UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor's project

Aleksander Junge

Reactive Benchmarking

Benchmarking in the Futhark compiler

Date: June 13 2022

Advisor: Troels Henriksen

Faculty: Faculty of Science
Institute: Department of Computer Science
Author: Aleksander Junge
Title: Reactive Benchmarking
Advisor: Troels Henriksen
Date: 13-06-2022

1 Abstract

This project describes a new benchmarking technique that automatically stops benchmarking when the collected data is deemed sufficient to provide a reliable measurement. We implement this in the context of *Futhark*. This automation is useful as it frees up time for the developer who would otherwise have to adjust the number of iterations to match the execution time of the program in question. The proposed solution consists of two phases; A first phase runs the benchmark for half a second. Afterwards, a second phase calculates statistics of the collected data, and keeps running more iterations until the result is deemed reliable. The new tool was evaluated by comparing it with manually specified iteration counts on a large benchmark suite, and was found to reach a reliable result faster than its old counterpart, as it more evenly splits its time between programs of short and long duration. The new tool provides a mean $\overline{x}$ that on average varies with a relative standard deviation of 3.8% between repetitions of the same benchmark.

Table of content

2 Introduction

This section will first define what is understood by empirical benchmarking. Secondly, introduce the reader to the *Futhark* language and compiler. Third, present a disambiguation of terminology used in the project, before identifying the scope of the project. Lastly, we will finish off the introduction with an outline of what's to come in the remaining report.

2.1 Empirical Benchmarking

Benchmarking is the practice of measuring the performance of a computer program. The primary metric of interest is usually the execution time, which will allow the developer to compare the relative speed of two implementations of the same program, or calculate other metrics such as the floating-point operations per second (FLOPS).

What are common applications of benchmarking?

- A developer may want to understand the performance effect of a change to his program.
- A developer may want to compare the implementation of a program written in two different languages.
- A programming language researcher may introduce a new algorithm, or optimization in the compilation process and want to see what impact it has on performance.
- Benchmarking may be done to measure the underlying performance of the hardware. E.g. running the same program on two different machine configurations.

Note that for each application, the question to be answered by benchmarking is usually concerned with estimating the performance impact of some change. In fact, Kalibera et al.[1] found that out of 122 papers published in *ASPLOS, ISMM, PLDI, TACO and TOPLAS*, 65 included evaluation seeking to quantify a performance change.

Benchmarking is inherently empirical, as we base our performance estimate on quantitative observations and measurements of the program. Thus I will refer to the person conducting the benchmark as 'experimenter'.

If there were no variation between iterations of a benchmark, we could happily conclude benchmarking after collecting a single measurement, sadly this is not the case. Progams may be non-deterministic, and execution time can vary dramatically. The challenge for the experimenter is how to best address this uncertainty, and provide reliable estimates of performance

in a reasonable amount of time, without compromising the validity of the results.

Thus benchmarking is not a trivial feat for a user with limited knowledge of statistics. The goal of this project is in part to automate the work for the user, such that less statistical knowledge is required to obtain and interpret benchmark results.

2.2 Futhark

Futhark is a programming language designed for writing efficient code on massively parallel hardware. It is a data-parallel and purely functional array language, with an optimising compiler that generates either GPU code through *OpenCL* and *CUDA*, or multi-threaded CPU code. It is not intended to replace general purpose languages, but may be easily integrated into projects written in e.g. *C* or *Python*. *Futhark* has a built-in benchmarking tool (henceforth referred to as "$\mathrm{futhark\ bench}$"), which we will describe in greater detail in section 4. For more information on Futhark see [2].

2.3 Terminology

Here we will disambiguate a few terms used throughout the report.

Iteration An iteration is the lowest level at which a benchmark is repeated. It corresponds to running the code to be measured once.

Repetition A repetition of a benchmark, is the act of running the benchmark program e.g. $\mathrm{futhark\ bench}$, which may perform multiple iterations of the code to be measured.

Plateau A group of consecutive benchmark iterations that have a similar performance level. (i.e. different from that of another group)

2.4 Scope

Jones et al.[3] describes benchmarking as a process that involves three kinds of variables affecting the outcome of the benchmark. A controlled variable is a variable that the experimenter is in direct control of (such as the chosen platform or compiler options). A random variable is as the name suggest random (e.g. the number of hardware interrupts during an iteration or the order of scheduling), these random factors usually lead to distributions with a long tail (skewed to the right)[4]. The last kind of variable is an uncontrolled variable which is fixed for the duration of the experiment, (this includes factors such as randomized algorithms used in compilation),

which once the binary is compiled will remain fixed and may introduce a bias.

This project won't include any treatment of uncontrolled variables. As the *Futhark* compiler doesn't use randomness in the compilation process, meaning that compiling the same program on the same machine will always yield the same binary. It is the job of the experimenter to identify possible sources of bias.

We will consider controlled variables as part of the benchmark. Thus, when we benchmark a program, it is not only the code to be run which is being measured, but also implicitly its input size and the options with which it is run. Together these factors provide a framework that should remain constant during benchmarking.

The last variable that concerns us is the random variable. It is in the face of randomness that we have to provide an estimate of program performance. Later we will look to characterize *independent* and *time-dependent* noise, each of which influences execution time and will need to be addressed by a new tool. The underlying sources behind the noise are not of primary interest, but will be explored to some extent in section 5.

The tool developed in this project rests on the idea that to combat noise, we should collect more samples. Thus if we are given a set of execution times that has been distorted by the presence of noise, we believe that the solution will ultimately be to collect more samples, in order to gain a clearer picture of the true underlying distribution.

So what is reactive benchmarking? It is a term coined by my advisor, to describe a way of benchmarking that reacts to the data collected so far, to determine whether to keep running more iterations or terminate (having established trust in the statistic of interest, e.g. mean execution time). Developing such a tool as well as considering the underlying principles that would allow for the development of similar tools, is the primary objective of this project. The main properties we want from a new tool include:

- Automatically determine when to stop a benchmark. E.g. when the sample mean $\overline{x}$ reasonably approximates the 'true mean' μ, avoiding manually specifying that all programs in the benchmark should run *x* times.
- Provide an indication of how closely $\overline{x}$ approximates μ. This might be accomplished by a confidence-interval, which provides a span around $\overline{x}$ that we with e.g. 95% confidence would expect μ to fall within.

2.5 Project outline

Section 3 covers examples of related work, here we look at other benchmarking libraries that have some degree of automatic benchmarking. Section 4 describes the Futhark compiler's old benchmarking tool (the state of the benchmarking tool at the time of beginning this project). Section 5 looks at data collected by `futhark bench`, to identify patterns that needs to be addressed by a new tool. Section 6 describes the new benchmarking tool. Section 7 displays experimental results, comparing the new tool with the old one. Section 8 discusses the obtained results, and makes suggestions for future work. Section 9 concludes this project.

3 Related Work

In this section we will look at two open-source libraries for benchmarking C++ and Haskell code respectively. The main goal is to gain inspiration. Both libraries provide sophisticated options such as varying the input size to estimate asymptotic complexity or measure garbage collector statistics. However, we will limit ourselves to look at how they do the actual timing, how they see the challenge of noise, how they deal with it, and what sort of statistics they report.

3.1 Google benchmark

Google has an open-source benchmark library on github[5], for benchmarking *C++* code snippets (in a manner similar to unit tests).

Before benchmarking, *google benchmark* prints system information such as the size of every cache level, and a snapshot of the *system load*, measured as the average number of processes in the system run queue over a period of time (1, 5 and 15 minutes).

It collects results at two levels; 'iterations' and 'repetitions'. An iteration of a benchmark is a single execution of the code snippet to be measured, however in recognition that a single iteration may be quite noisy (as we shall see in section 5), they continue running iterations until $0.5s$ has elapsed, averaging them to a single measurement. Such a $0.5s$ average of iterations is a single 'repetition'.

Google benchmark first does a warm-up repetition, where it runs (and discards) $0.5s$ of iterations. Following the warm up repetition, the default setting is to collect and report a single repetition's average. However, they state in the user guide that *"benchmarks are often noisy and a single result may not be representative of the overall behavior"*[5], and thus allow an option to specify a desired number of repetitions. It will then run the specified number of repetitions, and do statistics on the collected data at a repetition granularity (reporting the mean, median, and standard deviation).

In Figure 1, we see 2 different invocations of a benchmark (using 20 repetitions), which constructs and copies a vector of 10^7 *ints*. Despite having discarded a warm-up repetition, the first few repetitions (each 13 iterations averaged), still underperforms subsequent repetitions. Note also the discrepancy in performance between the two invocations.

Google benchmark measured a load over the past minute of 1.18 for the invocation seen in blue, and 0.41 for the orange, which could explain part of the discrepancy.

In summary, *google benchmark* requires some work from the user to get a reliable result (specifying repetition count). Additionally it may not be intuitive that the statistics are not done directly on the individual iterations, but on the repetitions of $0.5s$. Lastly we saw how providing a measure of the system load could aid in diagnosing the noise level of a benchmark.

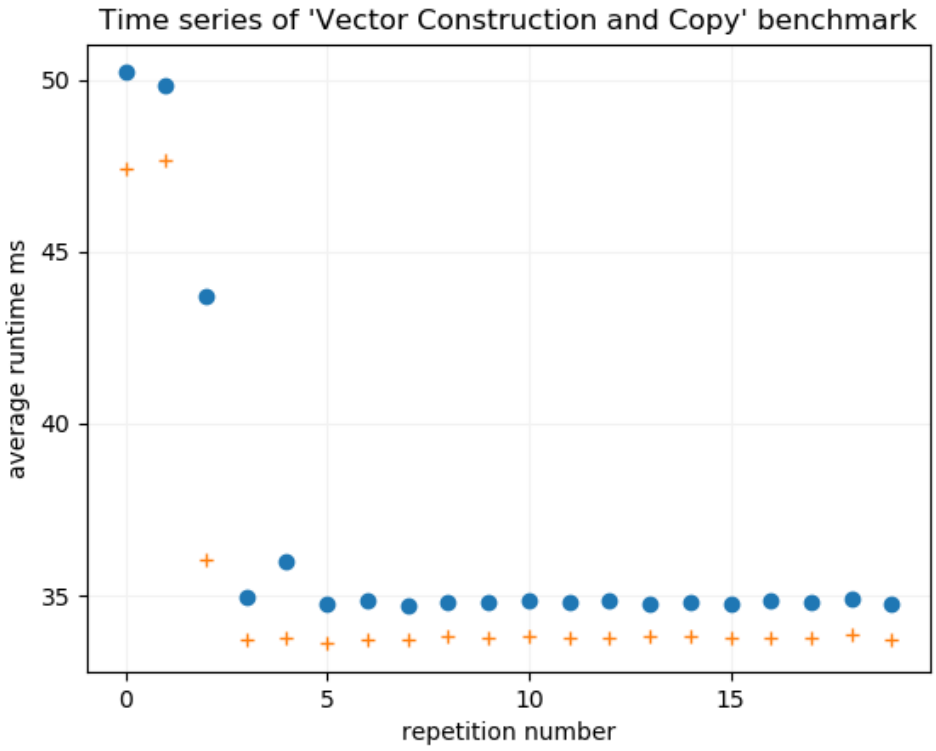


Figure 1: Two different invocations of *google benchmark* (blue/orange respectively), on a benchmark constructing and copying a vector in C++. Measured using 20 repetitions, each repetition is the average of 13 iterations.

3.2 Criterion

Criterion is a benchmarking library for Haskell[6]. By default, *Criterion* runs the benchmark for 5 seconds, though it is possible to specify a different duration. It then prints statistics, including the standard deviation and two different means: the sample mean $\overline{x}$ as well as an estimate of the mean using ordinary least-square regression *OLS*.

Figure 2 shows the output of a benchmark of a small function calculating the 5th *fibonacci* number.

The plot on the left shows a *kernel density estimate*. It's interpretation is similar to a histogram, having frequency of occurrence along the y-axis and execution time along x-axis.

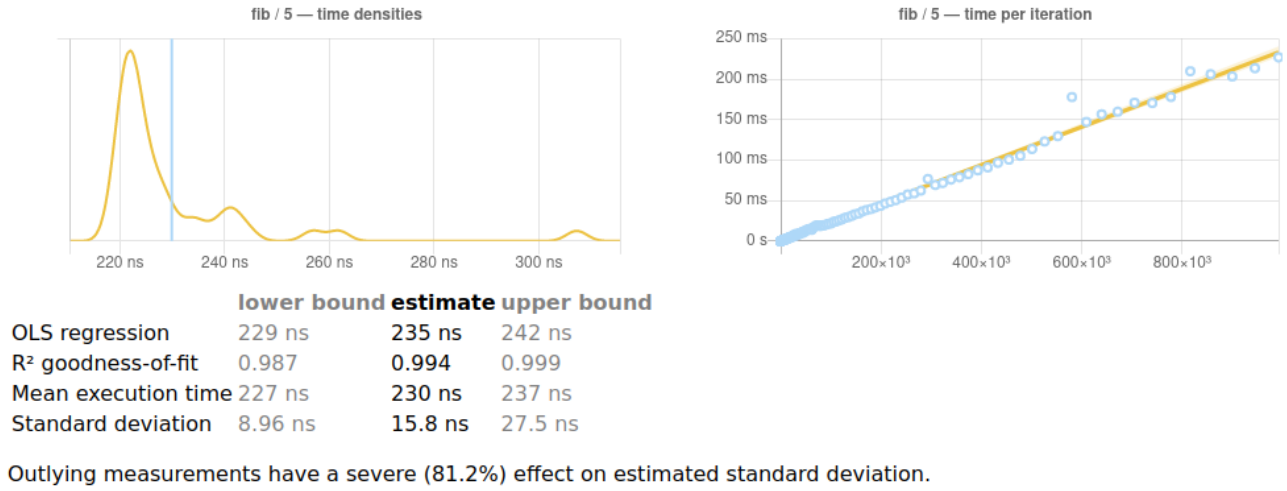


Figure 2: The plots and statistics produced by Criterion, for a benchmark of a function calculating the 5th *fibonacci* number.

The right plot is not so intuitive. Since every benchmark takes a different amount of time, its impossible to say beforehand how many iterations will be required to reach $5s$. Thus while benchmarking, they periodically need to check the timer to see if $5s$ has elapsed, and if not, determine how many iterations to do before checking again. The next number of iterations to run is found by multiplying the last count by 1.05. Each point in the right graph, represents the samples collected between checks of the timer. If we have a point with e.g. 1000 iterations (x-axis), then the following point has $1000 \cdot 1.05 = 1050$ iterations. On the y-axis is the total execution time of the corresponding set of iterations.

The "*Mean execution time*" of $230ns$, was obtained as:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

However, they also use ordinary least-square regression (OLS) to estimate a mean. In other words they try to fit a straight line to the plot on the right, by minimizing the residual sum of squares. If the measurements were taken with minimal noise, it should be possible to fit a straight line to the points as each point on the x-axis has 1.05 times more measurements than its predecessor, its expected execution time should likewise be 1.05 times higher. The mean can be found as the coefficient of the slope of the line.

The R^2 *goodness-of-fit* is a measure of how well the line fits the points, and thus also gives

an idea of the noise level.

Criterion additionally provides a % measure of how much outliers contributed to the standard deviation (as we will see in section 5 standard deviation is vulnerable to big outliers). However, it is hard to interpret what is meant with an '81.2% effect' other than they say it is severe.

Lastly, the measurements all have an upper and lower bound found by using a technique known as bootstrapping. This technique simulates the sampling process by random resampling (with replacement) from the original sample. A bootstrap sample

$$X^* = (x_1^*, x_2^*, ..., x_n^*)$$

is drawn from the original sample of size n, with replacement such that e.g.

$$(x_1^* = x_3, x_2^* = x_3, x_3^* = x_1)$$

is a possible resample of an $n = 3$ sample. To provide an upper and lower bound for the mean $\overline{x}$, they draw $100,000$ of such samples and for each calculate the statistic of interest:

$$\overline{x}^* = \frac{1}{n}\sum_{i=1}^{n} x_i^* \tag{1}$$

Rearranging these bootstrap means from lowest to highest. They obtain a lower bound of a 95% confidence interval as the $2.5th$ percentile of this sorted list of means, similarly the upper bound is equal to the $97.5th$ percentile. Additionally they perform bias-correction and acceleration to get the final confidence interval, however that is beyond the scope of this walkthrough.

In summary, *Criterion* uses some more advanced statistic techniques which requires some understanding of statistics. However, they provide a lot of insight into both the actual distribution of the samples in the form of the plots, as well as the noise present in the sampling process (R^2, confidence intervals and impact of outliers measurement). However, it is still up to the user to judge if these measurements present an acceptable estimate, or if the benchmark should be repeated to perhaps obtain less noisy estimates - which may be challenging for a user with limited knowledge of statistics.

4 Prior benchmarking system

The Futhark compiler has a simple benchmarking tool, which works by typing:

```
$ futhark bench file.fut
```

Before collecting any samples, the tool does a 'warm-up' iteration. This is done to get the code and data into cache, before collecting the subsequent samples. By default, `futhark bench` will run *"file.fut"* 10 times, outputting the mean of these 10 iterations along with the *relative standard deviation* (RSD), defined as the ratio of the sample standard deviation s to the sample mean $\overline{x}$:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$$

$$RSD = \frac{s}{\overline{x}} \tag{2}$$

Additionally, we are informed of how much the fastest iteration beat the mean execution time by, as well as how much slower the slowest iteration was. Below is an example output:

```
file.fut                61μs  (RSD: 0.124; min: -41%; max: +182%)
```

It is possible to specify the desired number of iterations, for example:

```
$ futhark bench --runs=500 file.fut
```

will run file.fut 500 times. To do more statistics, the user may dump the raw results to a *json* file, and do further analysis on their own.

5 Examining the data

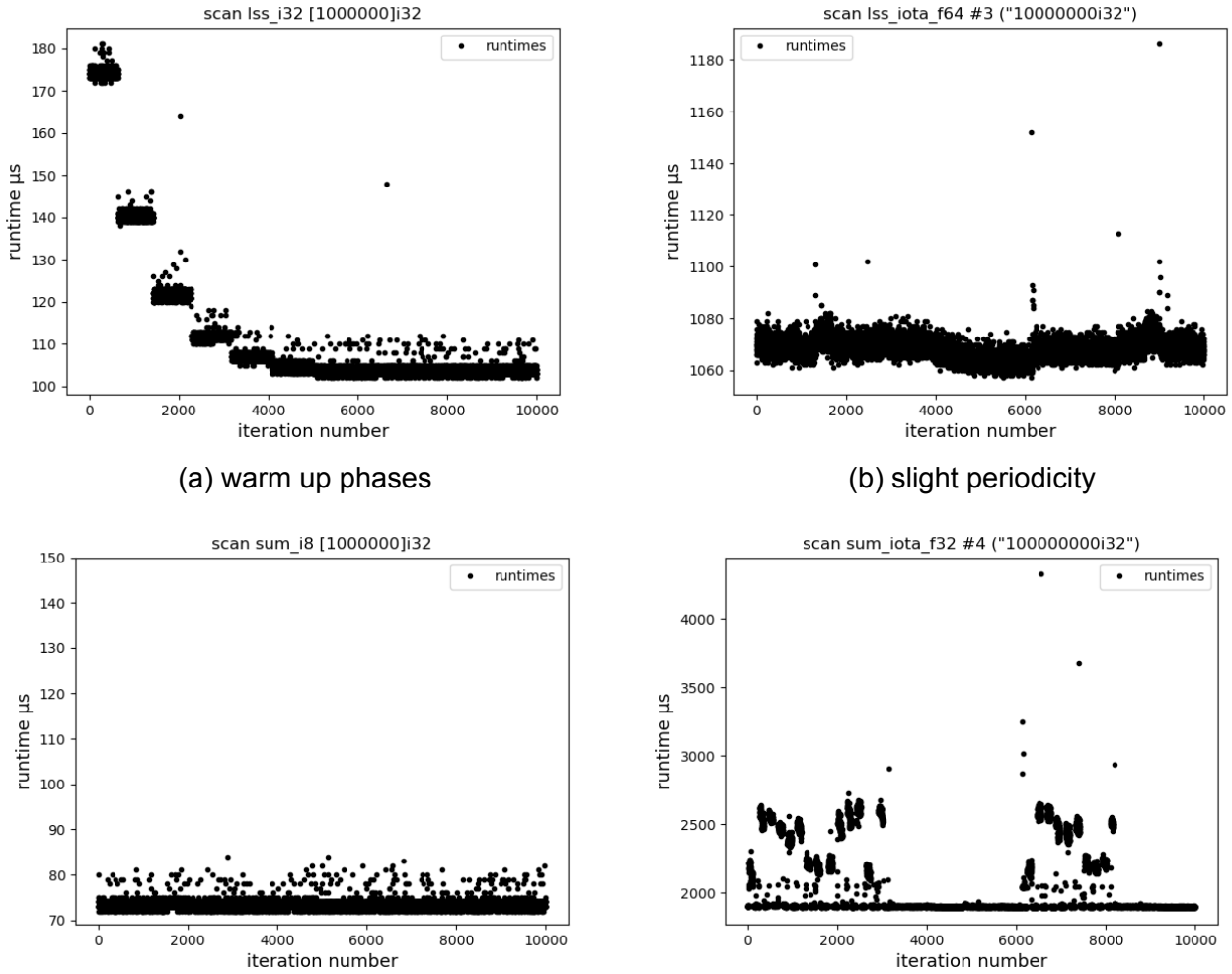
In this section we will take a thorough look at benchmarking data from running $\mathrm{futhark\ bench}$. The aim is to give an idea of how the iterations of a single benchmark vary. We will start by presenting the data at a high level. Next, we will attempt to quantify the variation that characterize many benchmarks. Lastly, the section is concluded with a description of how to measure the presence of dependencies and 'plateaus'.

5.1 The data

Figure 3 displays a time series plot of each of the four benchmarks. Each trace shows a quite unique pattern. Benchmark *a)* seems to have multiple phases of performance improvement until around iteration number 4000, where it reaches a more steady state. Benchmark *b)* seems more stable throughout its whole execution, yet the performance is still time-dependent as seen e.g. from iteration 4000 to 6000 where the average execution time drops slightly, before rising again. Benchmark *c)* has some noise, spread out evenly through its execution, with no signs of periodicity.
Benchmark *d)* shows the result of running a benchmark while periodically running other processes on the GPU. Benchmarking with a high background load is undesirable as it doesn't provide a clear picture of the code being measured, so we include it to give an idea of what such a trace looks like. More specifically another GPU-benchmark was run simultaneously, in the periods around iteration 0-3000, and 6000-8000. It is notable that in these periods there are clusters of execution times with a significant drop in performance. The benchmark suite which was run at the same time, contained multiple smaller benchmarks, taking on the order of $100ms$, and in-between these small benchmarks are periods of non-GPU work, which is the reason that performance improve in between the clusters.

Another way to view the data is using histograms (see figure 4). For benchmark *a)*, we clearly see each of the initial warm up phases as a spike in the tail of the distribution, however most of the samples fall in a group on the left. Benchmark *b)*, almost follows a normal distribution, this may be because it has a much longer average execution time and is thus more robust against random factors such as interrupts etc. (being averaged out over the span of execution).

Benchmark *c)* is slightly skewed to the right. As seen in the time-series data, this is not because of a warm up period but rather more sporadic spikes in the execution time. These likely come from random factors such as scheduling or interrupts. Having a shorter execution time per iteration, this benchmark is more vulnerable to this type of random noise. Lastly, benchmark *d)* also has a right-tail, containing predominantly measurements taken while the background load was high (another benchmark running). Having a long execution time, didn't help against this



(a) warm up phases

(b) slight periodicity

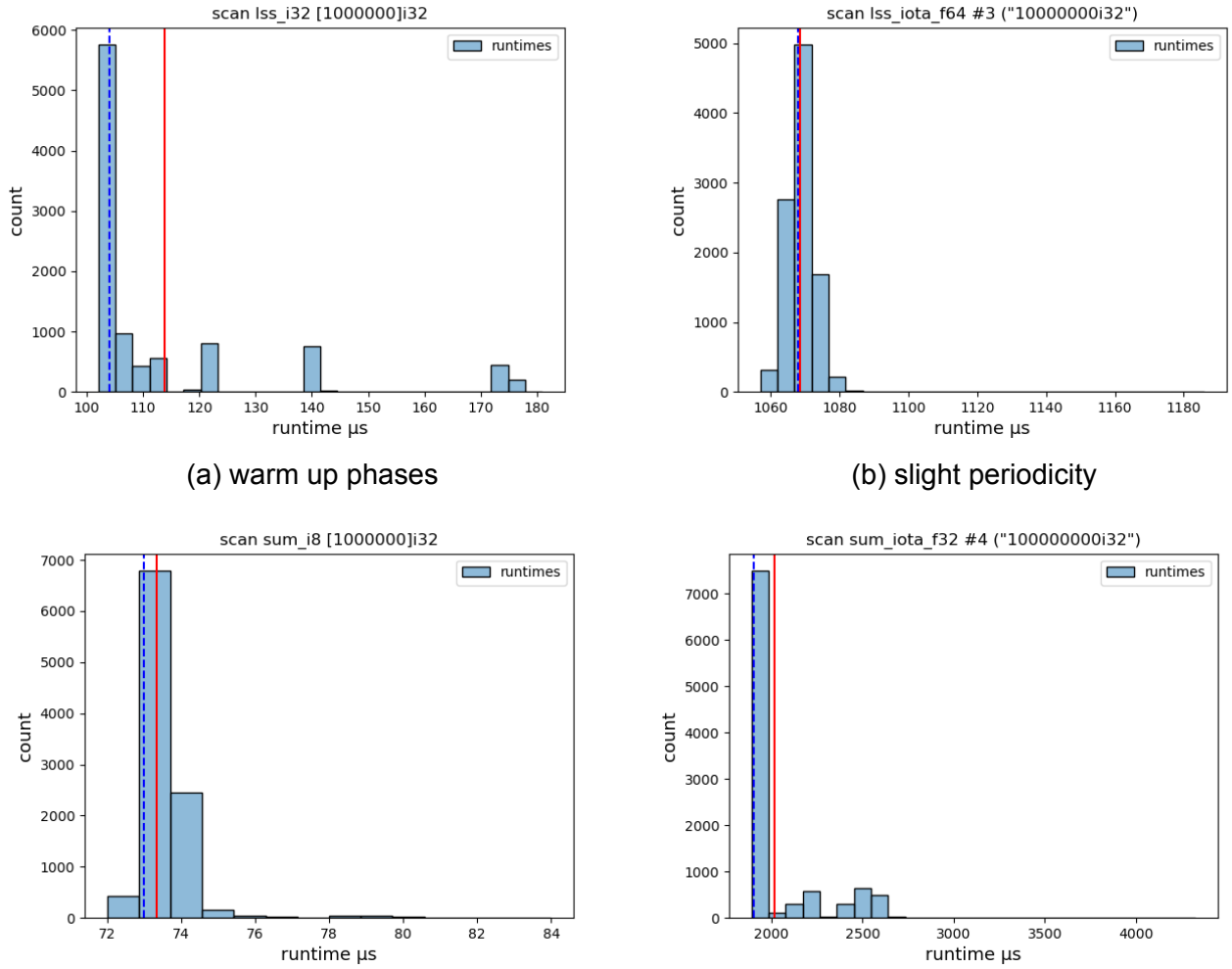
(c) *i.i.d* noise

(d) high background noise

Figure 3: Time series plot of 4 benchmarks. Each benchmark was run for 10,000 iterations. Along the x-axis is the iteration number, and on the y-axis is the execution time in μs. *OpenCL* was used as the backend for compiling to the *GPU*. For more information on benchmarking methodology, see Appendix A.

type of noise, as it is very time-dependent (hence some iterations will have high background load, while others none at all).

Looking at the mean (red line) and median (blue dotted line), we see that in all cases the median usually stays in the 'highest frequency' group on the left. The mean on the other hand, is dragged to the right by the tail of the distribution. It may be tempting to resolve the question of finding a mean execution time, by simply using the median instead. As it is seems to handle noise better. However, this approach has at least two problems. The first is in terms of how to interpret the result. While the mean is the expected average value, the median is harder to explain. Perhaps it can be seen as the value one would expect to most frequently appear



(a) warm up phases

(b) slight periodicity

(c) *i.i.d* noise

(d) high background noise

Figure 4: Histogram of the 4 benchmarks (using the same data). The red line is the mean of the execution times while the blue dotted line displays the median execution time (50th percentile)

if throwing an n-sided dice with each iteration as a face on the dice. A more serious problem is if the distribution is multi-modal, and we report a median that is not representative at all.[7] Consider the distribution presented in figure 5.

Here the median is found in the mode on the right, as the *50th* percentile lies in this cluster. The mean on the other hand, is sort of in-between. Although this benchmark is ”artificial”, in the sense that it was conjured up by increasing the background load, it goes to show that the median is not a solution if we want a robust tool (robust in the sense that it should be able to handle data following an arbitrary distribution, in particular we can’t make any normality assumptions).

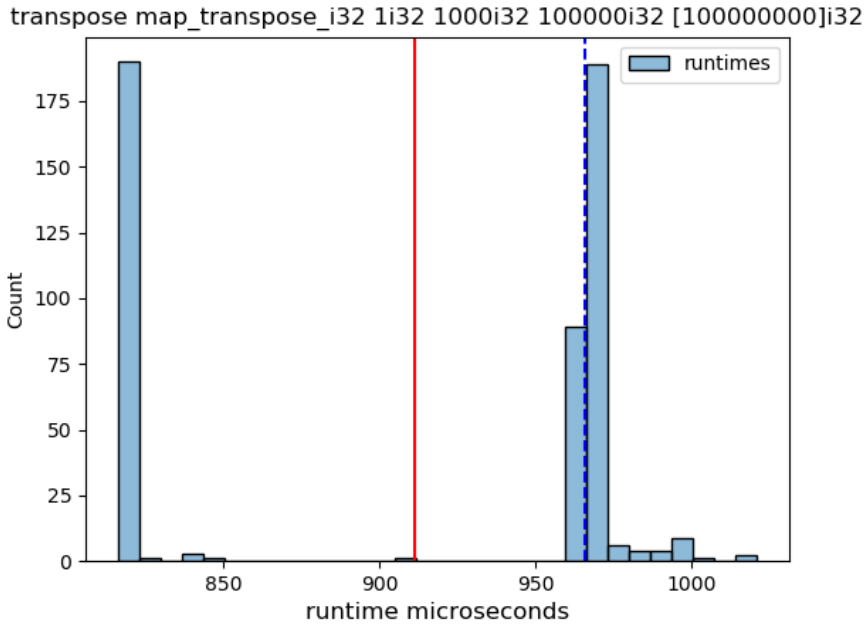


Figure 5: A histogram of a *transpose* benchmark, showing a clear bi-modal distribution. The benchmark was conducted with high GPU-load. The blue dotted-line shows the median, while the red line is the sample mean.

5.2 Quantifying variation

One thing that stands out is the high amount of variation in the benchmarks. In all four benchmarks we observed samples that stood out in one way or another. Now we will look at how to quantify this variation.

A first measure that comes to mind is the relative standard deviation (see equation 2), which says something about how far from the mean the data on average is spread out. The benchmarks had the following RSDs in order: $16.7\%, 0.4\%, 1.2\%, 11.2\%$. Unsurprisingly, number 1 and 4 had quite high variation while number 2 and 3 had much smaller numbers. However, RSD doesn't tell us anything about where the observed variation comes from. Consider table 1, here we see two quite different datasets that have the same mean and RSD. In the first dataset the data is spread out evenly around the mean, while in the second dataset almost all the variation comes from a single outlier.

Samples	$\overline{x}$	RSD	G
$1, 3, 5, 7, 9, 11, 13, 15$	8	61.2%	0.328
$6, 6, 6, 6, 6, 7, 7, 7, 21$	8	61.2%	0.204

Table 1: Two fictive datasets, showing μ, RSD and gini G.

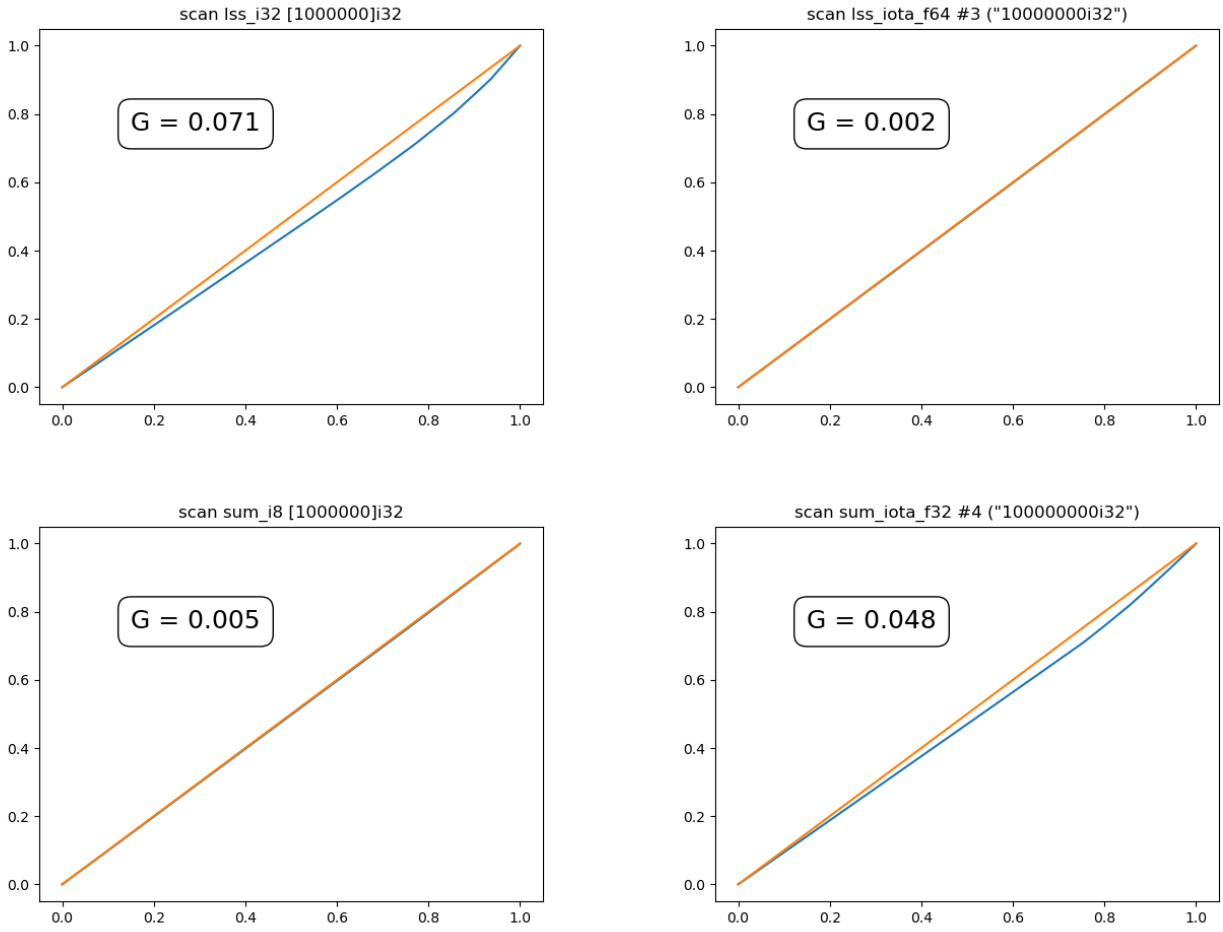


Figure 6: Lorenz curve and gini-coefficient (G) for each of the 4 scan benchmarks. Orange curve is perfect equality (all samples identical runtime), while the blue line shows the lorenz-curve.

Another way to look at variation is using a Lorenz-curve, commonly used in economics when measuring inequality of income. A point on the curve (x, y) with $x, y \in [0, 1]$ is defined as the contribution of the bottom fraction x to the total sum of execution times $\sum_{i=1}^{n} x_i$. Perfect equality is shown as a straight diagonal line, such that e.g. the bottom $20th$ percentile (0.2 on the x-axis), contributes exactly 20% to the total execution time.[8] Figure 6 shows such a curve for each benchmark. A related statistic that provides a measure of inequality in the dataset is the gini-coefficient G. It is defined as 2 times the area between the perfect equality line and the lorenz curve, or in terms of the samples x:

$$G = \frac{\sum_{i=1}^{n}\sum_{j=n}^{n}|x_i - x_j|}{2n^2\overline{x}} \tag{3}$$

Table 1 also displays the gini coefficient G. It measures relatively less variation than RSD in the dataset containing an outlier. This is because RSD contains a squaring operation and

thus weighs outliers more heavily, while G uses the absolute value and is thus less affected by a big value.

It is hard to say which term is more useful. As RSD puts more emphasize on outliers it is perhaps a sensible choice if we think big outliers is the most common type of noise that we want to avoid. However, judging from the time series plot of the 4 benchmarks in figure 3, it's unclear if this is really the case.

In either case, consider the following problem: We have collected the first 700 samples of benchmark 1, reproduced in figure 7.

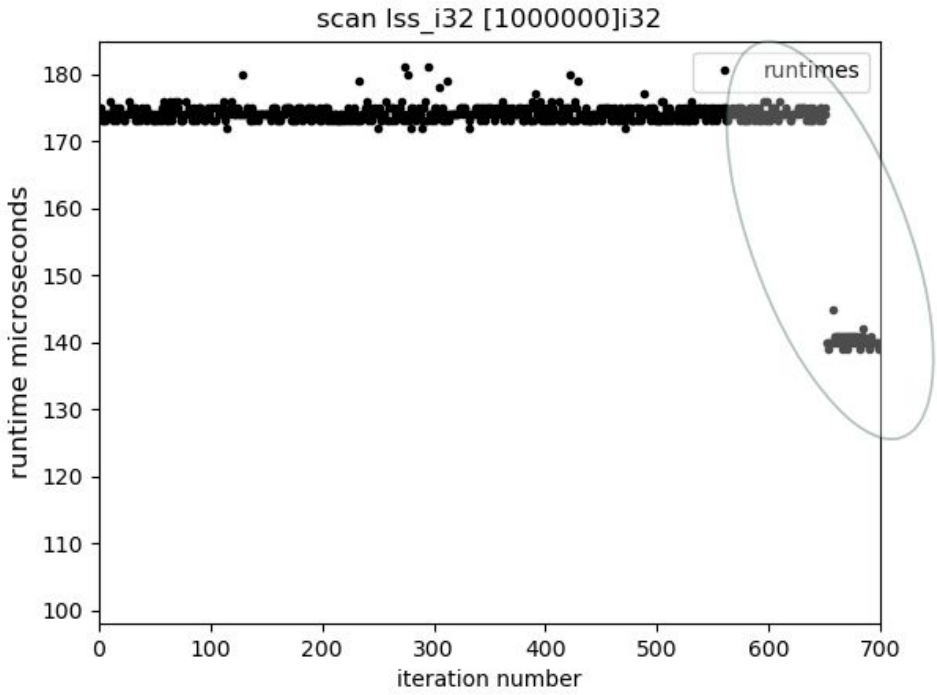


Figure 7: First 700 iterations of the 1st benchmark, $RSD = 5.01\%$. Note the shift in performance (circled) from the initial plateau.

The RSD of this subset of the benchmark is a meager 5.01%. Inspecting the plot we clearly see that the first $\sim$650 samples may have been part of some warm up phase (indeed the next 3400 samples is too), meaning that we would like to continue running the benchmark - as it has yet to converge. But the RSD doesn't capture this time-dependent performance of the data, and reports a low variation. To better capture these plateaus we can look to the serial autocorrelation, discussed in the next section.

5.3 Measuring periodicity

Figure 8 displays lag-plots of the 4 benchmarks. A lag is a fixed displacement of a time series, the h'th lag is the observation that happened h time points before i, e.g. $lag_1(x_5) = x_4$.[9] The original set of execution times is displayed along the x-axis, while the y-axis is the same set of data 'lagged' once.

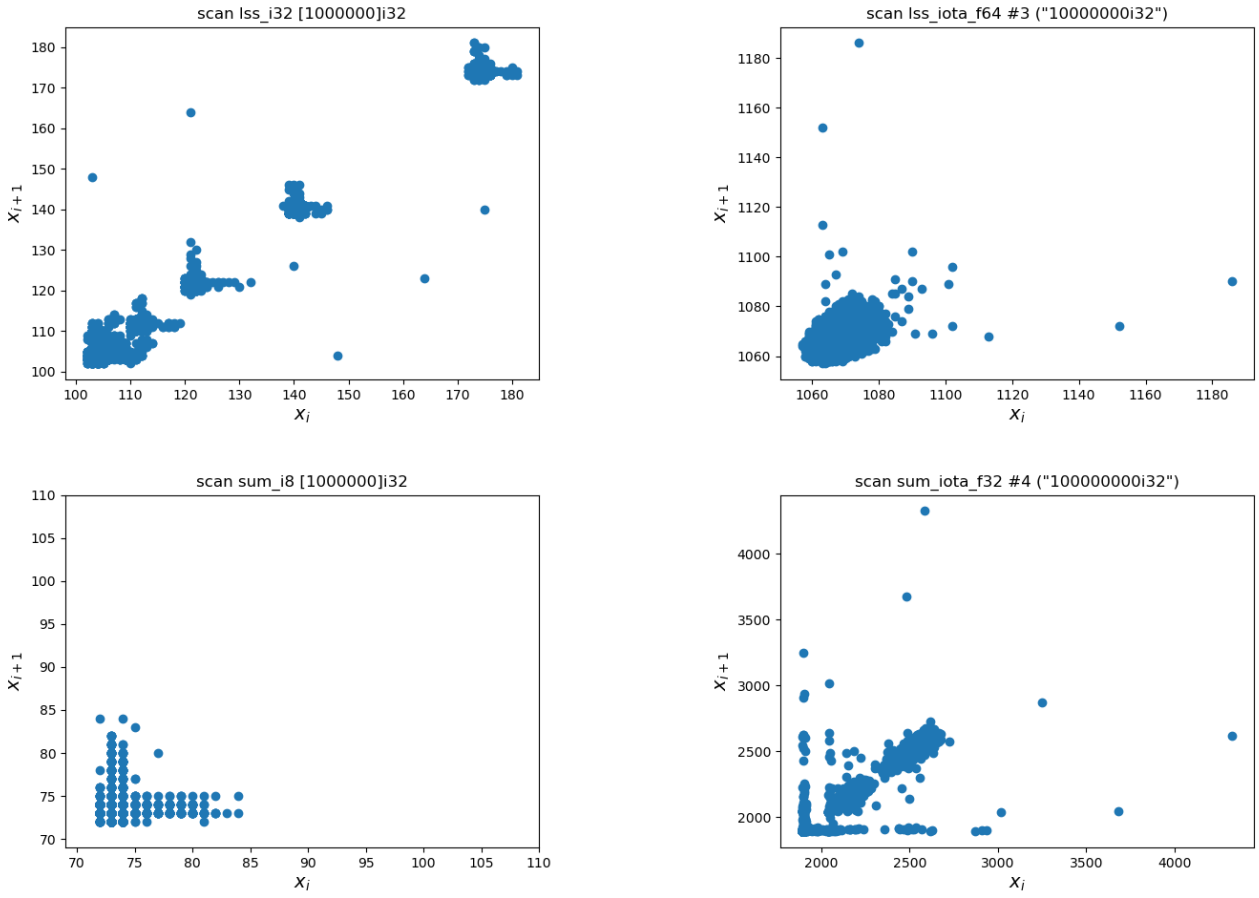


Figure 8: Lag plot, showing the runtime in μs along the x-axis and the same dataset shifted 1 sample to the right along the y-axis.

In the first benchmark (containing many warm up phases), most points are scattered along the diagonal. Thus sample x_i is a good predictor of x_{i+1}, each of the warm up plateaus are clearly visible as its own cluster. The second benchmark is more clustered in a big oval group, however it also has a slight diagonal shape, again indicating that sample x_i is correlated with x_{i+1}. The third benchmark's lag plot tells a different story. Here we see that if we have observed a slow iteration ($> 76\mu s$), then the next iteration is unlikely to be slow - which indicates that the performance drops are singular events, not correlated with the next iteration of the benchmark (the reason for the grid-like structure is that the resolution of the timer is limited to $1\mu s$). The last benchmark also has a significant diagonal component, however there are also times where x_i

can't predict x_{i+1}, as seen in the horizontal and vertical lines (likely measurements taken on the onset or offset of a performance plateau - x_i having a high background load and x_{i+1} none).

These lag plots can give a visual cue of whether a benchmark sample is independent or has dependencies. Another way to detect dependencies in the sample is an autocorrelation plot, which displays the correlation between our time series X_t and a lagged version of the same series X_{t-h}, for some lag h

$$cor(X_t, X_{t-h}) = \frac{E\big[(X_t - \overline{x})(X_{t-h} - \overline{x})\big]}{s^2} \tag{4}$$

A positive correlation indicates that if x_i is a large value, then x_{i+h} is probably also large. Conversely for a negative correlation, a large x_i would suggest that x_{i+h} is relatively small. See figure 9 for autocorrelation plots of the 4 benchmarks.

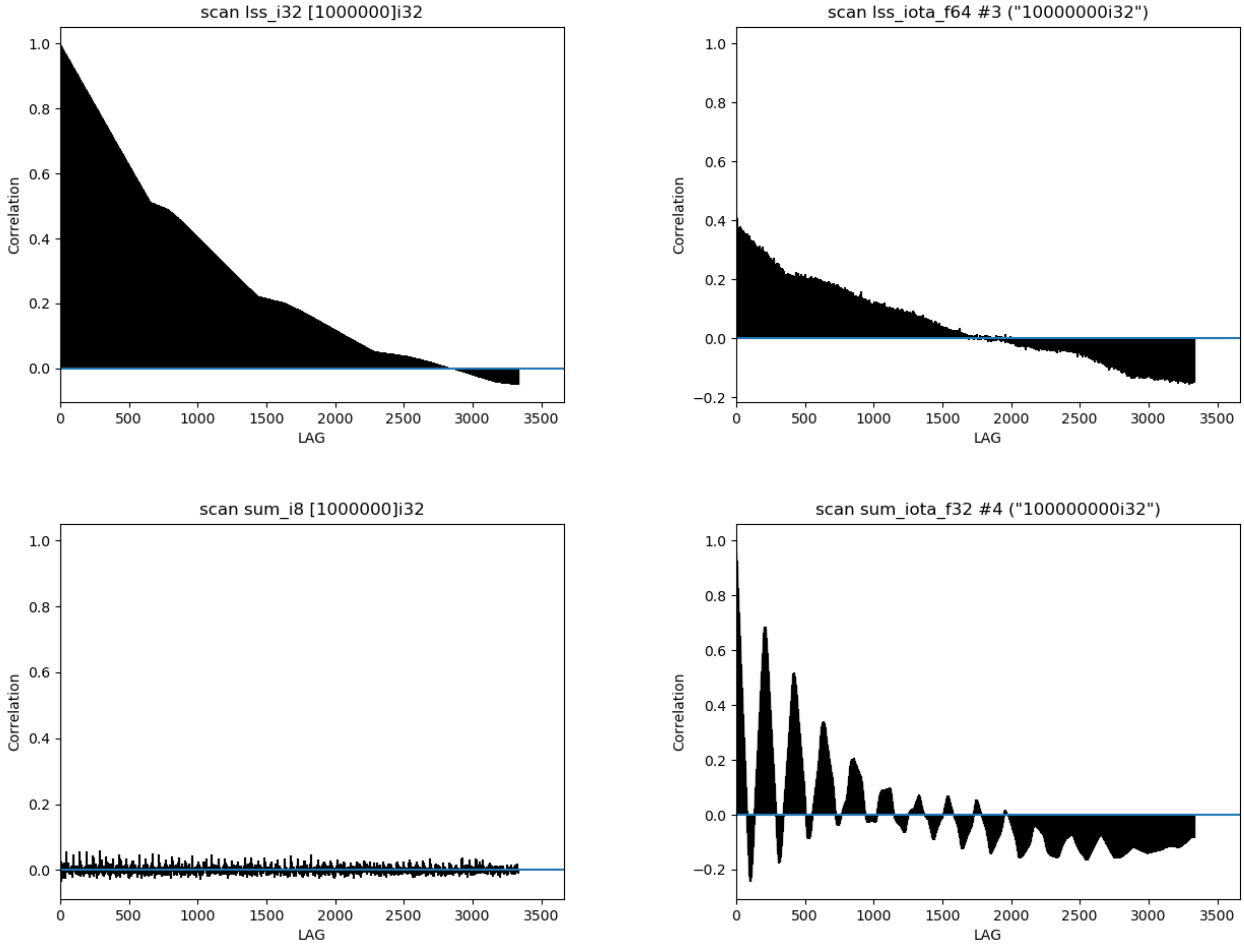


Figure 9: Autocorrelation plots of the 4 benchmarks. x-axis is the lag at which the correlation was calculated, while the y-axis shows the value of the correlation at that lag.

The first benchmark has a very high autocorrelation, slowly dropping until lag $\sim$2700 where the correlation is 0. This tells us that the further away from an iteration x_i we move, the less predictive power (correlation) it has, until reaching x_{i+2700} where they are uncorrelated. Similarly for benchmark 2, however the magnitude of the correlation is a lot smaller, which makes sense as the time-dependency of the data was less severe. Benchmark 3 seems to contain only *independent and identically distributted* (*i.i.d*) noise, as each iteration is uncorrelated with the others. The last benchmark shows an interesting periodical phenomenon. Remembering the time series plot, the background noise was clustered in small groups of around $100ms$, which is very visible on this plot. Lag 1-100 has a high correlation, which decreases fast and then turns negative as an iteration x_{i+120} is more likely to be in a different performance cluster from x_i.

Thus the autocorrelation enables us to say something about how each sample x_i relates to a subsequent sample x_{i+h}. We may use this to detect 'plateaus' in the plots, i.e. as seen in figure 7, where the best predictor of x_i is x_{i+1}.

5.4 Summary

In summary, we saw how benchmarks vary significantly from iteration to iteration. We identified two different kinds of noise, one which was random throughout the whole execution (e.g. benchmark 3), and another which was time-dependent (either the result of increasing the background load, or due to the presence of a warm up phase). We postulated that the first kind of noise, is a bigger problem for shorter running benchmarks, as e.g. the impact of an extra interrupt is not amortized well over a short duration of execution, whereas a longer running benchmark will average out these sources of random noise. Conversely, the time-dependent noise is a problem for benchmarks of all durations, as it has a more 'on/off' character, where it in periods significantly worsens the performance and in other periods is completely absent.
Then we moved on to characterize the variation in the benchmarks, first using the relative standard deviation and then the gini-coefficient, noting that RSD puts more emphasize on large outliers. However, we saw that these two measures had a significant shortcoming in that they were unable to detect the presence of multiple plateaus of performance.
Lastly we found a way to quantify the plateaus, as the autocorrelation of the samples at lag 1. Where a high correlation suggests the presence of strong inter-iteration dependencies.

6 A new benchmarking tool

Now that we have studied benchmarking tools used in practice in section 3 and seen the non-deterministic nature of the data in section 5, we are ready to construct a new tool.

In this section we will first look at how to define a notion of accuracy, which we may use to determine when to complete a benchmark. Next, we will describe the new tool developed as part of this project, and the design considerations that led to it.

6.1 Accuracy and convergence

As we want to run benchmarks until we have an accurate result, we should make clear what is meant by 'accurate'. We may define accuracy, as a measure of how close the sample mean $\overline{x}$ is to the true mean μ. However, the problem with this definition is that by definition μ is unknown. The Law of large numbers

$$\overline{x}_n \to \mu \qquad as\, n \to \infty$$

states that the sample mean tends toward the true mean μ, as the number of samples approaches infinity. However, to use the Law of large numbers each trial must be *i.i.d*, an assumption that fails in our setting. It is the 'independence' part that is not satisfied, as we saw in section 5, benchmarks often have a strong serial autocorrelation.

However, despite being somewhat idealized, adopting this notion allows us to use the statistical term known as standard error as a proxy for accuracy. The standard error (SE), estimates how closely the sample mean $\overline{x}$ represents the true mean μ.

To describe standard error, let's imagine that we have access to the true distribution F:

$$x \sim (\mu_F, \sigma_F^2)$$

Taking the mean $\overline{x}$ of a sample from F, would have the same expected mean of μ, but its variance would decrease by a factor of the sample size n:

$$\overline{x} \sim (\mu_F, \frac{\sigma_F^2}{n})$$

The standard error can be seen as the standard deviation of this sample mean.[10]

$$SE = \frac{\sigma}{\sqrt{n}}$$

To normalize the standard error, we may divide by the mean obtaining the relative standard error (RSE):

$$RSE = \frac{\sigma}{\mu\sqrt{n}}$$

However, as we don't have access to the true mean μ or standard deviation σ of the population, we may estimate it by using the sample standard deviation s and mean $\overline{x}$:

$$RSE = \frac{s}{\overline{x}\sqrt{n}} \tag{5}$$

The accuracy of this estimator depends on the sample size, as for very small sample sizes it will systematically underestimate the standard deviation of the population. For $n = 6$ the bias is about 5%, and drops rapidly for larger n (in our case, n will usually be many times higher).

6.2 Design considerations

Both *google benchmark* and *Criterion* used time as the sole criteria for when to stop benchmarking ($0.5s$ for a repetition of *google benchmark*, and $5s$ total sampling time for *Criterion*). Having observed the way that many benchmarks have warm up periods, it makes sense to incorporate an element of real time measurement in order to give it some time to 'get warm'.
If we conversely were to only look at data pertaining to the variation of the data, we could end up terminating the benchmark prematurely, as the variation on a single plateau can be quite low. Thus to give it time to reach a second plateau (if present at all), a minimum time threshold makes sense.

More specifically in the case of $\mathrm{futhark\ bench}$, it seemed that whenever there was a warm up phase, it usually took around *0.5-1s*. It also seemed that some benchmarks were more susceptible to having a warm up phase than others. However, as of now we have no way of determining whether a benchmark will contain a warm up phase a priori, and other architectural explanations remain plausible, e.g. that the GPU is switching on some kind of turbo boost where it draws more power as it comes under load, thus increasing performance after a short period.

Once we have run our benchmark for the initial time period, we should determine whether it contained anomalies such as a warm up phase, and if so, what to do about it. In section 5, we saw how the autocorrelation could detect the presence of dependencies between iterations (and hence warm up periods). Thus we could calculate the autocorrelation at lag 1 to identify a warm up period.

Next, we should determine what to do if such a period is present. Intuitively, we should keep on collecting more samples to reduce the effect of a period that is unlikely to be representative of

normal performance. But how much should we continue running? And should we always keep running more iterations if the autocorrelation remains high? Once there are multiple plateaus, the autocorrelation will remain high for a long time despite subsequent iterations sharing an execution time within a narrow range.

Thus, it would make sense to factor in how many iterations we have already done before we keep running indefinitely. The RSE which can be considered as the standard deviation of the mean $\overline{x}$, has the property that it decreases with the square root of the sample size (see equation 5). E.g. reducing the error by a factor of two requires four times as many samples, while reducing it by a factor of 100 requires $10,000$ times more samples. Using the RSE has the added benefit that it also measures the variation, thus a low RSE would come from either a low variation or a very large sample size n, or a combination.

Thus if we incorporated the RSE as a measure, in combination with the autocorrelation, we could set various termination conditions. For instance if the autocorrelation is high (i.e. there are dependencies that may introduce bias), we would require a very low RSE as compensation before terminating. Conversely, if the data seemed to only suffer from *i.i.d* noise, we would be satisfied that the RSE is relatively higher. The challenge lies in making sure the initial timed period is long enough to capture a warm up phase.

6.3 A new tool

Having gone through the design considerations, we will now summarize how the new tool works. At a high level, the procedure can be seen in figure 10. We start by running iterations for half a second (or at least 10 iterations), before entering the 'convergence phase'.

The convergence phase starts by checking whether the collected data satisfy a set of 'convergence criteria', which can be seen in figure 10. The criteria consists of a combination of autocorrelation and RSE thresholds. As mentioned earlier, a high autocorrelation demands a lower RSE as compensation. If the initial $0.5s$ of data doesn't meet the criteria, we discard all the iterations collected so far as they probably contained a warm up phase. This enables the tool to 'restart' but hopefully now in a warm state. This means the tool finishes a lot sooner, as the autocorrelation would otherwise remain high for a long time.
We get the next iteration count as $\frac{n}{2}$ iterations where n was the previous number of iterations. Once it has completed these, we recheck if the results satisfy any of our criteria. If the data doesn't meet the criteria, we perform another $\frac{n}{2}$ iterations and recheck. In the rare case that not only the autocorrelation remains high, but also the RSE, we also impose a benchmarking time

limit of 5 minutes.

Our next desired property in addition to automatic convergence, was to provide an indicator of how accurately the sample mean $\overline{x}$ represents μ (or at least a measure of the variation involved).
To do this, we may look to *Criterion* for inspiration. They use bootstrapping to obtain a non-parametric confidence interval (i.e. it doesn't use a parameterized statistical distribution to obtain the confidence interval, as most of these have an underlying *i.i.d* assumption).

With bootstrapping, we rearrange the resampled means $\overline{x}^*$ in order from low to high (see equation 1), and obtain a 95% confidence interval as the 2.5th and 97.5th percentile of this sorted list. This also provides a measure of variation, as data that is spread out will produce a wider confidence interval.

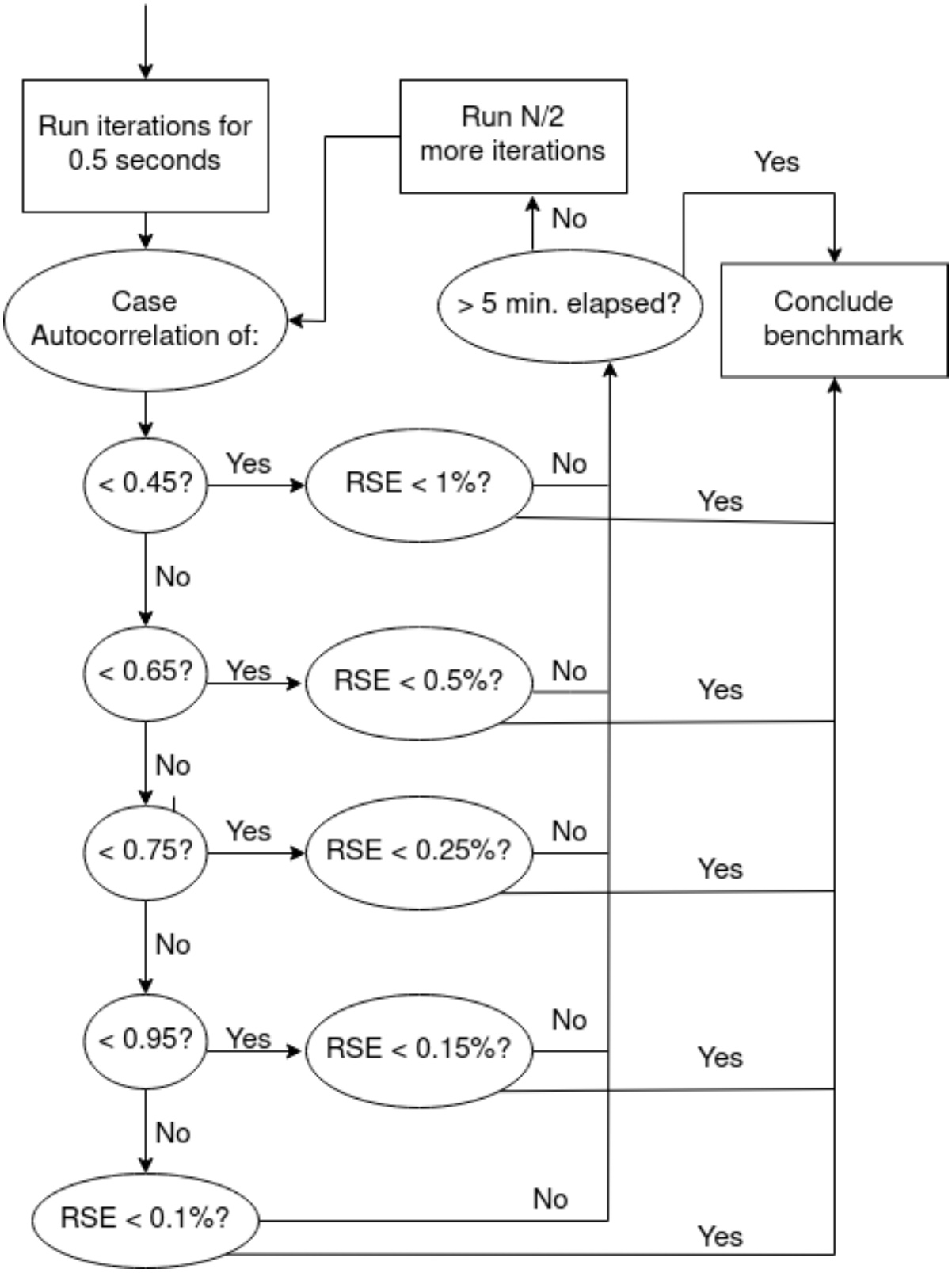


Figure 10: A schematic of the solution. All implementation details have been stripped away, focusing on the algorithm at a high level.

7 Experimental Results

In this section we will evaluate the accuracy of our new tool, as well as the speed with which it may conclude benchmarking. Our main point of comparison will be the old tool, varying the iteration count from the default 10 iterations, to a more conservative $10,000$.

When measuring accuracy, there are two things we desire; *reliability* and *validity*. If our new tool is reliable, it means that asked the same question (i.e. given the same benchmark under the same conditions) it should produce the same result each time. However, although a tool may be reliable, it may just reliably give the wrong answer each time. Thus we also want the result to be 'valid' in the sense that we got what we were asking for. We will use the mean of 15 repetitions $10,000$ iterations each, as a baseline to test for validity, however we will first determine how reliable the new tool is.

7.1 Reliability

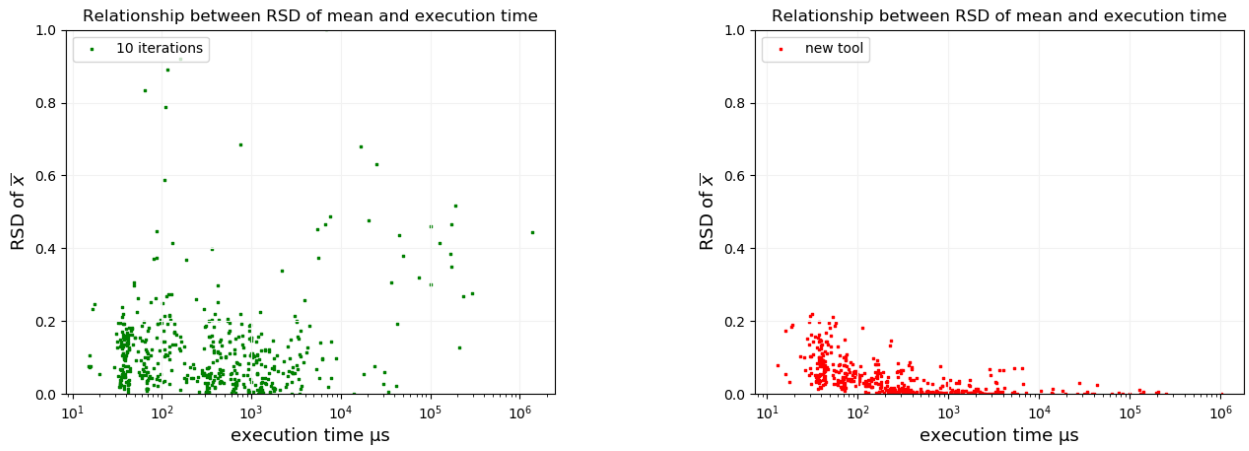
Remembering that RSE could be considered the standard deviation of the mean, we could perform the same benchmark a number of times, collecting the sample means $\overline{x}$ and calculate the standard deviation of these means. We would thus obtain an indicator of how closely each estimate of μ fall to each other - if they all end up tightly spaced (A low deviation), the result may be considered reliable. Conversely if we obtain means that differ significantly, the tool can be said to be unreliable.

Figure 11 is a collection of benchmarks, each represented by a dot (see appendix A, for a reference on the benchmark suite). Each benchmark was repeated 15 times, calculating the mean $\overline{x}_i$, and the variation between these means using RSD. The x-axis displays the average execution time $\frac{1}{15}\sum_{i=1}^{15}\overline{x}_i$, while the y-axis is the RSD of the sample means. The benchmark suite contained benchmarks with an execution time ranging from $15\mu s$ to over $1s$ (note the logarithmic x-axis).

The old tool using 10 iterations (plot *a)*, had an average RSD of $\overline{x}$ of 15%, which is highly unreliable. Applying the CLT, we can say that having observed one repetition giving a mean $\overline{x}$, we would expect the mean of another repetition to fall in the range: $[\overline{x} \pm 30\%]$ with a 95% probability. Using 10 iterations to obtain $\overline{x}$ left too much to chance, e.g. a warm up phase perhaps present in one repetition but not another, or a hardware interrupt creating an outlier that isn't amortized well over 10 iterations.

On the other hand, the new tool had a much lower average RSD of $\overline{x}$ of 3.8%, this tells us that when we have obtained a sample mean $\overline{x}$, we can expect a subsequent repetition of the same benchmark to produce a sample mean that 95% of the time will lie in the interval $[\overline{x} \pm 7.6\%]$.

Another thing that stands out is the relationship between the RSD and execution time. As execution time increases, the variation in collected sample means seem to fall. The difference is quite remarkable, for programs with an execution time $< 250\mu s$, the average RSD of $\overline{x}$ was 6.6% while for those over $250\mu s$ it was just 1.2%. The exact reason for this relationship is unknown, it is possible that part of the explanation comes from the robustness of longer-running benchmarks against *i.i.d* noise, or perhaps short-running benchmarks need a lot of iterations to reach a high sustained performance (perhaps only achieved at some point after the initial timed phase is discarded, which may not occur for every repetition of the same benchmark).



(a) 10 iterations

(b) New tool

Figure 11: RSD of the sample mean (y-axis), against the average execution time of the 15 repetitions(x-axis). The full benchmark suite (listed in appendix A) was used.

Figure 12, displays the same relationship between RSD of $\overline{x}$ and execution time as above, however now using $10,000$ iterations (only done on *scan.fut* and *transpose.fut* of the benchmark suite, as it took a significant amount of time to run 15 repetitions using $10,000$ iterations). Using a conservative $10,000$ iterations gives a lower variation, but also took a significant amount of time. The time spent on each benchmark was proportional to its execution time, thus despite the robustness of longer-running benchmarks, the old tool spends most of its time benchmarking these.

Using $10,000$ iterations the average RSD reaches a low 2.4%, (giving a 95% confidence interval for another repetition of $[\overline{x} \pm 4.8\%]$). The same tendency for higher variation in $< 250\mu s$

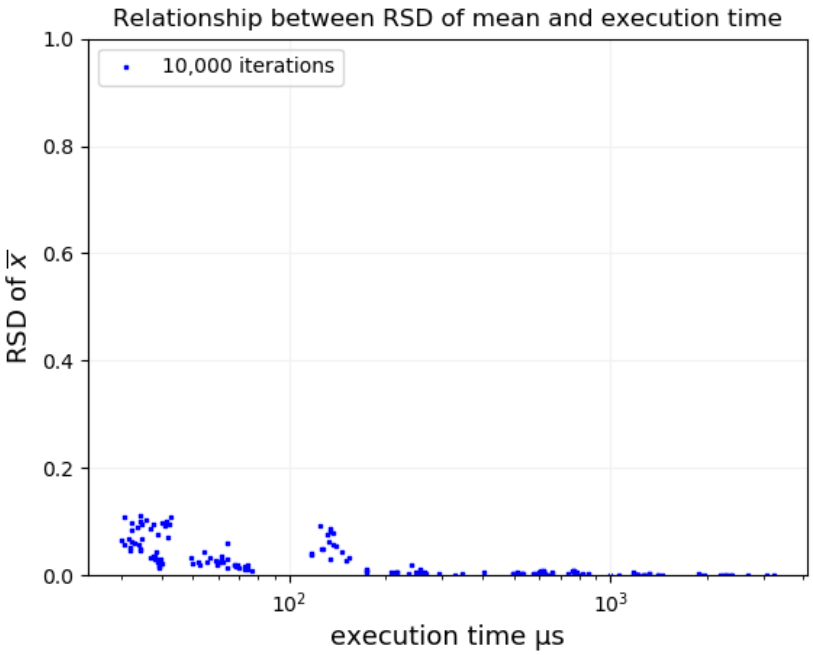


Figure 12: RSD of the sample mean (y-axis), against the average execution time of the 15 repetitions(x-axis), using $10,000$ iterations. The benchmarks found in *scan.fut* and *transpose.fut* from the benchmark suite was used.

programs that we saw when using the new tool, also shows when using $10,000$ iterations; here the variation is 4.5% while for those above $250\mu s$ it was 0.3%.

Thus in terms of reliability, $10,000$ iterations is $\sim$*1.58x* more reliable than the new tool (2.4% average RSD of $\overline{x}$ vs. 3.8%), but how do they compare in the amount of time they took to finish benchmarking *scan.fut* and *transpose.fut*? The new tool spent 4 *minutes* and *16* seconds, while $10,000$ iterations took *23 minutes and 44 seconds*, thus the *1.58x* gain in reliability of using $10,000$ iterations, came at a *5.6x* speed penalty.

7.2 Validity

Now that we have looked at the amount of variation in the sample mean $\overline{x}$ of the same tool (i.e. how reliably it gives the same answer to the same question), let's see to what degree the obtained sample means agree with a $10,000$ iteration baseline (averaged out over 15 repetitions). We allow for some margin of error, as $10,000$ iterations is also only an estimate of μ.

Figure 13 displays on the y-axis the ratio of $\overline{x}$ between the conservative estimate using $10,000$ iterations (indexed to 1), the old tool using 10 iterations (green) and the new tool (red). The benchmark suite used is *transpose.fut*.

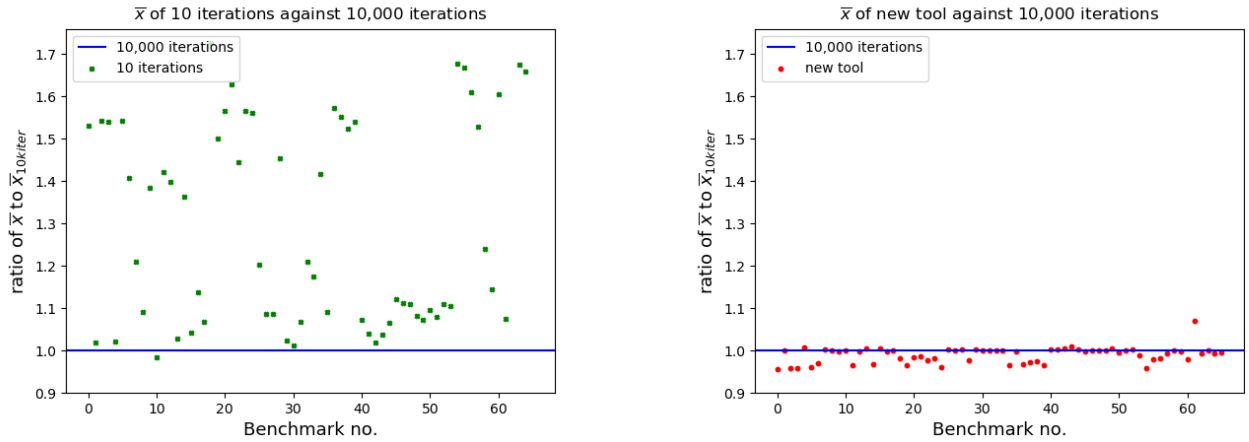


Figure 13: Comparing the sample mean $\overline{x}$ across different tool configurations, on the *transpose.fut* benchmark suite. The sample mean using $10,000$ iterations is used as index (blue). Each dot represents a unique benchmark which was repeated 15 times using either 10 iterations (green) or letting the new tool determine iteration count (red).

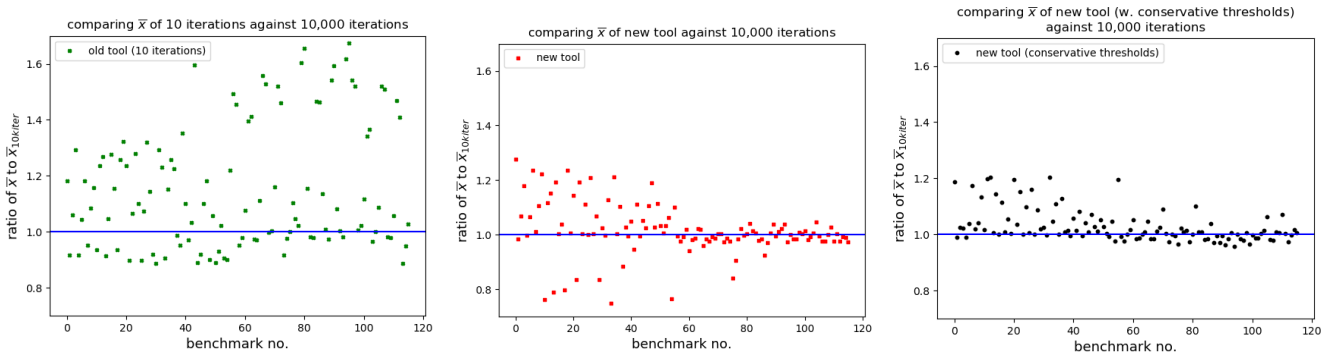
The mean obtained using 10 iterations seems to consistently overstate the execution time. This is likely due to the presence of warm up phases, using 10 iterations is seldom enough to reach a more stable state. On the other hand, the new tool agrees more closely with the results obtained using $10,000$ iterations.

As we spent *12 minutes and 21 seconds* using $10,000$ iterations and only *1 minute and 53 seconds* using the new tool, and achieved quite similar means (with some margin of error), the combination of (autocorrelation, RSE) thresholds can be said to work quite nicely for the *transpose.fut* suite.

However, for the *scan.fut* collection of benchmarks, the results using the new tool deviate a bit more from the $10,000$ iteration "baseline", as seen in figure 14.

Again, the new tool *(b)* is definitely closer to the baseline than the default 10 iterations *(a)*, which overstates the mean most of the time. On the x-axis, the benchmarks are ordered by execution time (as measured with $10,000$ iterations) from low to high. Thus it seems that it is mostly for benchmarks of short duration that there is a discrepancy, and as execution time increase the new tool's mean approaches that obtained using $10,000$ iterations.

To obtain a higher degree of accuracy, we may change the weighting used between RSE and autocorrelation, for instance doubling the RSE requirements and increasing the minimum benchmarking time from $0.5s$ to $1s$, which produces the ratios seen in *(c)*. Using these more



(a) 10 iterations (b) new tool (c) conservative RSE thresholds

Figure 14: Comparing the sample mean $\overline{x}$ across different tool configurations, on *scan.fut* benchmark suite. The sample mean of the old tool with $10,000$ iterations is used as index (blue). Each dot represents a unique benchmark which was repeated 15 times using either 10 iterations *(a)*, new tool determine iteration count *(b)*, or the new tool with more conservative weights *(c)*.

conservative thresholds gave a sample mean that closer approximates that using $10,000$ iterations, again with increasing proximity as execution time increases. However, it is ultimately a tradeoff between speed and accuracy. Running $10,000$ iterations took *11 minutes and 25 seconds*, the new tool took *2 minutes and 23 seconds*, and the new tool with conservative weights took *6 minutes and 52 seconds*.

7.3 Comparing autocorrelation

One factor that enables the new tool to finish faster than the old tool, is the near elimination of warm up phases, which it would take a lot of iterations to reduce the effect of. To verify this, we should check that autocorrelation in the results have decreased when using the new tool. Our hypothesis is that discarding the initial warm up period, should yield an overall lower autocorrelation.

Figure 15 compares the autocorrelation at lag 1 (y-axis) of benchmarks from the *scan.fut* suite. Each point's autocorrelation has been averaged over 15 repetitions of the corresponding benchmark. The benchmarks have been sorted using the autocorrelation of $10,000$ iterations with the old tool as key.

The old tool averaged an autocorrelation of 0.52, while the new tool averaged 0.18. Thus as we expected, the new tool produces samples that have less inter-iteration dependencies, as warm up phases have been removed. However, part of the reason may also be that we are spending more time doing $10,000$ iterations, which simply leaves more time for dependencies to arise.

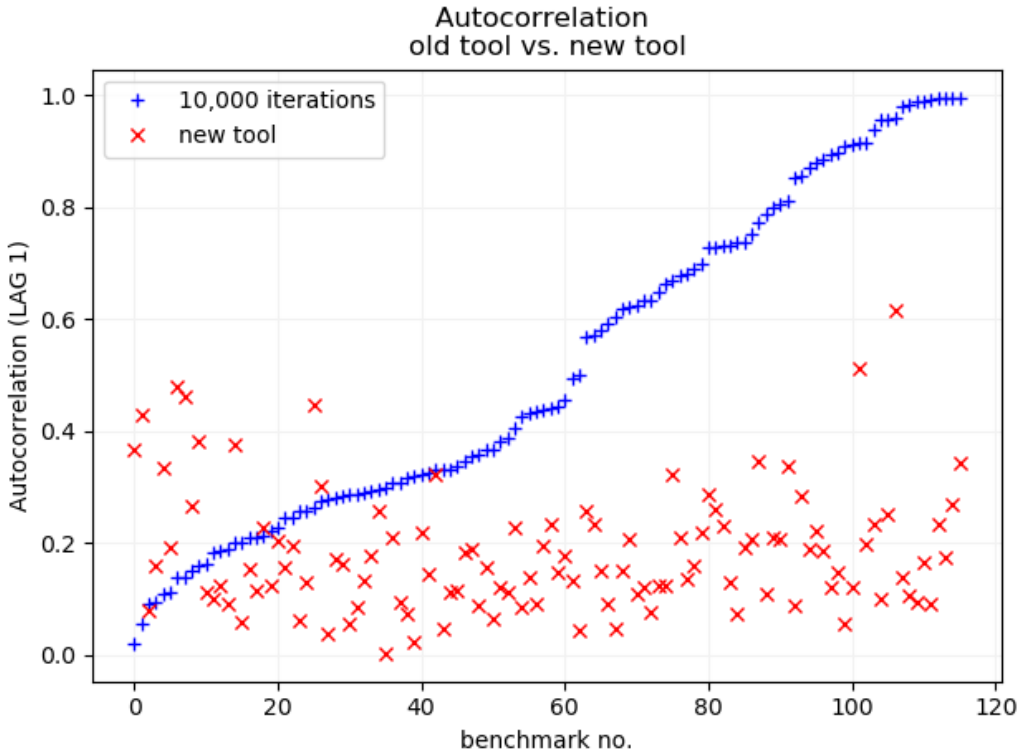


Figure 15: An autocorrelation comparison. Each point in the plot is the correlation at LAG 1 averaged over 15 repetitions of a benchmark, using the new tool (red) or $10,000$ iterations of the old tool (blue).

When using $10,000$ iterations, it is notable that despite averaging the autocorrelation over 15 repetitions, many benchmarks still register a very high autocorrelation. Meaning that they must have contained serial dependencies (most likely warm up phases), for most if not all of their 15 repetitions (as the autocorrelation is always stricly less than 1, getting an average of close to 1 would require: $\frac{1}{15}\sum_{i=1}^{15} ac_i = 1 \;\Rightarrow\; \forall i : ac_i = 1$).

7.4 Summary

In summary, we first looked at reliability as how much the sample mean varied from repetition to repetition of a benchmark. 10 iterations presented an extremely unreliable result, while $10,000$ iterations was quite reliable but also time-consuming - especially for longer-running benchmarks. A middle ground was found in the new tool, which only spent a fraction of the time, but still achieved a low variation in $\overline{x}$ across benchmark repetitions. We also made the observation that as execution time rose, the RSD of the sample mean fell.

To say something about the validity, we compared the sample mean obtained across the different configuration of tools. Using $10,000$ iterations of the old tool as a baseline, we saw that for the *tranpose.fut* benchmark suite, the new tool obtained very similar results in a fraction of the time. While for another suite, it sometimes differed as far as $\sim 15\%$, most notably when the execution time was low. We saw how altering the weights used between autocorrelation and RSE could reduce this deviation, and give the tool more time to converge (at a speed penalty).

Lastly, we saw how eliminating warm up phases enabled the tool to spend less time in order to obtain an accurate result.

8 Discussion

In this section we will first discuss whether the new tool lives up to its goals of automation and easing the job of benchmarking for the user. Lastly we will make suggestions for future work.

8.1 Reflection on solution

So does the new tool present a better solution for the user? As we saw in section 2.1, most users are concerned with a relative performance comparison of different setups. As the new tool sometimes deviates up to $\sim 15\%$ from the performance found using a higher iteration count, it could suggest that the error is too high to be useful for performance comparisons, especially if the performance difference to be investigated is small (e.g. $\sim 5\%$). However, it was mostly benchmarks with an execution time under $\sim 200\mu s$ that had such deviation, while the deviation was almost negligible for benchmarks over $400\mu s$. Thus it is mostly a problem for performance comparisons of very short-running programs.

However, it is inherently difficult to stop benchmarking at the 'right time' as we have no way of knowing if the performance seen so far is indicative of the performance to come in the next iteration. As we have seen, there exists plateaus of performance, and these are only apparent once we have actually left the plateau - thus trying to determine whether there exists another plateau, which has a higher sustained performance than the current, would require seeing that which has yet to happen. Thus, another way to view the solution produced in this project, is that it may serve as a predictor of the execution time of the next iteration of a benchmark. (For all intends and purposes it predicts it quite well, as the recent past is usually a good indicator of the near future - but with exceptions).

A problem is also that it may be hard to understand the performance characteristics of the benchmark from a single number, as its underlying distribution may change the story significantly. For instance, it would be beneficial for the user to also know the modality of the distribution, as well as the noise-level which was present while benchmarking. We will address these issues, and a few others in the future work section.

8.2 Future work

In the following we will look at four ideas for future work.

8.2.1 Data insight

The confidence-interval of the new tool may reveal whether the data is right-skewed or how much variation there was (a wider interval), however making such interpretations takes effort. Perhaps it would be better to give the confidence-interval in percentage of the mean, e.g. $[-1.3\%; +2.7\%]$, as it is easier to judge than absolute values. However, further analysis of the distribution of data would still require the user to inspect the raw *json* file (where the timed phase may have been removed, further complicating analysis).

To give the user a more thorough feel of the distribution underlying the results, it should be possible to develop a framework for plotting results. *Criterion* does this by giving the option of writing to a *.html* file that can be rendered in a browser.

As some users may find it tedious to write to a *.html* file and then open that separately, another path may be to output e.g. a histogram in the terminal itself, using a tool like *spark*[11].

Another measure that could bring a lot of insight, would be to provide an indicator of the modality of the distribution of execution times. If there were time-dependent noise such as a warm up phase or periodic heavy background load, it would show up as its own mode (e.g. as seen in figure 5). To detect this, one could use the *mvalue* approach:

$$m(x) = \frac{1}{M} \sum_{i=2}^{n} |x_i - x_{i-1}|$$

where M is the maximum bin value of the histogram, and n is the number of bins. Roughly speaking this measures modality as the number of elevation increases and decreases in a histogram. A high *mvalue* (e.g. > 2.8) indicates multi-modality.[12] Another technique, which may be better suited to deal with particularly noisy samples is to measure the valleys rather than the peaks[13].

8.2.2 Improving benchmarking reliability of short-running programs

In section 7, we saw that when benchmarking programs of $< 250\mu s$ duration, the average RSD of $\overline{x}$ was 6.6%, while for those above $250\mu s$ it was a more stable 1.2%. Thus the new tool produce more uncertain results for shorter-running programs.

There could be many reasons for this, it could just be an artefact of the more tail-heavy distribution which is characteristic of short-running programs as they are more vulnerable to *i.i.d* noise. Another factor could be that a small difference in $\overline{x}$ of e.g. $2\mu s$ influences a benchmark of short duration more than a long-running one (2% for a $100\mu s$ benchmark, but only 0.2% for a $1000\mu s$ benchmark).

To increase the reliability of the tool when benchmarking programs of short duration, it could be considered to find a way of running more iterations of these, without spending more time on the slower programs. One idea could be to use the cube root $\sqrt[3]{n}$ rather than $\sqrt{n}$ in the denominator of RSE. This would mean that running more iterations has less of an impact as a termination criteria. For instance for 4096 iterations, using the cube-root would increase the RSE by a factor of 4, while for a benchmark of 64 iterations just a factor of 2.

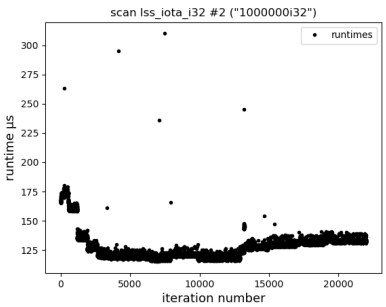
A future line of study could also seek to uncover if there is something else which is characteristic about a program of short-duration, which makes it a challenge to benchmark.

8.2.3 Aiming for *i.i.d*

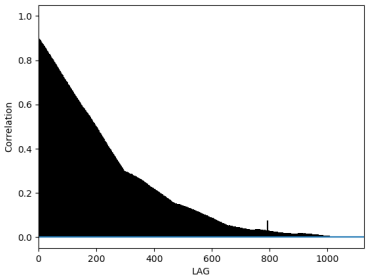
As seen in section 7, when running multiple repetitions of the new tool, the mean $\overline{x}$ obtained would have an average RSD of 3.8%. However, in our selection of RSE weights, we specified that we would like to stop benchmarking only when the RSE is below 1% (or lower if the autocorrelation is high). As the RSE can be interpreted as the expected standard deviation of $\overline{x}$, we see that there is a notable difference between the ideal of 1% deviation, and the actual 3.8% observed. This is due to the fact that the iterations aren't *i.i.d*, but rather have periods of higher and lower performance.

If we could find a way to only save the execution time of an iteration once we are sure that it is uncorrelated with the previous iteration, then we would be able to give an unbiased estimate of μ (and use the *CLT* to say something about the accuracy obtained, rather than have to repeat experiments to determine it).

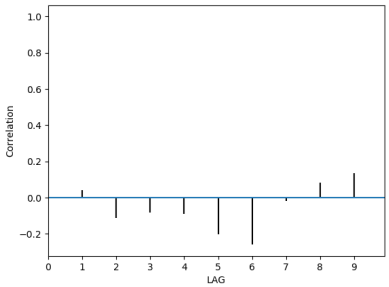
As we saw in the autocorrelation plots of figure 9, due to the (sometimes abrupt) time-dependent trends in the data, an iteration may be correlated with another iteration many lags later, e.g. an iteration x_i may still be correlated with iteration x_{i+1000}. To obtain *i.i.d* samples, we would have to discard a large fraction of all iterations, so that an iteration x_i can't serve as a predictor of x_{i+h}.



(a) time series



(b) autocorrelation



(c) autocorrelation keeping only 1 in 1024 iterations.

Figure 16: An example of discarding iterations to obtain a set of data with less serial autocorrelation. The benchmark is taken from *scan.fut* using $22,000$ iterations.

One way to go about doing this, would be to discard a fraction of iterations and then check the autocorrelation of the remaining. If there are still dependencies (e.g. *correlation > 0.25*) at some lag, we continue discarding data. At some point, (e.g. after keeping $\frac{1}{512}$ iterations) the autocorrelation will show that the iterations are uncorrelated. See figure 16 for an example. Here we see a benchmark with a high autocorrelation. If we keep only 1 in 1024 iterations, the autocorrelation reduces significantly *c)*.

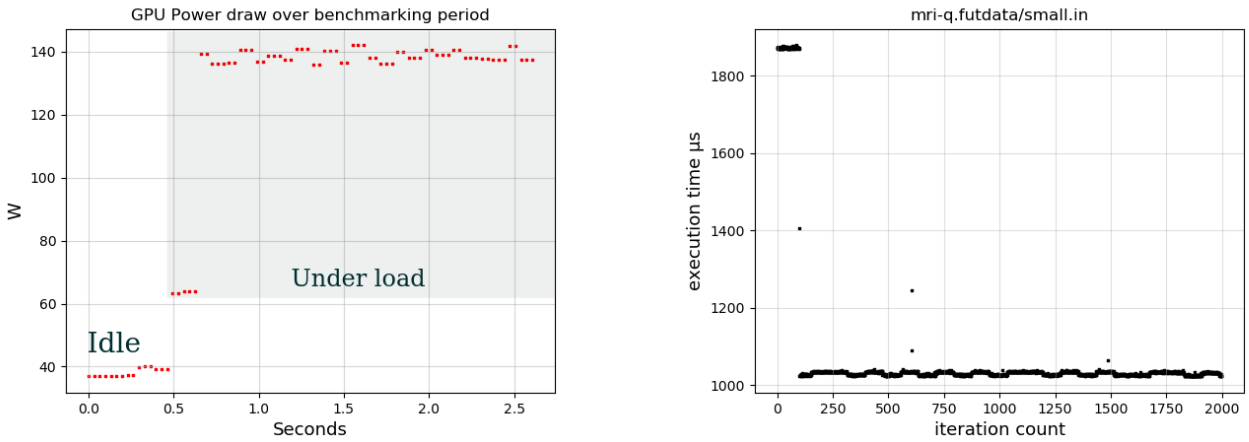
8.2.4 Understanding sources of noise

As we saw in section 5, the iterations of a benchmark have a high degree of time-dependent noise. Identifying whether a benchmark is likely to contain a warm up period or other time-dependencies *a priori*, will enable the tool make more informed decisions about how long to run the benchmark for.

One might hypothesize that the initial warm up phase, is correlated with the state of the GPU. As a first step we may check the power drawn by the GPU over the duration of a benchmark, to see if they are correlated. Figure 17 displays in plot *a)* the power drawn in *W*, and in *b)* the time series plot of the benchmark.

The first $0.5s$ was captured before the benchmark began, once the benchmark begins there is about $200ms$ with $\sim 63W$ being drawn, after which it consumes $\sim 140W$ for the rest of the benchmark. The correlation seems striking, as the first $200ms$ of the benchmark also has a significantly lower performance than the remainder. Even the small fluctuations in power draw, seem to show up as small performance swings in the time series plot.

This is however just a glance at a potential cause of warm up phases. Remembering fig-



(a) Power draw

(b) Time series plot

Figure 17: Examining relationship between the power draw of the GPU and execution time. *a)* displays the power trace in *W*, the first 0.5 seconds are with an idle load, then around $\sim 0.5s$ the benchmark starts and runs for a little over 2 seconds. *b)* shows the time series plot of the benchmark. The trace was obtained using the query functionality of *nvidia-smi*

ure 15, we saw that some benchmarks had a high autocorrelation for all of their 15 repetitions. Thus a line of investigation could seek to uncover what makes the performance of these benchmarks more time-dependent than others.

9 Conclusions

In summary, the tool developed in this project frees the user from specifying the iteration count needed to obtain reliable measurements. It does this by first running through a timed phase, which it discards if it is deemed too noisy. This led to measurements with less time-dependencies, which enabled the tool to finish sooner, as it would otherwise take a lot of iterations to reduce the effect of e.g. a warm up phase. Secondly, it enters a convergence phase where it looks to the RSE and autocorrelation to determine if the measurements are good enough to stop benchmarking, or if more iterations are needed. As the new tool doesn't rely on a specified iteration count, it is able to spend its time more evenly between benchmarking short and long-running programs, and can thus finish benchmarking sooner than the old tool.
Lastly, we saw how the techniques and statistical indicators used by the tool are heuristic in nature. For example the weights between RSE and autocorrelation could be used to adjust the trade-off between speed and accuracy, while increasing the initial timed phase may enable the tool to discover another plateau of higher performance, which it otherwise would have missed.

Although arriving at reliable measurements now takes less work and reasoning than when using its predecessor, the new tool is by no means a silver bullet. Trying to provide a single number that reflects the performance of a program is a dimensionality reduction that potentially losses information along the way, hence it is perhaps best to interpret the result as a predictor of the execution time to come, rather than an absolute truth.

A Benchmarking methodology

- The benchmark suites used in this project are all taken from https://github.com/diku-dk/futhark-benchmarks.
- All benchmarks are run on the same machine with an Nvidia A100 Tensor core GPU with 40 GB of memory, and an AMD Epyc 7352 24-core CPU.
- Unless otherwise specified, all benchmarks are run (to the extent possible) without other processes drawing resources from the GPU, this was done by checking for running processes using *nvidia-smi* while benchmarking (the machine is a shared work-station accessed through ssh).
- The kind of use of the benchmarked program we want to estimate, is that achieved through repeated steps in a simulation i.e. each execution of the benchmark is done in a 'warm' state, where the program code and data is already resident in memory. Flushing the cache between runs is thus not desirable.

References

[1] T. Kalibera and R. Jones. *Quantifying Performance Changes with Effect Size Confidence Intervals*. 2020. DOI: 10.48550/ARXIV.2007.10899. URL: https://arxiv.org/abs/2007.10899.

[2] *The Futhark Programming Language*. https://futhark-lang.org/. 2022.

[3] T. Kalibera and R. Jones. "Rigorous Benchmarking in Reasonable Time." In: 48.11 (2013), pp. 63–74. DOI: https://doi.org/10.1145/2555670.2464160.

[4] T. Hoefler and R. Belli. "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results." In: (2015), pp. 1–12. DOI: https://doi.org/10.1145/2807591.2807644.

[5] Google Inc. et al. *benchmark*. https://github.com/google/benchmark. 2022.

[6] Bryan O'Sullivan. *Criterion*. https://github.com/haskell/criterion. 2022.

[7] Manikandan S. "Measures of central tendency: Median and mode." In: 2.3 (2011), pp. 214–215. DOI: doi:10.4103/0976-500X.83300.

[8] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2010. ISBN: 978-2-940222-40-7.

[9] NCSS Statistical software. *Chapter 164: Lag Plots*. https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Lag_Plots.pdf. 2019.

[10] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Springer, 1993. ISBN: 978-0-412-04231-7.

[11] holman. *spark*. https://github.com/holman/spark. 2022.

[12] Brendan Gregg. *Frequency Trails: Modes and Modality*. https://www.brendangregg.com/FrequencyTrails/modes.html. 2020.

[13] Andrey Akinshin. *Lowland multimodality detection*. https://aakinshin.net/posts/lowland-multimodality-detection/. 2020.