Universiteit Leiden
The Netherlands

Opleiding Informatica

Multi-objective optimization for auto-tuning GPU kernels

Maric A. Blommaert

Supervisors:
Ben van Werkhoven & Stijn Heldens

BACHELOR THESIS

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

28/01/2026

Abstract

In this thesis an extension to the auto-tuning framework Kernel Tuner is presented that makes it capable of tuning multiple objectives simultaneously. This makes it possible to obtain the configurations of the application being tuned that sit at trade-off points with respect to the selected objectives, which allows software engineers to make better informed decisions about the configurations that is most fitting for their situation. The extension makes this possible by integrating two Multi-Objective Evolutionary Algorithms (MOEAs), NSGA-II and NSGA-III, into the framework. The algorithms are tested on the pre-evaluated search spaces of nine mixed-precision GPU kernels using three and five objectives and their performance is compared to Random Search (RS) and each other. The results show that NSGA-II and NSGA-III are on average able to find solution sets that are up to 60% better than RS given the same evaluation budget and are able to achieve the same solution quality as RS in half the time (a 100% speedup) for evaluation budgets that allow them to have a sufficient number of generations.

1 Introduction

Graphics Processing Units (GPUs) have become an indispensable part of the High-Performance Computing (HPC) landscape and are the computational backbone of the Artificial Intelligence (AI) and Machine Learning (ML) revolution that is currently taking place. They have taken this position because GPUs are a type of hardware accelerator that offers massive parallel compute power relative to conventional CPUs while remaining application-agnostic. It is also exactly these characteristics that make GPUs so effective for the highly parallel workloads often found in HPC, AI, and ML applications.

The sections of application code that run on GPUs are called GPU kernels, or just kernels for short, and are difficult to implement such that they run efficiently on different input datasets and GPU architectures [1]. A kernel's performance, implementation details, and the architecture it is executed on are tightly coupled, which means that small changes in the implementation choices can generally drop its performance by an order of magnitude and similar effects can occur when the kernel is executed on another architecture [2]. This generally makes it difficult to implement kernels that have portable performance.

The factors that need to be considered during the implementation of each kernel are numerous, so there are many knobs that can tuned. The work-group dimensions the kernel is executed with is one of the main tuning parameters and has a wide range of possible values for each dimension with which it runs correctly, but for which the performance varies greatly. But there are many other parameters that need to be tuned, such as:

- the algorithms and data structures used for the computations
- the tile sizes used for loop tiling
- the factor by which loops are unrolled
- the arithmetic precision of the input, output, and intermediate values of the computations
- the layout of the data in memory

- the clock frequency of the processors

Each parameter has multiple possible values, making the search space very large. It is also non-contiguous because not every point in the parameter space corresponds to a valid implementation. These properties make it practically infeasible for a developer to manually tune the kernels, as such automatic performance tuning (auto-tuning) is necessary to achieve optimal and portable performance [1].

Another aspect that the developer needs to consider is the performance objectives of the kernel. The most familiar one is the total execution time, but there are other and they can be in conflict with each other. The energy related objectives are especially interesting because they are important to both HPC and low-energy embedded systems, so if a kernel can be implemented using less energy while maintaining about the same execution time than that is something developers in these areas would want to know. Another objective a developer might want to consider is the numerical accuracy of the computation. Lowering the accuracy would allow the use of smaller data types that have lower arithmetic precision. This would in turn make the data items smaller, making more items fit in the same memory space and allowing for more items to be transferred using the bandwidth. Together this would increase the arithmetic intensity and reduce the total execution time. It can also reduce the total amount of energy used because less data will have to be transferred overall. On the other hand, the reduction in accuracy also means the results have a larger error, so this trade-off is something some applications may be able to use and others will not. These examples show that performance tuning naturally has multiple objectives that developers may want and need to consider to get a clearer picture of the trade-offs available to them.

In the language of multi-objective optimization this means that the implementations of interest are those on the *Pareto front* with respect to the performance objectives we are interested in. That is, the implementations that are no worse than any other implementation with respect to every objective. These are also called *non-dominated*, or *Pareto optimal*.

The complexity of tuning GPU kernels combined with the fact that performance considerations have multiple, sometimes competing, objectives naturally leads to the research question of this thesis: *how do we implement multi-objective automatic performance tuning for GPUs using Kernel Tuner and do the selected algorithms do better than random search?*

In this thesis I will present an implementation of a multi-objective automatic performance tuner for GPU kernels. It is integrated into Kernel Tuner, a open-source auto-tuning tool [3]. This means the code is open-source and available to the public. The algorithms made available in Kernel Tuner are NSGA-II and NSGA-III and the solution sets they produce will the compared to random search to assess if these algorithms are a promising direction for further research on multi-objective auto-tuning.

Section 2 gives the required background on multi-objective optimization followed by a description of auto-tuning and finally an overview of Kernel Tuner. Section 3 provides an overview of different auto-tuning framework. Then comes Section 4 which starts by describing genetic algorithms, then NSGA-II and NSGA-III and finally a description of the implementation. Section 5 first describes the dataset that was used to obtain the results followed by the results themselves and a discussion of them. Finally Section 6 gives an overview of what has been done, states conclusions drawn from this work, and provides possible directions of future work.

2 Background

This section describes multi-objective optimization, auto-tuning, and Kernel Tuner. The multi-objective section will give a short overview of the topic which is required to properly understand the added challenges when compared to single-objective optimization. After that comes an overview of auto-tuning and finally Kernel Tuner, the framework that I extend in this thesis.

2.1 Multi-objective optimization

Multi-objective optimization (MO) is concerned with solving the optimization problems that have multiple objective functions that all need to be optimized simultaneously. This allows for the expression of optimization problems where there are conflicting goals without having to decide what objectives are more important than others before optimization can happen like would be the case with goal programming. The presence of conflicting goals generally makes it impossible for a single solution to be the best with respect to every objective. Instead there is a set of solutions that are optimal with respect to a subset of the objectives, but which cannot be improved for one objective without degrading it for another. These solutions correspond to the fundamental trade-offs that exist for a MO problem and is where a decision needs to be made about what is most important.

A multi-objective optimization problem (MOP) can be stated as:

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{f}(x) = [f_1(x), f_2(x), ..., f_n(x)] \\ \text{s.t.} \quad & x \ = (x_1, x_2, ..., x_m) \in X \end{aligned} \tag{1}$$

where

- n is the number of objectives and m the number of decision variables.
- $\boldsymbol{f}(x) : X \to Y \subseteq \mathbf{R}^n$, the vector-valued objective function that is formed by the combination of the objectives. It maps feasible solutions to their objective vector in the objective space, Y.
- $X \subseteq X^+ = \prod_{i=1}^{m} X_i$, the feasible set, which is the subset of the Cartesian product of the domains of the decision variables, X^+, that only contains the feasible solutions. The feasibility of a solution depends on the constraints imposed on the problem.

What follows are some definitions that are required to talk multi-objective optimization:

Given two feasible solutions $x_1, x_2 \in X$, x_1 **dominates** x_2, written $x_1 \prec x_2$, iff is at least as good as x_2 in every objective and there is at least one objective in which x_1 is better than x_2. Another way of saying that x_1 dominates x_2 is to say that x_1 is a **Pareto improvement** of x_2.

Given a subset of the feasible set $X' \subseteq X$ and a solution in this subset $x \in X'$, x_1 is **non-dominated** in X' iff there does not exist any other solution in X' that dominates x. A **Pareto optimal solution** is a solution that is non-dominated in the entire feasible set, such solutions are also said to be **Pareto efficient**.

Continuing with the set X', the **non-dominated set** X^* of X' is the set of non-dominated solutions in X'. The **Pareto set** is the non-dominated set of the entire

feasible set, which is the same as the set of Pareto optimal solutions. The **Pareto front** is the image of the Pareto set in the objective space.

The Pareto set and Pareto front of a MOP can now be stated as:

$$X^* = \arg\min_{x \in X} \boldsymbol{f}(x) \tag{2}$$

$$Y^* = \{\boldsymbol{f}(x^*) \mid x^* \in X^*\} \tag{3}$$

where X^* is the Pareto set and an x is considered minimal is it is non-dominated.

The optimization problems will, going forward, be considered minimization problems because auto-tuning GPU kernels generally takes the form of trying to minimize the amount of resources (e.g. time, energy, memory space) used during its execution.

The **ideal** and **nadir vectors** are two vectors in the objective space whose components are the best and worst (resp.) values for each objective function found in a non-dominated set. Given a non-dominated set X^* the ideal ($\mathbf{y}_{\text{ideal}}$) and nadir ($\mathbf{y}_{\text{nadir}}$) vectors are defined as follows for a minimization problem:

$$\mathbf{y}_{\text{ideal}} = [f_1^{\min}, f_2^{\min}, ..., f_m^{\min}] \tag{4}$$

$$\mathbf{y}_{\text{nadir}} = [f_1^{\max}, f_2^{\max}, ..., f_m^{\max}] \tag{5}$$

where $f_i^{\min} = \min_{x \in X^*} f_i(x)$ and $f_i^{\max} = \max_{x \in X^*} f_i(x)$ for $1 \leq i \leq m$. The hypercube defined by the ideal and nadir gives important information about the range of possible objective values and is required to normalize the objective space [4].

2.2 Auto-tuning

"Autotuning refers to the automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation." — [5]

It is important to note that implementation does not just refer to the sequence of instructions of the program to be executed but also to the configuration of the computing system it will be executed on. This means that all non-functional aspects of an implementation over which there is sufficient control can in principle be tuned and as such be considered tuning parameters.

The parameters selected to tune a program combine to form the **parameter space** X defined as the Cartesian product of the value sets X_i of these parameters:

$$X = \prod X_i.$$

An element $x \in X$ is called a **parameter configuration** and together with the program P has a map to an implementation variant $(P, x) \mapsto P_x$ if it satisfies the constraints imposed by the user, software, and hardware. The **search space** that needs to be explored is then the subset of X containing the configuration for which such mappings exists. In other words, the search space is the set of parameter configurations that correspond to a valid implementation variant.

2.2.1 Tuning parameters

Tuning parameters may generally be divided into two groups, the system parameters and the program parameters [6].

The **system parameters** do not alter the code of the program itself, but do effect how the code is executed on the computing system. Examples of system parameters are:

- the thread block size and the extent of each dimension;
- the clock frequency of the processor and memory; and
- the number of communication channels between the host and GPU.

The **program parameters** are the transformations applied to the program to produce a program variant while keeping them semantically equivalent, although an increase in round off error may be permissible depending on the problem domain. Each transformation can further have its own parameters that can be tuned. Examples of program parameters are:

- loop unrolling, which has the unrolling factor as parameter;
- loop tiling, which has the tile dimension as parameter;
- the data types and structures used to represent the data being processed; and
- the algorithms used to perform the computations.

The order that the transformations occur in is also important, because changing the order of application can produce different programs with different performance characteristics. As a very simple example take the program:

```
FOR (i := 0; i < N; i := i + 1)
    FOR (j := 0; j < M; j := j + 1)
        f(i, j);
```

First applying loop interchange of `i` and `j` and then unrolling the inner loop 2 times produces:

```
FOR (j := 0; j < M; j := j + 1)
    FOR (i := 0; i < N; i := i + 2)
        f(i,     j);
        f(i + 1, j);
```

While performing these transformations in the opposite order produces:

```
FOR (j := 0; j < M; j := j + 2)
    FOR (i := 0; i < N; i := i + 1)
        f(i, j);
        f(i, j + 1);
```

If `f()` were to now access an array using the indices the performance could be vastly different depending on the array's layout. This shows that unless all transformations are independent (i.e. do not effect the same region of code) their order of application should be considered another tuning parameter.

2.2.2 Evaluation

There are generally two approaches to evaluate a variant [5]. The first and most straightforward approach is to compile and benchmark the configuration and measure the characteristics of interest, e.g. execution time and energy usage. The second approach is to create a mathematical model that is able to predict the objective values without having to compile or execute the program. Such a model can either

be constructed using heuristics or through machine learning methods. The advantage of this latter approach is that the benchmarking step is avoided which takes a lot of time, making tuning faster. The disadvantage however is that the model takes a non-trivial amount of work to create, its accuracy needs to be validated, and it needs to be updated each time the tuning parameters change. This generally makes model-based evaluate techniques less flexible than the model-free ones.

2.2.3 Exploration

The size of the parameter space and in turn the search space increases quickly as the number of tuning parameters and their possible values they can have increases. This makes the exhaustive evaluation of all possible configurations infeasible for practical applications of auto-tuning. Instead optimization algorithms have to be used so that only a small subset of the search space has to be explored to find a good configuration.

A wide variety of optimization algorithms have been employed for auto-tuning [7]. A constraint that the use of optimization algorithms imposes however, is that the user now needs to decide what performance metrics are going to be the optimization objectives that will steer the tuning process, whereas with exhaustive exploration a wide range of metrics could be recorded and the selection done after the fact. A further limitation is that many optimization algorithms are designed to only work with a single objective, so when multiple objectives need to be taken into account they have to be transformed in a single one, which inevitably means that information will be lost. This is a major issue because, just like in economics, it is rarely the case that we only care about a single objective, we want the code to run as fast as possible, use as little energy as possible, minimize the response time, maximize the throughput, etc. The use of multi-objective optimization algorithms for the auto-tuning process solves this latter issue by being able to search for the program variants that sit at trade-off points between the different objectives, as it is rarely the case that there is a single variant that is the best for all objectives.

2.3 Kernel Tuner

Kernel Tuner is an extensible and generic auto-tuning framework written in Python [3]. It is capable of tuning CUDA, OpenCL, and HIP kernels and can even be used to tune C host side code that launches these kernels.

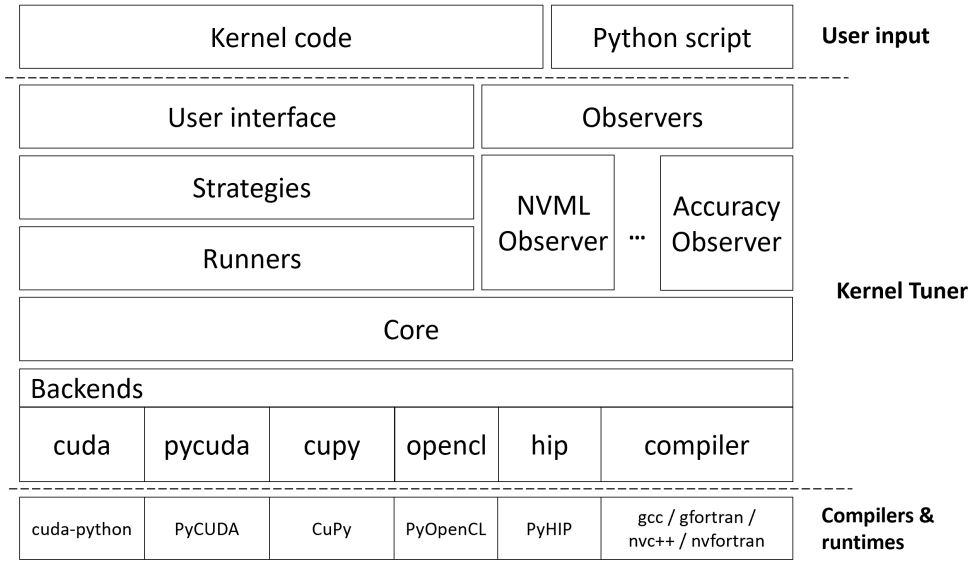


Figure 1: Block diagram of the architecture of Kernel Tuner.

The ability to tune both host and device code allows for the optimization of host-device communication and across different GPU kernels. The latter is very useful for applications that contain a pipeline of different kernels.

Kernel Tuner offers a wide variety of optimization algorithms, which they refer to as **strategies**, such as Basin Hopping, Genetic Algorithms, and Bayesian Optimization, but by default it uses the "brute force" strategy, which explores the search space exhaustively.

To control the tuning process of Kernel Tuner a Python script is needed that sets up the tuning problem and calls the tuner. Among the things this script prepares and passes onto the tuner are the program code, problem size, the list of arguments used to call the program, and the tuning parameters with their possible values. There are other settings like the tuning objectives, the search strategy, and the settings of the strategy, but these are optional, so unless specified the defaults will be used. The full list of parameters that are available to control the tuning process can be found in the documentation of Kernel Tuner [8].

Kernel Tuner evaluates a parameter configuration by compiling the program using the configuration's parameter values and benchmarking it on the inputs passed by the script. What is recorded in the results of the benchmark depends on the **observers** and **metrics** that are being used. The **observers** record values that are measured from the execution of the program configuration or can only be known about the program after it has been compiled, these include the execution time, energy usage, the number of registers used by each thread, and performance counters. **Metrics** on the other hand are values that are calculated from the values recorded by the observers or from the parameter values. Each program configuration is benchmarked multiple times in a row, 7 by default, to reduce the effect of the variance of the properties that are being measured, like execution time, and their mean is also recorded in the result.

A consequence of evaluating the configurations by benchmarking is that the time taken to execute the different program configurations dominates the total time it

takes to tune.

A result cache is maintained during tuning so that the benchmark results can be reused when the strategy tries to evaluate an already benchmarked configuration again instead of redoing the work, greatly speeding up the tuning process. It makes it possible to use optimization algorithms that require the objective functions to be deterministic, which would not be the case if the same kernel was benchmarked multiple times due to the stochasticity inherent in the measurement of objectives like execution time and energy consumption.

Kernel Tuner is able to simulate the tuning process on exhaustively explored search spaces using the result cache system. This allows for different strategies and the same strategy with different settings to be tried on the search space without having to benchmark the program configurations for each tuning run with different setup.

One of the limitations that Kernel Tuner currently has and that this thesis tries to resolve is that it only supports tuning for a single objective, so when multiple objectives are of interest scalarization techniques need to be used to reduce them to one.

3 Related work

In this section I will discuss a number of auto-tuning frameworks, the first 2 works are genetic auto-tuning frameworks that have been used to both tune CPU, GPU, and mixed workloads, the 3 works after that mainly focus on tuning GPU applications, but may also be capable of tuning CPU applications, and the last 3 works have multi-objective auto-tuning as their primary focus. The works that I looked at do generally not incorporate compiler techniques such as performance heuristics or optimization passes, but instead empirically measure the characteristics of the program variants which are produced by changing the source program and the compiler settings used to compile the programs to find the best program configuration.

OpenTuner [9] is one of the first generic auto-tuners. It can accept multiple objectives, but either scalarizes them into a single objective or in the case of two objectives can optimize one while thresholding the other. This means OpenTuner is still essentially a single objective auto-tuning framework.

Rasch *et al.* [10] introduce the Auto-Tuning Framework (ATF). ATF is a generic auto-tuning framework written in `C++` that support a range of target languages and search techniques. It allows for the expression of dependencies between tuning parameters making it especially useful for situations where this is important, like with the GEMM operator for which the shape of the accumulation matrix depends on the shape of the multiplied matrices. Its handling of multi-objective optimization is however lacking, it is possible to specify multiple objectives like with OpenTuner, but it is required that the solutions can be placed in a total ordering based on their objective values for the optimization algorithms to work correctly. This means the domination relation cannot be used as it only induces a partial ordering on the space of solutions and it cannot produce a Pareto set of configurations at the trade-off points.

CLTune by Nugteren and Codreanu [11] is a single-objective auto-tuner designed to work with OpenCL applications. It is written in `C++` and allows the user to pick between the exhaustive search, random search, simulated annealing, and particle swarm optimization algorithms to explore the search space.

Kernel Tuner Toolkit [12] is a auto-tuning framework written in `C++` with an API based on that of CLTune and its search space generation derives from ATF. Its primary focus is GPU kernels and it supports the OpenCL and CUDA kernels. It is however able to also tune for th CPU and the ability to optimize pipelines of kernels and the communication between host and device were all considered during its design.

Kernel Tuner [3] as I've already described in Section 2.3 is the auto-tuning framework I chose to extend with multi-objective capabilities. Because it is written in Python it is more easily accessible to a wide variety of applications, especially data science and machine learning applications. Kernel Tuner and ATF are both very generic and extensible, but Kernel Tuner has a stronger focus on the tuning of GPU applications.

Chen and Hollingsworth [13] introduce ANGEL, a hierarchical optimization technique that is able to accept multiple objects and is tailored to auto-tuning for HPC applications. The proposed technique first gives each objective a priority ranking and then a single-objective algorithm is used to optimize the objective in priority order. Constraint information is passed on from higher to lower objectives which makes solutions invalid that have a value for a higher objective that exceeds a specified relative difference with the best known value for this objective. Because the multiple objectives are in essence scalarized, only using a hierarchical technique, only a single solution is returned as optimal, which has the downside that the user does not have access to a range of trade-offs to pick between.

Nardi *et al.* [14] introduce HyperMapper 2.0, a design space exploration framework that is able to work with multiple objective simultaneously. Design space exploration is similar to auto-tuning, but approaches the problem from the angle of computer system design and engineering design optimization more broadly. The proposed framework uses a white-box model and Design of Experiments (DoE) methods to learn how the design variables effect the objective values. They use their framework to explore the design space of implementing a computation on an FPGA. The framework can also be used to auto-tune GPU kernels, as was done in [15].

Cheema and Khan [16] introduce an auto-tuning framework called MOKAT that uses NSGA-II, a evolutionary multi-objective algorithm, and apply it to a 2D convolution kernel to simultaneously optimize its runtime performance and energy efficiency. It is able to tune OpenCL and CUDA kernels and uses NVIDIA NVML to obtain the the power sensors values on the device. It does not support any other way of measing power, making it unable to measure power metric on AMD and Intel GPUs. Unlike the other frameworks, MOKAT does not seem to be open-source, making it both difficult to studie and assess its capabilities besides those mentioned in the paper directly.

4 Design and Implementation

In this section I will first describe the application of multi-objective optimization to the problem of auto-tuning GPU kernels, then genetic algorithms in Section 4.1, NSGA-II in Section 4.2, NSGA-III in Section 4.3, and finally the implementation in Section 4.4.

In real-world application we rarely only care about a single facet of the application's performance and computational resources it consumes. Instead, a wide variety of aspects need to be considered to satisfy the non-functional desires and requirements of the application. Take a cloud computing environment for example, in this space

the service cost is often directly proportional to the amount of resources used so to minimize the cost a configuration of the application that uses the least amount of resources is desirable. On the other hand, we still want maximal throughput, minimal response time, or a combination of both while using as little memory as we can. All these, often conflicting, desires mean that to understand the trade-offs that exist in the design space of our application we need to consider a variety of objectives, making multi-objective optimization a natural choice for tuning applications.

Multi-objective optimization problems do not generally have a single solution that minimizes all objectives simultaneously. Instead there is a Pareto set, a set of solutions that cannot be altered to decreased the cost in any objective without also increasing the cost of at least one other objective (see Section 2.1).

The feasible region of the parameter space, i.e. the search space, cannot be fully determined statically because the validity of a parameter configuration can only be fully known after it has been successfully compiled and launched. This is because of the hardware restrictions imposed by the GPU that is being tuned on and the guarantees, or the lack thereof, that successful compilation gives.

The objective values of a parameter configuration are determined in multiple ways, the major aspects such as execution time and energy consumption are measured empirically. Other aspect such as memory footprint can often be calculated from the parameters themselves using knowledge about the kernel that is being tuned. There are also objective values that can be derived from others, such as average power draw, which is defined as the total energy consumption divided by the execution time.

A consequence of the empirical nature of the objective space is that the true Pareto set cannot be known unless the search space is exhaustively evaluated, which is prohibitively expensive for practical applications. Instead the true Pareto set can only be approximated using a variety of optimization algorithms, two of which are explored below. Not knowing the true Pareto set causes issues when it needs to be known how good an approximation is in an absolute sense, but it is still possible in a relative manner, this is explored further in Section 5.

4.1 Genetic algorithms

The Pareto set can be approximated using various algorithms, such as Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D) [17], Reference Vector guided Evolutionary Algorithm (RVEA) [18], and Non-dominated Sorting Genetic Algorithm I, II, and III (NSGA-I, NSGA-II, and NSGA-III) [19]–[21]. NSGA-I is the earliest algorithm in the NSGA family, but has been completely superseded by NSGA-II. Only NSGA-II and NSGA-III have been explored in this thesis to keep the score and the required implementation effort reasonable.

NSGA-II and NSGA-III are genetic algorithms and the latter also makes use of reference directions. Genetic algorithms are meta-heuristics that for the purposes of this thesis can be described using the outline shown in Algorithm 1. For the purposes of this thesis it can be assumed that $\nu_{\text{par}} = \nu_{\text{off}}$ and $\rho_{\text{cross}} = 1$.

This same structure also applies to its multi-objective variant, but instead of maintaining a single best solution that approximates the optimal solution, a set of solutions is maintained that are all Pareto optimal within the population, effectively approximating the true Pareto set.

An important aspect of applying genetic algorithms to a problem is how the solutions

Algorithm 1 Genetic algorithm outline

▷ *Hyperparameters* ◁
ν_{init} = the size of the initial population.
$\nu_{\text{par}},\ \nu_{\text{off}}$ = the number of parents and the number of offspring.
$\rho_{\text{cross}},\ \rho_{\text{mul}}$ = the rate of crossover and the rate of mutation.
ν_{pop} = the size of the population that survives to the next generation.

1: $P_0 \leftarrow \text{SAMPLE}(\nu_{\text{init}})$
2: $\text{EVALUATE}(P_0)$
3: $t \leftarrow 0$
4: **while not** $\text{TERMINATE}(P_t)$ **do**
5: ▷ *Mating phase* ◁
6: $M_t \leftarrow \text{SELECT-MATES}(P_t, \nu_{\text{par}})$
7: $Q_t \leftarrow \text{CROSSOVER}(M_t, \nu_{\text{off}}, \rho_{\text{cross}})$
8: $\text{MUTATE}(Q_t, \rho_{\text{mut}})$
9: $\text{REPAIR}(Q_t)$
10: ▷ *Survival phase* ◁
11: $\text{EVALUATE}(Q_t)$
12: $P_{t+1} \leftarrow \text{SELECT-SURVIVORS}(P_t \cup Q_t, \nu_{\text{pop}})$
13: $t \leftarrow t + 1$
14: **end while**

are represented. Nothing special is done in this regard, largely because Kernel Tuner already uses a representation that was easily adapted to work with multiple objectives. Formally it can be assumed that a solution is an element of the Cartesian product of the tuning parameter domains.

What follows is a description of the operators mentioned in Algorithm 1:

Sampling The SAMPLE() operator selects a sample of the search space, there are more sophisticated ways of sampling that take the objectives into account but these are not explored in this thesis, instead a simple uniform random sample is used.

Evaluation The EVALUATE() operator determines and assigns fitness scores to the individuals of the population. In the case of auto-tuning this corresponds to benchmarking the implementation variant in case it was not found in the result cache.

Termination The TERMINATE() operator decides when to end optimization loop should end. Only fixed budget termination is considered in this thesis.

Selection The selection operators SELECT-MATES() and SELECT-SURVIVORS() pick the population members which are going to produce offspring and survive to the next generation respectively using the fitness score assigned to them. The algorithm most commonly used to implement SELECT-MATES() is Tournament Selection [22]. Tournament Selection works by taking ν_{par} simple random samples with replacement from P_t with each sample being size k. The members of each sample are then ranked according to some comparison function. After that an individual is picked from each sample to form the mating population M_t. The method of picking individuals can either be probabilistic or deterministic. The probabilistic method chooses

- the highest rank individual with probability p,

- the second highest with probability $p(1-p)$,
- the third highest with $p(1-p)^2$,
- et cetera

while the deterministic method always picks the one with the highest rank, which is the same as the probabilistic method with $p = 1$. This means that normally Tournament Selection has the hyperparameters k and p in addition to ν_{par}, but for the purposes of this thesis $k = 2$ and $p = 1$, eliminating these variables again. The comparison function will also not be considered hyperparameter, because it is an integral part of the design of NSGA-II and NSGA-III.

The SELECT-SURVIVORS() is required because only a subset of the combined population $P_t \cup Q_t$ can survive to the next generation. In the case of NSGA-II and NSGA-III it also serves the purpose of selecting those members from the combined population that are simultaneously the most promising solutions and together are as diverse as possible to make convergence to the Pareto front more likely.

Crossover The CROSSOVER() operator pairs solutions in the mating population M_t and combines each pair to produce two offspring solutions in the offspring population. There are many possible ways of combining two solutions to produce new ones, of them the ones I selected are single-point, two-point, and uniform crossover [23], [24] because they work well for the tuple representation used to represent the solutions. Figure 2 shows a simple visualization of single-point crossover.

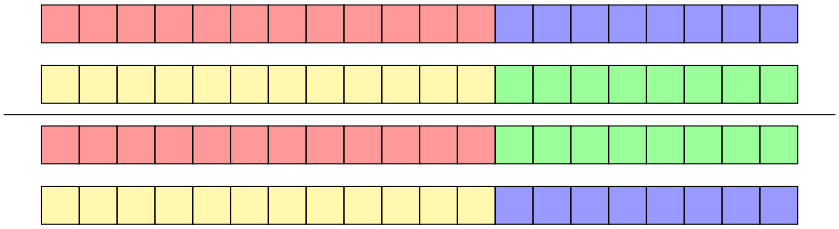


Figure 2: Single-point crossover: The cells that make up each row represent a variable for which a value needs to be selected. The top two rows are the parents solutions and the bottom two the produced offspring. The colored regions indicate from which parent the values were selected for the variables of the offspring.

The nature of the search spaces dealt with in auto-tuning makes this a major source of invalid configurations and is also why the REPAIR() operator is important to have a decent ratio of proposed solutions to valid solutions, which is explained below.

Mutation The MUTATE() operator has the chance to modify solutions with probability called the mutation rate. The general intent of this operator is to introduce new variance into the population.

A design is chosen where each solution has a chance of being swapped with a valid solution that differs in exactly 1 tuning parameter, i.e. has a Hamming distance of 1, from the original. A solution is only allowed to be swapped with a valid one in its neighborhood because a parameter configurations can easily become invalid when modified, so allowing invalid solutions would mean that either a large number of the solutions in a generation can become invalid or the mutation rate would have to be

kept low to prevent this. Keeping the mutation rate low would in turn mean less variance is introduced in the population, which would reduce the exploration that occurs. Allowing MUTATE() to produce invalid solutions would also make it possible for the repair to revert the solution back to its pre-mutation state, rendering the work useless. The effect of increasing or decreasing the mutation rate would also become less predictable because the frequency at which mutated solutions are reverted back to their pre-mutation state would counteract the mutation rate. A benefit of the chosen design is that the original and modified solutions are always very similar which should allow it to function as a local search.

Repair The REPAIR() operator takes an invalid solution and attempts to turn it into a valid one.

In the design an attempt is made to repair an invalid solution by looking for a valid solution in three increasingly wide neighborhoods one by one. The invalid solution is replaced by the first valid solution found, but if there are no valid solutions in the neighborhood then the solution is left unchanged.

4.2 NSGA-II

NSGA-II [20] is a Multi-Objective Evolutionary Algorithm (MOEA) which has three main goals (1) good convergence, (2) good variety, and (3) good coverage [25]. Good convergence means that the difference between the true and approximate Pareto fronts reliably decreases or is maintained which would result in the approximation settling down given sufficient generations. If this is not the case the algorithm would be able to get stuck in a loop of the approximation getting closer and further away from the true Pareto front. Good variety means that the algorithm maintains sufficient diversity in its population so the entire Pareto front remains reachable. Good coverage means that the approximate Pareto fronts produced cover a sufficient portion of the Pareto front. The latter two are closely connected because if a diverse population can be maintained until the end it is also likely that said population will cover a significant part of the true Pareto front.

Algorithm 2 NSGA-II

1: $P_0 \leftarrow \text{SAMPLE}(\nu_{\text{init}})$
2: $\text{EVALUATE}(P_0)$
3: $P_1 \leftarrow \text{FIRST-GENERATION}(P_0)$
4: $t \leftarrow 1$
5: **while not** $\text{TERMINATE}(P_t)$ **do**
6: $\quad M_t \leftarrow \text{SELECT-MATES}(P_t, \nu_{\text{par}})$
7: $\quad Q_t \leftarrow \text{CROSSOVER}(M_t, \nu_{\text{off}}, \rho_{\text{cross}})$
8: $\quad \text{MUTATE}(Q_t, \rho_{\text{mut}})$
9: $\quad \text{REPAIR}(Q_t)$
10: $\quad \text{EVALUATE}(Q_t)$
11: $\quad R_t \leftarrow P_t \cup Q_t$
12: $\quad P_{t+1} \leftarrow \text{SELECT-SURVIVORS}(R_t, \nu_{\text{pop}})$
13: $\quad t \leftarrow t + 1$
14: **end while**

NSGA-II with a REPAIR() operator added is shown in Algorithm 2 and is practically identical to Algorithm 1 except for Lines 3 and 4, because the first generation

Algorithm 3 NSGA-II's survivors selection

```
1: procedure SELECT-SURVIVORS(R_t, N)
2:     F ← FAST-NON-DOMINATING-SORT(R_t)
3:     S_t ← ∅; l ← 1
4:     while |S_t| + |F_l| < N do
5:         S_t ← S_t ∪ F_l
6:         l ← l + 1
7:     end while
8:     ASSIGN-CROWDING-DISTANCE(S_t ∪ F_l)
9:     if |S_t| + |F_l| = N then
10:        S_t ← S_t ∪ F_l
11:    else
12:        SORT-DSC(F_l, ≺_n)
13:        S_t ← S_t ∪ TAKE(N − |S_t|, F_l)
14:    end if
15:    return S_t
16: end procedure
```

is a special case. Its SELECT-SURVIVORS() operator shown in Algorithm 3 uses the objective values of the population to determine for each individual how many dominate it, called its rank, and which solutions it dominates. This is used to quickly sort the population into least non-dominated levels, as described in Algorithm 4. It also determines how crowded individuals in the population are using a metric they call the crowding distance. The crowding distance is calculated using the Manhattan distance an individual has to its two neighbors per objective. The neighbors are the first objective values smaller than and larger than the individual's value after the values of the objective have been min-max normalized and sorted. The minimal and maximal individuals, i.e. those that only have one neighbor for one of the objectives, are assigned infinite crowding distance. The internal individuals' crowding distance is the mean of their Manhattan distance for each objective. (See [20] for a more complete description.) The rank and crowding distance are combined in the Crowded-Comparison Operator ($\prec_n$) which prefers individuals with lower rank over those with a higher one, but if their ranks are the same it picks the one with the highest crowding distance. This operator is used in both SELECT-MATES() and SELECT-SURVIVORS(), although the latter mostly relies on non-dominated sorting and only uses the comparison operator to select a part of the survivors. These mechanisms combined allow the algorithm to have both good convergence and diversity properties.

The algorithm's steps are now briefly described in word in addition to the algorithms that have already been given. The first generation (computed by FIRST-GENERATION()) is a special case because the crowding distance is calculated during the survival phase so the first mating selection cannot yet make use of the complete crowded-comparison operator, instead the rank is used on its own. After the first generation the algorithm goes as follows: Given generation t we have parent population P_t of size N. Mating pairs are then selected from P_t to procreate offspring Q_t

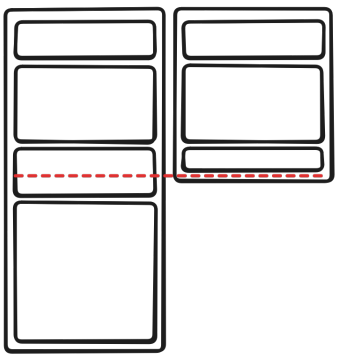


Figure 3: Left box is R_t. Right box is S_t after selection. Subboxes are levels F_i. Red line is the cutoff point N.

also of size N. Populations P_t and Q_t are then combined to form population $R_t = P_t \cup Q_t$ of size $2N$. R_t is now partitioned into non-dominated levels F_1, F_2, etc. using Algorithm 4, where F_1 is the non-dominated set of R_t, F_2 the non-dominated set of $R_t - F_1$, and so on for the rest of the levels. The survivors S_t are chosen in two steps: (1) starting from F_1, non-domination levels are selected one by one until their union's size would either be equal to or exceed N, name the last level F_l and $S_t = \bigcup_{i=1}^{l-1} F_i$. This means the levels $l+1$ and up are all rejected and will not survive until the next generation. (See Figure 3.) (2) If $|S_t| + |F_l| = N$ then the members of F_l are added to S_t. If this is not the case their combined size would exceed N, so only a subset of the members F_l can be added. F_l is now sorted using Crowded-Comparison Operator ($\prec_n$) in descending order and individuals are selected from the start and added to S_t until it reaches size N.

Algorithm 4 Fast non-dominated sorting algorithm from [20]

```
1:  procedure FAST-NON-DOMINATED-SORT(P)
2:      F_1 ← ∅
3:      for each p ∈ P do
4:          S_p ← ∅                          ▷ Set of individuals that p dominates
5:          n_p ← 0                          ▷ Count of individuals that dominate p
6:          for each q ∈ P do
7:              if p ≺ q then
8:                  S_p ← S_p ∪ {q}
9:              else if q ≺ p then
10:                 n_p ← n_p + 1
11:             end if
12:         end for
13:         if n_p = 0 then
14:             p_rank ← 1
15:             F_1 ← F_1 ∪ {p}
16:         end if
17:     end for
18:     i ← 1
19:     while F_i ≠ ∅ do
20:         F_{i+1} ← ∅
21:         for each p ∈ F_i do
22:             for each q ∈ S_p do
23:                 n_q ← n_q − 1
24:                 if n_q = 0 then
25:                     q_rank ← i + 1
26:                     F_{i+1} ← F_{i+1} ∪ {q}
27:                 end if
28:             end for
29:         end for
30:         i ← i + 1
31:     end while
32: end procedure
```

4.3 NSGA-III

NSGA-III [21] is a Many-Objective Evolutionary Algorithm (MaOEA). MaOEAs are MOEAs that have four or more objectives. NSGA-III is similar to NSGA-II but has significant differences in its selection operator to better deal with higher-dimensional objective spaces. The hypervolume of the objective space grows quickly as the number of objectives increases, making the solutions more sparse in the objective space. This makes the crowding-distance metric used by NSGA-II a less effective measure of solution density because it uses neighboring solutions to approximate this. NSGA-III like NSGA-II partitions the combined population R_t into non-dominating levels, orders them from least to most dominated, and selects the levels just like NSGA-II does. It does however differ in the way it selects members from the last level F_l in case $|S_t| + |F_l| > N$. Instead of crowding distance it uses reference points to maintain the diversity of the population. Using reference points to maintain diversity is more robust against the curse of dimensionality because its approximation of the solution density doesn't depend on neighboring solutions, which get further and further away as the number of objectives increases.

The reference points lie on the unit hyper-plane in the objective space, this means that for any reference point $\mathbf{x}$ the sum of its components, $\sum \mathbf{x} = 1$. To make the objective vectors of $S_t \cup F_l$ compatible with the reference points they are normalized using their ideal and nadir vectors. The members of $S_t \cup F_l$ are then associated to the reference point they are closest to, where closeness is measured by the perpendicular distance an objective vector has when treated as a point to the lines passing through the origin and reference points, as shown in Figure 4. This is also why they are sometimes called reference directions instead of reference points. Using the information about the number of solutions associated with each reference point it keeps track of the least populous ones while selecting the individuals from F_l. It does the selection one at a time based on which individual is closest to a least populated reference point and associates the individual to it, increasing the population count of the reference point. This process is repeated until the number of individuals required to make $|S_t| = N$ have been selected from F_l.

The reference points can be either generated using techniques like Das and Dennis's structured approach [26] and the Riesz s-Energy based method [27] or manually specified. Manually specifying the reference points allows the user to guide the search process because NSGA-III is likely to identify Pareto optimal solutions close to the specified reference points. This means the reference points can be used to encode preference information and as such become part of the decision making process [21].

4.4 Implementation

In this section I describe the implementation of my multi-objective optimization extension to Kernel Tuner. The implementation can be neatly separated into two parts, the modifications made to Kernel Tuner to accommodate multiple objectives (Section 4.4.1) and the new strategy module that gives access to multi-objective optimization algorithms (Section 4.4.2).

4.4.1 Multiple objectives

The `tune_kernel()` function, a part of Kernel Tuner's user interface, is extended so the user can specify multiple objectives to guide the strategy and their respective optimization directions, i.e. minimize or maximize.

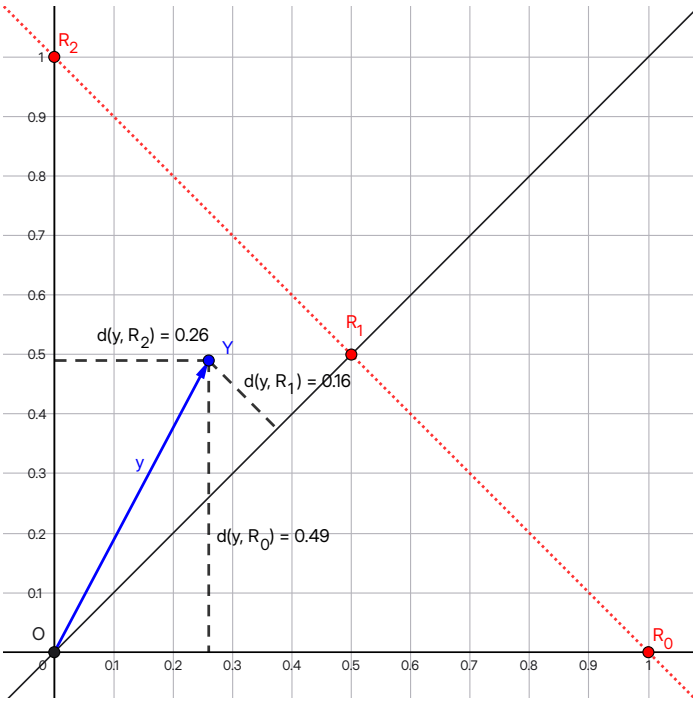


Figure 4: The perpendicular distance distance between objective vector y and reference points $\mathrm{R}_0, \mathrm{R}_1, \mathrm{R}_2$. Y is the point vector y corresponds to and $d(y, \mathrm{R}_\mathrm{i})$ is the perpendicular distance point Y has to the line OR_i.

I have also modified the result cache (Section 2.3) because the original format does not work well when multiple objectives are involved. In the original format a result is stored in a Python dictionary that contains the tuning parameters, observed values, and derived metrics. The issue is that the field used as the objective is also used to store the error received if the configuration associated with the result fails to be evaluated. This might work fine when you are dealing with a single objective, but when dealing with multiple objectives it is unclear which one to pick. A naive solution would be to always pick the first objective specified, but this creates an unnecessarily dependence on the order of the objectives. Instead I chose for an approach that uses a special "error" key that is only added to results that failed to be evaluated. This makes checking if a cached result is erroneous as simple testing if the "error" keys is in the dictionary. A collateral effect of changing the cache format is that all the places that use it need to be adapted for it.

The code Kernel Tuner uses to get the objective value from a result is also modified to allow for multiple objectives and returns the objective vector taking optimization directions of the objectives into account.

The runners, which compile and benchmark the GPU kernels, did not need to be modified to work with multiple objectives. This is because the runners depend on the observers and metrics, but do not interact with the objectives specified by the user.

4.4.2 Strategy module

Kernel Tuner implements strategies using a modular interface, making it easy for users to add their own. Each strategy is a Python module that implements the `tune()` function. The `tune()` function takes a search space, runner, and the tuning options as parameters and returns all the results gathered during tuning.

The new strategy module uses the Pymoo [28] framework for its implementation of the NSGA-II and NSGA-III optimization algorithms. Pymoo is a framework written in Python for multi-objective optimization and is designed in an object-oriented style that makes it easy to adapt to various applications. In the framework, optimization problems and algorithms are represented by classes as are most other parts of Pymoo. The algorithms are further more factored into various operators, genetic algorithms for example can be constructed using a variety of sampling, selection, crossover, and mutation operators. After an algorithm has been constructed it still needs to be set-up for the optimization run you want to do. This step sets the optimization problem, the termination condition, and has various other parameters to configure the run. A Pymoo problem class is implemented to represent the tuning problems of Kernel Tuner, all this class does is call Kernel Tuner's solution evaluation machinery. It also makes the true Pareto front accessible to Pymoo in tuning simulation mode, this is not accessible during normal tuning because the search space needs to be known to determine the true Pareto front.

Pymoo is primarily designed for continuous optimization problems so I had to implement custom sampling and repair operators that are aware of the structure of Kernel Tuner's search spaces. It would have also been possible to model the entire search space using components from Pymoo, but I decided against it because this would have also required that all the constraint machinery in Kernel Tuner would have to be reimplemented using Pymoo and it was unclear if this would result in any tangible benefits.

Pymoo provides multiple crossover and mutation operators that can be used for discrete optimization problems. These operators do not normally need special knowledge about the search space, but as was described in the design section I still chose to write a custom mutation operator. The crossover operators can be used as is however and I decided to give easy access to the uniform crossover, single-point crossover, and two-point crossover methods, because they worked on the solution representation without the need to make any additional changes to the case. When it comes the the selection operators, the defaults that Pymoo provides for NSGA-II and NSGA-III are used as is, which is Tournament selection for SELECT-MATES() and the algorithms are described above for SELECT-SURVIVORS(). This decision is made because both the mating and survivor selection operators are tightly coupled with the implementations of the algorithms, making it hard to swap them freely for another selection algorithm without exposing an excessive amount of the implementation details to the user. It is thus left to future work to make these hyperparameters configurable by the user.

5 Evaluation

I reused the dataset produced for [29] to perform my experiments. The dataset contains results for the search spaces of 9 kernels that have been exhaustively evaluated with a focus on mixed-precision computing, so there are multiple tuning parameters that affect the data types of variables and the numerical error on the computed output values. The exact number of tuning parameters and information on the size of

the search spaces can be found in Table 7 in Appendix A.2. A total of 5 metrics were recorded during the evaluation of the search spaces of the kernels, 2 of them pertaining to computational resources and 3 pertaining to numerical precision:

- The execution time of the kernels, the value for this metric is determined by executing the kernel seven times and taking the average of the individual measurements.
- The memory footprint, this represents the amount of memory the kernel occupies on the device.
- The mean relative error (MRE):

$$\text{MRE} = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y'_i|}{|y'_i|}$$

- The normalized root mean squared error (NRMSE):

$$\text{NRMSE} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}|y_i - y'_i|^2}}{\frac{1}{n}\sum_{i=1}^{n} y'_i}$$

- The normalized mean absolute error (NMAE):

$$\text{NMAE} = \frac{\frac{1}{n}\sum_{i=1}^{n}|y_i - y'_i|}{\frac{1}{n}\sum_{i=1}^{n}|y'_i|}$$

What follows is a description of the kernels that were tuned:

Bessel A Bessel kernel that computes the approximation of the modified Bessel function of the first kind using the follow formula:

$$I_0(x) = 1 + \sum_{k=1}^{k_{\max}} t_k \quad \text{where} \quad t_0 = 1 \quad \text{and} \quad t_k = t_{k-1} \cdot \frac{x^2}{4k^2}$$

The tuning parameters are the number number of terms evaluated (i.e. the value of $k_{\max}$) and the data types of the inputs and output, the intermediate result, and the computations.

Convolution2D A kernel that computes the 2D convolution of an input image. Each thread block loads a tile of the image into shared memory and the threads in each thread block cooperate to process the image elements and store the results. The tuning parameters are:

- the thread block dimensions
- the tile dimensions
- the option to use shared memory
- the minimum number of thead block per SM, this parameter prevents the overutilization of registers in an SM
- the data types of the input and output image, the filter weights, the internal accumulators, and shared memory.

The kernel was benchmarked by applying a 17×17 filter to an 8192×8192 image.

K-means A kernel that performs distance computation step of the K-means clustering algorithm, i.e. it computes the the distance all N data points have to the K cluster centers. Each thread is assigned multiple data points to spread the workload. The tuning parameters are the data types of the input data points, cluster centers, and the output distances. The kernel was benchmarked by processing 5×10^6 data points with 15 features for 40 cluster centers.

LavaMD The LavaMD kernel computes the inter-atomic forces among particles in 3D space to simulate their molecular dynamics. The 3D space is divided into fixed-size cubes and only the forces between particles in the same cube and adjacent cubes are computed. Each cube gets assigned to a thread block and the threads in each thread block the cooperatively process the particles in the cube and its adjacent cubes. The tuning parameters are the particle positions, the particle charges, the internal accumulators, and the output forces. The kernel was benchmarked by processing 8.6×10^5 particles and the cubes are $15 \times 15 \times 15$.

Inverse Kinematic (InvK) An Inverse Kinematic kernel that computes the angle that each joint of a robot arm needs to have to reach a given target point. The tuning parameters are:

- the number of threads per block
- the number of points processed per thread
- The data types of the input targets, the output angles, and the intermediate variables

The kernel was benchmarked by processing 50×10^6 targets in parallel for a four-joint robotic arm.

Newton-Raphson A Newton-Raphson kernel that computes the roots of 4th-degree polynomials using the iterative Newton-Raphson method for 10 iterations. The tuning parameters are:

- The number of threads per block
- The number of polynomials processed per thread
- The data types of the polynomial coefficients, input root estimates, output roots
- The data types used for the first 5 and last 5 iterations, which they can be different but do not have to be

The kernel was benchmarked by processing 10^8 polynomials in parallel.

Coulombic The Coulombic kernel that computes the electric potential of a collection of charged particles at discrete points on a 2 dimensional grid. The tuning parameters are:

- The data types of the particle positions, particle charges, and electric potentials
- the thread block dimensions
- the number of grid points per thread
- the unrolling factor for the loop that iterates over the particles

The kernel was benchmarked using 10^4 particles and 4×10^6 grid points.

MRI-Q The MRI-Q kernel that performs an inverse non-uniform Fourier transform. It is commonly used in the field of MRI (magnetic resonance imaging) to produces an

image from the input frequency domain data. The data types of the input magnitude-and-phase values, k-space frequency domain positions, spacial domain positions, and output image can be tuned. The kernel was benchmarked using 16×16×16 dimensions for the k-space positions and 64×64×64 dimension for the spacial positions.

Black-Scholes A kernel that computes the call and put prices of options according to the Black-Scholes model from quantitative finance. The kernel was benchmarked by processing 2×10^8 options in parallel.

The kernels were auto-tuned on the Nvidia A100 and the AMD Instinct MI250X GPUs.

- The A100 GPUs have 40GB of memory and were in DAS6 nodes located in the ASTRON-site [30], a part of the distributed supercomputer of the Netherlands. There nodes also have two 16-core AMD EPYC 7282 CPUs and 128 GB of memory. The software used is Rocky Linux 8.9, CUDA 12.2.1, Python 3.11.5, and Kernel Tuner 1.0.
- The MI250Xs have 128 GB of HBM2e memory and are in nodes from LUMI [31], a pre-exascale system located in Finland. Each of the 2,978 GPU nodes has one 64-core AMD Trento CPU and four GPUs. The MI250Xs has two compute dies but they only used a single one per GPU to gather the data. The software used is SUSE Linux Enterprise Server 15 with ROCm 6.0.3, Python 3.9.13, and Kernel Tuner 1.0.

Auto-tuning on these pre-explored search spaces was simulated on a AMD Ryzen AI 7 350 CPU. This CPU has 8 cores 4 of which have the Zen 5 µ-architecture and the other 4 the Zen 5c µ-architecture. Every simulation was run on a single Zen 5 core with simultaneous multi-threading (SMT) disabled, to remove the influence of the scheduler and make sure the results can be compared fairly. Frequency scaling was however not disabled, because this is an integral part of modern day CPUs, so disabling it would limit the hardware's capabilities on the task. Lastly the software used was Fedora Linux 43 and CPython 3.12.12 compiled using GCC 15.2.1.

It is important to note that normally not every accuracy metric is applicable to every kernel because of consideration about their numerical range, but this fact has been ignored during the following analysis and any solutions that had NaN values for any of the objective values were removed and not considered valid solutions. This was done so that the dataset could be treated as having 5 fully fledged objectives instead of the number of objectives depending on the kernel. The decision to ignore the nature of the measurements does make the analysis less representative of real world situations, but having 5 complete objectives to work with was deemed more important to assess the multi-objective capabilities of the algorithms.

5.1 Quality indicators

When compared to single-objective optimization, working with multiple objectives greatly complicates the analysis of solutions because there is generally no natural total ordering of single solutions, instead there is the partial order induced by the domination relation called the Pareto order and there are Quality Indicators (QIs) that assigns a single number to a set of solutions with that number representing a scoring of its quality. Examples of QIs include the Hypervolume Indicator (HV) [32], the Generational Distance (GD) [33], the Inverted Generational Distance (IGD) [34], and their augmentation variants GD^+ and IGD^+ [35]. An aspect of QIs is the

extend to which the order they induce on the solution sets is compatible with the Pareto order, namely a QI $I(x)$ is called Pareto compliant if given two solutions sets A and B that are not equal it follows from every element $b \in B$ having an element $a \in A$ that weakly dominates it $a \preceq b$ that $I(A) < I(B)$ and weakly Pareto compliant if it follows from this relation that $I(A) \leq I(B)$ [36]. If a QI satisfies this property then better solution sets correspond will have higher scores in case of full Pareto compliance and a score that is no worse in case of weak compliance.

IGD^+ , a weakly Pareto-compliant QI was used to score the quality of the produced solution sets. It is defined as follows: given two set objective vectors A and Z IGD^+ measures how "close" A is to Z as follows:

$$\text{IGD}^+(A, Z) = \frac{1}{|Z|} \sum_{z \in Z} \min_{a \in A} d^+(a, z),$$

where

$$d^+(a, z) = \sqrt{\sum_{i=1}^{m} [\max(a_i - z_i, 0)]^2}.$$

This definition assumes the objectives are being minimized. For the purposes of this thesis A is the set of objective vectors (of a prefix) of the list of solutions evaluated during a run and Z the true Pareto front of the search space. So IGD^+ will measure how "close" the current approximation of the Pareto front is to true front.

IGD^+ was chosen over HV, which is strongly Pareto-compliant, making HV have better theoretical properties than IGD^+ , because unlike HV, IGD^+ does does not have exponential runtime complexity in the number of objectives, making it much cheaper to compute for many-objective problems. IGD^+ is also similar to HV in its ability to measure convergence and diversity as long as the true Pareto front is known, which is the case in this studie [37].

Going forward it should be assumed that the objective vectors have been normalized using the ideal and nadir vectors of the true Pareto front of their search spaces before their IGD^+ score was computed.

5.2 Hyperparameters

In order to fairly compare the optimization algorithms across different tuning problems, reasonable values need to be chosen for the algorithms' hyperparameters. Of all the possible values for the hyperparameters the ones considered are shown in Table 1. These are tested on 6 tuning spaces (2 GPUs × 3 kernels), with 3 objectives for NSGA-II and 5 for NSGA-III, for every evaluation budget of interest.

Kernels: Bessel, Convolution, and MRI-Q

GPUs: Nvidia A100 and AMD Instinct MI250X

Objectives: NSGA-II: Time, memory footprint, and MRE.
NSGA-III: NRMSE and NMAE in addtion to the ones NSGA-II has.

Budgets: 50, 100, 150, and 200

This is done by doing a gridsearch over all the combinations of tuning problems (kernel × GPU × objectives), hyperparameter tuples, and evaluation budgets for each algorithm, with each experiment repeated 20 times.

Algorithm	Hyperparameter	Domain	Domain size
NSGA-II	Population size	$\{20, 40, 80\}$	3
	Crossover method	{TWO-POINT-CROSSOVER}	1
	Crossover probability	$\{1.0\}$	1
	Mutation method	$\{custom\}$	1
	Mutation probability	$\{0.05, 0.1, 0.2\}$	3
		Number of possible configurations	9
NSGA-III	Population size	$\{20, 40\}$	2
	Crossover method	{TWO-POINT-CROSSOVER}	1
	Crossover probability	$\{1.0\}$	1
	Mutation method	$\{custom\}$	1
	Mutation probability	$\{0.05, 0.1, 0.2\}$	3
	Reference directions method	{Riesz s-Energy}	1
	Number of reference directions	= population size	1
		Number of possible configurations	6

Table 1: Hyperparameters per algorithm and the domain of each parameter.

I will now describe the procedure that was used to select a hyperparameter configuration for each each algorithm at each budget by giving the steps used on the data for NSGA-II at a single budget, but the steps are the same for the other algorithm, budget pairs: The IGD^+ score of each repetition was computed, producing a table the following signature:

Configuration	Kernel	GPU	Score

$9 \times 3 \times 2 \times 20 = 1080$ rows

The median and interquartile range (IQR) of scores was then computed while grouping by the Configuration, Kernel, and GPU columns to aggregate the 20 repetitions of each experiment, producing:

Configuration	Kernel	GPU	Median	IQR

$9 \times 3 \times 2 = 54$ rows

The mean was then taken by grouping by the Configuration and Kernel columns to get an average score and spread for the kernel on both GPUs per configuration, producing:

Configuration	Kernel	E(Median)	E(IQR)

$9 \times 3 = 27$ rows

The mean was then taken again, but this time grouping by Configuration column to finally produce a single value for the average and a single one for the spread per configuration. The columns were respectively renamed y_1 and y_2 because "mean of mean of medians" and "mean of mean of IQRs" is quite long and cumbersome.

Configuration	y_1	y_2

9 rows

The averaging was done in this way because I wanted to both produce a single value to represent the average score and spread for each configuration, but at the same time wanted it to be more robust against more extreme values and this was a simple way to achieve that goal. The median could have also been used, but I judged that the potential outliers should not be too extreme and that underlying distribution should not be very skewed, so a mean of means seemed fitting.

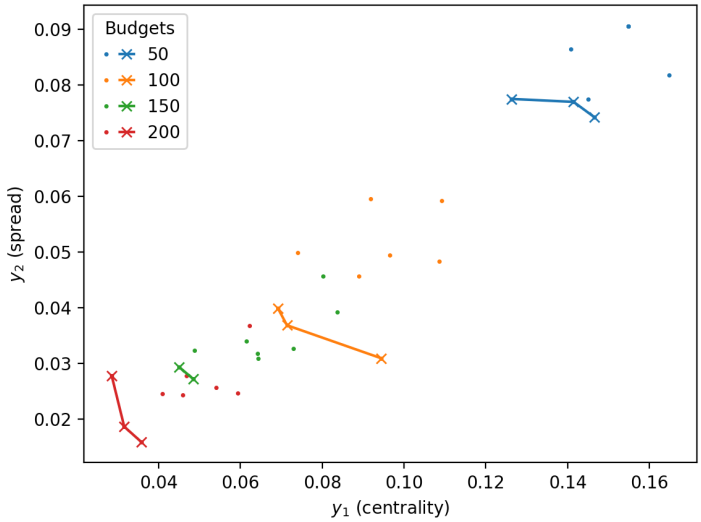
The best configuration for the budget should produce results that have the best average score and give confidance that the scores it produces on a variety of search spaces does not vary excessively. To find this configuration first the Pareto front was calculated by minimizing both y_1 and y_2 and then a configuration on the Pareto front was picked using the following simple decision procedure: If the number of configurations for a budget is odd then select the middle one else the number is even in which case a coin is tossed to pick one of the two middle points.

This procedure was repeated for each algorithm and budget producing the Pareto fronts shown in Figure 5 and Figure 6 for NSGA-II and NSGA-III respectively with the selected configurations highlighted in the table on the right. It can be seen on the plots that for a budget of 50 the Pareto front distributions are quite horizontal in shape, with the one for NSGA-III almost being collinear, while the front distributions for a budget of 200 are much more vertical in shape. The horizontal shape corresponds with a large variance in the average IGD^+ score and small variance in the average spread, this is not unexpected for a low budget and can be explained by the fact that there has been at most one round of evolution, so the algorithms did not yet have a chance to properly converge, making the variance in the scores high, which together causes y_1 to vary a lot and y_2 to be high and stable. The vertical shape of the front distributions when the budget is 200 can be explained through similar reasoning: the solution population would have undergone a number of rounds of evolution before termination, giving it time to converge towards the true Pareto front, making the variance in the score lower, which together causes y_1 to be low and stable, but y_2 is now not kept high by the lack of convergence, making the differences in variance more apperent. This analysis is however speculative and further research is required to properly understand the effects of changing the hyperparameter values on tuning problems. It is also notable that when the budget is 150 the fronts have fewer than 3 points and in the case of NSGA-III only 1, but I do not have an explanation for this phenomenon. Table 6 can be referenced to get a clear overview of what hyperparameter values were selected for the algorithms per budget.

It is worth noting that the size of the hyperparameter space is rather small, or at least, given the flexibility of genetic algorithms it could be much bigger. It was a conscious decision to keep the search small, it is after all not the goal of this thesis to find the optimal configuration for NSGA-II or NSGA-III, but to extend Kernel Tuner with multi-objective capabilities. The goal of this analysis should then be taken to show the promise of these techniques and to do further research using them now that there easily accessible implementation that can be applied to auto-tuning problems.

5.3 Algorithm comparison

To assess the quality of the solutions produced by NSGA-II and NSGA-III for certain budgets tuning was simulated on all the pre-evaluated search spaces for 30 repetitions using the best hyperparameter value for each budget. This was done for the same two sets of objectives used to selected the best hyperparameter configurations (see Table 6), which from now on will simply be referred to as the 3 and 5 objective sets.



Budget	Pop size	P(muta)	y_1	y_2
50	20	0.10	0.126214	0.077449
50	20	0.20	0.141475	0.076920
50	40	0.05	0.146606	0.074142
100	20	0.10	0.069203	0.039828
100	20	0.05	0.071388	0.036830
100	40	0.05	0.094395	0.030919
150	20	0.20	0.044986	0.029355
150	20	0.10	0.048416	0.027212
200	20	0.10	0.028532	0.027749
200	20	0.20	0.031582	0.018643
200	20	0.05	0.035726	0.015902

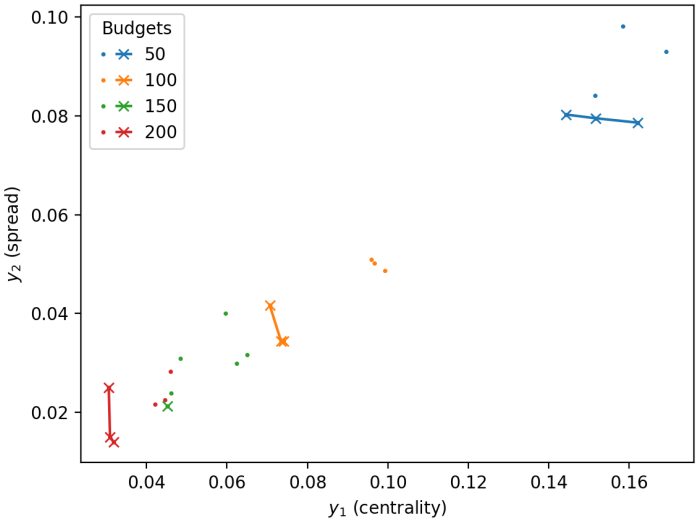
Figure 5: The left plot shows the objective values (y_1, y_2) of each configurations considered for NSGA-II per budget, the crosses connected by a line correspond with the non-dominated configurations and the dots with the dominated ones. The table on the right also shows the configurations in the Pareto set and the highlighted rows are the configurations that were selected.

The same experiments were done with the Random Search (RS) algorithm which was used as a baseline.

RS is a very simple algorithm: it takes a random sample the size of the function evaluation budget without repetition from the search space, evaluates each solution in the sample, and finally produces the Pareto front of this sample as the result. This simplicity is also why it makes a good baseline, because if a proposed optimization algorithm is unable to do better than RS it indicates that the extra effort the algorithm has to do compared to RS does not result in observable improvement.

As was done to find the best hyperparameter configuration, the IGD^+ score of each repetition was computed and the median score taken for further comparison. These values are shown in the top plot of Figure 7 for the experiment using the Bessel search space on the AMD Instinct MI250X with 3 objectives. To further determine how much better NSGA-II and NSGA-III did than RS for a certain budget their median score for each budget was subtracted from RS's median and then divided the latter. This value can be interpreted as representing how much better the solutions found by the algorithms are relative to the quality of the solutions found by RS. The plots for the rest of the experiments can be found in Appendix A.3.

These plots and Table 2 show that RS, on average, has similar performance as NSGA-II or NSGA-III and sometimes does better than the latter when the evaluation budget is 50, the lowest budget that was tested with, while at 100 evaluations NSGA-II and NSGA-III seem to consistently do better than RS, with them both being better in 35/36 cases or 97% with the notable exception Figure 7. It is from around a 150 evaluations that the algorithms start always outperform RS. The similar performance



Budget	Pop size	P(muta)	y_1	y_2
50	20	0.20	0.144258	0.080259
50	40	0.10	0.151696	0.079509
50	40	0.05	0.162116	0.078624
100	20	0.20	0.070623	0.041672
100	20	0.10	0.073415	0.034445
100	20	0.05	0.074109	0.034436
150	20	0.05	0.045112	0.021210
200	20	0.05	0.030591	0.025012
200	20	0.20	0.030916	0.014982
200	20	0.10	0.031789	0.014012

Figure 6: The same type of plot and table as shown in Figure 5 for NSGA-III.

Budget	**NSGA-II (SD)**		**NSGA-III (SD)**	
50	11.19	(± 19.46)	1.23	(± 13.64)
100	34.60	(± 15.16)	34.36	(± 15.90)
150	49.87	(± 15.07)	52.95	(± 17.01)
200	59.76	(± 17.45)	64.29	(± 14.21)

Table 2: The mean and standard deviation (SD) of the percentage median improvements of NSGA-II and NSGA-III over RS for the IGD^+ score obtained for the solutions sets they produced per evaluation budget that was tested with.

to RS for a budget of 50 can be explained by the fact that NSGA-II and NSGA-III have not yet had a change to have multiple round of evolution before the budget is already exhausted. When the budget is 100 a population of size 20 has 4 evolution rounds, so this is where the effects algorithms really start to show, which is also reflected in the results. This trend continues and by 150 evaluations there are no more problems on which RS does better than NSGA-II or NSGA-III.

It can also be observed in the plots and summarized in Table 3 that the difference between NSGA-II and NSGA-III is minor and that NSGA-III does not do consistently better than NSGA-II even on the ”many-objective” tuning problems. I suspect that this is because 5 objectives is too few for NSGA-II to really start to suffer from the problems that come with objective spaces that have high dimensionality and for NSGA-III to start showing its adaptations for these types of problems, but it requires further research to answer this question with certainty. NSGA-III does however seem to have a tendency to outperform NSGA-II as the budget exceeds 150.

Next the time the algorithms require to tune the kernels was investigated by looking at the number of evaluations and the time the algorithms spent benchmarking,

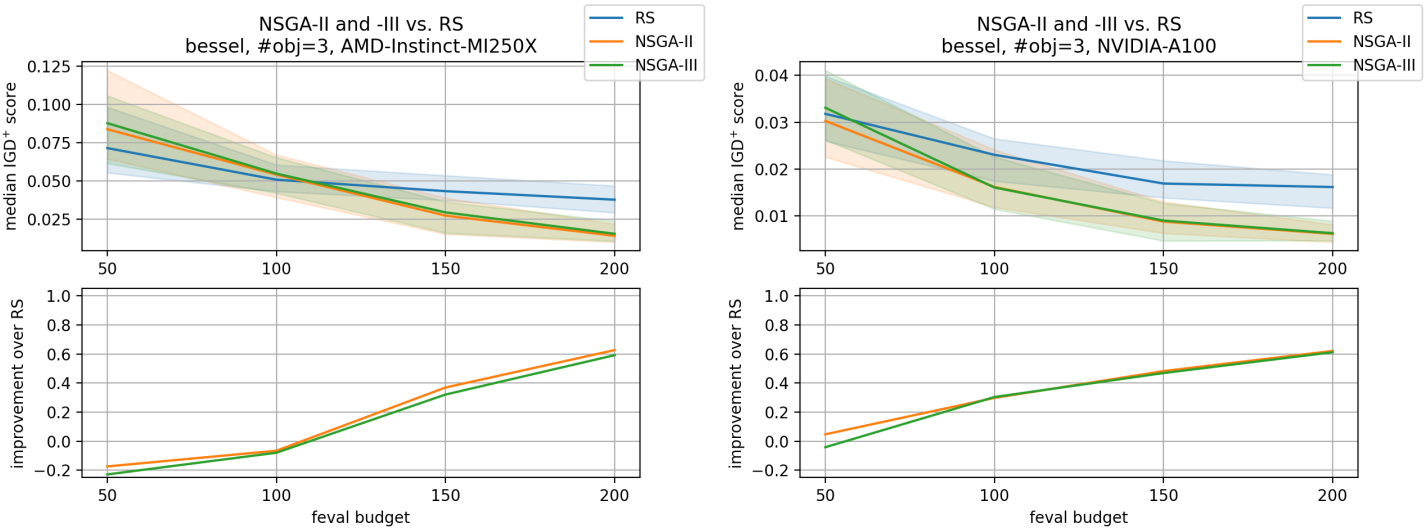


Figure 7: The top shows the median IGD^+ score per algorithm and budget with the shaded region displaying the range between the first and third quartile (i.e. the IQR). The bottom plot show the relative improvement of NSGA-II and NSGA-III over RS, $y = 0$ indicates the same performance as RS and $y = 1$ corresponds to obtaining a median score of 0.

verifying the results of, and compiling the kernels, which I will from now on refer to as "evaluation time" (Kernel Tuner calls it "simulated time"). First I looked at how much faster NSGA-II and NSGA-III are than RS to obtain a certain IGD^+ score and then if any speedup in the evaluation time was obtained for the same number evaluation budget.

To assess how much faster NSGA-II and NSGA-III are than RS in reaching a certain IGD^+ score I looked at how many variants they needed to evaluate to obtain a median score that was at least as good as RS achieved when using the full 200 evaluations. Using this information I was able to compute the amount of evaluation time both algorithm needed to get this score with which I was able to determine the speedup over RS. On average both were twice as fast as RS in reaching the RS's median score at 200 evaluations, with NSGA-II having a 122% ($\pm$ 43%) and NSGA-III a 117% ($\pm$ 34%) speedup over RS. The average speedup the algorithms have per kernel is shown in Table 4. In absolute terms NSGA-II can speedup the tuning process between 1:09 and 8:10 minutes and NSGA-III between 1:19 and 7:16 minutes on these kernels. These results clearly show that NSGA-II and NSGA-III are capable of cutting the time that it requires RS to achieve a certain quality score in half, which would improve the experience of engineers using these tools in their development pipeline.

To assess the amount of time the algorithms spend tuning the kernel for certain budgets the total evaluation time of each run was analyzed. The results of this analysis are succinctly displayed in Table 5, these improvements might look minor at first, especially with their large standard deviation values, but when it is taken into account that RS would have spent anywhere from 2 minutes and 15 seconds to 11 minutes and 14 second on evaluating kernels for a budget of 200 an improvement of about 10% is quite welcome, especially if auto-tuning is to become an integral part of

Budget	No. objectives 3 & 5 (SD)		5 (SD)	
50	−15.01	(± 24.56)	−17.24	(± 26.27)
100	−2.19	(± 21.64)	−0.11	(± 18.25)
150	5.16	(± 28.17)	6.49	(± 30.51)
200	9.89	(± 27.04)	9.55	(± 28.69)

Table 3: The mean and standard deviation (SD) of the percentage median improvements of NSGA-III over NSGA-II for the IGD^+ score obtained for the solutions sets they produced per evaluation budget that was tested with. The **3 & 5** column shows these values for the scores that were obtained when the algorithms were run with each objective set and the **5** column only the scores when they were run with the 5 objective set.

Kernel	NSGA-II (SD)		NSGA-III (SD)	
Bessel	121.86	(± 9.65)	109.89	(± 15.31)
Convolution2D	137.98	(± 42.63)	108.48	(± 39.09)
K-Means	94.70	(± 49.91)	114.50	(± 42.04)
LavaMD	137.95	(± 32.58)	129.11	(± 32.81)
InvK	96.26	(± 29.44)	89.02	(± 23.78)
Newton-Raphson	137.19	(± 17.38)	148.62	(± 6.85)
Coulombic	117.43	(± 58.79)	105.11	(± 23.23)
MRI-Q	107.22	(± 30.95)	111.13	(± 28.48)
Black-Scholes	147.13	(± 80.46)	140.78	(± 56.18)

Table 4: The mean and standard deviation (SD) of the percentage speedup NSGA-II and NSGA-III have over RS per kernel when they have to read the same score as RS is able to achieve with a budget of 200 evaluations.

Budget	NSGA-II (SD)		NSGA-III (SD)	
50	2.25	(± 6.97)	0.30	(± 3.59)
100	8.78	(± 7.66)	7.82	(± 6.92)
150	9.87	(± 7.86)	9.93	(± 7.75)
200	11.94	(± 9.19)	11.15	(± 7.84)

Table 5: The mean and standard deviation (SD) of the median percentage improvement of NSGA-II and NSGA-III over RS for the time spent benchmarking, verifying the results of, and compiling the GPU kernels per evaluation budget.

the software development pipeline. It should be noted however that this most likely a result of time being one of the objectives that is being minimized during tuning, so it should not be surprising that NSGA-II and NSGA-III spend less time evaluating kernels as their populations evolve to contain solutions that are faster while RS takes a random sample and does not make any decisions to direct the optimization process. For SOO this would mean that we can only have these benefits if time is the objective being tuned for, but MOO does not have this limitation, with it the time objective could always be added purely to speed up the tuning process, even if it is not one of the objectives of interest. This does however mean a part of the tuning budget will consequently be used to try and explore the direction in the space of solutions that corresponds with improvements in the time objective, but with MOO it is at least an option that can be considered, unlike SOO.

6 Conclusion

In this thesis, I describe the design and implementation of an extension to the auto-tuning framework Kernel Tuner that makes it capable of doing multi-objective optimization. The extension makes the NSGA-II and NSGA-III available which allows developers to tune their GPU kernels for multiple objectives simultaneously without requiring any form of scalarization of the objective functions. This makes a more hands-off approach possible because no knowledge of the underlying search space is required to tune multiple objectives, which is not the case when multiple objectives are scalarized so they can work with a single-objective optimization algorithm.

To evaluate the quality of the solution sets produced by NSGA-II and NSGA-III was then analyzed by comparing them to Random Search for specific evaluation budgets. To do this analysis the best hyperparameter values for the algorithms were first determined from a limited set of options. Using these hyperparameter values the algorithms were applied to the pre-evaluated search space of nine kernels and two devices, making for a total of 18 spaces used for the evaluation. Because the search spaces are pre-evaluated the true Pareto fronts are known, which made it possible to use the IGD^+ metric to score the quality of the solutions found. These scores were then compared to the ones obtained by Random Search on the same problems and evaluation budgets. This test showed that for lower evaluation budgets NSGA-II and NSGA-III are about tie with Random Search, but starting from a budget of around 100 function evaluations they start to consistently outperform it and by 150 evaluations this is always the case. This leads to the conclusion that the algorithms need at least 4 round of evolution to consistently outperform Random Search with the hyperparameter configurations that were used, further research is however required

to find the best default hyperparameter values so that the algorithms can be broadly applicable without needing much effort on the user's side. The amount of time the algorithms spend evaluating kernels was also analyzed. The first test showed that NSGA-II and NSGA-III on average only need half the time RS requires to obtain a median IGD^+ score that RS obtains using 200 evaluations and the second test led to the interesting conclusion that MOO makes it possible to add the minimization of time as one of the objectives purely to reduce the time spend on auto-tuning.

This work makes multi-objective auto-tuning more easily accessible to a wider audience by being implemented in an extensible framework written in the Python programming language. Future work could, as previously mentioned, make more of the hyperparameters NSGA-II and NSGA-III have accessible to the user, explore better default settings for the algorithms, or compare their performance to different multi-objective optimization algorithm when applied to auto-tuning GPU kernels. How simultaneous multi-objective optimization as done in this work, compares to iteratively tuning for a single objective using conventional single-objective optimization algorithms also needs to be investigated. It is in any case my hope that this work will allow for this field to gain a broader audience by allowing user to more directly encode their tuning goal by using multiple objectives.

References

[1] Ben van Werkhoven, Willem Jan Palenstijn, and Alessio Sclocco, "Lessons learned in a decade of research software engineering gpu applications," in *International Conference on Computational Science, ICCS 2020*, Springer International Publishing, Jan. 2020, pp. 399–412, ISBN: 9783030504366. DOI: 10.1007/978-3-030-50436-6_29.

[2] Sclocco, Alessio, Bal, Henri E., Hessels, Jason, Leeuwen, Joeri van, and Nieuwpoort, Rob V. van, "Auto-tuning dedispersion for many-core accelerators," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 952–961. DOI: 10.1109/IPDPS.2014.101.

[3] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019. DOI: https://doi.org/10.1016/j.future.2018.08.004.

[4] K. Deb, K. Miettinen, and S. Chaudhuri, *An estimation of nadir objective vector using a hybrid evolutionary-cum-local-search procedure*, 2009.

[5] Balaprakash, Prasanna, Dongarra, Jack, Gamblin, Todd, *et al.*, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018. DOI: 10.1109/JPROC.2018.2841200.

[6] J. Durillo and T. Fahringer, "From single- to multi-objective auto-tuning of programs: Advantages and implications," *Scientific Programming*, vol. 22, no. 4, p. 818 579, 2014. DOI: https://doi.org/10.3233/SPR-140394. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.3233/SPR-140394. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.3233/SPR-140394.

[7] Schoonhoven, Richard, van Werkhoven, Ben, and Batenburg, K Joost, "Benchmarking optimization algorithms for auto-tuning gpu kernels," *IEEE Transactions on Evolutionary Computation*, 2022.

[8] *Kernel tuner documentation*, online. [Online]. Available: https://kerneltuner.github.io/.

[9] J. Ansel, S. Kamil, K. Veeramachaneni, *et al.*, "Opentuner: An extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Canada, Aug. 2014. [Online]. Available: https://commit.csail.mit.edu/papers/2014/ansel-pact14-opentuner.pdf.

[10] A. Rasch, M. Haidl, and S. Gorlatch, "Atf: A generic auto-tuning framework," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec. 2017, pp. 64–71. DOI: 10.1109/HPCC-SmartCity-DSS.2017.9.

[11] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Manycore Systems-on-Chip*, (Turin, Italy), IEEE, Sep. 2015, pp. 195–202. DOI: 10.1109/MCSoC.2015.10.

[12] 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 Generation Computer Systems*, vol. 108, pp. 161–177, 2020, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.02.069. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19327360.

[13] R. S. Chen and J. K. Hollingsworth, "Angel: A hierarchical approach to multi-objective online auto-tuning," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, (Portland, OR, USA), ser. ROSS '15, New York, NY, USA: Association for Computing Machinery, 2015, ISBN: 9781450336062. DOI: 10.1145/2768405.2768409. [Online]. Available: https://doi.org/10.1145/2768405.2768409.

[14] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2019, pp. 347–358.

[15] B. Bodin, L. Nardi, H. Wagstaff, P. H. J. Kelly, and M. O'Boyle, "Algorithmic performance-accuracy trade-off in 3d vision applications," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 123–124. DOI: 10.1109/ISPASS.2018.00024. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS.2018.00024.

[16] S. Cheema and G. Khan, "Gpu auto-tuning framework for optimal performance and power consumption," in *Proceedings of the 15th Workshop on General Purpose Processing Using GPU*, ser. GPGPU '23, Montreal, Canada: Association for Computing Machinery, 2023, pp. 1–6, ISBN: 9798400707766. DOI: 10.1145/3589236.3589241.

[17] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, Dec. 2007, ISSN: 1941-0026. DOI: 10.1109/TEVC.2007.892759.

[18] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, "A reference vector guided evolutionary algorithm for many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 773–791, 2016. DOI: 10.1109/TEVC.2016.2519378.

[19] N. Srinivas and K. Deb, "Muiltiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, Sep. 1994, ISSN: 1063-6560. DOI: 10.1162/evco.1994.2.3.221.

[20] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T., "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002. DOI: 10.1109/4235.996017.

[21] Deb, Kalyanmoy and Jain, Himanshu, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014. DOI: 10.1109/TEVC.2013.2281535.

[22] T. Blickle and L. Thiele, "A mathematical analysis of tournament selection," in *Proceedings of the 6th International Conference on Genetic Algorithms*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 9–16, ISBN: 1558603700.

[23] G. Syswerda, "Uniform crossover in genetic algorithms," Jan. 1989.

[24] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, Apr. 1992, ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001. [Online]. Available: https://doi.org/10.7551/mitpress/1090.001.0001.

[25] N. A. Rashed, Y. H. Ali, T. A. Rashid, and A. Salih, *Unraveling the versatility and impact of multi-objective optimization: Algorithms, applications, and trends for solving complex real-world problems*, 2024. arXiv: 2407.08754 [cs.NE]. [Online]. Available: https://arxiv.org/abs/2407.08754.

[26] I. Das and J. E. Dennis, "Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems," *SIAM Journal on Optimization*, vol. 8, no. 3, pp. 631–657, 1998. DOI: 10.1137/S1052623496307510. eprint: https://doi.org/10.1137/S1052623496307510. [Online]. Available: https://doi.org/10.1137/S1052623496307510.

[27] J. Blank, K. Deb, Y. Dhebar, S. Bandaru, and H. Seada, "Generating well-spaced points on a unit simplex for evolutionary many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 1, pp. 48–60, Feb. 2021, ISSN: 1941-0026. DOI: 10.1109/TEVC.2020.2992387.

[28] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020. DOI: 10.1109/ACCESS.2020.2990567.

[29] S. Heldens and B. van Werkhoven, "Kernel float: Unlocking mixed-precision gpu programming," *ACM Transactions on Mathematical Software*, Dec. 2025, Just Accepted, ISSN: 0098-3500. DOI: 10.1145/3779120. [Online]. Available: https://doi.org/10.1145/3779120.

[30] H. Bal, D. Epema, C. de Laat, *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016. DOI: 10.1109/MC.2016.127.

[31] L. consortium., *Lumi supercomputer*. 2024. [Online]. Available: https://www.lumi-supercomputer.eu/.

[32] E. Zitzler and L. Thiele, "Multiobjective optimization using evolutionary algorithms — a comparative case study," in *Parallel Problem Solving from Nature — PPSN V*, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 292–301, ISBN: 978-3-540-49672-4.

[33] D. A. Van Veldhuizen, "Multiobjective evolutionary algorithms: Classifications, analyses, and new innovations," AAI9928483, Ph.D. dissertation, USA, 1999, ISBN: 0599283165.

[34] C. A. Coello Coello and M. Reyes Sierra, "A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm," in *MICAI 2004: Advances in Artificial Intelligence*, R. Monroy, G. Arroyo-Figueroa, L. E. Sucar, and H. Sossa, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 688–697, ISBN: 978-3-540-24694-7.

[35] H. Ishibuchi, H. Masuda, Y. Tanigaki, and Y. Nojima, "Modified distance calculation in generational distance and inverted generational distance," in *Evolutionary Multi-Criterion Optimization*, A. Gaspar-Cunha, C. Henggeler Antunes, and C. C. Coello, Eds., Cham: Springer International Publishing, 2015, pp. 110–125, ISBN: 978-3-319-15892-1.

[36] J. G. Falcón-Cardona, S. Zapotecas-Martínez, and A. García-Nájera, "Pareto compliance from a practical point of view," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '21, Lille, France: Association for Computing Machinery, 2021, pp. 395–402, ISBN: 9781450383509. DOI: 10.1145/3449639.3459276. [Online]. Available: https://doi.org/10.1145/3449639.3459276.

[37] H. Ishibuchi, R. Imada, N. Masuyama, and Y. Nojima, "Comparison of hypervolume, igd and igd+ from the viewpoint of optimal distributions of solutions," in *Evolutionary Multi-Criterion Optimization: 10th International Conference, EMO 2019, East Lansing, MI, USA, March 10-13, 2019, Proceedings*, East Lansing, MI, USA: Springer-Verlag, 2019, pp. 332–345, ISBN: 978-3-030-12597-4. DOI: 10.1007/978-3-030-12598-1_27. [Online]. Available: https://doi.org/10.1007/978-3-030-12598-1_27.

[38] J. O. Torring, B. van Werkhoven, F. Petrovc, F.-J. Willemsen, J. Filipovic, and A. C. Elster, " Towards a Benchmarking Suite for Kernel Tuners," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 724–733. DOI: 10.1109/IPDPSW59300.2023.00124. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW59300.2023.00124.

[39] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[40] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design Test*, vol. 34, no. 2, pp. 60–68, 2017. DOI: 10.1109/MDAT.2016.2630270.

[41] J. Stratton, C. Rodrigues, I. Sung, *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[42] NVIDIA Corporation, *CUDA Samples*. [Online]. Available: https://github.com/NVIDIA/cuda-samples.

A Appendices

A.1 Appendix: The hyperparameters of the algorithms

	Budgets			
NSGA-II	50	100	150	200
Population size	20			
Crossover method	TWO-POINT-CROSSOVER			
Crossover probability	1.0			
Mutation method	*custom*			
Mutation probability	0.2	0.05	0.2	0.2
NSGA-III	50	100	150	200
Population size	40	20	20	20
Crossover method	TWO-POINT-CROSSOVER			
Crossover probability	1.0			
Mutation method	*custom*			
Mutation probability	0.1	0.1	0.05	0.2
Reference directions method	Riesz s-Energy			
Number of reference directions	40	20	20	20

Table 6: The selected hyperparameter values per algorithm for each budget.

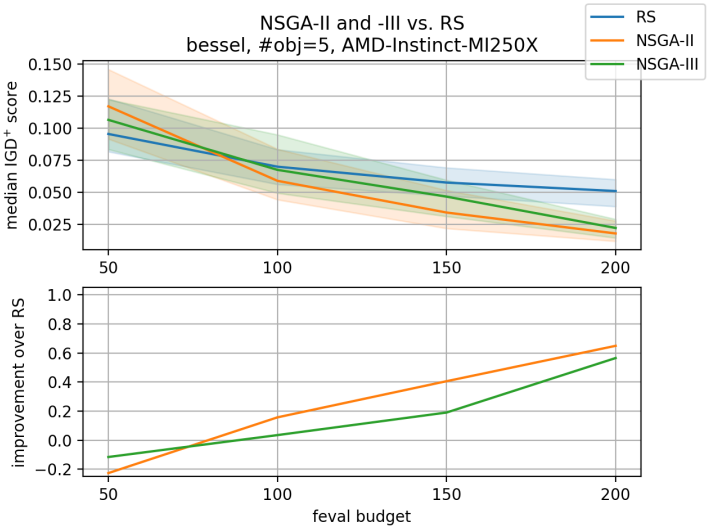
A.2 Appendix: The tunable parameters per GPU kernel

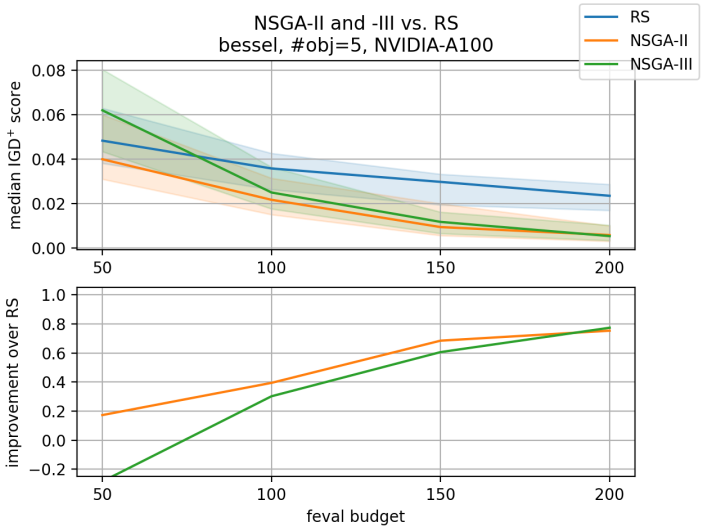
Name	**Field**	**Source**	**Error Metric**	**No. params**	**No. configs**
Bessel	Mathematical Physics	Kernel Float [29]	MRE	8	15,120
Convolution2D	Image Processing	BAT [38]	NRMSE	11	72,112
K-Means	Machine Learning	Rodinia [39]	MRE	6	3,328
LavaMD	Molecular Dynamics	Rodinia [39]	NMAE	8	14,896
InvK (Inverse Kinematic)	Robotics	AxBench [40]	NMAE	7	4,641
Newton-Raphson	Numerical Analysis	AxBench [40]	NRMSE	8	5,572
Coulombic	Electrical Engineering	Parboil [41]	MRE	8	20,736
MRI-Q	Medical	Parboil [41]	NRMSE	8	29,916
Black-Scholes	Finance	CUDA Samples [42]	NRMSE	7	6,383

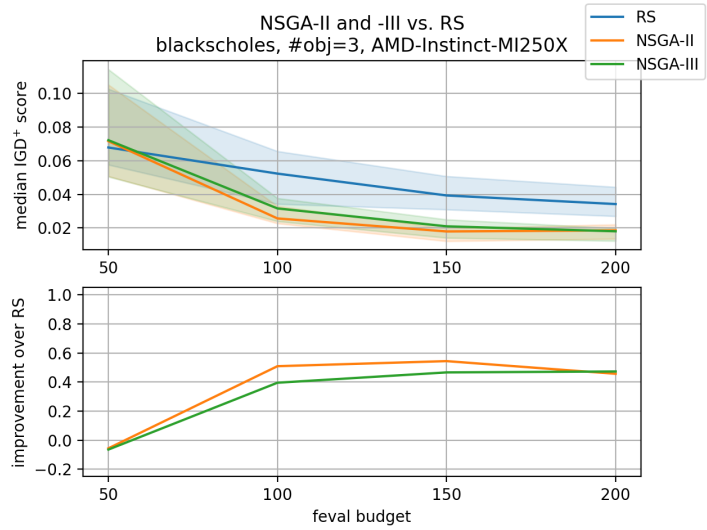
Table 7: Benchmarks used for performance evaluation. The exact number of valid configurations depends on hardware capabilities, numbers shown for Nvidia A100.

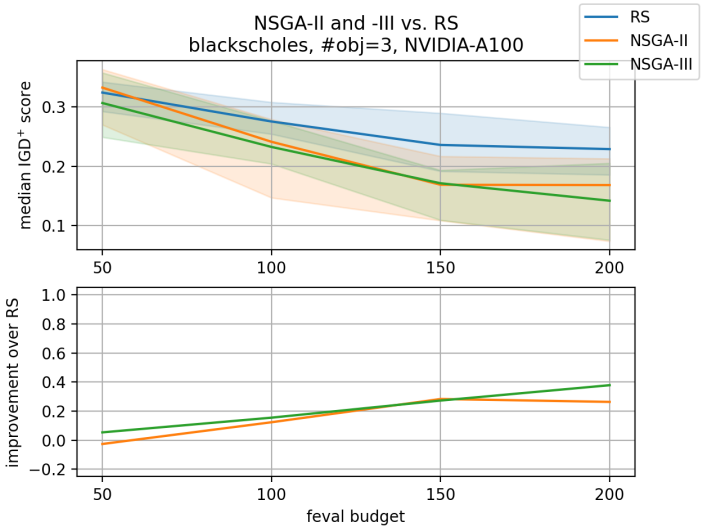
A.3 Appendix: The scores compared to random search

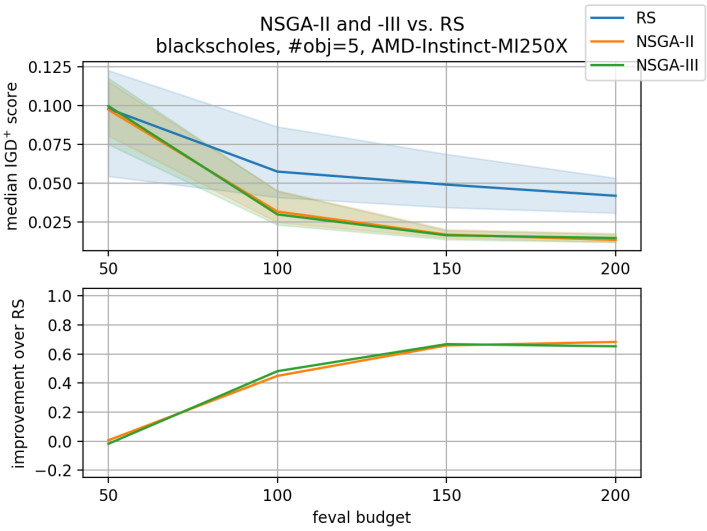
Refer to Figure 7 to understand how to read the plots.

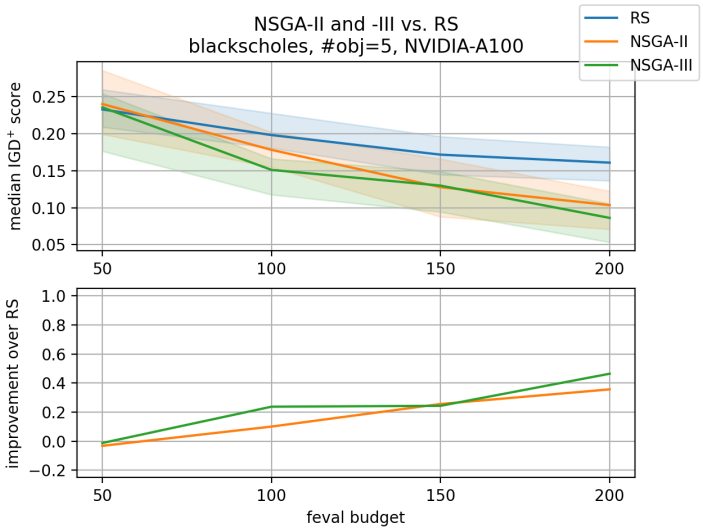

NSGA-II and -III vs. RS
bessel, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

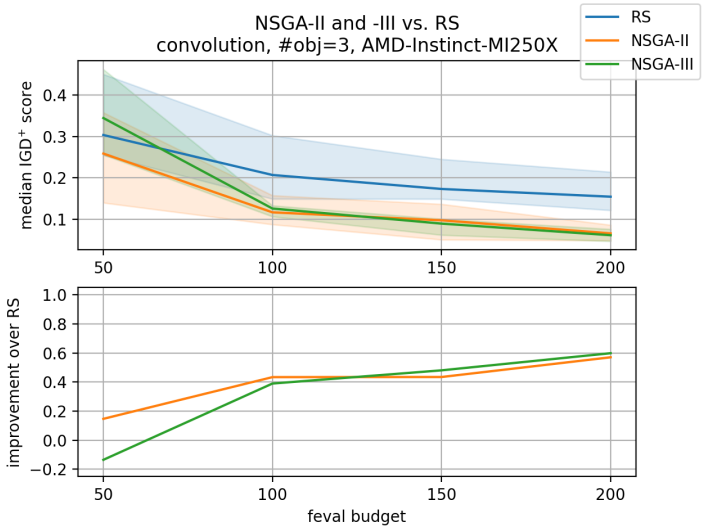

NSGA-II and -III vs. RS
bessel, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

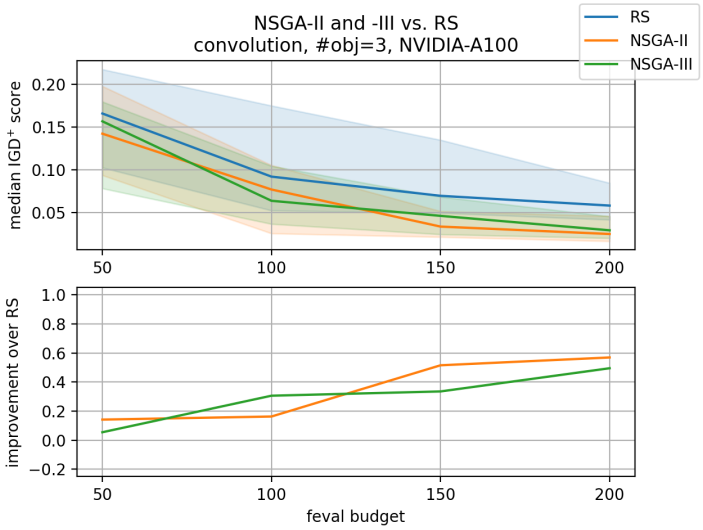

NSGA-II and -III vs. RS
blackscholes, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

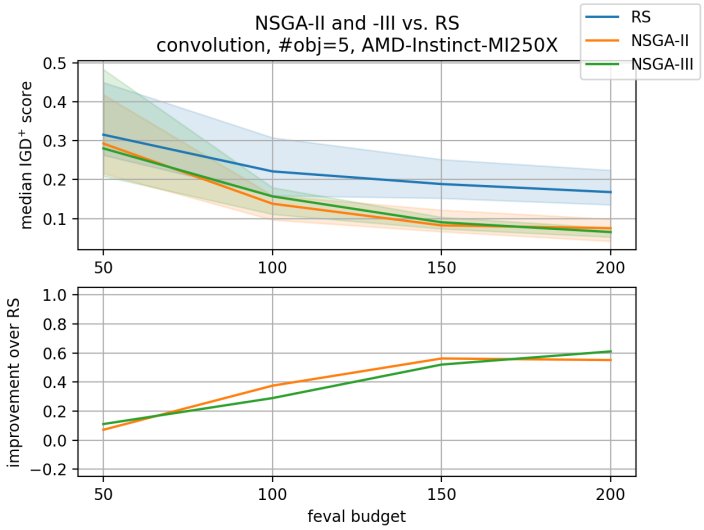

NSGA-II and -III vs. RS
blackscholes, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

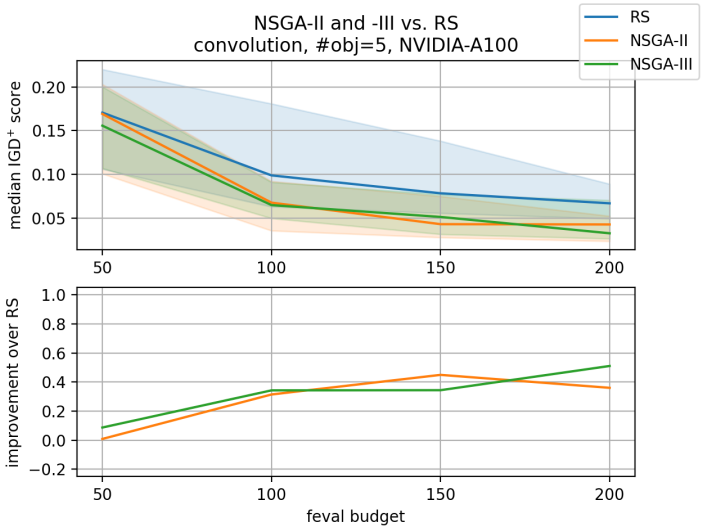

NSGA-II and -III vs. RS
blackscholes, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

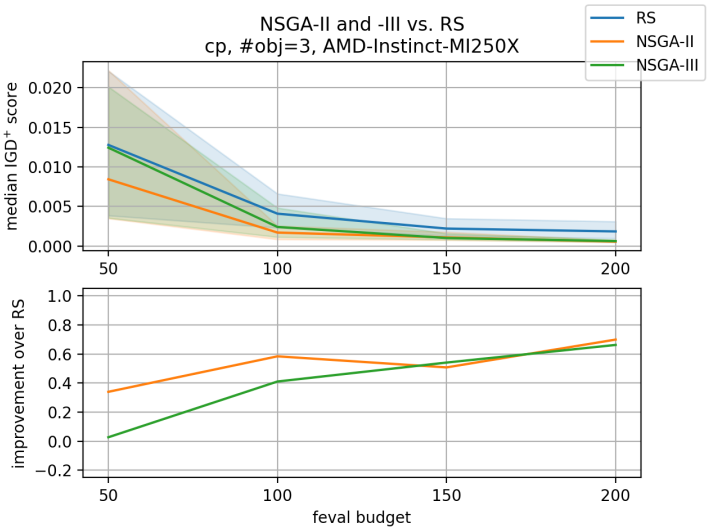

NSGA-II and -III vs. RS
blackscholes, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

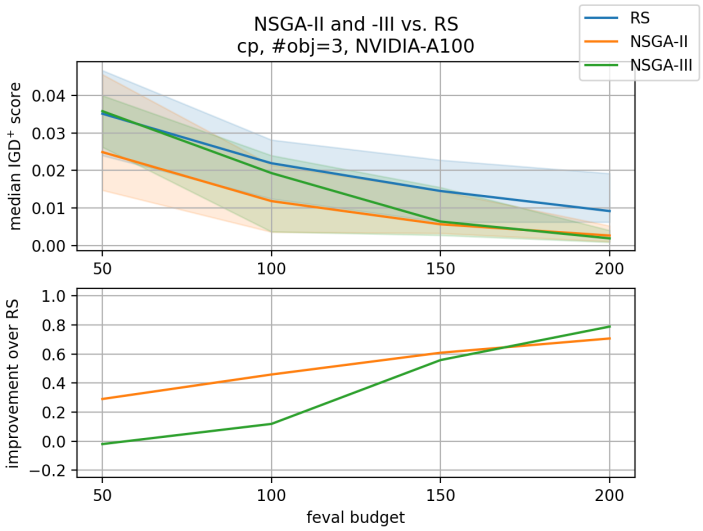

NSGA-II and -III vs. RS
convolution, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

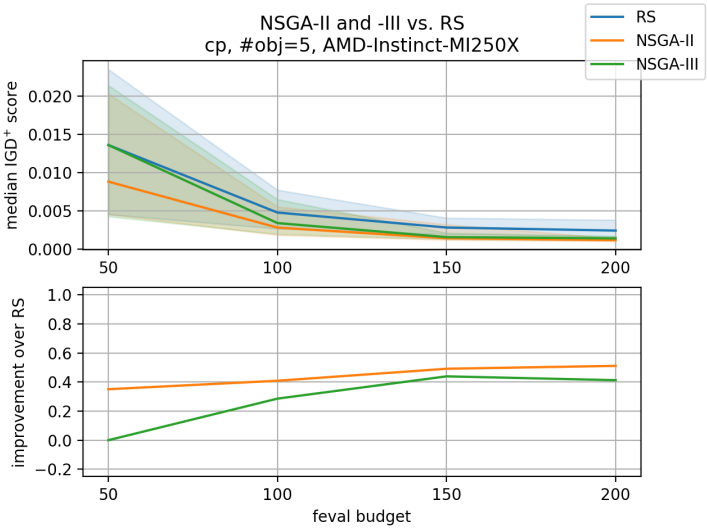

NSGA-II and -III vs. RS
convolution, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

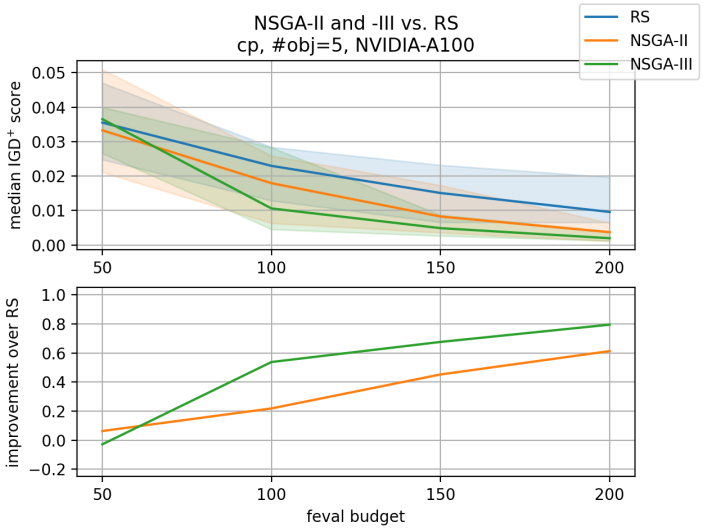

NSGA-II and -III vs. RS
convolution, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

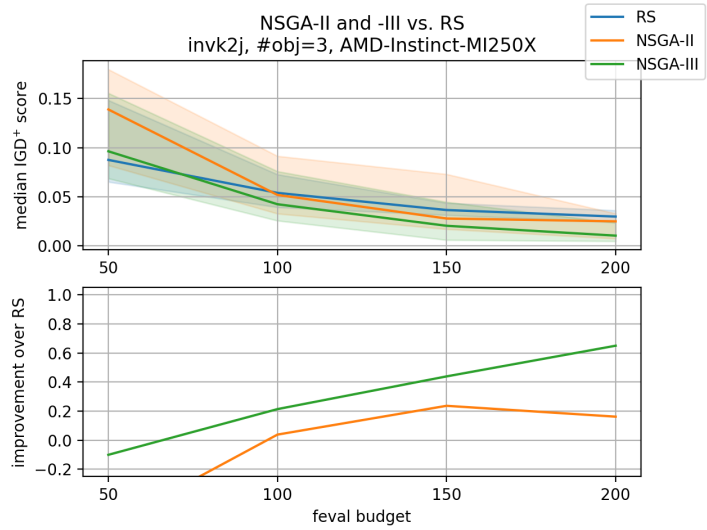

NSGA-II and -III vs. RS
convolution, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

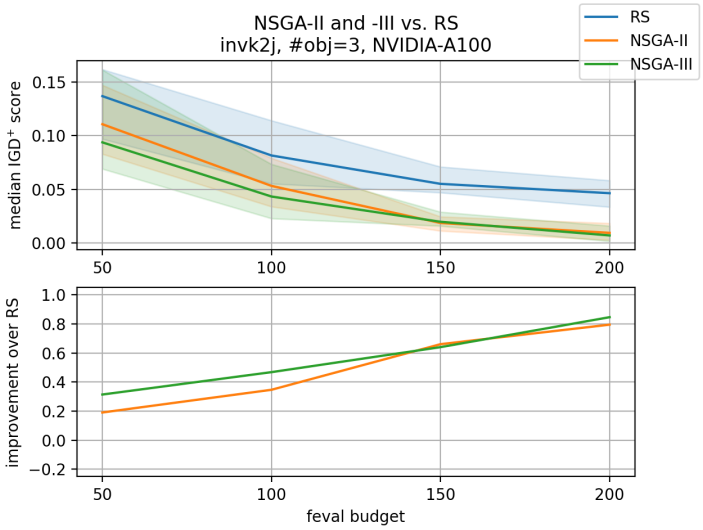

NSGA-II and -III vs. RS
cp, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

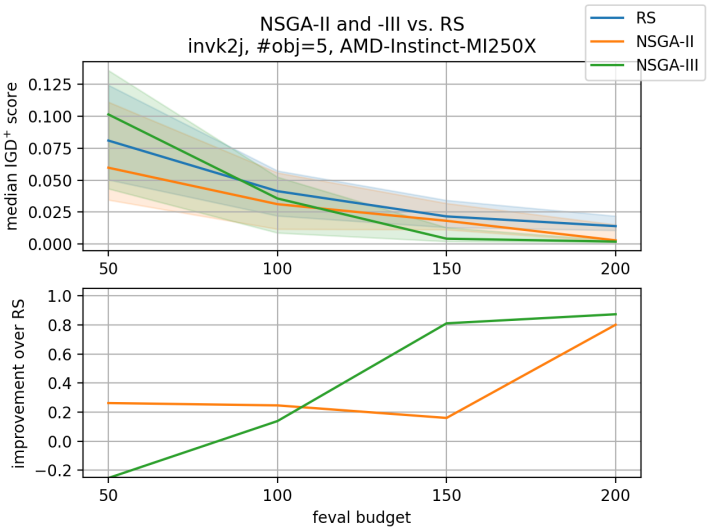

NSGA-II and -III vs. RS
cp, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

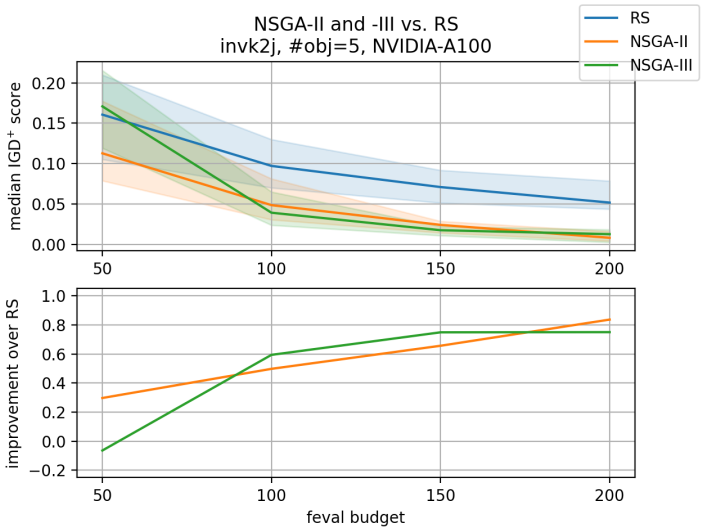

NSGA-II and -III vs. RS
cp, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

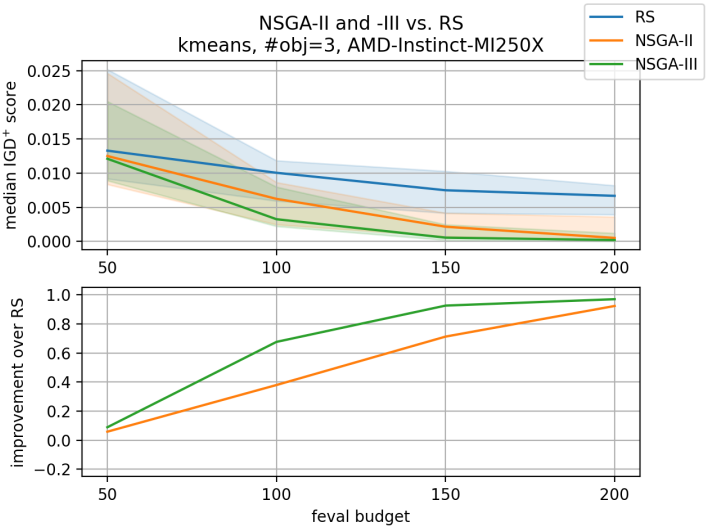

NSGA-II and -III vs. RS
cp, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

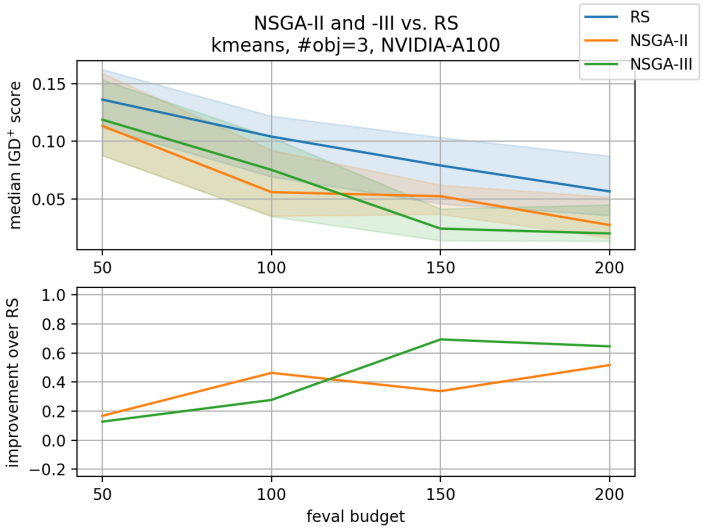

NSGA-II and -III vs. RS
invk2j, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

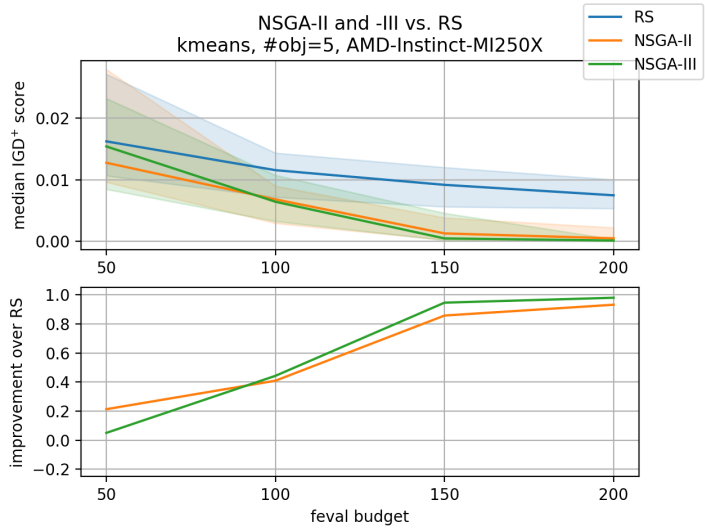

NSGA-II and -III vs. RS
invk2j, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

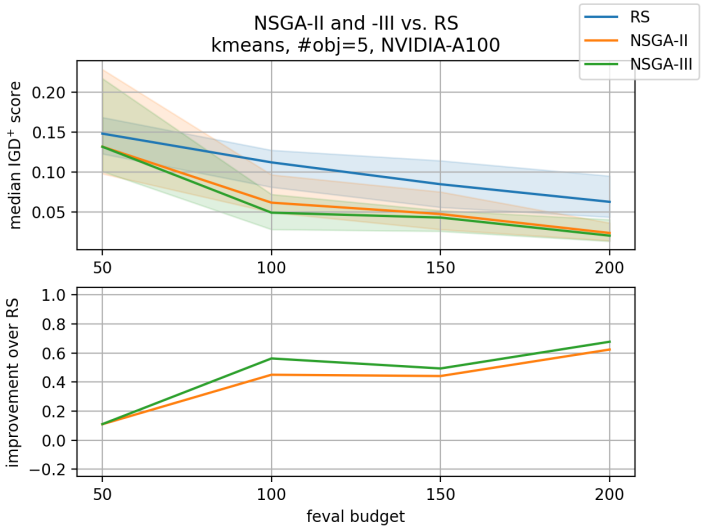

NSGA-II and -III vs. RS
invk2j, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

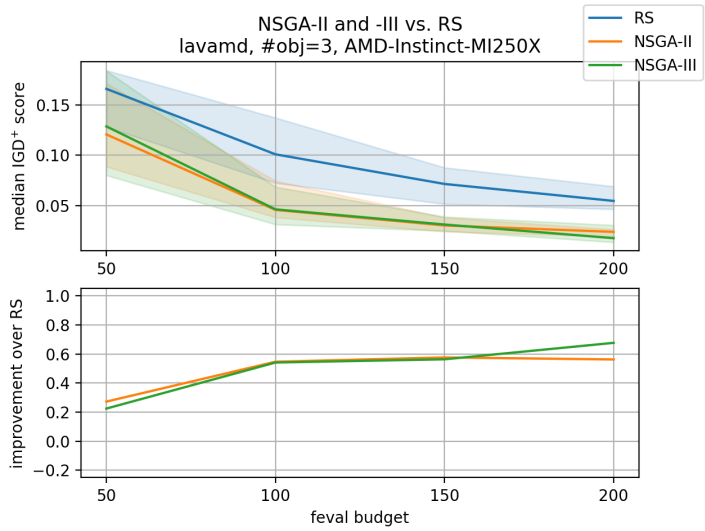

NSGA-II and -III vs. RS
invk2j, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

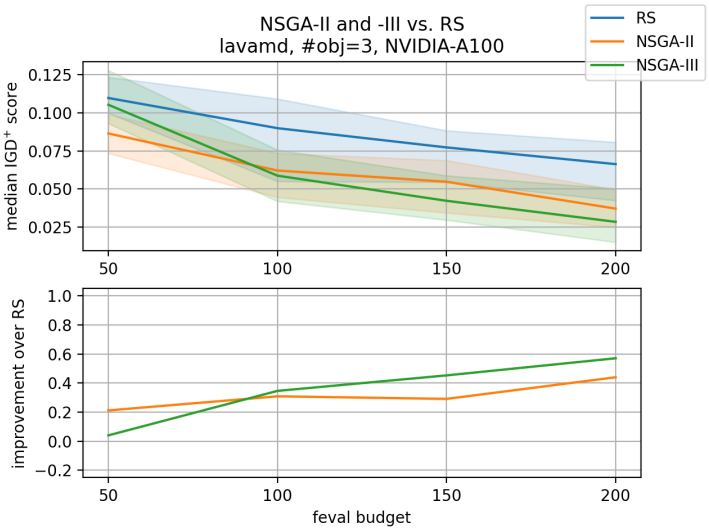

NSGA-II and -III vs. RS
kmeans, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

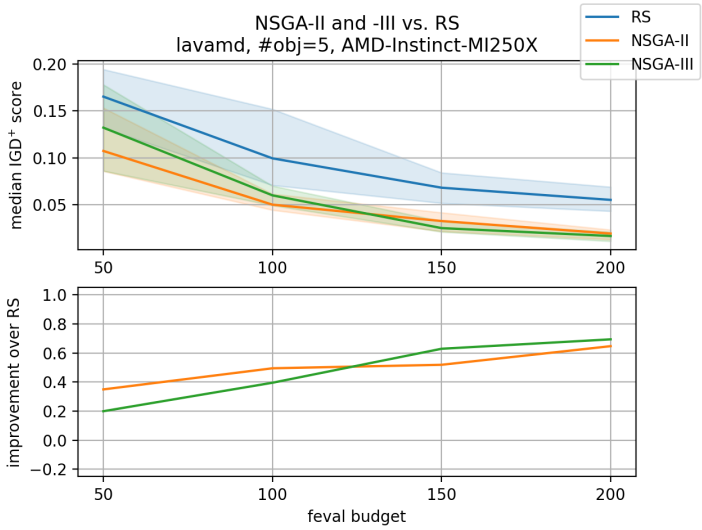

NSGA-II and -III vs. RS
kmeans, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

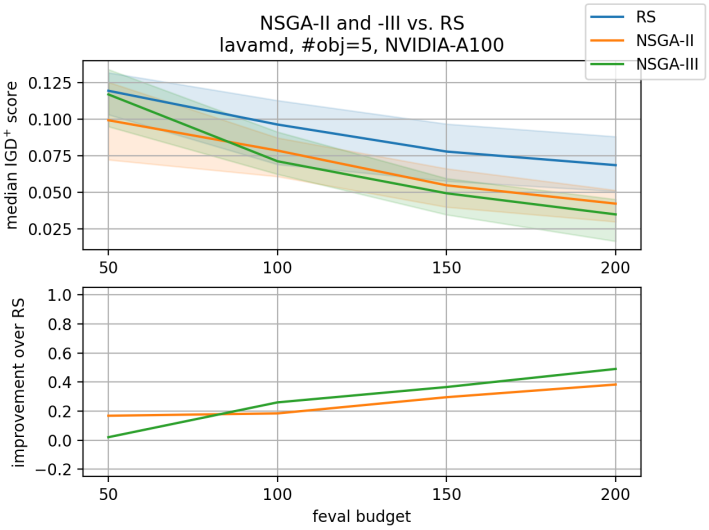

NSGA-II and -III vs. RS
kmeans, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

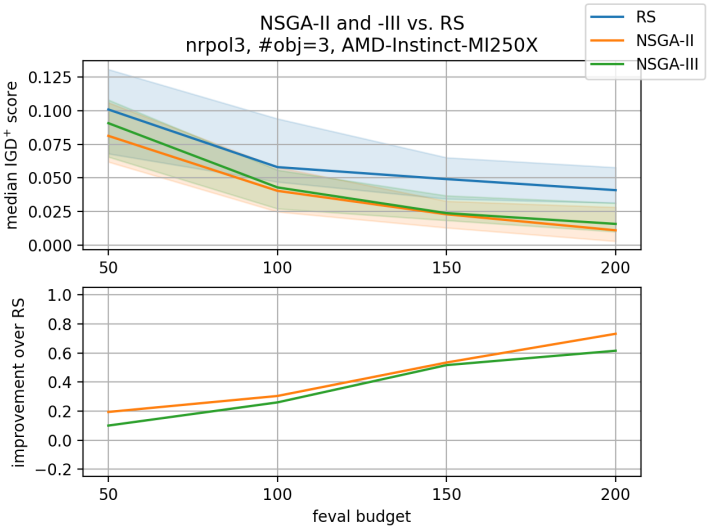

NSGA-II and -III vs. RS
kmeans, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

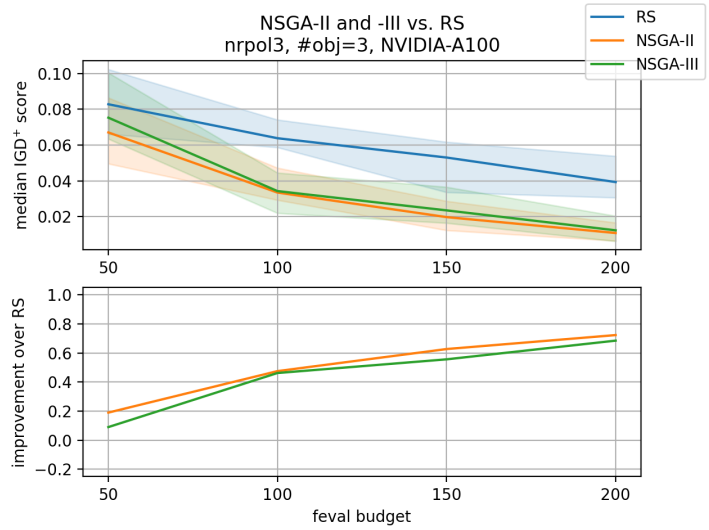

NSGA-II and -III vs. RS
lavamd, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

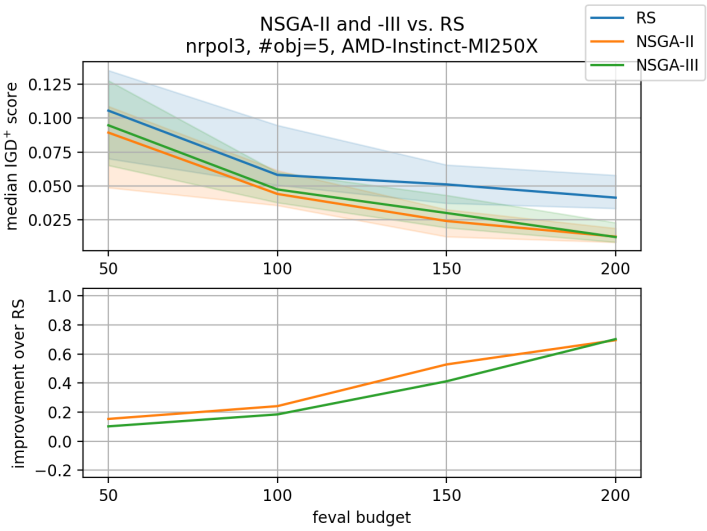

NSGA-II and -III vs. RS
lavamd, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

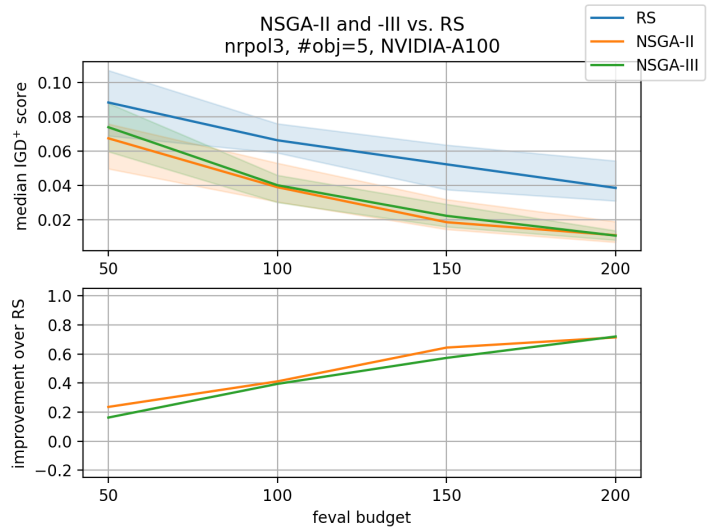

NSGA-II and -III vs. RS
lavamd, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

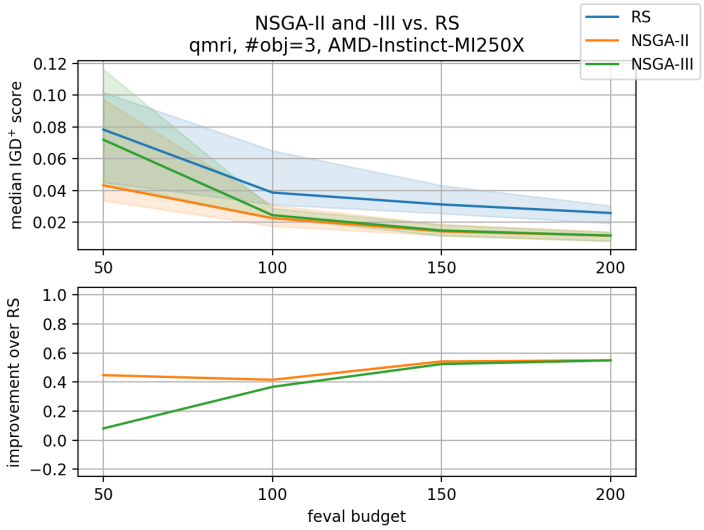

NSGA-II and -III vs. RS
lavamd, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

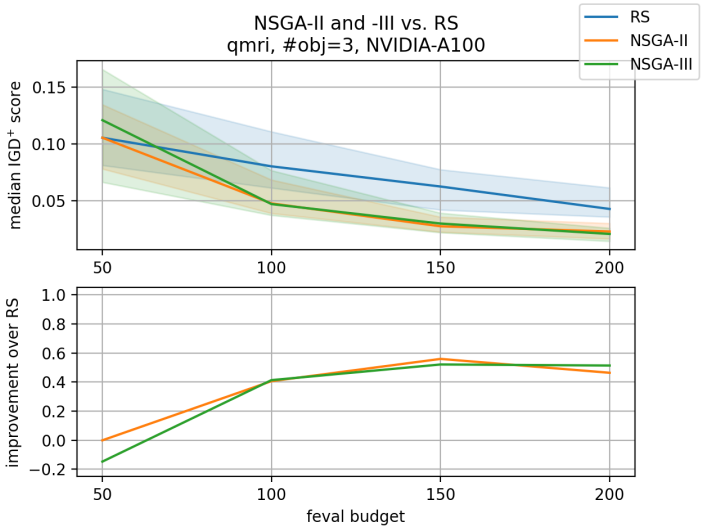

NSGA-II and -III vs. RS
nrpol3, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

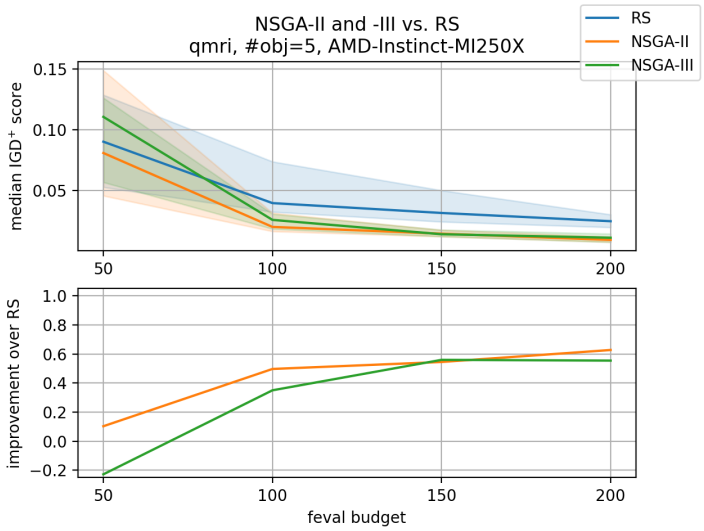

NSGA-II and -III vs. RS
nrpol3, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget

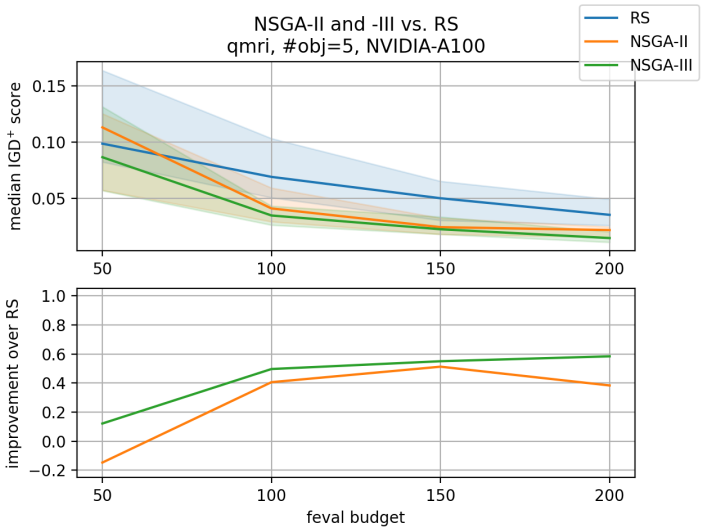

NSGA-II and -III vs. RS
nrpol3, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget


NSGA-II and -III vs. RS
nrpol3, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget


NSGA-II and -III vs. RS
qmri, #obj=3, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget


NSGA-II and -III vs. RS
qmri, #obj=3, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget


NSGA-II and -III vs. RS
qmri, #obj=5, AMD-Instinct-MI250X
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget


NSGA-II and -III vs. RS
qmri, #obj=5, NVIDIA-A100
RS
NSGA-II
NSGA-III
median IGD⁺ score
improvement over RS
feval budget