J Supercomput
https://doi.org/10.1007/s11227-018-2252-6

Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware

Dénes Bán[1] · **Rudolf Ferenc**[1] · **István Siket**[1] · **Ákos Kiss**[1] · **Tibor Gyimóthy**[2]

Abstract Heterogeneous computer environments are becoming commonplace so it is increasingly important to understand *how* and *where* we could execute a given algorithm the most efficiently. In this paper we propose a methodology that uses both static source code metrics, and dynamic execution time, power, and energy measurements to build gain ratio prediction models. These models are trained on special benchmarks that have both sequential and parallel implementations and can be executed on various computing elements, e.g., on CPUs, GPUs, or FPGAs. After they are built, however, they can be applied to a new system using only the system's static source code metrics which are much more easily computable than any dynamic measurement. We found that while estimating a continuous gain ratio is a much harder problem, we could predict the gain *category* (e.g., "slight improvement" or "large deterioration") of porting to a specific configuration significantly more accurately than a random choice, using static information alone. We also conclude based on our benchmarks that parallelized implementations are less maintainable, thereby supporting the need for automatic transformations.

✉ István Siket
siket@inf.u-szeged.hu

Dénes Bán
zealot@inf.u-szeged.hu

Rudolf Ferenc
ferenc@inf.u-szeged.hu

Ákos Kiss
akiss@inf.u-szeged.hu

Tibor Gyimóthy
gyimothy@inf.u-szeged.hu

1 Department of Software Engineering, University of Szeged, Szeged, Hungary

2 MTA-SZTE Research Group on Artificial Intelligence, Department of Software Engineering, University of Szeged, Szeged, Hungary

Published online: 02 February 2018

Keywords Green computing · Heterogeneous architecture · Performance optimization · Power-aware execution · Configuration selection

1 Introduction

As technological advancements make GPUs—or other alternative computation accelerators—more widespread, it is increasingly important to question whether the CPU is still the most efficient option for running specific applications. In this paper we describe a method for deriving prediction models that can select the execution configuration best suited for a given algorithm with regards to one of three different aspects: time, power, and energy consumption. These models are built by applying various machine learning methods where the predictors are calculated from the source code (using static analysis techniques during compilation time) and the output of the models is an estimate of the gain (in terms of time, average power, or energy) the algorithm could reach if it was executed on a specific processing element compared to the sequential execution on a single core CPU. This estimate could be a real number (if the machine learning approach is a regression) or an interval the exact ratio would fall in (if it is a classification).

To build the desired prediction models, first we took a number of algorithms—referred to as benchmarks—that were implemented in various languages, e.g., in C or OpenCL C, allowing us to execute them on different computing units, like the CPUs, GPUs, or FPGAs. We manually identified and tagged the phases in these benchmarks, including the kernels—i.e., the main computational parts—as well as the initialization/cleanup and data transfer steps with markers for a dynamic runtime and energy consumption measurement framework and a static analyzer. After this, we extracted multiple size, coupling, complexity, and control flow based static source code metrics from the kernels of these analyzed systems, and aggregated these to system level for every benchmark. Then we performed measurements on the time, energy, and power required to run these kernels—and the other delineated parts—on different platforms and with different input sizes. Finally, we applied multiple machine learning methods that use the calculated data to build the models—one for each phase, aspect, platform, measurement aggregation method and machine learning technique.

There are only static prerequisites for using the created models; if users want to apply them to a new system, they only need to extract static source code metrics as listed in this paper and input them to one of the models to predict the best platform for running the kernel and also to predict the expected gain. In this paper we describe a possible method for creating such models through a concrete experiment and discuss their benefits as well as possible ways for improving them even further. We also apply a previously established maintainability model to both the sequential and parallel implementations of the benchmark programs for comparison.

The two research questions we aim to answer are the following:

- *RQ1* Can the performance gain of porting an algorithm to another computation element be predicted using static information only?
- *RQ2* How does parallelization affect the maintainability of a subject system?

In order to encourage further research in this area, we provide source code for the RMeasure library [19] and the tagged benchmarks [9] along with their static metrics and dynamic measurements [8].

The paper[1] is organized as follows: in the next section we discuss related work. Then, in Sect. 3 we describe our methodology in detail. In Sect. 4 we introduce the used benchmarks, while in Sect. 5 we describe the way we performed the dynamic measurements. Afterward, in Sect. 6 we describe the static metrics extraction in detail, along with the metric normalization and model evaluation. In Sect. 7 we show the results that we have achieved. Finally, in Sect. 8 we draw conclusions and outline future work.

2 Related work

As heterogeneous execution environments became more and more prevalent in recent years, it also became increasingly important to study their individual and relative performances. There is a multitude of related work in the area with fundamentally different approaches.

Some researchers tried to characterize a particular platform alone. For example, Ma et al. [22] focused only on GPUs and built statistical models to predict power consumption. Brandolese et al. [10] concentrated on CPUs by statically analyzing C source code and estimating their execution times. For the OpenMP environment, Li et al. [21] derived a performance model, while Shen et al. [28] compared OpenMP to OpenCL using some of the same benchmark systems we used. Note that although we share some source benchmarks with Shen et al., we focus on predicting performance instead of analyzing the actual, dynamic performance of concrete implementations. For FPGAs, Osmulski et al. [24] introduced a tool to evaluate the power consumption of a given circuit without needing to actually test them. It is also evident from these studies that most of this type of research targets a single aspect (time or power). We on the other hand, consider multiple platforms and multiple aspects as our goal is to predict the optimal environment from static information alone.

Others are more closely related to our current work as they focus on cross-platform optimization. Yang et al. [32] generalized the expected behavior of a program on another platform by extrapolating from partial execution measurements while Takizawa et al. [30] aimed at energy efficiency by dynamically selecting the execution environment at run time. Unlike these works, we use dynamic information only for building the prediction models which then can be used with static data alone. Another, even more similar approach is presented by Grewe and O'Boyle [17], aiming to partition tasks between the CPU and the GPU using static program features and machine learning. Their methodology heavily utilizes memory layout and data-related metrics, and considers the effect of distributing the subject algorithms among multiple platforms in different percentages (instead of completely porting them to one of the target platforms only). Our study, on the other hand, prefers code structure and control flow

[1] This journal paper is an extended version of our earlier conference paper [7].

based metrics, compares speedup to native implementations, and incorporates power and energy measurements, as well.

A subset of these cross-platform works concentrate on compiled or intermediate program representations. Kuperberg et al. [20] analyzed components and platforms separately to avoid a combinatorial explosion. They built parametric models for performance prediction, but it requires microbenchmarks for each platform and works with Java bytecode only. Marin and Mellor-Crummey [23] also processed application binaries and built architecture-neutral models which were then used to estimate cache misses and execution time on an unknown platform. One key difference of these studies compared to our approach is that we use the source code of the training benchmarks and not their compiled forms.

Still other research touched on the maintainability of parallelized implementations. Pflüger and Pfander [25] performed a fine-tuning case study on their SG++ library while trying to preserve source code maintainability. They also concluded—among other lessons learned—that maintainability deterioration is a natural side effect of performance optimization and that automatic code generation and domain specific languages could help substantially. Another study in this area was done by Brown et al. [11] who examined that starting with a higher abstraction level language and then transforming to heterogeneous platforms could yield comparable or even better performance without degrading developer productivity. While these works considered a more subjective measure of maintainability, we aim to quantify the objective differences between sequential and parallel versions.

3 Methodology

This section contains the detailed description of our concept of a quantitative prediction model and how it is built. Using source code metrics produced by static source code analysis, our model is able to predict quite adequately not only the computing unit that allows the fastest or most energy efficient execution of a given program but also the amount of improvement in terms of performance, power, and energy consumption that can be expected. For even finer grained measurements, we only considered the core of the algorithms, the computing kernels represented in each benchmark program and none of their preparation steps, e.g., OpenCL platform or device initializations, etc. We achieved this by "tagging" the appropriate parts of the benchmarks with a special macro pair. We also used this tagging approach to separate the dynamic measurements into initialization/cleanup, data transfer, and kernel execution stages. The model is built following these steps:

- Extract multiple size, coupling, complexity, and control flow based metrics from the tagged kernels of the analyzed systems.
- Collect measurements of the time and power required to run the parts of these systems on different platforms and with different input sizes.
- Use various machine learning algorithms to build models that are able to predict the gain that a kernel with a specific set of metric values can produce when migrated to a given platform.

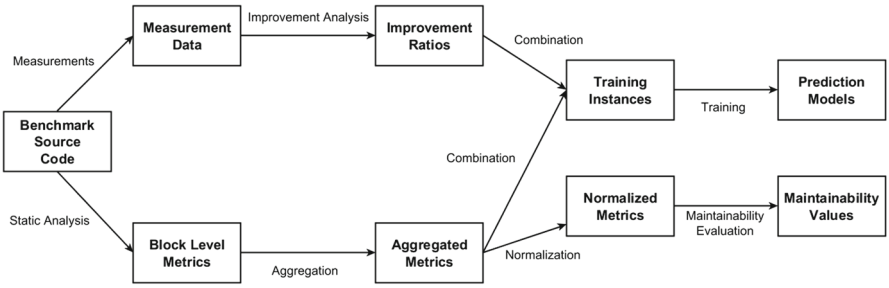

Fig. 1 Main steps of the model creation process

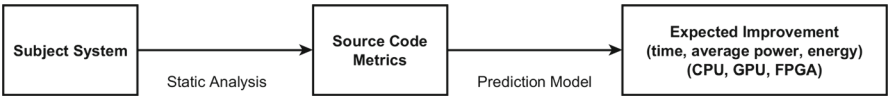

Fig. 2 Usage of a previously built model on a new subject system

Additionally, we compare the sequential and parallel benchmark implementations to study possible trends in their maintainability scores. The steps and intermediate states of our methodology are outlined in Fig. 1. Each of these steps will be detailed in their dedicated sections:

- The selected benchmarks in Sect. 4,
- the dynamic measurements in Sect. 5,
- the static analysis in Sect. 6.1,
- the selected metrics relevant for representing the encapsulated algorithms in Sect. 6.2,
- the metric aggregation process and its result in Sect. 6.3, where a single set of metrics is collected for every benchmark,
- the metric normalization and the model evaluation in Sect. 6.4,
- the combination of different dynamic measurements into gain ratios in Sect. 7.1,
- the model training and its results in Sects. 7.2 and 7.3, where we use a number of machine learning algorithms to build the prediction models we aim for, and finally,
- the maintainability change between sequential and parallel benchmark versions in Sect. 7.4.

Once a prediction model is in place, new systems can be analyzed to predict how much improvement one can expect when migrating their sequential implementation to another platform. Figure 2 depicts the steps of applying a model to a new subject system (unknown to the trained model). To determine the expected gain of a new system on a specific platform, only the same source code metrics need to be calculated (via static analysis) that we used for training the model, and based on them the model can compute an estimated improvement ratio.

4 Benchmarks

The subject systems for our model training came from three self-contained benchmark suites: *Parboil*, *Rodinia*, and *PolyBench/ACC*. The Parboil suite [29] provides a combination of sequential, OpenCL, and OpenMP implementations for 11 programs. Rodinia [12] contains 18 benchmark programs with OpenCL and OpenMP implementations but without the sequential equivalents. PolyBench/ACC [16] is an extended version of PolyBench [26] that contains 29 programs in multiple implementations. In this work, we measured a subset of these three benchmark suites (some programs were excluded either because of dynamic problems—they were not implemented in all necessary languages or could not be executed on all necessary platforms—or static issues like a faulty build or inherent include errors). The final number of systems that have both metric data and measurements (for both CPU, GPU, and FPGA) is 3 for Parboil (mri-q, spmv, and stencil), 4 for Rodinia (bfs, hotspot, lavaMD, and nn), and 9 for PolyBench (atax, bicg, convolution-2d, doitgen, gemm, gemver, gesummv, jacobi-2d-imper, and mvt).

5 Measurements

In order to train our configuration prediction models, we needed to obtain dynamic measurements for execution time, power consumption, and energy usage. We emphasize that although these measures are labeled "dynamic," they have no connection to, e.g., memory usage, caching, aliasing, or any other similar runtime characteristics commonly found in "dynamic" program analysis tools. The qualifier is intended only as a comparison to the static nature of the source code metrics, and because time, power, and energy measurements require program execution. We compiled the benchmarks with `g++` 4.8.2 using standard `-fopenmp` or `-lOpenCL` flags and ran them on an Ubuntu 14.04 LTS installation on a hardware platform built from 2 Intel Xeon E5-2695 v2 CPUs (30M Cache, 2.40 GHz), 8 × 8 GB of DDR3 1600 MHz memory, a Supermicro X9DRG-QF mainboard, an AMD Radeon R9 290X VGA card, and an Alpha Data ADM-PCIE-7V3 FPGA card. Execution time could have been easily checked using software-based timers only. Power and energy, on the other hand, required a more sophisticated approach. So we additionally applied a universal hardware-extension solution and used our own open-source RMeasure library [19] that provides a unified API hiding the implementation details.

Section 5.1 briefly overviews some of the already available performance and energy consumption measurement methods while Sect. 5.2 introduces RMeasure and how it incorporates these methods. Finally, Sect. 5.3 discusses measurement precision.

5.1 Measurement methods

For the purpose of this overview, we classify methods either as *internal*—if the component under measurement can introspect its own behavior and expose the information typically via performance counter registers—or as *external*—if some external hardware is needed for the measurement.

The most well-known internal measurement method is Intel's running average power limit (RAPL) [5] solution introduced in their Sandy Bridge microarchitecture, which gives access both to cycle count and energy consumption data for different physical domains—like sockets, core and uncore elements, and DRAM—through model-specific registers (MSRs). The two major GPU manufacturers, AMD and NVIDIA, both provide libraries and APIs to access similar hardware performance counters of their graphics processors. However, the publicly accessible AMD GPU Performance API [3] provides no access to power or energy consumption counters, while the NVIDIA Management Library (NVML) [1] is able to report the current power draw only for the high-end boards, like Tesla K10/20/40 cards. Internal methods are not limited to the x86 world only, recent ARM cores have built-in performance monitoring units as well. However, up until the latest ARMv8 processors, these are performance-only with no unified access to power data.

When internal methods are not available—as visible from the above paragraph, this happens mostly for power usage monitoring— external solutions have to be applied. The physics behind most of such external metering methods is similar: a shunt resistor is inserted into the power line of a component, the voltage drop is measured on this resistor, and an instrumentation amplifier is used to make this voltage readable by conventional ADCs (such as used by embedded devices, microcontrollers, or even external test equipments, e.g., oscilloscopes). Knowing the value of the resistor and the voltage of the power rail, the momentary power of the measured component is easily computed with the $P = U_{\text{rail}} * (U_{\text{drop}}/R_{\text{shunt}})$ formula at any given sampling point, while integrating these results over time gives the energy consumption. Some ARM devices have measurement points, to which an ARM Energy Probe [4, Chapter 11] can be attached that works based on this concept and emits measurement result on a USB interface. Some accelerator cards are also instrumented for power measurements using this technique. E.g., the Xilinx Virtex VC709 FPGA development board has shunt resistors inserted into all internal power rails, and the resulting analog values are fed to a DC/DC converter controller chip, which reports power usage information digitally via the external Power Management Bus serial interface.

Since not all computation devices in our platform support a built-in power and energy measurement method, we designed and implemented a universal solution based on the above principles. We designed a printed circuit which can be conveniently placed inside the platform and holds the shunt resistor and amplifier needed for measuring a single computation device or power line. For each computation device, we used one of these circuits. To make the insertion of the circuits into the power lines the least intrusive and reversible, we did not cut the wires of the power supplies, but we obtained different extension cords and modified them to be used with the measurement PCBs. For both CPU sockets, their 8-pin EPS12V power connectors are intercepted. For the GPU card, as it draws power both from the PCI-Express slot and from an additional PCI-Express power connector, both its rails are routed to a PCB (the former with the help of a PCI riser). Finally, we used a computer-controlled multi-channel measurement device, a PicoScope 4824 oscilloscope, to capture the output of the PCBs over time.

5.2 The RMeasure Library

The main goal of our RMeasure performance and energy monitoring library [19] is to provide a unified interface for retrieving performance and energy consumption data about the system, independent of the applied and/or available measurement methods. Thus, the interface handles built-in (e.g., performance counter-based) and external (e.g., shunt and oscilloscope-based) measurements alike and hides all implementation details.

The core interface of the library consists of only a few base classes, which represent the concept of a measurement method (e.g., RAPL counter-based or PicoScope-based) and stand for an actual measurement and its results. All supported measurement methods expose what components of the system it can measure and what kind of information it is able to provide. The components of the system are identified by their HPP-DL component IDs [27]. The HPP-DL path notation provides a manufacturer- and architecture-independent abstraction layer to specify measured hardware components. The measured information can be an arbitrary combination of the following:

- energy consumption (in Joules),
- minimum, maximum, and average power (in Watts),
- elapsed time (a.k.a. wall-clock time), and time spent in kernel or in user mode (in seconds).

The API of RMeasure is intentionally simple; however, it can have several components working together under the hood in a full configuration. The main component of RMeasure exposes the public API. However, there are certain tasks that need to be separated from the main part of RMeasure. Specifically, if the external oscilloscope-based measurement method is enabled, the control service of the scope—whose responsibility is to control the oscilloscope via the PicoScope API [2], configures the sample rate and the channels, runs in a gap-less continuous streaming mode and retrieves the raw data—needs to be run on a separate unit, because processing the data requires significant CPU power that could distort the measurements if ran on the measured computer. The RAPL-based internal measurement method also has specific needs, since accessing the machine specific registers needs root privileges. Therefore, it is useful to be organized into a separate service. The setup of a full measurement configuration is shown in Fig. 3.

5.3 Measurement precision

Since a service is constantly running in the background on the same computer as the measured code (at least for RAPL counters), it causes additional CPU load and therefore additional power consumption, which can have an effect on the precision of the measurements. To understand the introduced overhead, we took two sets of measurements, one using the service, and another with a slightly modified library setup where no services were running on the measured system. In the latter case, the application directly accessed the RAPL energy counters, thus requiring root permission. According to the results, the overhead on energy consumption, average power and running time were all below 5% on average, which we deemed acceptable. Therefore, we stuck

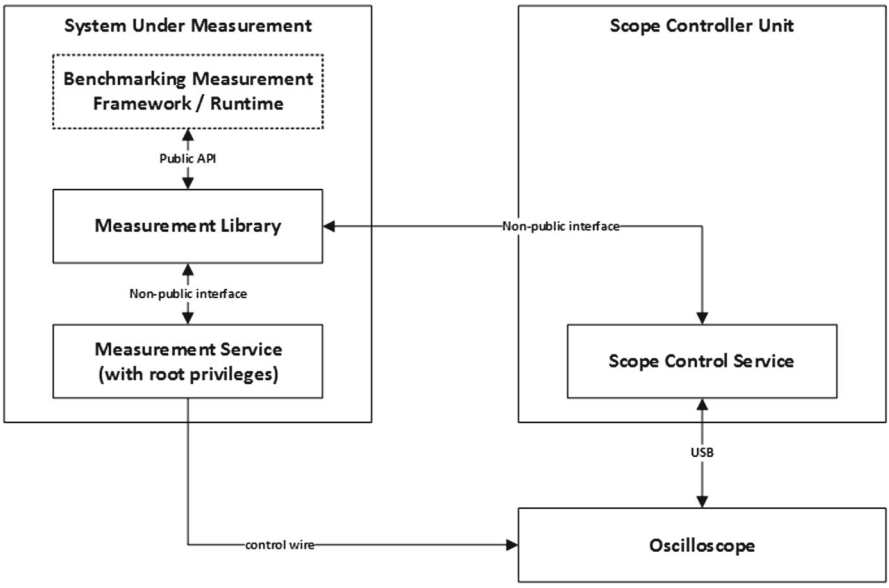

Fig. 3 RMeasure Library overview

to the service-based approach, as that is more universally applicable (no need for root privileges).

6 Metrics extraction

In this section we describe the process of static analysis to calculate static source code metrics. As outlined in Sect. 3, this static source code information is used both to predict the expected improvement of a given aspect (time, average power, or energy) for a target execution configuration and to compare the maintainability of the sequential and parallel implementations. We list all the selected metrics used in the machine learning algorithms as predictors, present how we aggregated the block level metrics to system level, how we normalized them into the [0, 1] interval, and finally, how we calculated the corresponding maintainability scores.

6.1 Static analysis

For metrics calculation, we ran our static code analysis tool [13] on all three benchmark suites. Instead of using method or function level granularity for metrics as "atoms," we used block level metrics to isolate the characteristics of the kernels and exclude every "wrapper" and "initializer" functionality. We calculated these block level metrics by analyzing only the appropriate source code parts between our special tagging macros.

This analysis was performed on both the sequential and OpenCL variants of every benchmark because even though the prediction models require metrics only from the

former (the "before" state of a hypothetical parallel transformation), we also needed metrics from the latter (the "after" state) to be able to compare them for our second research question. It should be mentioned that, as OpenCL C is very close to standard C syntax, we treated the source code of the OpenCL variants as C for the sake of the analysis, skipping nonconforming syntactic elements (e.g., `__kernel` and `__global` tokens).

Please note that the current approach does not use any dynamic information from the source code yet, metrics are static, and do not contemplate runtime problems such as memory aliasing, caching and memory allocation.

6.2 Metric definitions

The metrics we computed and used as predictors for the classifications and regressions are listed below. It should be noted that the word "block" may refer to either basic blocks (which is a control flow concept) or the above-mentioned tagged source code blocks. To help differentiate between the meanings, we always add a "(tagged)" prefix in the ambiguous cases. Also note that metrics starting with "ft" are adopted directly from the feature list of the Milepost GCC compiler [15].

- *Lines of code (LOC)* is the count of every line in a block.
- *Logical lines of code (LLOC)* is the count of all non-empty, non-comment lines in a block.
- *Nesting level (NL)* for a block is the maximum of the control structure depth. Only *if, switch, for, while and do…while* instructions are taken into account.
- *Nesting level else-if (NLE)* for a block is the maximum of the control structure depth. Only *if, switch, for, while and do…while* instructions are taken into account but *if…else if* does not increase the value.
- *McCabe's cyclomatic complexity (McCC)* is defined as the number of decisions within the specified block plus 1, where each *if, for, while, do…while and ?: (conditional operator)* counts once, each N-way *switch* counts $N + 1$ times and each *try* with *N catch*es counts $N + 1$ times. (E.g., *else* does not increment the number of decisions.)
- *Number of statements (NOS)* is the number of statements inside a block.
- *Number of outgoing invocations (NOI)* for a block is the number of all function invocations inside it.
- *Loop nesting level (LNL)* is the maximum loop depth inside the block. (The same as NL, but without the *if*s, *switch*es, *try*s and ternary operators). We also computed LNL1, LNL2, and LNL3 that contain the number of loops that were at depths 1, 2, and 3, respectively.
- *Number of expressions (EXP)* is the number of expressions in the block.
- *Number of array accesses (ARR)* is the number of array subscript expressions in the block. Also, ARR% is defined as the ratio ARR/EXP.
- *Number of multiplications (MUL)* is the number of multiplications (* or *=) in the block. Also, MUL% is defined as the ratio MUL/EXP.
- *Number of additions (ADD)* is the number of additions (+ or +=) in the block. Also, ADD% is defined as the ratio ADD/EXP.

- *ft1* is the number of basic blocks in the (tagged) block.
- *ft2* is the number of basic blocks with a single successor.
- *ft3* is the number of basic blocks with two successors.
- *ft4* is the number of basic blocks with more than two successors.
- *ft5* is the number of basic blocks with a single predecessor.
- *ft6* is the number of basic blocks with two predecessors.
- *ft7* is the number of basic blocks with more than two predecessors.
- *ft8* is the number of basic blocks with a single predecessor and a single successor.
- *ft9* is the number of basic blocks with a single predecessor and two successors.
- *ft10* is the number of basic blocks with a two predecessors and one successor.
- *ft11* is the number of basic blocks with two successors and two predecessors.
- *ft12* is the number of basic blocks with more than two successors and more than two predecessors.
- *ft13* is the number of basic blocks with number of instructions less than 15.
- *ft14* is the number of basic blocks with number of instructions in the interval [15, 500].
- *ft15* is the number of basic blocks with number of instructions greater than 500.
- *ft21* is the number of assignment instructions in the (tagged) block.
- *ft22* is the number of binary integer operations in the (tagged) block.
- *ft23* is the number of binary floating point operations in the (tagged) block.
- *ft25* is the average number of instructions in basic blocks.
- *ft33* is the number of switch instructions in the (tagged) block.
- *ft34* is the number of unary operations in the (tagged) block.
- *ft40* is the number of assignment instructions with the right operand as an integer constant in the (tagged) block.
- *ft41* is the number of binary operations with one of the operands as an integer constant in the (tagged) block.
- *ft42* is the number of calls with the number of arguments greater than 4.
- *ft45* is the number of calls that return an integer.
- *ft46* is the number of occurrences of integer constant zero.
- *ft48* is the number of occurrences of integer constant one.

Another, slightly different metric is the *Input size*, i.e., the relative size of the input the encapsulated algorithm will process. We categorized input sizes into five possible bins: mini, small, medium, large, and extra large. Note that these were already given with our subject benchmarks and as "small" is relative to the algorithm in question, we cannot give exact thresholds.

Also note that all of these metrics can be statically computed. Nevertheless, they can help in predicting dynamic behavior, as we will demonstrate in Sect. 7.

6.3 Metrics aggregation

The output of the static analysis is a set of metrics for every block in the sequential and OpenCL configuration for every benchmark system—except for the input size, which is already system level. These represent the captured algorithms and are the

correct basis for further study because the dynamic metrics also express how much improvement can be expected compared to the sequential configuration.

To aggregate these metrics into a system-level set for each benchmark, we combined the metrics of multiple blocks. The method of combination is customizable per metric, and we chose the most naturally expressive for each:

- addition minus one for McCC (the minus one accounts for the default execution path the separate block gets on its own and is now not needed),
- maximization for NL, NLE and LNL,
- recalculation for averages like ARR% and ft25 (i.e., their numerators and denominators are aggregated separately and the average is computed again at the end), and finally
- addition for the others, as they are all counts of different occurrences.

This way we got one single set of metric values for every benchmark, capturing many of its characteristics.

6.4 Metrics normalization and maintainability evaluation

The metrics we calculated so far are complete but absolute and therefore cannot be compared to each other. E.g., we have no way to tell what a `McCC` of 5 or a `NL` of 3 *means* compared to each other. For this reason, we normalize each metric value into the [0, 1] interval using empirical cumulative distribution functions (or ECDFs) [31]. This method produces relative numeric values that show the ratio of how many of the available data points are smaller than a certain metric. These values are relative because they depend on the context they were evaluated in.

Then, using these normalized metrics as a base, we perform a weighted aggregation to produce more abstract scores, in multiple steps. First, we compute intermediate values from certain static metrics, namely analysability, modifiability, reusability, testability, and modularity. Then aggregate those intermediate values further to reach a single maintainability indicator. Which source code metric influences which intermediate characteristic, how much, and how those are combined into a final result is dependent on expert votes.

The method, the votes, and the maintainability model itself is discussed in detail in the REPARA report D7.4: Maintainability models of heterogeneous programming models [14]. This experiment could be considered a replication of those results, only on an extended and more fine-tuned benchmark set.

7 Results

In this chapter we describe the prediction models we built using the static and dynamic data outlined above. We also present the validation results of the models created by different machine learning algorithms. The results are validated with 10-fold cross-validation [6]. Finally, we compare the maintainability scores of the “before” and “after” versions of a hypothetical parallel transformation.

7.1 Training instances

After we have obtained measurements for each aspect (time, average power, energy) in each configuration (sequential, OpenCL on CPU, OpenCL on GPU, and OpenCL on FPGA) for each code region (initialization/cleanup, data transfer, or kernel execution) for each input size (mini, small, medium, large, or extra large) of each benchmark system, the question is how fast (or energy efficient) a given algorithm will be.

However, improvement is a characteristic hard to describe in absolute terms because static metrics alone are not expected to fully describe the dynamic behavior of a program. For example, it might happen that two separate programs yield the same source code metric values, but can have significantly different running times. If our models learned from one of them that migrating to OpenCL on GPU can produce a shorter runtime that would not mean anything unless we also knew how much of an improvement that decrease is compared to its original runtime. This is why instead of absolute measures (like seconds or Joules) we used relative values (ratios).

So, after aggregating the source code metrics (detailed in Sect. 6.3) we converted the dynamic measurements to the above-mentioned ratios that could be classes in a machine learning experiment. We did so by dividing the values measured on a parallel computation unit (e.g., the runtime of a kernel) by their original, sequential counterparts: values below one signaled improvement and values greater than one indicated deterioration. We calculated these ratios for every input size of every benchmark and then combined them with the static metrics to finalize our training databases, each containing over 50 instances.

We also experimented with different measurement aggregation methods that affect *what* exactly do we consider the power/energy consumption of a given program execution: One way is to take only the values of the chosen hardware itself into account (denoted as "Single" in later tables). Another is to always add the CPU's measurements to the total, since there needs to be a CPU in the system to send tasks to the selected accelerator (denoted as "With CPU"). Finally, we can view the system as a whole and sum the total power/energy consumption that the different hardware components produced (denoted as "All").

The fine granularity of the tagging provides yet another possible dimension to the study: do we predict the improvement for the kernel only ("Kernel") or for the whole ("Full") program (including initializations and data transfers)? We could also create training datasets for the separated initialization/cleanup ("Init") and data transfer ("Transfer") phases.

This results in a training set for each phase–platform–measurement aggregation method–aspect tuple. As an example, part[2] of the Kernel-Single-GPU-Time training instances can be seen in Table 1.

[2] The full tables are part of the Online Appendix [8]."

Table 1 Training instances from the kernels of all benchmark suites with Single-GPU-Time improvement ratios

Benchmark	LOC	LLOC	NL	NLE	McCC	…	Input size	Ratio
Poly_atax	16	14	4	2	2	…	1	1250.85
Poly_atax	16	14	4	2	2	…	2	22.96474
Poly_atax	16	14	4	2	2	…	3	1.094502
Poly_atax	16	14	4	2	2	…	4	1.068928
Poly_bicg	17	15	3	2	2	…	1	5287.375
Poly_bicg	17	15	3	2	2	…	2	40.54651
Poly_bicg	17	15	3	2	2	…	3	1.509625
Poly_bicg	17	15	3	2	2	…	4	1.482663
Poly_conv2d	16	12	2	2	2	…	1	1844.5
Poly_conv2d	16	12	2	2	2	…	2	2.265508
Poly_conv2d	16	12	2	2	2	…	3	0.252187
Poly_conv2d	16	12	2	2	2	…	4	0.739263
Poly_conv2d	16	12	2	2	2	…	5	0.682129
…	…	…	…	…	…	…	…	…

7.2 Machine learning

Using the datasets like the one shown in Table 1, we were able to run various machine learning algorithms to build models that can predict the gain ratios based on the source code metrics. The tool used for machine learning was `Weka` [18].

We have experimented with both classification and regression algorithms. While the regression models were trained for the continuous improvement ratios, the classification algorithms required classes. Thus, we have applied a discretizing preprocessing filter to our training data to divide the ratios into 5 (and 3) bins, or "improvement categories." These bins ranged from "large deterioration" to "large improvement" with automatically computed thresholds. This discretization and bin selection represents a compromise between our previous approach of only choosing the best platform and the regression algorithms that aim to exactly estimate improvement.

7.3 Validation of the models

In the following we show the results of the experiments where we applied our full set of source code metrics plus the input size as predictors. The achieved accuracy values for the Full, Kernel, Init and Transfer phases are shown in Tables 2, 3, 4 and 5, respectively. Each of these tables has three layers of headers for the measurement aggregation method (Single, With CPU, All), the target platform (CPU, GPU, or FPGA) and the measured dynamic aspect (*T*ime, *P*ower or *E*nergy), while the rows show how each tested algorithm performed on the corresponding problem. The rows are separated into three groups for regression algorithms, 5 bin and 3 bin classifications. All three row

Table 2 Full prediction accuracies of the built models

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E
ZeroR	0.65	0.45	0.65	0.39	0.45	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.37	0.40	0.65	0.38	0.65
LinReg	0.65	0.45	0.65	0.39	0.19	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.30	0.40	0.65	0.38	0.65
Mult.Perc.	0.01	0.37	0.01	0.12	0.66	0.44	0.09	0.07	0.10	0.01	0.37	0.01	0.12	0.42	0.15	0.09	0.34	0.12	0.01	0.45	0.03	0.12	0.62	0.15	0.09	0.15	0.10
REPTree	0.10	0.45	0.12	0.63	0.26	0.83	0.23	0.36	0.25	0.10	0.45	0.12	0.63	0.39	0.70	0.23	0.05	0.02	0.10	0.39	0.12	0.63	0.26	0.70	0.23	0.08	0.25
M5P	0.30	0.13	0.32	0.65	0.50	0.79	0.47	0.10	0.48	0.30	0.13	0.32	0.65	0.17	0.72	0.47	0.13	0.48	0.30	0.12	0.32	0.65	0.28	0.71	0.47	0.01	0.48
SMOreg	0.06	0.25	0.08	0.15	0.70	0.10	0.04	0.37	0.04	0.06	0.25	0.08	0.15	0.65	0.15	0.04	0.20	0.04	0.06	0.30	0.08	0.15	0.64	0.16	0.04	0.20	0.04
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	37.21	27.91	58.14	53.33	60.00	53.33	34.48	27.59	37.93	37.21	27.91	58.14	53.33	51.11	60.00	34.48	41.38	48.28	37.21	25.58	58.14	53.33	28.89	60.00	34.48	34.48	41.38
NaiveBayes	25.58	30.23	20.93	22.22	28.89	22.22	17.24	37.93	24.14	25.58	30.23	20.93	22.22	17.78	17.78	17.24	31.03	20.69	25.58	30.23	20.93	22.22	20.00	17.78	17.24	24.14	13.79
Logistic	27.91	23.26	30.23	51.11	31.11	40.00	27.59	37.93	31.03	27.91	23.26	30.23	51.11	33.33	46.67	27.59	31.03	24.14	27.91	27.91	30.23	51.11	33.33	46.67	27.59	24.14	24.14
SMO	39.53	27.91	27.91	40.00	28.89	42.22	27.59	41.38	17.24	39.53	27.91	27.91	40.00	26.67	44.44	27.59	17.24	3.45	39.53	23.26	27.91	40.00	17.78	44.44	27.59	13.79	10.34
ZeroR	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48
J48	65.12	41.86	65.12	71.11	57.78	60.00	55.17	37.93	55.17	65.12	41.86	65.12	71.11	57.78	71.11	55.17	44.83	55.17	65.12	46.51	65.12	71.11	44.44	71.11	55.17	34.48	55.17
NaiveBayes	32.56	37.21	32.56	33.33	40.00	40.00	58.62	41.38	58.62	32.56	37.21	32.56	33.33	44.44	33.33	58.62	41.38	58.62	32.56	37.21	32.56	33.33	46.67	33.33	58.62	37.93	58.62
Logistic	34.88	51.16	34.88	66.67	46.67	60.00	62.07	48.28	62.07	34.88	51.16	34.88	66.67	60.00	66.67	62.07	41.38	62.07	34.88	53.49	34.88	66.67	57.78	66.67	62.07	41.38	62.07
SMO	39.53	46.51	39.53	53.33	60.00	42.22	55.17	55.17	55.17	39.53	46.51	39.53	53.33	46.67	53.33	55.17	55.17	55.17	39.53	44.19	39.53	53.33	53.33	53.33	55.17	41.38	55.17

T time, *P* power, *E* energy

Table 3 Kernel prediction accuracies of the built models

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E
ZeroR	0.48	0.46	0.48	0.36	0.46	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.44	0.35	0.65	0.41	0.65
LinReg	0.48	0.46	0.48	0.36	0.32	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.23	0.35	0.65	0.41	0.65
Mult.Perc.	0.01	0.13	0.02	0.63	0.63	0.56	0.07	0.16	0.16	0.01	0.13	0.02	0.63	0.46	0.67	0.07	0.39	0.13	0.01	0.14	0.02	0.63	0.50	0.61	0.07	0.48	0.06
REPTree	0.08	0.10	0.08	0.74	0.27	0.70	0.19	0.18	0.21	0.08	0.10	0.08	0.74	0.00	0.77	0.19	0.07	0.13	0.08	0.47	0.08	0.74	0.21	0.74	0.19	0.19	0.21
M5P	0.01	0.04	0.01	0.67	0.05	0.67	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.02	0.64	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.11	0.67	0.43	0.16	0.44
SMOreg	0.04	0.07	0.05	0.00	0.53	0.00	0.07	0.12	0.07	0.04	0.07	0.05	0.00	0.62	0.01	0.07	0.04	0.06	0.04	0.12	0.05	0.00	0.63	0.01	0.07	0.08	0.06
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	32.56	18.60	44.19	48.89	35.56	46.67	68.97	20.69	68.97	32.56	18.60	44.19	48.89	22.22	48.89	68.97	17.24	68.97	32.56	34.88	37.21	48.89	48.89	48.89	68.97	48.28	68.97
NaiveBayes	18.60	20.93	13.95	26.67	31.11	22.22	34.48	24.14	34.48	18.60	20.93	13.95	26.67	20.00	33.33	34.48	17.24	34.48	18.60	13.95	6.98	26.67	28.89	33.33	34.48	27.59	34.48
Logistic	41.86	25.58	44.19	33.33	37.78	40.00	41.38	27.59	41.38	41.86	25.58	44.19	33.33	33.33	35.56	41.38	24.14	41.38	41.86	25.58	25.58	33.33	31.11	35.56	41.38	27.59	41.38
SMO	32.56	20.93	20.93	33.33	28.89	40.00	34.48	20.69	34.48	32.56	20.93	20.93	33.33	20.00	42.22	34.48	17.24	34.48	32.56	23.26	20.93	33.33	28.89	42.22	34.48	10.34	34.48
ZeroR	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48
J48	55.81	32.56	55.81	53.33	62.22	53.33	55.17	48.28	55.17	55.81	32.56	55.81	53.33	46.67	48.89	55.17	41.38	55.17	55.81	32.56	55.81	53.33	35.56	48.89	55.17	37.93	55.17
NaiveBayes	37.21	39.53	37.21	44.44	44.44	57.78	34.48	31.03	34.48	37.21	39.53	37.21	44.44	40.00	48.89	34.48	17.24	34.48	37.21	39.53	37.21	44.44	42.22	48.89	34.48	17.24	34.48
Logistic	65.12	37.21	65.12	48.89	55.56	53.33	68.97	37.93	68.97	65.12	37.21	65.12	48.89	57.78	53.33	68.97	27.59	68.97	65.12	37.21	65.12	48.89	53.33	53.33	68.97	34.48	68.97
SMO	53.49	46.51	53.49	48.89	55.56	51.11	44.83	41.38	44.83	53.49	46.51	53.49	48.89	40.00	55.56	44.83	17.24	44.83	53.49	46.51	53.49	48.89	44.44	55.56	44.83	27.59	44.83

T time, *P* power, *E* energy

Table 4 Initialization/cleanup prediction accuracies of the built models

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E
ZeroR	0.43	0.41	0.43	0.37	0.46	0.37	0.56	0.42	0.56	0.43	0.41	0.43	0.37	0.35	0.37	0.56	0.51	0.56	0.43	0.41	0.43	0.37	0.34	0.37	0.56	0.53	0.56
LinReg	0.43	0.41	0.43	0.14	0.32	0.07	0.56	0.38	0.56	0.43	0.41	0.43	0.14	0.26	0.13	0.56	0.22	0.56	0.43	0.41	0.43	0.14	0.18	0.13	0.56	0.53	0.56
Mult.Perc.	0.07	0.11	0.06	0.03	0.11	0.02	0.06	0.76	0.07	0.07	0.11	0.06	0.03	0.07	0.03	0.06	0.62	0.06	0.07	0.16	0.07	0.03	0.05	0.02	0.06	0.74	0.06
REPTree	0.42	0.00	0.42	0.07	0.08	0.07	0.56	0.57	0.56	0.42	0.00	0.42	0.07	0.17	0.07	0.56	0.49	0.56	0.42	0.18	0.42	0.07	0.17	0.07	0.56	0.81	0.56
M5P	0.35	0.16	0.35	0.03	0.21	0.01	0.70	0.61	0.70	0.35	0.16	0.35	0.03	0.11	0.04	0.70	0.74	0.70	0.35	0.10	0.35	0.03	0.12	0.04	0.70	0.69	0.69
SMOreg	0.01	0.30	0.01	0.04	0.32	0.10	0.08	0.77	0.08	0.01	0.30	0.01	0.04	0.06	0.09	0.08	0.70	0.08	0.01	0.32	0.01	0.04	0.06	0.09	0.08	0.77	0.08
ZeroR	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	9.38	3.13	19.23	19.23	19.23	18.52	18.52	18.52	6.25	9.38	3.13
J48	32.69	28.85	34.62	38.89	44.44	38.89	37.50	28.13	37.50	32.69	28.85	34.62	38.89	20.37	44.44	37.50	9.38	40.63	32.69	34.62	32.69	38.89	22.22	44.44	37.50	25.00	40.63
NaiveBayes	28.85	23.08	25.00	25.93	18.52	16.67	34.38	25.00	34.38	28.85	23.08	25.00	25.93	18.52	20.37	34.38	9.38	28.13	28.85	11.54	23.08	25.93	16.67	24.07	34.38	9.38	28.13
Logistic	42.31	21.15	46.15	35.19	42.59	37.04	50.00	31.25	50.00	42.31	21.15	46.15	35.19	24.07	25.93	50.00	25.00	46.88	42.31	26.92	36.54	35.19	27.78	31.48	50.00	25.00	46.88
SMO	42.31	13.46	50.00	22.22	29.63	20.37	31.25	9.38	31.25	42.31	13.46	50.00	22.22	20.37	22.22	31.25	6.25	50.00	42.31	21.15	57.69	22.22	14.81	20.37	31.25	12.50	50.00
ZeroR	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25
J48	53.85	50.00	59.62	53.70	44.44	62.96	81.25	62.50	81.25	53.85	50.00	59.62	53.70	37.04	53.70	81.25	37.50	81.25	53.85	38.46	59.62	53.70	44.44	57.41	81.25	43.75	81.25
NaiveBayes	46.15	40.38	44.23	27.78	27.78	25.93	59.38	50.00	59.38	46.15	40.38	44.23	27.78	42.59	42.59	59.38	50.00	59.38	46.15	38.46	44.23	27.78	53.70	37.04	59.38	50.00	59.38
Logistic	50.00	53.85	53.85	51.85	61.11	61.11	65.63	65.63	65.63	50.00	53.85	53.85	51.85	53.70	53.70	65.63	50.00	65.63	50.00	48.08	53.85	51.85	57.41	55.56	65.63	46.88	65.63
SMO	63.46	46.15	65.38	44.44	57.41	53.70	65.63	56.25	65.63	63.46	46.15	65.38	44.44	46.30	46.30	65.63	43.75	65.63	63.46	40.38	65.38	44.44	48.15	46.30	65.63	43.75	65.63

T time, *P* power, *E* energy

Table 5 Data transfer prediction accuracies of the built models

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E	T	P	E
ZeroR	0.44	0.62	0.44	0.41	0.36	0.38	0.53	0.50	0.53	0.44	0.62	0.44	0.41	0.38	0.40	0.53	0.53	0.54	0.44	0.60	0.44	0.41	0.38	0.40	0.53	0.52	0.53
LinReg	0.02	0.16	0.04	0.05	0.57	0.24	0.53	0.50	0.53	0.02	0.16	0.04	0.05	0.78	0.12	0.53	0.53	0.54	0.02	0.46	0.00	0.05	0.81	0.11	0.53	0.52	0.53
Mult.Perc.	0.05	0.19	0.10	0.02	0.66	0.38	0.22	0.09	0.16	0.05	0.19	0.10	0.02	0.75	0.12	0.22	0.12	0.44	0.05	0.15	0.07	0.02	0.77	0.11	0.22	0.11	0.17
REPTree	0.15	0.61	0.16	0.08	0.56	0.32	0.53	0.46	0.53	0.15	0.61	0.16	0.08	0.20	0.15	0.53	0.53	0.27	0.15	0.63	0.16	0.08	0.15	0.14	0.53	0.52	0.18
M5P	0.19	0.43	0.18	0.14	0.47	0.36	0.04	0.02	0.05	0.19	0.43	0.18	0.14	0.80	0.19	0.04	0.05	0.03	0.19	0.51	0.19	0.14	0.82	0.19	0.04	0.04	0.01
SMOreg	0.31	0.23	0.33	0.17	0.46	0.41	0.02	0.19	0.02	0.31	0.23	0.33	0.17	0.77	0.23	0.02	0.08	0.03	0.31	0.34	0.33	0.17	0.78	0.22	0.02	0.10	0.03
ZeroR	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	12.50	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25
J48	48.08	44.23	48.08	38.89	61.11	50.00	37.50	40.63	37.50	48.08	44.23	48.08	38.89	44.44	44.44	37.50	12.50	40.63	48.08	40.38	48.08	38.89	42.59	44.44	37.50	12.50	40.63
NaiveBayes	13.46	26.92	13.46	7.41	25.93	22.22	28.13	46.88	28.13	13.46	26.92	13.46	7.41	27.78	16.67	28.13	15.63	25.00	13.46	28.85	13.46	7.41	35.19	16.67	28.13	12.50	25.00
Logistic	28.85	30.77	28.85	40.74	38.89	37.04	31.25	37.50	31.25	28.85	30.77	28.85	40.74	44.44	37.04	31.25	18.75	31.25	28.85	32.69	28.85	40.74	40.74	37.04	31.25	25.00	31.25
SMO	28.85	28.85	28.85	42.59	40.74	38.89	18.75	46.88	18.75	28.85	28.85	28.85	42.59	48.15	44.44	18.75	18.75	25.00	28.85	28.85	28.85	42.59	50.00	44.44	18.75	18.75	25.00
ZeroR	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25
J48	51.92	42.31	51.92	61.11	72.22	55.56	50.00	34.38	50.00	51.92	42.31	51.92	61.11	50.00	55.56	50.00	34.38	43.75	51.92	44.23	51.92	61.11	50.00	61.11	50.00	34.38	43.75
NaiveBayes	26.92	38.46	26.92	37.04	44.44	42.59	31.25	43.75	31.25	26.92	38.46	26.92	37.04	40.74	42.59	31.25	31.25	34.38	26.92	38.46	26.92	37.04	40.74	37.04	31.25	31.25	34.38
Logistic	53.85	36.54	53.85	48.15	59.26	42.59	43.75	40.63	43.75	53.85	36.54	53.85	48.15	51.85	42.59	43.75	46.88	53.13	53.85	30.77	53.85	48.15	51.85	48.15	43.75	46.88	53.13
SMO	55.77	30.77	55.77	61.11	62.96	66.67	46.88	40.63	46.88	55.77	30.77	55.77	61.11	61.11	66.67	46.88	40.63	50.00	55.77	34.62	55.77	61.11	61.11	61.11	46.88	40.63	50.00

T time, *P* power, *E* energy

groups start with Weka's ZeroR algorithm that can be considered a baseline for the given problem, i.e., algorithms that outperform this accuracy are said to have predictive power in this context. For easier visual parsing, the cells of the tables are colored with five different shades to signal higher precision.

Note that regression cells represent the absolute values of correlation coefficients of the cross-validation, while the classification values are percentages of the correctly classified instances. (We use absolute values because in this case we are interested in the strength of the correlations, not their direction.) Also note that random choice on a 5 or 3 bin classification would yield 20 or 33.33% accuracy, respectively (which the vast majority of classifiers still outperform), but the baseline can be (and is) worse than random choice as there the model always picks the most represented class in the *training* data, which guarantees nothing in the *test* data. For example, if a data set with 7 blacks and 3 whites as its classes were separated into training and test data sets where each training instance is black and each test instance is white, ZeroR would always predict black based on the training data and it would be 0% accurate on the test set. Additionally, although cross-validation repeats this training-test separation n times, the average of the results could still be lower than random choice depending on the separations and the starting distribution of the classes.

Globally, 886 of our 1404 models produced meaningful (i.e., at least 5%) improvement over the baseline performance, and 867 of these used at least 2 predictor metrics. (This second check was implemented to root out a few models encountered during random manual validation that were simple constants or relied only on InputSize.) Additionally, we collected statistics for the most frequently used metrics in the models. The top 10 start with the all important InputSize—used in 98% of the models—, followed by ARR%, LOC, ft25 (average number of instructions in basic blocks), ft48 (number of occurrences of integer constant one), ft7 (number of basic blocks with more than two predecessors), EXP, ARR, LNL1, and MUL, respectively.

To gain further insight into the effect the different dimensions (i.e., phase, aspect, etc.) have on prediction accuracy, we also computed model success distributions for each dimension separately. Note that, in order to make this discussion more concise, $x/y/z$ will mean that "out of all possible z models, y managed to outperform the baseline by at least 5%, x of which used at least 2 predictors from the available set." These could be thought of as "better/good/count."

- Source code phase
  - Initialization/Cleanup: 212/220/351
  - Kernel execution: 224/224/351
  - Data transfer: 206/206/351
  - Full: 225/236/351
- Measurement aggregation method
  - Single: 295/301/468
  - With CPU: 286/292/468
  - All: 286/293/468
- Execution platform
  - CPU: 269/269/468
  - GPU: 325/336/468

  - FPGA: 273/281/468
- Measurement aspect
  - Time: 282/291/468
  - Power: 302/304/468
  - Energy: 283/291/468
- Machine learning technique
  - Regression: 58/77/540
  - 3 bin classification: 405/405/432
  - 5 bin classification: 404/404/432

These figures show that every dimension has at least a limited effect on the models. Considering source code phase, kernel execution and the full program are a little easier to estimate than either initialization/cleanup or data transfer. This is to be expected, though, since "kernel" and "full" are the two phases containing the kernels the predictor metrics are based on. Regarding measurement aggregation, concentrating on a single execution platform is proven simpler than accounting for other parts of the hardware system as well. A similar slight edge can be observed for the power aspect, compared to both time and energy, while the GPU platform has an even more pronounced advantage over both CPUs and FPGAs. The most important difference, however, is evident along the machine learning technique dimension, specifically that barely 10% of the regression models managed to outperform the baseline, while this ratio is over 90% for both classification types. This suggests, not surprisingly, that an exact improvement ratio is much harder to estimate than an interval it will fall in.

Regression models frequently resorted to using only a constant value or a function of a single input metric, which is a clear sign of undertraining, but after disregarding these, we still had a few promising cases. However, these belonged almost exclusively to GPUs. E.g., the highest precision among the "full" regressions—which is also the highest value increase compared to its ZeroR counterpart—is the REPTree model for Single-GPU-Energy estimation. It reaches an absolute 0.83 correlation coefficient, representing a 0.41 improvement. The most precise "kernel" regression is also a REPTree—this time for WithCPU-GPU-Energy—with a value of 0.76, representing another 0.41 improvement. The pattern of these two tables suggests that REPTree and M5P are more appropriate for time and energy prediction, while Multilayer Perceptron and SMOreg are more successful for average power. This is no longer true for the "initialization/cleanup" phase, where only FPGAs have notable models. M5P seems the most capable for all three aspects, but the best models are the All-FPGA-Power REPTree with a 0.81 precision and Single-FPGA-Power SMOreg with a 0.35 increase. As for the "data transfer" models, only the GPU-Power columns stand out. The best case scenario here is the All-GPU-Power M5P model with an accuracy of 0.82, which is a 0.44 improvement.

Regarding classification models, we no longer see the superiority of GPU prediction. The most easily discernible global observation is that the overwhelming majority is a significant upgrade compared to either the ZeroR reference or a random choice. We can also notice that while regressions were more prone to "column patterns"—i.e., the measurement aggregation method, the platform, or the aspect mattered more in the columns than the algorithms in the rows, leading to higher concentrations of precise

models above or below each other—classifications lean toward "row patterns"—i.e., once the source code phase is chosen, higher accuracy correlates more with the algorithm. For the sake of brevity in further model discussion, (vs. $x\%/y\%$) will mean "compared to a ZeroR of $x\%$ and a random choice of $y\%$."

"Full" classifications are lead by J48 models, which display up to 60% accuracy on 5 bins (vs. 11.11%/20%), once for GPU power (Single) and twice for GPU energy (With CPU and All). For 3 bins, this value is up to 71.11% (vs. 22.22%/33.33%), but here we note that Logistic regression is a close second. The best "kernel" models come from these two algorithms again; J48 on 5 bins is at times 68.97% (vs. 0%/20%) for FPGA time and energy, while Logistic regression reaches the same 68.97% on 3 bins (vs. 34.48%/33.33%), at the same places. For the "initialization/cleanup" phase, the best choices are SMO on 5 bins for All-CPU-Energy with 57.69% (vs. 19.23%/20%), and J48 on 3 bins for FPGA time and energy modeling with 81.25% (vs. 31.25%/33.33%). Finally, the most accurate "data transfer" classifications are J48 trees, both times for Single-GPU-Power prediction: 61.11% on 5 bins (vs. 18.52%/20%) and 72.22% on 3 bins (vs. 25.93%/33.33%).

In conclusion, by predicting the improvement category significantly more accurately than either a baseline performance or a random choice, our classification algorithms clearly demonstrated that static metrics have predictive power and skill in this domain. Therefore, we can answer our first research question in the affirmative.

Although these findings can hardly be considered widely generalizable due to the small number of training instances, the main result of this study is the streamlined process by which they were produced. With the described infrastructure in place, making the model more precise is largely just a matter of integrating more benchmark source code into the analysis. We would also like to emphasize the fact that every benchmark [9], calculated metric, measurement, machine learning result [8] and even the measurement library [19] are opened to the public so we invite replication or further expansion.

7.4 Maintainability changes

As far as extending the available benchmark set, automatic kernel transformators or other parallelized source code generators would greatly help. Is it worth developing such algorithms, however, or should we simply manually maintain a dedicated parallel implementation? To try and explore this question from the source code side, we compared the calculated abstract characteristics of the sequential and parallel versions of our benchmarks. The changes in intermediate values (analysability, modifiability, reusability, testability, and modularity) and in the final Maintainability score of the whole system and of the separated kernel regions are shown in Tables 6 and 7.

According to the data in Table 6, we can answer our second research question: Maintainability experiences a distinct negative change as a result of parallelization. When we look at Table 7, however, we see a much less pronounced negative effect, which, at times, even turns positive. Similarly to the conclusions of the original study [14], we speculate that this is because, even though such a transformation can deteriorate the maintainability of the kernels themselves, its most powerful effect is the boilerplate

Table 6 Maintainability changes on the system level

	Analy.	Modif.	Reusa.	Test.	Modul.	Maint.
mri-q	− 0.388	− 0.405	− 0.432	− 0.360	− 0.448	− 0.407
spmv	− 0.667	− 0.685	− 0.676	− 0.658	− 0.653	− 0.668
stencil	− 0.225	− 0.237	− 0.325	− 0.199	− 0.428	− 0.283
atax	− 0.338	− 0.354	− 0.406	− 0.298	− 0.472	− 0.375
bicg	− 0.342	− 0.358	− 0.412	− 0.308	− 0.471	− 0.379
conv2d	− 0.332	− 0.346	− 0.405	− 0.296	− 0.469	− 0.370
doitgen	− 0.372	− 0.388	− 0.476	− 0.324	− 0.582	− 0.429
gemm	− 0.269	− 0.283	− 0.352	− 0.237	− 0.435	− 0.315
gemver	− 0.325	− 0.343	− 0.417	− 0.294	− 0.494	− 0.375
gesummv	− 0.290	− 0.304	− 0.343	− 0.262	− 0.384	− 0.317
jacobi2d	− 0.420	− 0.433	− 0.491	− 0.373	− 0.560	− 0.456
mvt	− 0.339	− 0.353	− 0.396	− 0.304	− 0.444	− 0.368
bfs	− 0.352	− 0.367	− 0.431	− 0.319	− 0.497	− 0.393
hotspot	− 0.226	− 0.235	− 0.308	− 0.167	− 0.371	− 0.261
lavaMD	− 0.271	− 0.276	− 0.315	− 0.244	− 0.352	− 0.292
nn	− 0.429	− 0.434	− 0.485	− 0.364	− 0.560	− 0.456

Table 7 Maintainability changes on the kernel level

	Analy.	Modif.	Reusa.	Test.	Modul.	Maint.
mri-q	− 0.234	− 0.240	− 0.321	− 0.225	− 0.395	− 0.282
spmv	0.139	0.135	− 0.069	0.188	− 0.308	0.019
stencil	0.145	0.144	− 0.205	0.220	− 0.617	− 0.059
atax	− 0.109	− 0.136	− 0.283	− 0.087	− 0.435	− 0.208
bicg	− 0.200	− 0.222	− 0.329	− 0.162	− 0.449	− 0.272
conv2d	− 0.065	− 0.075	− 0.228	− 0.002	− 0.431	− 0.161
doitgen	0.147	0.131	− 0.228	0.226	− 0.653	− 0.072
gemm	0.120	0.110	− 0.123	0.175	− 0.391	− 0.019
gemver	− 0.161	− 0.187	− 0.429	− 0.119	− 0.708	− 0.319
gesummv	− 0.041	− 0.055	− 0.199	− 0.033	− 0.341	− 0.131
jacobi2d	− 0.035	− 0.057	− 0.347	0.019	− 0.691	− 0.220
mvt	− 0.148	− 0.174	− 0.302	− 0.115	− 0.443	− 0.236
bfs	0.067	0.063	− 0.150	0.095	− 0.408	− 0.064
hotspot	− 0.035	− 0.041	− 0.390	0.043	− 0.774	− 0.235
lavaMD	− 0.158	− 0.165	− 0.303	− 0.150	− 0.434	− 0.239
nn	0.015	0.017	0.006	0.013	0.007	0.012

and added necessary infrastructure it brings to the system as a whole. This can be considered another point in favor of automatic parallel transformations as that way developers could work on a more maintainable version of the source code while still being able to reap the benefits of modern accelerators and parallel platforms.

8 Conclusions and future work

The goal of this paper was to present our work addressing the creation of prediction models that are able to automatically determine not only the optimal execution configuration of a program (i.e., sequential or OpenCL, CPU, GPU or FPGA) but how much improvement we can expect that way. For this, we developed a highly generalizable and reusable methodology for producing such models. Moreover, these models do not depend on dynamic behavior information so they can be easily applied to classifying new subject systems.

Building these models required a set of algorithms that were each implemented on every relevant target platform. After thorough research, we found three independent benchmark suites containing multiple systems that fulfilled this criterion. To be able to build the necessary models, we also needed to measure the time, power, and energy consumption of the algorithms on different configurations. For this, we used our own open-source RMeasure library and universal hardware extensions to measure the power and energy consumption of the hardware components. We then successfully applied our methodology on these systems to create prediction models based on different machine learning approaches, using source code metrics as predictors. The resulting models are quantitative which means that they can predict the optimal execution configuration and also the ratio of how much better it is compared to the other alternatives.

Nevertheless, there are opportunities for improving the model building process in the future. One of these is increasing the number of instances on which the models are based. Another factor can be adding even more predictor metrics. We will try to derive even more potentially representative characteristics by manual inspection of typical properties of the kernels and refine the learning methods by fine tuning and validating their parameters.

We also conducted a replication of the maintainability study by Ferenc et al. [14] on the sequential and parallel versions of the kernels and concluded that the maintainability of parallelized implementations is significantly lower. However, this does not necessarily show—or at least not as strictly—in the kernels themselves, suggesting that the introduced boilerplate is to blame.

Overall, we consider the results of this paper encouraging. Despite the small number of subject systems, we were able to demonstrate that statically computed source code metrics are appropriate and useful for configuration selection. The models are promising by themselves, but we feel that the main result of this paper is the methodology behind their creation. We now have a flexible, expandable and configurable infrastructure in place and the generalizability of its output models depend only on the number of initial benchmark systems we use for training. Additionally, our modified benchmarks

are also capable of maintainability assessment, the results of which are a step toward justifying and motivating the development of automatic kernel transformations.

Acknowledgements The authors would like to thank Péter Molnár and Róbert Sipka for their extensive help with dynamic measurements. This work was supported by the European Union FP7 Project "REPARA—Reengineering and Enabling Performance And poweR of Applications" (Project No. 609666), and by the EU-funded Hungarian national Grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things."

References

1. (2014) NVIDIA Management Library (NVML)—Reference Manual. NVIDIA Corporation, TRM-06719-001 _vR331
2. (2014) PicoScope 4000 Series (A API)—Programmers Guide. Pico Technology Ltd., ps4000apg.en r1
3. (2015) AMD GPU Performance API—User Guide. Advanced Micro Devices, Inc., v2.15
4. (2015) ARM DS-5 Version 5.21—Streamline User Guide. ARM, ARM DUI0482S
5. (2015) Intel 64 and IA-32 Architectures Software Developer's Manual: vol 3B. Intel Corporation, Order Number 253669
6. Arlot S, Celisse A (2010) A survey of cross-validation procedures for model selection. Stat Surv 4:40–79
7. Bán D, Ferenc R, Siket I, Kiss Á (2015) Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. In: Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2015). IEEE, pp 178–183
8. Bán D, Ferenc R, Siket I, Kiss Á, Gyimóthy T (2017) Performance, power, and energy prediction models. http://www.inf.u-szeged.hu/~ferenc/papers/PerformancePowerEnergyModels/
9. Bán D, Sipka R, Dobi I (2017) Tagged parallel benchmarks. https://github.com/sed-inf-u-szeged/TaggedParallelBenchmarks
10. Brandolese C, Fornaciari W, Salice F, Sciuto D (2001) Source-level execution time estimation of C programs. In: Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES). ACM, New York, NY, USA, pp 98–103
11. Brown KJ, Sujeeth AK, Lee HJ, Rompf T, Chafi H, Odersky M, Olukotun K (2011) A heterogeneous parallel framework for domain-specific languages. In: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, pp 89–100
12. Che S, Boyer M, Meng J, Tarjan D, Sheaffer J, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. IEEE International Symposium on Workload Characterization (IISWC). IEEE Computer Society, Washington, DC, USA, pp 44–54
13. Ferenc R et al (2014) Static analysis techniques for AIR generation. Deliverable D2.2, REPARA
14. Ferenc R et al (2015) Maintainability models of heterogeneous programming models. Deliverable D7.4, REPARA
15. Fursin G, Kashnikov Y, Memon AW, Chamski Z, Temam O, Namolaru M, Yom-Tov E, Mendelson B, Zaks A, Courtois E, Bodin F, Barnard P, Ashton E, Bonilla E, Thomson J, Williams CKI, O'Boyle M (2011) Milepost GCC: machine learning enabled self-tuning compiler. Int J Parallel Program 39:296–327
16. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: Innovative Parallel Computing (InPar). IEEE, pp 1–10
17. Grewe D, O'Boyle MFP (2011) A static task partitioning approach for heterogeneous systems using OpenCL. In: Proceedings of the 20th International Conference Compiler Construction (CC). Springer, Berlin, Heidelberg, pp 286–305
18. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. In: SIGKDD Explorations, ACM, vol 11, pp 10–18
19. Kiss Á, Molnár P, Sipka R (2017) RMeasure performance and energy monitoring library. https://github.com/sed-inf-u-szeged/RMeasure
20. Kuperberg M, Krogmann K, Reussner R (2008) Performance prediction for black-box components using reengineered parametric behaviour models. In: Proceedings of the 11th International Symposium on Component-Based Software Engineering. Springer, pp 48–63

21. Li D, de Supinski B, Schulz M, Cameron K, Nikolopoulos D (2010) Hybrid MPI/OpenMP power-aware computing. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS). IEEE, pp 1–12
22. Ma X, Dong M, Zhong L, Deng Z (2009) Statistical power consumption analysis and modeling for GPU-based computing. In: In Proceedings of SOSP Workshop on Power-aware Computing and Systems (HotPower)'09
23. Marin G, Mellor-Crummey J (2004) Cross-architecture performance predictions for scientific applications using parameterized models. In: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems. ACM, pp 2–13
24. Osmulski T, Muehring JT, Veale B, West JM, Li H, Vanichayobon S, Ko SH, Antonio JK, Dhall SK (2000) A probabilistic power prediction tool for the Xilinx 4000-series FPGA. In: Proceedings of the IPDPS 2000 Workshops on Parallel and Distributed Processing. Springer, pp 776–783
25. Pflüger D, Pfander D (2016) Computational efficiency vs. maintainability and portability. Experiences with the sparse grid code sg++. In: Proceedings of the Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE). IEEE, pp 17–25
26. Pouchet LN (2011) Polybench: the polyhedral benchmark suite. http://www-roc.inria.fr/~pouchet/software/polybench
27. Sánchez LM et al (2014) Target platform description specification. Deliverable D3.1, REPARA
28. Shen J, Fang J, Sips H, Varbanescu A (2012) Performance gaps between OpenMP and OpenCL for multi-core CPUs. 41st International Conference on Parallel Processing Workshops (ICPPW). IEEE Computer Society, Washington, DC, USA, pp 116–125
29. Stratton JA, Rodrigues C, Sung IJ, Obeid N, Chang LW, Anssari N, Liu GD, Mei W, Hwu W (2012) Parboil: a revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign
30. Takizawa H, Sato K, Kobayashi H (2008) SPRAT: Runtime processor selection for energy-aware computing. In: IEEE International Conference on Cluster Computing. IEEE, pp 386–393
31. Van Der Vaart A (1998) Asymptotic statistics, Cambridge series in statistical and probabilistic mathematics, vol 3. Cambridge University Press, Cambridge
32. Yang L, Ma X, Mueller F (2005) Cross-platform performance prediction of parallel applications using partial execution. In: Proceedings of the ACM/IEEE SC 2005 Conference on Supercomputing. IEEE Computer Society, Washington, DC, USA, p 40