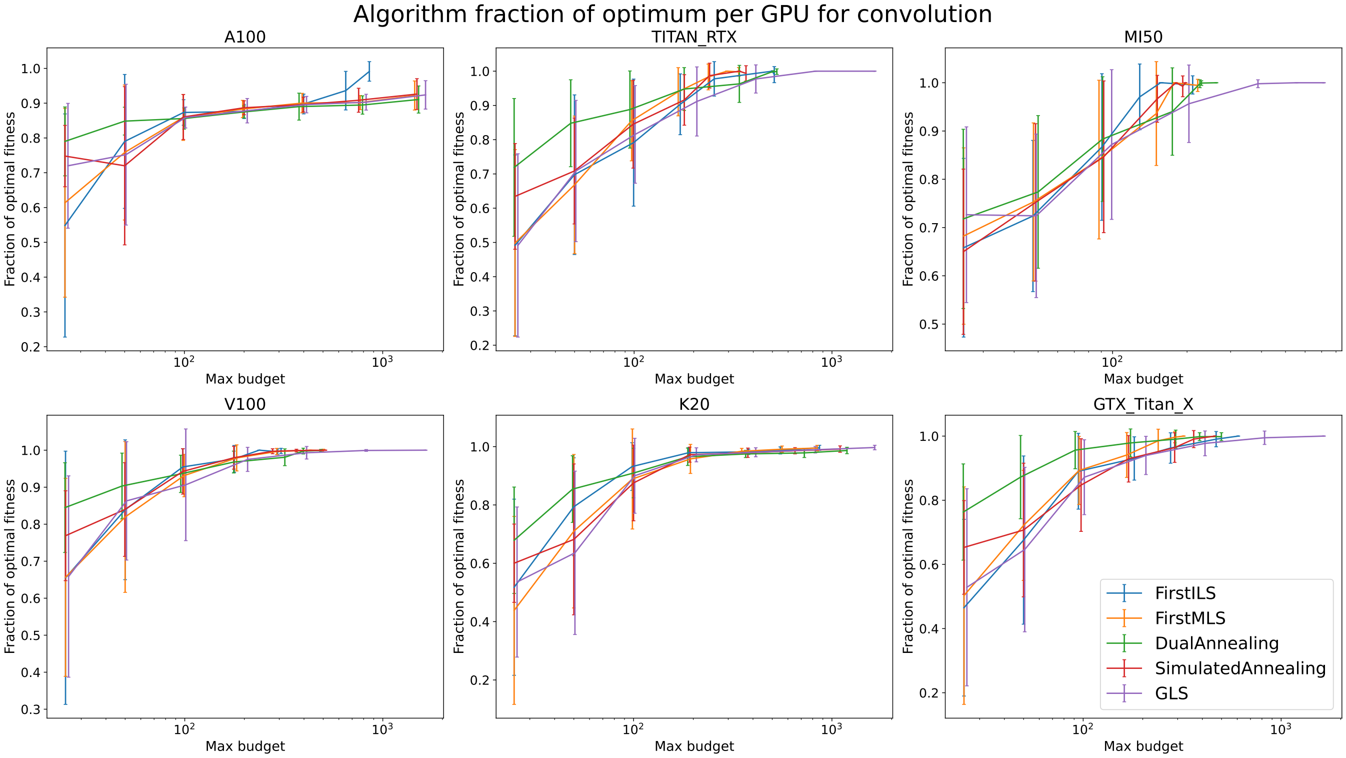


Fig. 10: **Convolution:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

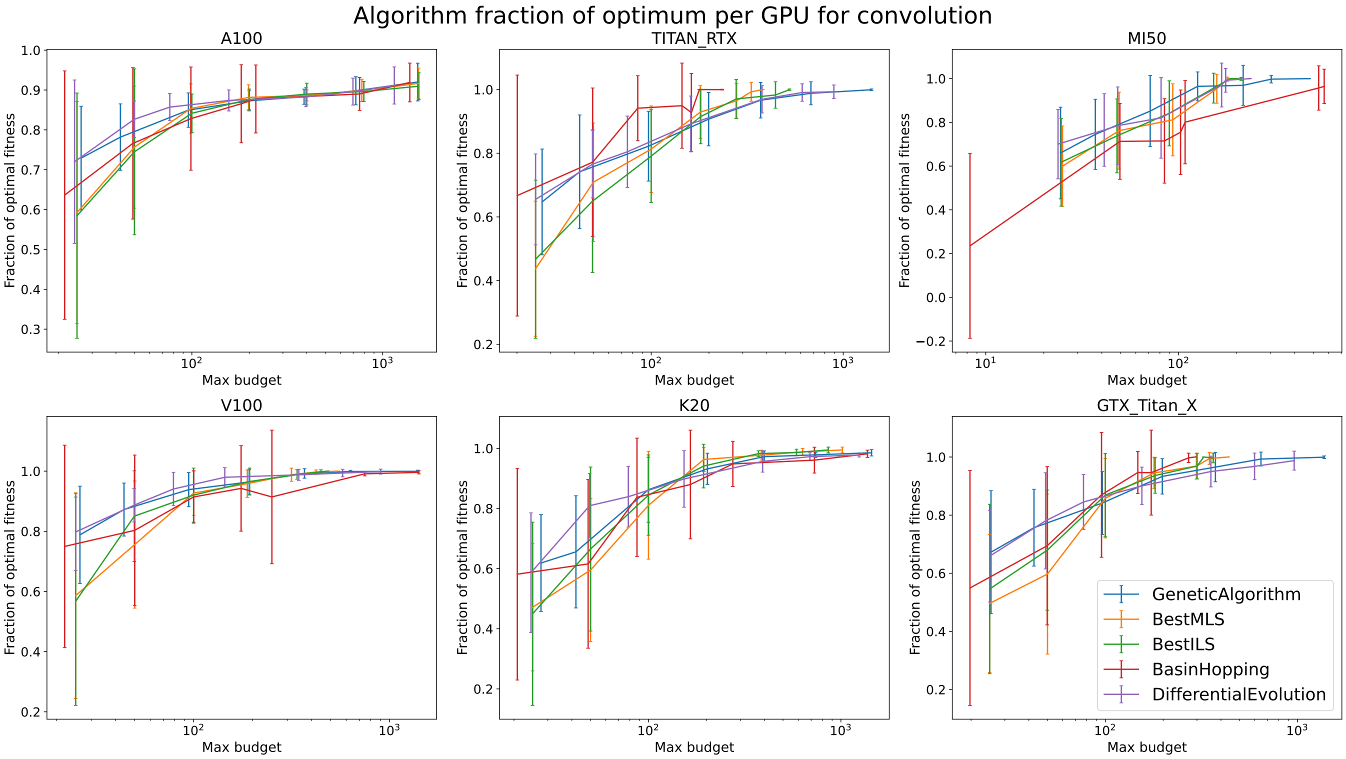


Fig. 11: **Convolution:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

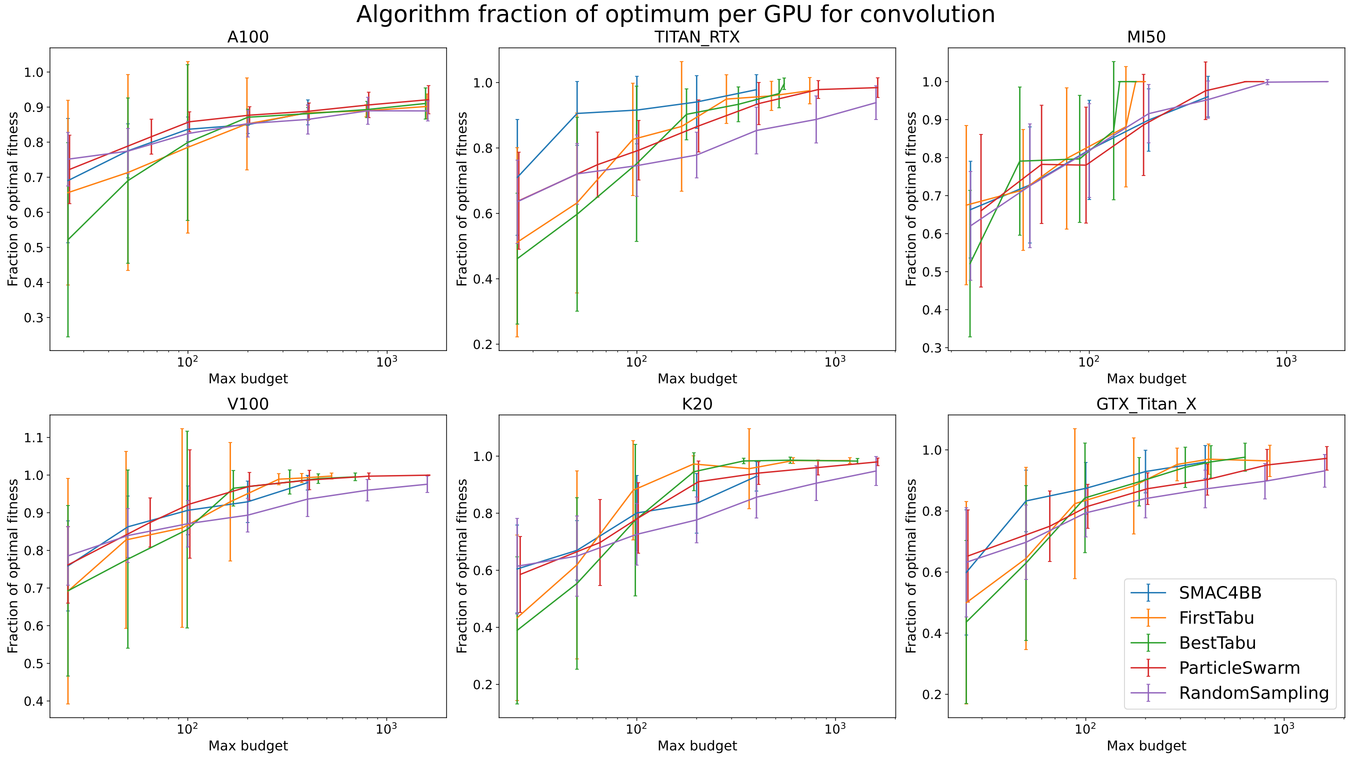


Fig. 12: **Convolution:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

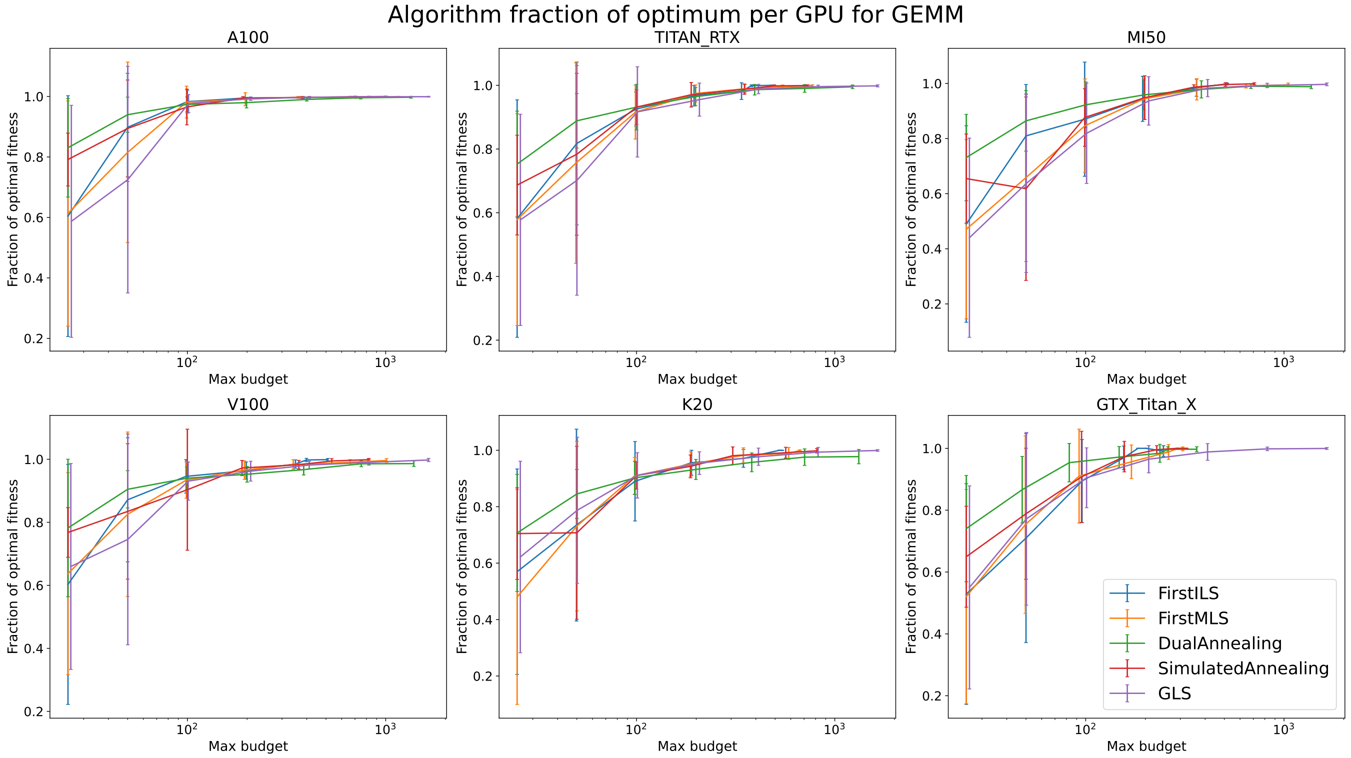


Fig. 13: **GEMM:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

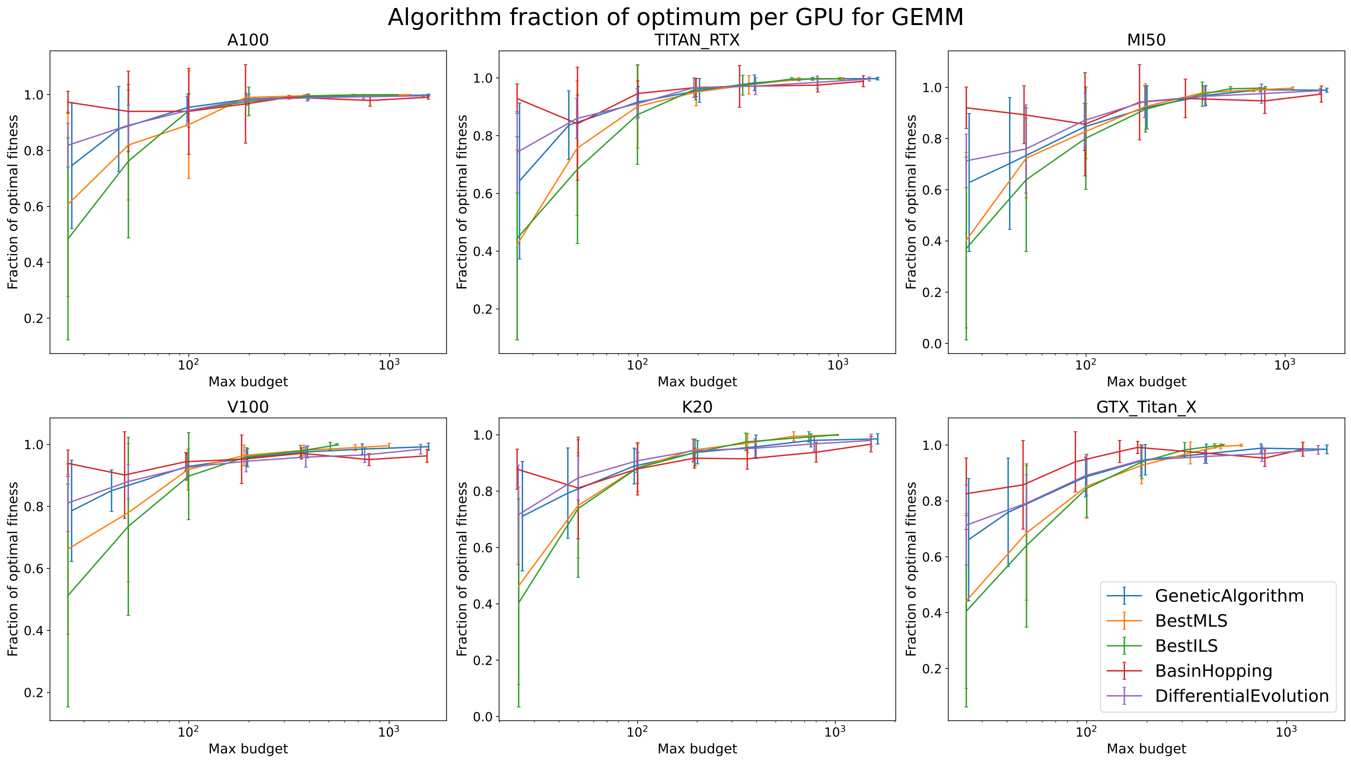


Fig. 14: **GEMM:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

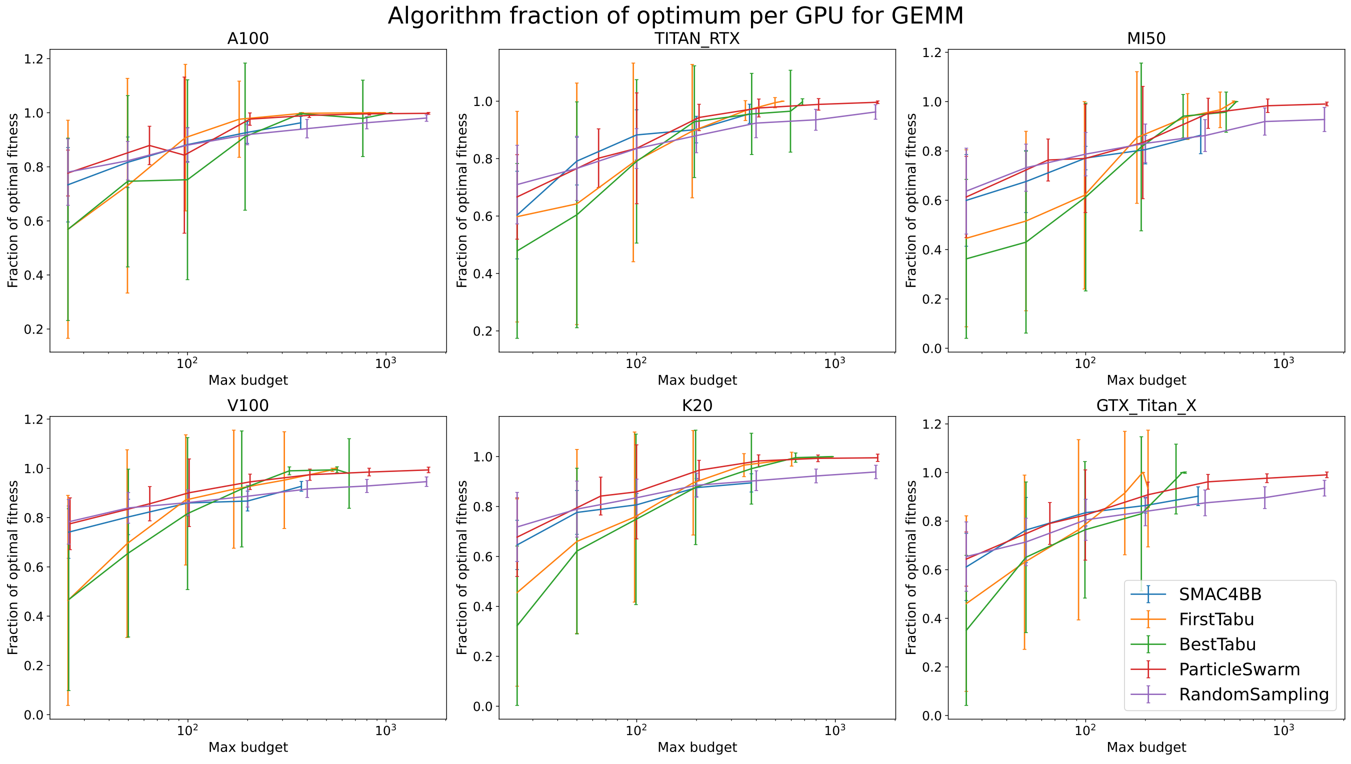


Fig. 15: **GEMM:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

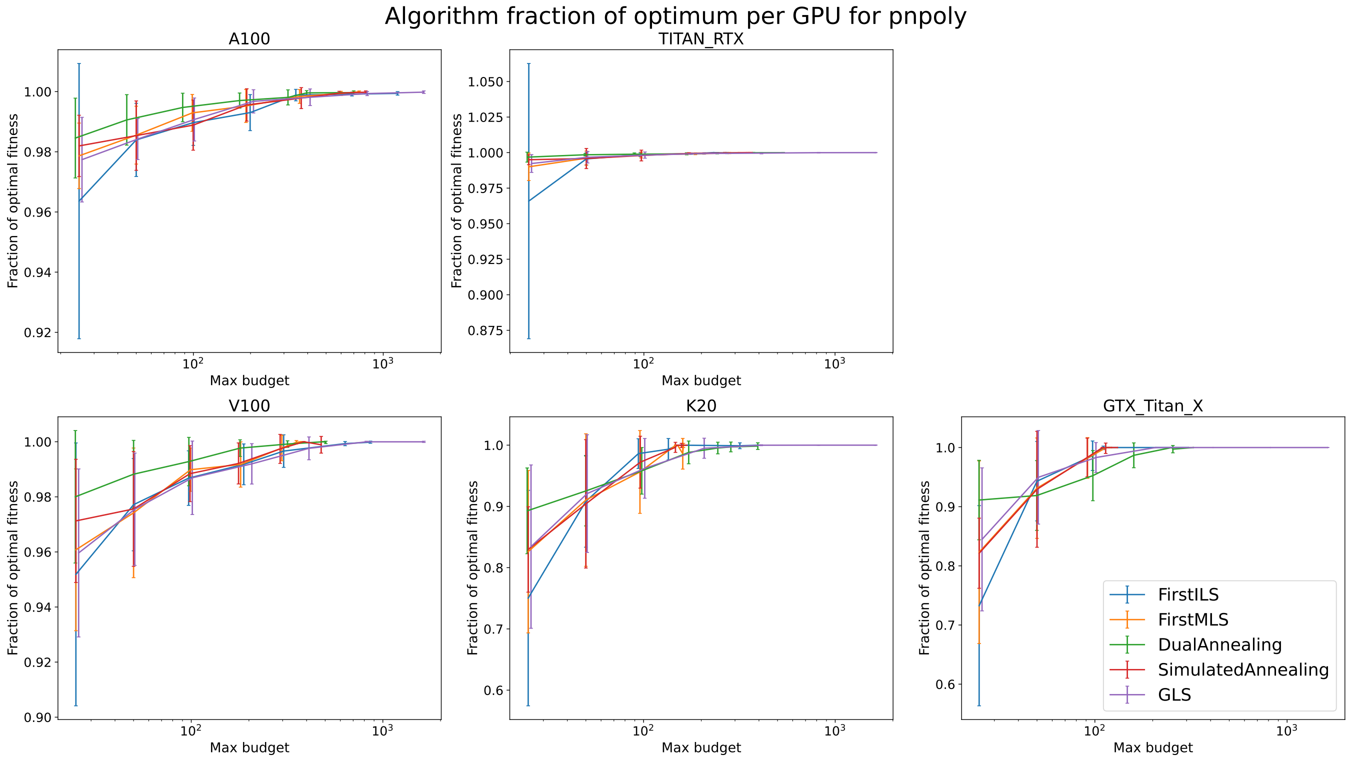


Fig. 16: **Point-in-polygon:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.

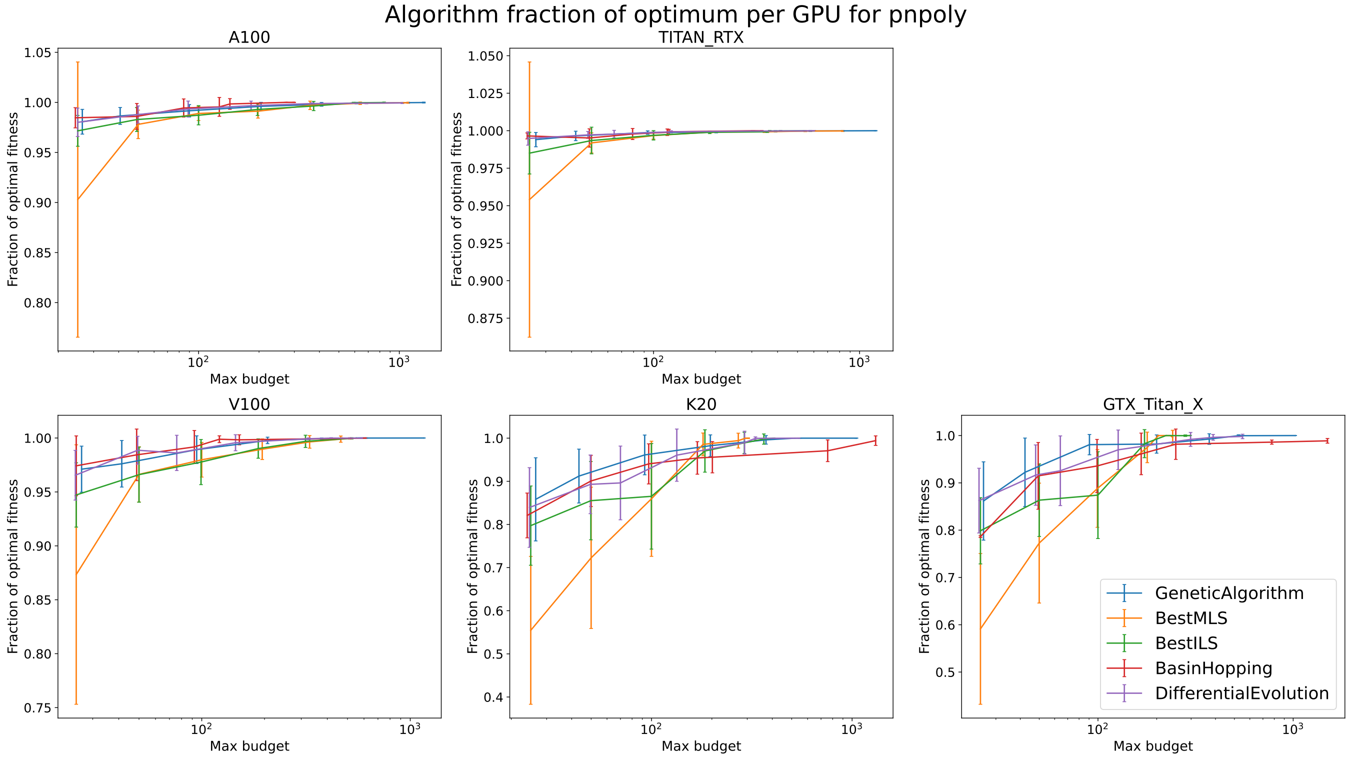


Fig. 17: **Point-in-polygon:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.

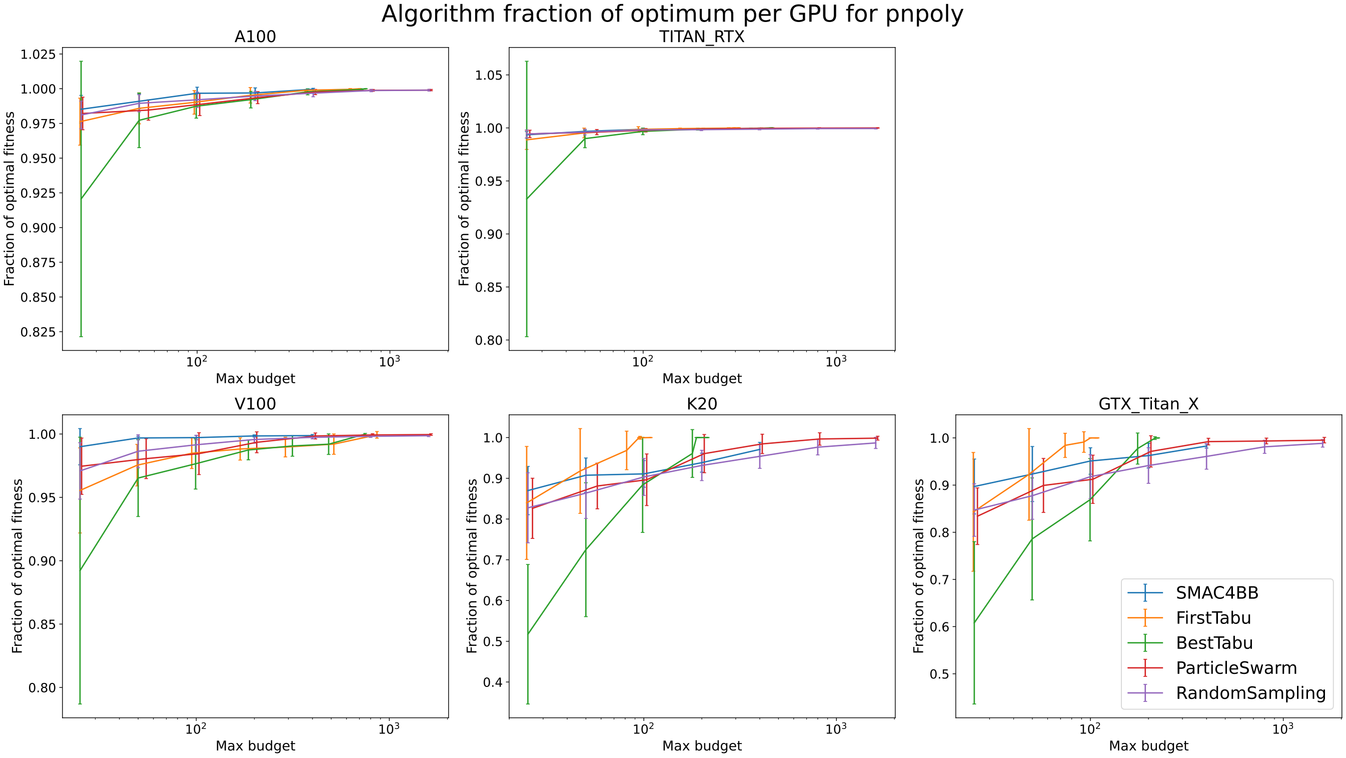


Fig. 18: **Point-in-polygon:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.