Decision Support for GPU Acceleration by Predicting Energy Savings and Programming Effort

Charalampos Marantos[a,*], Lazaros Papadopoulos[a], Angeliki-Agathi Tsintzira[b], Apostolos Ampatzoglou[b], Alexander Chatzigeorgiou[b], Dimitrios Soudris[a]

[a]*School of Electrical and Computer Engineering, National Technical University of Athens, Greece*
[b]*Department of Applied Informatics, University of Macedonia, Greece*

Abstract

As the number of heterogeneous embedded systems used in IoT applications increases, there is a lack of software tools to assist developers to meet the challenge of reducing energy consumption. Indeed, there are only few performance prediction tools for heterogeneous systems in the literature and they typically focus on the prediction of speedup by acceleration. In this work, we propose a methodology for analysing CPU applications in order to estimate the potential Energy gains by offloading a piece of code on an embedded GPU. The proposed methodology provides several features beyond the state of the art of existing predictors, including the combination of static analysis and dynamic instrumentation approaches and the prediction of the programming effort of developing the CUDA kernel of a CPU code, using advanced metrics. The methodology is supported by a tool-flow and it is demonstrated and evaluated on modern heterogeneous embedded systems (Nvidia), where shows classification accuracy above 75%. The results show that the proposed methodology can assist application developers in the early design choice of investing effort to acceleration considering the expected Energy Savings and the Effort required to develop acceleration-specific code.

Keywords: Energy consumption, Embedded systems, GPU acceleration, Software Design

1. Introduction

The continued growth of Internet of Things (IoT) networks and the ever-increasing number of sensors and mobile devices are expected to lead to 41.6 billion connected devices by 2025 [9]. The interconnected devices are predicted to contribute to the transmission of 79.4 zettabytes of data over the Internet, further increasing the demands for more electric power and having negative impact on the CO_2 emissions generated by electricity production [9][21].

Accelerators integrated to edge computing devices target not only to performance improvements, but also energy efficiency [6]. Indeed, the growth of embedded applications enabled by image processing and machine learning algorithms, contributed to the evolution of embedded system architectures towards multiprocessing and heterogeneity. Acceleration units, such as GPUs and FPGAs enable efficient execution of computationally demanding algorithms, thus avoiding data transmission to other layers of IoT networks (e.g. Fog, Cloud), to further improving the energy efficiency of the whole network. Typical examples include Nvidia Tegra, which integrates an embedded GPU [19], Intel/Movidius Myriad [11] and CEVA [4] which integrate DSP cores along with various accelerators, tailored to image processing and neural network applications. Since these platforms are very low power they target markets in which autonomy is important, such as drones, mobile devices and wearables.

More and more tools and services are being introduced in order to properly use these devices from the software level and without the need for specialized hardware knowledge. Typical examples include high level languages like CUDA and OpenCL, that are used by software engineers in order to use the acceleration capabilities of the targeted heterogeneous devices. However, software developers still need advice about how and

*Corresponding author
Email addresses: hmarantos@microlab.ntua.gr (Charalampos Marantos), lpapadop@microlab.ntua.gr (Lazaros Papadopoulos), angeliki.agathi.tsintzira@gmail.com (Angeliki-Agathi Tsintzira), ampatzoglou@uom.edu.gr (Apostolos Ampatzoglou), achat@uom.edu.gr (Alexander Chatzigeorgiou), dsoudris@microlab.ntua.gr (Dimitrios Soudris)

when to use these features, while a significant programming effort is required to effectively exploit the available accelerators and bring the computational load of such complex algorithms within their power envelope. As a result, bringing the design principles of energy efficient development closer to the software engineering perspective is becoming an active research topic [8].

Several tools and frameworks have been proposed to address the challenges of heterogeneity, by assisting application developers in the tedious process of exploiting accelerators. A promising approach enabled by machine learning techniques is the performance prediction: Tools that fall into this category estimate the potentiality of a piece of CPU code to exploit an accelerator. More specifically, the tools predict the performance of a piece of CPU code on an acceleration unit, by analyzing the CPU code only. Thus, they guide application developers in the early design choice of offloading a piece of code to an accelerator without requiring tedious redevelopment effort for testing and/or access to the actual hardware.

Predictors proposed in the literature mainly rely on dynamic instrumentation of application binaries to extract features that feed a machine learning model that provides predictions. Dynamic instrumentation requires the execution of the application and normally induces significant time overhead. One possible solution might be to use static analysis of source code to extract features faster than the dynamic instrumentation, however, the accuracy of the prediction may be much lower or the granularity of the prediction extremely coarse (even binary). The choice of one approach over the other depends on whether tool developers prioritize prediction accuracy over user friendliness and prediction speed, or vice versa.

Porting CPU applications to accelerators usually requires significant programming effort. Often, the program needs to be restructured, the organization of data structures should be rearranged and memory management issues impose significant challenges. This is especially true when developing CUDA code for accelerating parallel parts of applications. However, non of the existing available predictors considers the effort required to develop the accelerated version of a piece of CPU code.

This work describes a methodology supported by a tool-flow that contributes to addressing the limitations of existing prediction approaches related to GPU acceleration. By proposing and combining both static and dynamic analysis techniques, the introduced approach leverages the advantages of both techniques and compromises prediction granularity and time required to generate a prediction. Additionally, instead of providing predictions for speedup by GPU acceleration, it focuses on energy consumption gains, which is an equally important quality, especially in the context of heterogeneous embedded systems [25], however usually neglected by the existing predictors on the application's software level. In particular, this work proposes the use of performance-related features for building energy consumption prediction models. Finally, it provides predictions about the programming effort required to develop CUDA code by analyzing the corresponding CPU code, based on sophisticated metrics. More specifically, the estimation relies on the *effort* indicator based on the Halstead metrics, which more accurately expresses the programming effort than the (typically used) Lines-of-Code (LOC). The proposed approach is expected to significantly assist application developers in decision making with respect to accelerating CPU code and in allocating programming time and effort efficiently.

Summarizing, the proposed methodology:

- Combines multiple approaches for energy consumption prediction (both static and dynamic) and exploits the advantages of both of them by compromising prediction granularity and time overhead to generate a prediction, while relying only on static analysis when feasible. Extensions of previous works are combined with the proposal of new methods into a single tool-flow.
- Provides results in multiple dimensions, by offering predictions in qualities other that execution time: Energy consumption and programming effort, useful for reducing application design time.
- Estimates the programming effort required to develop CUDA code based only on the analysis of the initial corresponding CPU code, by using well-established indicators.

The rest of the paper is organized as follows: Section 2 presents the related work. The proposed methodology is described in Section 3 and it is evaluated in Section 4 in terms of prediction accuracy. Finally, in Section 5 we draw conclusions.

2. Related Work

The most recent performance prediction tools (i.e. after 2015) directly related to the proposed approach are depicted in Table 1. They analyze CPU code to predict potential performance gains by GPU acceleration.

Table 1: Comparison against related recently designed approaches

Approach	XAPP [1]	CGPredict [26]	[2]	Proposed
Analysis Type	Dynamic instrumentation	Dynamic instrumentation	Static Analysis	Combination (advantages of both)
Predicted value	Speed-up	Speed-up	Speed-up	Energy Gain & Progr. Effort
Provide results only based on static analysis	No	No	No, dynamic info also needed	Yes
Training Dataset	Benchmarks	Benchmarks	Benchmarks	Benchmarks & Synthetic
Ability to reproduce the analysis	No	No	No	Dataset and models provided on git repo
Software Engineering Perspective	No	No	No	Effort Prediction

XAPP [1] and CGPredict [26] analyze CPU code based on dynamic instrumentation. However, XAPP leverages machine learning techniques for prediction, while CGPredict is based on analytical models. A first step towards a static analysis approach using random forest classification with two output classes has also been proposed recently to trade accuracy for fast and user friendly predictions [2]. Other recent works that belong to the broad category of performance predictors are the Compass and Automatch tools [14, 10]. Compass includes a static analysis framework, which analyzes C code and generates application performance models, while Automatch detects application characteristics and through the generation of analytical modeling predicts performance of execution in various accelerators.

It is observed that most predictors rely on dynamic instrumentation techniques for extracting features that feed a machine learning model that generates predictions. Although dynamic instrumentation is a well established analysis and profiling technique, which is supported by many widely-used tools (e.g. Valgrind [18], Pin [22]), it suffers from large execution time overhead. Additionally, the predictors that entirely rely on dynamic instrumentation inherit the limitations and the constraints of the instrumentation tools that integrate, such as specific configurations and required source code modifications. Therefore, some first approaches to design predictors that rely on static analysis of source code have recently been proposed [2], requiring some dynamic information, such as the number of loop iterations by the user. They trade prediction accuracy or prediction granularity level for user friendliness and short analysis time overhead. However, the methodology described in this work proposes combination of static with dynamic analysis approaches to a single tool-flow, to exploit the advantages of both approaches, to provide flexibility to application developers and to use static analysis only, when feasible, to avoid large time overhead.

Although the existing approaches provide predictions in terms of execution time, they do not consider energy consumption predictions. Indeed, energy efficiency is an important quality especially in embedded systems domain and a critical design constraint. Therefore, in this work, we extend existing approaches towards the prediction of energy consumption gains by acceleration on heterogeneous embedded devices. This is performed by investigating the potential use of machine learning features already proposed in the literature for building energy consumption (instead of performance) prediction models.

Finally, the existing tools predict only execution time gains without considering the programming effort that is required to achieve the predicted gains. Few attempts to quantify the programming effort of accelerating applications can be found in the literature, which are either based on empirical investigations [16], or rely on relatively simple metrics. A typical example of such a metric, is the Lines-of-Code (LOC) of accelerator-specific code (i.e. CUDA, OpenCL) versus the LOC of the corresponding CPU code [17]. However, using LOC may not be a good indicator of programming effort [24]. In addition, these approaches measure effort of existing code, while it is very important to know the effort required before writing the new code.

The advantages of the present work over the relevant approaches are summarized below:

- The proposed approach combines both static and dynamic analysis approaches and exploits the advantages of both of them into a single tool-flow.
- The proposed static analysis component relies entirely on analysing source code using text analytics techniques. Existing works that leverage static analysis techniques (e.g. [2]) require parameters related to dynamic information (such as the number of loop iterations or the direction of branches).
- Regarding the dynamic estimation component, we extend the existing approaches ([1, 3]) that analyze CPU code to provide speedup predictions, towards estimating the potential energy gains too. This is achieved by studying the correlation of the used features with energy consumption.
- To the best of the authors' knowledge this is a first

approach towards designing a tool that estimates programming effort of developing GPU code using the CPU code as input.

3. Proposed Methodology

This section provides an overview of the proposed methodology for predicting the energy consumption gains and the programming effort by acceleration. Then, it describes each one of the components of the methodology in detail.

3.1. Overview

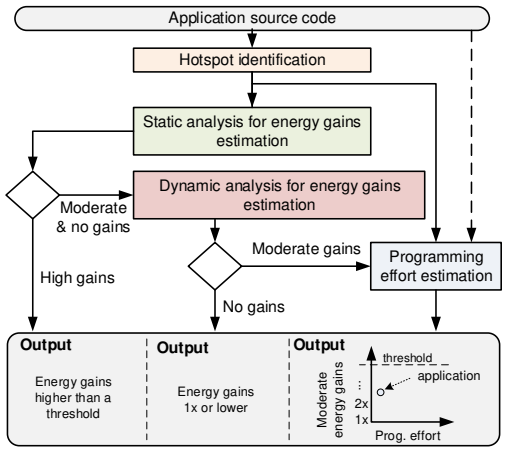


Figure 1: Overview of the proposed methodology

The proposed methodology is depicted in Figure 1. The input of the methodology is the source code of the CPU application and the output is the predicted energy gains by acceleration, as well as the programming effort needed to develop the CUDA version of the CPU application code. The methodology consists of the following steps:

- **Hotspot identification**: The most computationally intensive source code blocks in terms of CPU cycles are identified.

- **Prediction of acceleration gains by static analysis**: It classifies hotspots into the "High gains" or "Moderate/no gains" categories based on the expected energy consumption gains by offloading on the GPU. For hotspots classified into the "High gains", no further analysis is performed.

- **Prediction of acceleration gains by dynamic analysis**: Hotspots classified into the "Moderate/no gains" category by static analysis are further analyzed by leveraging the dynamic instrumentation approach. A classification step identifies hotspots for which no gains by acceleration are expected. Then, a regression step provides a fine-grained prediction of expected energy gains (i.e. prediction of gains by acceleration, as a percentage of energy consumed on the CPU) for the remaining hotspots.

- **Programming effort prediction**: Hotspots classified into the moderate gains category by dynamic analysis are analyzed to predict the programming effort needed to develop the acceleration-specific code based on the corresponding CPU code.

The output of the methodology is shown in Figure 1. A CPU application under analysis is classified into one of the three categories: "No gains", "Moderate" or "High gains" with respect to the predicted energy consumption gains by GPU acceleration. Fine-grained energy consumption prediction, along with programming effort prediction is provided for the applications classified into the "Moderate gains" category.

The methodology predicts the programming effort required to develop CUDA code, only for the applications classified into the "Moderate gains" category. We make the assumption that for applications classified into "High gains", the CUDA version will be developed no matter how much programming effort is required. However, for applications classified into the "Moderate gains", programming effort may affect developers' decision about developing CUDA or not, especially if the predicted energy gains by the regression step are relatively low.

3.2. Hotspot identification

In the context of the proposed methodology, hotspot is defined as block of CPU source code, in which significant number of CPU cycles are spent, compared to the application's total. Each identified hotspot is considered a candidate to be offloaded on the accelerator. Therefore, each hotspot should be analyzed to predict energy gains by acceleration and the programming effort to develop the accelerator-specific code.

The hotspot identification flow is shown in Figure 2. By generating the Abstract Syntax tree (AST) of the application using CLANG, *for* and *while* code blocks are identified in the application source code. Then, the application is dynamically analysed to monitor CPU cy-

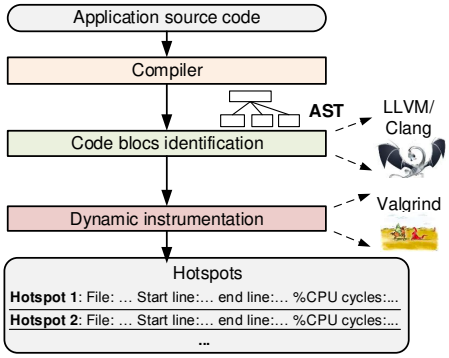


Figure 2: Hotspot identification flow

cles, by leveraging a widely used dynamic binary instrumentation profiler, such as *Callgrind* by the Valgrind suite. By combining the information generated by the dynamic analysis (i.e. *Callgrind* output) and the static analysis (i.e. statements identified by the AST processing), the number of CPU cycles spent in each statement is calculated. The code blocks defined by *for* and *while* statements in which the number of CPU cycles spent is above a threshold are considered *hotspots*. The threshold can be configured by developers.

Although this approach is independent of the physical platform that the application is executed, the execution time overhead due to binary instrumentation may be a limitation for large and complex applications. An alternative approach is the execution of the application on the actual embedded platform and the monitoring of the values of performance counters such as CPU cycles, if available. Tools such as *Linux perf* can be used to automate the monitoring (e.g. *perf* annotation mode). The overhead of this approach is negligible, however it is limited by the availability of performance counters on the physical platform.

Finally, developers familiar with the application under analysis, are often aware of the code blocks that are candidates for acceleration. Therefore, they may skip the hotspot identification step entirely and proceed directly to the static analysis step of the proposed methodology.

3.3. Prediction of energy gains by acceleration

This subsection describes the flow for the prediction of energy gains by offloading a hotspot to a GPU accelerator. The flow is depicted in Figure 3. The hotspots identified by the previous step of the methodology are further analyzed for monitoring the values of features, which are inputs to the prediction models for energy consumption estimation.

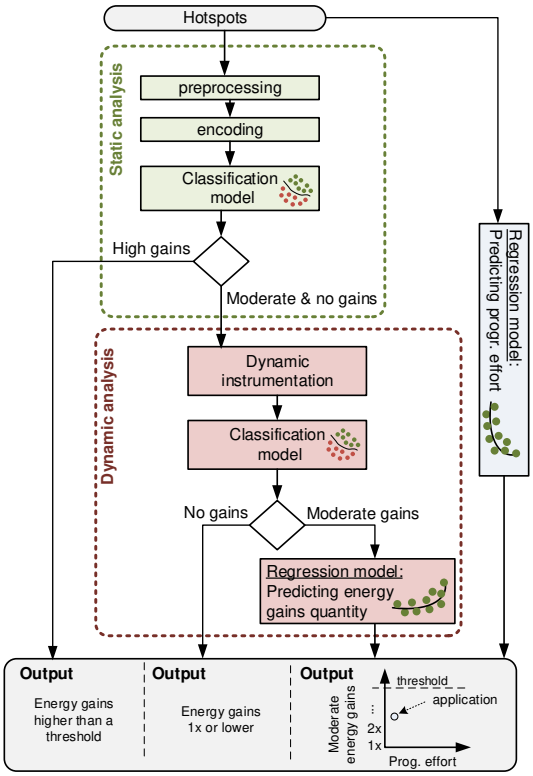


Figure 3: Estimation of energy gains by acceleration

The proposed tool-flow is based on the combination of static source code analysis and on dynamic instrumentation techniques to exploit the advantages of each approach. We first analyze the process we followed for building the dataset we used to train the estimation models of the static and dynamic approaches. Then, we describe the details of each approach.

Initially, the hotspots are statically analyzed (Section 3.3.2) using text analytics methods, which provide a coarse grained estimation of potential energy gains. The hotspots are classified into one of the following two categories: ”High-gains” or ”Moderate/no gains”. The hotspots for which ”Moderate/no gains” are predicted are further analyzed using analysis based on dynamic instrumentation approach for fine grained predictions, as described in Section 3.3.3.

3.3.1. Building the dataset

A major challenge when designing the introduced estimation models is the building of a high quality dataset. This is acknowledged by several related works [1, 26]. The main difficulty relies on the fact that the dataset must contain CPU source code, as well as the corresponding GPU kernel (i.e. the accelerated version of

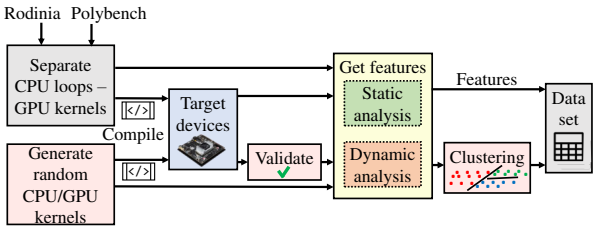


Figure 4: Dataset building

the specific CPU code). Apparently, there is a relatively small number of available benchmark suites that provide both CPU and the corresponding accelerated code. Therefore, in this work, we built a dataset that combines an equal number of synthetic benchmarks and real-world applications (i.e. non-synthetic and taken from existing benchmark suites) to reduce the danger over-fitting caused by a small dataset size that could lead to biased results. Figure 4 shows the steps we followed to build the dataset, which are detailed in the next paragraphs.

With respect to the synthetic part of the dataset, we developed a script that generates *for* loops that perform matrix and vector operations. The number of the matrices, the data types and the operations were randomly generated. CUDA kernels were generated including the same loop body as the CPU loops. Finally, to keep only valid datapoints in the dataset, after the execution of both the CPU and the GPU version, a test phase that performs cross-checking to all data structure values after executing the GPU kernel and the CPU loop was implemented. This validation mechanism checks if the matrix values are equal, meaning that no errors were occurred due to data conflicts in the parallel execution because of the inherent random characteristics of the code.

The real-world (i.e. non-synthetic) application datapoints of the dataset include kernels from the Polybench [20] and Rodinia benchmark suites [5]. These suites include a set of CPU applications, as well as the corresponding GPU version of each application. Following a similar approach for dataset building with other predictors [1], we increased the size of the dataset by modifying the input of some kernels (or the amount of data processed), resulting in a total number of 100 data-points (for the real-world part of the dataset). It is worth mentioning that, since the dataset includes same kernels with different inputs or a single application may include very similar kernels, we made sure that datapoints that belong to the same application will not be in training and test sets consequently.

The synthetic part of the dataset is further refined by eliminating very similar data-points that may cause overfitting. This process is performed through a k-means clustering pre-processing of the data-points and the selection of one data-point from each cluster. The number of generated and selected data-points is configured to be equal to the real-world data-points (i.e. 100), resulting in a training dataset of 200 data points in total. The whole dataset, which as stated earlier consists of 50% of synthetic benchmarks and 50% of real-world applications is publicly available [1].

Obviously, the quality of the CUDA code is expected to affect the quality of the dataset and, subsequently, the accuracy of the predictions. This is true for the real-world part of the dataset, since the CUDA synthetic part are relatively simple. Rodinia and Polybench are two benchmark suites that are widely used in many research works, both in the embedded and HPC domains and they are constantly maintained and improved. Therefore, it is reasonable to assume that the code quality of the CUDA code is relatively high, or at least, that it is close the code quality that experienced developers generate. Although the study of the impact of the quality of CUDA code on the accuracy of predictions is an interesting issue, it is beyond the scope of this work.

3.3.2. *Proposed Static analysis approach*

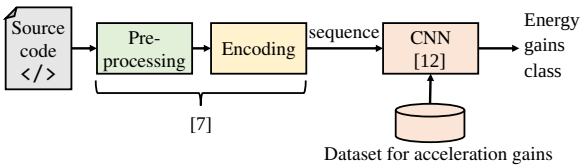


Figure 5: Flow of prediction of energy savings by acceleration based on static analysis

The static analysis flow receives as input the C/C++ CPU source code that is a candidate to be offloaded to a GPU by developing the corresponding CUDA kernel. The static analysis classifies the blocks of source code into one of the two categories, with respect to the expected gains by acceleration: ”High gains” and ”Moderate/No gains”. The goal of this first layer of analysis is to identify with negligible time overhead, the hotspots for which ”High gains” are energy expected (i.e. they will benefit a lot by GPU acceleration). The threshold between ”Moderate/No gains” and ”High gains” classes is selected so that the accuracy of the static analysis is maximized (i.e. the misclassifications are min-

[1] https://git.microlab.ntua.gr/hmar/Decision_Support_for_GPU_Accelerator_dataset

imized). This is demonstrated in the evaluation section. The accuracy of the static analysis significantly decreases when more that two output classes are defined. As a result, discrimination between "Moderate gains" and "No gains" is not possible by using static analysis. Therefore, the dynamic analysis is used for identifying hotspots for which "No gains" are expected and to provide find-grained predictions for "Moderate gains".

From technical perspective, the static analysis approach is based on text analytic approaches. It is inspired by existing work in the literature that analyses source code but for different purposes [7]. As shown in Figure 5, after pre-processing of the source code to remove information irrelevant to the code's structure such as comments, the source code is encoded into sequences of integers. Giving as a vocabulary all the source code derived from the given training dataset, a tokenizer mechanism is used in which each character corresponds to a vocabulary item, with the exception of common language words such as *if*, *for*, *while* etc. The encoded source code (word embeddings) is then provided as input to a classification model, which is trained based on a given dataset of pre-analyzed blocks of code. The prediction model used is a Convolutional Neural Network (CNN) for sequence classification [12]. The sequences retrieved from the aforementioned procedure have a maximum length of 500. The CNN includes 250 hidden neurons and uses filter sizes of 12. In contrast to related works that leverage static analysis (e.g. [2]) techniques, we use a more sophisticated approach that relies only on source code without any dynamic information (such as the number of loop iterations or the direction of branches). This choice may reduce the prediction accuracy, as the actual gains in absolute values are affected a lot by such execution-context metrics. However, considering that the orders of magnitude of the gains are not usually affected by execution-context metrics, we preferred to provide a coarse-grain classification based on static analysis, having the source code as the only input of the prediction model.

An interesting and potentially promising direction worth investigating is the enhancement of the static analysis approach with information obtained earlier by dynamic instrumentation. For example, during the hotspot identification, the CPU cycles are calculated for each code block that is a candidate for acceleration. This information could improve the classification accuracy of the static analysis step. However, in the context of this work, we decided to use a purely static analysis approach for coarse grained prediction and to provide as input the application source code, only. The reason is the fact that application developers familiar with the application under analysis are already aware of the hotspots, therefore, they are expected to skip the entire hotspot identification step.

3.3.3. Dynamic instrumentation

In this paragraph we describe the dynamic instrumentation techniques based sub-component of the proposed method for predicting the potential energy savings by acceleration. The goals of the dynamic analysis are i) to identify the hotspots for which "No gains" are expected using a classification model and ii) to perform a fine grained analysis to the hotspots classified into the "Moderate gains" using a regression model.

The CPU code blocks provided by the hotspots identification phase, are analyzed by profiling tools, such as Intel Pin tools, to monitor the values of accelerator-specific indicators. These indicators capture the extent by which the code behaviour can exploit the architectural features of the GPU accelerator. The selected subset of the most widely used indicators for GPU accelerators, as defined in the literature [1, 3] contains the following metrics:

- Total number of instructions in the code block
- Instruction level parallelism
- Number of cold memory references
- Number of single precision floating point ops
- Number of integer operations
- Number of control operations
- Number of memory operations
- Number of memory accesses with zero stride
- Branch divergence
- Data reuse
- Number of blocks that belong to the same page

The role of the dynamic analysis step is to provide fine grained acceleration gains prediction for hotspots classified into the "Moderate/No gains" category by the previously performed static analysis. The dynamic analysis is obviously more time consuming and this is the reason that it is applied only when fine grained predictions are required (e.g. prediction of gains by acceleration, as percentage of CPU energy consumption).

To select the features used in dynamic instrumentation estimation models, we must investigate whether they are suitable for predicting potential energy savings.

For this purpose, we used the stepAIC method. StepAIC is an automated method that identifies an optimal set of features by selectively adding and removing features in each step and using regression methods to evaluate the importance of each one. AIC stands for Akaike Information Criteria and quantifies the amount of information loss when a feature is removed. AIC estimates the prediction error and thus, the quality of each model.

Table 2: Importance of features in terms of relation to energy

Features	p-value
Instruction Level Parallelism	2.12e-06
Number of instructions	1.44e-07
Number of cold memory references	0.033269
Number of single precision floating point operations	0.035817
Number of integer operations	0.006868
Number of control operations	0.039069
Number of memory operations	5.14e-05
Number of memory accesses with zero stride	1.41e-06
Branch divergence	2.84e-11
Number of division operations	0.061380
Number of blocks accessed in the same page	2.75e-05

Table 2 shows the output of the StepAIC method. It presents the features with the highest relation to the energy consumption of the accelerated version, which are the ones for which the p-value is less or close to 0.05. For a typical statistic analysis, the null hypothesis (i.e. removing a feature from the features vector as it is not related with energy consumption) is rejected when $p < 0.05$ and not rejected when $p > 0.05$.

During an analysis of a hotspot, the values of the features presented in Table 2 are forwarded to the classification model. If the hotspot is classified into the "No gains" class, no further analysis is required. However, if the hotspot is classified into the "Moderate gains", the values are forwarded to a regression model, as it is important to predict the expected gains as accurate as possible. Therefore, the acceleration gains are predicted as a percentage of corresponding CPU energy consumption. For example, energy gains by acceleration 3× means that the accelerated code will consume 3 times less energy during execution, compared to the corresponding CPU code. Fine grained predictions, along with the corresponding information about programming effort will assist developers to decide if it worth developing CUDA code for the hotspots for which moderate energy consumption gains are predicted.

3.4. *Programming Effort Quantification and Proposed Estimation method*

The importance of quantifying and predicting the programming effort required to develop CUDA kernels based on corresponding CPU code was highlighted in Section 2. Furthermore, the need of having more reliable quantification of the effort creates the need of using more complex metrics for programming effort quantification than the lines of code (LOC).

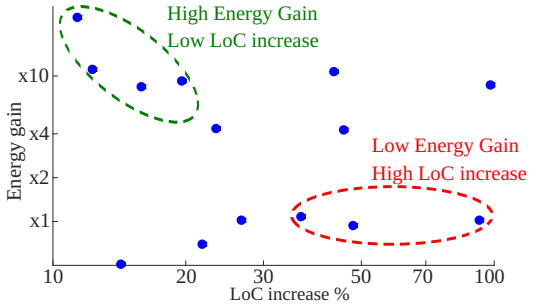


Figure 6: Energy gains vs LOC increase by GPU acceleration in applications of the Rodinia Benchmark Suite

Figure 6 shows the impact of GPU acceleration on a set of applications of the Rodinia suite [5], in terms of energy consumption, as well as the old (for demonstration and motivation purposes) metric of LOC increase of the CUDA version compared to the corresponding CPU version. The vertical axis is the energy gains by GPU acceleration measured in NVidia Jetson TX1 embedded heterogeneous platform [19], while the horizontal is the percentage of LOC increase. As stated earlier, the LOC has been used in the literature as a programming effort indicator [16, 17]. We notice that there are applications that require minor effort to be accelerated, however, the energy gains are very high. On the other hand, there are applications that require significant effort to be accelerated, however the energy gains are relatively trivial. For these applications, developers may decide not to invest in acceleration. Therefore, tool support for predicting both energy gains and programming effort will significantly assist decision making for application developers and contribute to an effective investment of programming time and effort.

However, related studies conclude that for the purpose of effort estimation, using LOC may underestimate the amount of programming effort required and that more sophisticated metrics should be used [24]. Halstead's metrics quantify both the number of distinct operators and distinct operands (their sum corresponds to the program's *Length* (N)), as well as the total number of occurrences of operators and operands (their sum corresponds to the program's *Vocabulary* (n)). The *Volume* of a Program, is obtained as $V = N * log_2 n$ and expresses (in an abstract manner) the size in bits, since the logarithm yields the minimum number of bits required

to represent all operators and operands.

Any piece code, could be theoretically written in its most abstract form, by invoking a hypothetical function that would deliver the same functionality. This compact form of the program would contain the name of the function and a grouping operator in terms of operators, and all unique operands (to be passed as arguments to that function). Thus, the minimal volume for any program can be defined as $V^* = (2 + n_2^*)log_2(2 + n_2^*)$. As a result, the ratio of the minimal volume over the actual volume is called *Level* $L = V^*/V$ and expresses the abstractness of a program. Program *Difficulty* is the inverse of the program level, while the ratio of volume over the level represents a measure of *Effort*, $E = L/V$, to write or understand a program as it considers both its size and abstractness.

According to a highly-acclaimed systematic literature review by Riaz et al. [23] Halstead's *Effort* (among other measures) are considered successful maintainability predictors. Halstead metrics have been also used to measure the development effort in a comparison of high-level parallel programming approaches [15]. They are specifically designed to estimate the time spent on writing an existing source code. Therefore, they are deemed suitable to compare programs providing the same functionality. Since the notion of maintainability captures the ease with which a software component can be modified to adapt to a changed environment, *Effort* can act as programming effort estimates for transforming CPU to accelerator-specific code.

In contrast to related approaches that quantify the programming effort of already written code, in the proposed methodology, the *Effort* is estimated using only the initial CPU code as input, in order to support design-decision for developers with respect to acceleration.

Hotspots classified into the "Moderate gains" category are analyzed in terms of programming effort that is required to develop the corresponding GPU code (e.g. CUDA, OpenCL). For hotspots classified into "High gains", this may not be considered necessary, since developers are expected to develop accelerator-specific code no matter how much effort is required.

To predict the programming effort required to transform a hotspot to the corresponding accelerator-specific code, we use the following indicator: Having as a baseline the Halstead's *Effort* required to develop a part of the CPU code, we predict the percentage of extra effort required to develop the accelerator-specific code. This metric will serve as indicator of the programming effort required to develop the accelerator-specific version of a CPU code. The metrics used as features in the regression prediction model are the following:

- number of hotspots
- LOC of application
- number of hotspots' LOC
- number of hotspots' statements
- distinct and total operators of CPU version
- CPU-code Complexity, Volume, Length, Difficulty

Table 3: Importance of features in terms of relation to Progr. Effort

Features	p-value
Number of hotspots	0.0707
LOC of application	0.0219
Number of hotspot's LOC	0.0219
Number of hotspot's statements	0.0129
Distinct operations (CPU)	0.0238
Total operations (CPU)	0.0308
Complexity (CPU)	0.069
Volume (CPU)	0.0362
Length (CPU)	0.0231
Difficulty (CPU)	0.0289

Table 3 quantifies the importance of the selected features with regards to the programming effort, after applying the stepAIC method.

The programming effort predictions are made by a regression model, which predicts the increased *Effort* for developing the accelerator-specific code in comparison with the CPU *Effort*. The model receives as input the values of the features listed above. The tools that have been used to collect the aforementioned source code metrics are the SonarQube open-source platform [2] for cyclomatic complexity and non-commented lines of code (NCLOC), the Halstead Metrics tool [3] for program Length, Vocabulary, Volume, Difficulty and Effort and a custom made tool for determining the dependencies of each C file on other system files (coupling).

It's worth mentioning here that the main contribution of this manuscript is not based on the use and monitoring of the Halstead metric itself, but rather on the design of a tool that predicts the effort required to develop a new (accelerated) version of the code, before development. To the best of the authors' knowledge, this is a first approach towards using the increase of Halstead effort to create a model that predicts the effort of developing GPU-accelerated code using the CPU-version as the only input, while other metrics could be also used.

[2] https://www.sonarqube.org/

[3] https://sourceforge.net/projects/halsteadmetricstool/

4. Evaluation

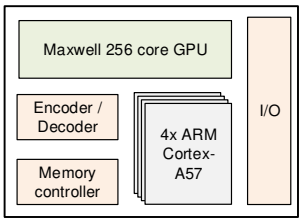


Figure 7: High-level schematic diagram of Tegra X1

The evaluation of the proposed methodology is based on the level of prediction accuracy of each component: The static analysis, the dynamic analysis and the programming effort prediction. The evaluation results are analysed for an heterogeneous embedded platform: NVidia Jetson Tegra X1 that integrates a MAXWELL embedded GPU. A high-level schematic diagram is shown in Figure 7. Energy consumption was measured using the installed power monitor (INA3221). Finally, the proposed tool is extended to support and provide results for two more Nvidia SoC embedded devices, namely Nvidia Jetson Nano and Nvidia Xavier NX in Section 4.3.

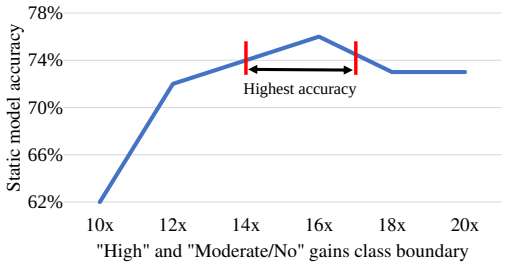


Figure 8: Selection of class threshold between "High gains" and "Moderate/No gains" so that the static model accuracy is maximized

After training the estimation models using the dataset described in Section 3.3.1, we defined the boundary between the "High gains" and the "Moderate/no gains" classes. As mentioned earlier, the boundary is selected so that the misclassifications of the static analysis model are minimized. Figure 8 shows the accuracy of the static prediction model for different boundary values. Accuracy is maximized for boundary value 16×, where it reaches 76%. Therefore, hotspots for which energy gains above 16× are predicted, are classified into the "High gains", while the rest of the hotspots are classified into the "Moderate/No gains" class. This threshold is only recalculated when the dataset changes in order to support additional platforms or to optimize the accuracy for the specified platform. As a result, this procedure takes place rarely, only when the models are retrained.

4.1. Demonstration of methodology

Below, we summarize the evaluation setup:

- *Models*: The static and dynamic models were introduced in Sections 3.3.2 and 3.3.3, respectively. Detailed model selection experiments are presented in Section 4.2.2 and 4.2.4.
- *Features*: The input of the static model is encoded C/C++ source code (3.3.2). The input of the dynamic model is features presented on Table 2.
- *Test dataset and evaluation process*: The test dataset consists of a total number of 100 applications hotspots from Rodinia and Polybench benchmark suites [5, 20]. For the accuracy evaluation, we followed the same approach with similar works in the literature (e.g. [2, 1]): When predicting the energy consumption gains for a single datapoint (i.e. a specific hotspot), we re-train the models with the a training dataset including all datapoints that belong to the rest of the applications (i.e. the remaining applications hotspots). This approach, which is based on a modified leave-one-out cross-validation (LOOCV) is widely used in the analysis of small dataset [2, 1, 13].
- *Evaluation platform*: NVidia Tegra X1 with an integrated power monitor (INA3221 sensor) [19].

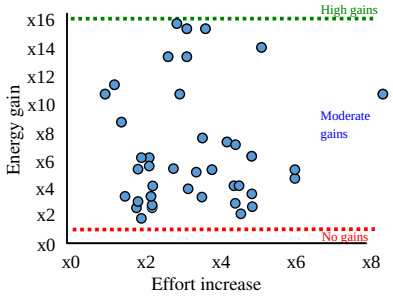


Figure 9: Output of the methodology for the Polybench and Rodinia hotspots that are classified into the "Moderate gains" category.

The output of the proposed methodology for the moderate gains is shown in Figure 9. The Figure shows the predicted energy gains on Tegra X1 vs. the predicted *Effort* required to develop the GPU version of each application in comparison to the corresponding CPU version. Each point corresponds to a single hotspot. 53

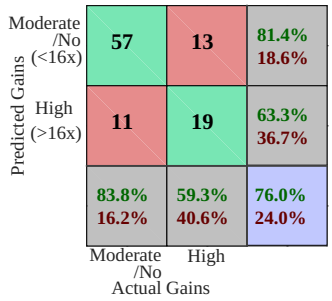


Figure 10: Predicted vs. actual energy gains class: Static analysis

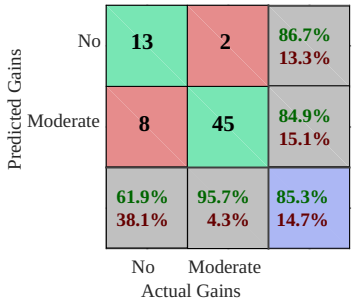


Figure 11: Predicted vs. actual energy gains class: Dynamic analysis

hotspots were classified into the ”Moderate gains” category, while 30 were classified into ”High gains” and for 15 hotspots ”No gains” were predicted.

4.2. Accuracy evaluation

In this subsection we evaluate the accuracy of the prediction models of the proposed methodology.

4.2.1. Static analysis component accuracy

As shown earlier in Figure 8, the accuracy of the static analysis component is 76%. The accuracy is further demonstrated in the confusion matrix of Figure 10. The rows represent the instances in a predicted class, while the columns represent the instances in an actual energy gains class. It shows that the model correctly classifies 76 datapoints (i.e. hotspots) out of 100.

Considering the fact that the static analysis receives as input only the hotspot source code, this level of accuracy is reasonable. As stated earlier, the accuracy of the static analysis component can be potentially improved by receiving as input, apart from the source code, information obtained by dynamic analysis. However, in this work we developed a purely static analysis component and traded classification accuracy for user friendliness.

4.2.2. Dynamic analysis component accuracy

The dynamic analysis component includes a classification step, to identify the hotspots for which no gains are predicted and a regression step to predict the energy gains by acceleration for the hotspots classified into the ”Moderate gains” category. The classification step is based on ensemble method that combines Extra Trees, Bagging Trees and Gradient Boosting (Figure 12). As shown in the confusion matrix of Figure 11, accuracy reaches 85.3%, which means that misclassification probability is lower than 15%.

The values of the dynamic analysis indicators (see Section 3.3.3) are forwarded to a classification model, which is used to identify the proper energy gains class

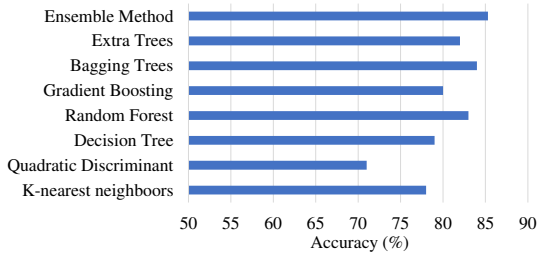


Figure 12: Comparison of accuracy of various classification models for dynamic analysis based estimation

for each given block of CPU code. The model predicts the number of times that the energy consumption is lower when the code is offloaded on the GPU, compared to the corresponding CPU execution. Thus, the model assigns each piece of CPU code to the proper class.

In order to select a suitable classification model, we compare the classification accuracy of various models. The accuracy of the evaluated models is depicted in Figure 12. Then, we utilize an Ensemble Voting technique that incorporates the best 3 models (Extra Trees, Bagging Trees and Gradient Boosting) achieving an energy gains prediction accuracy level of 85%.

For the regression step, we evaluated several algorithms in terms of accuracy, as shown in Figure 13,

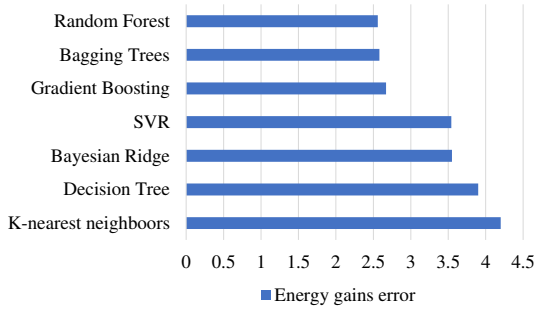


Figure 13: Energy gains prediction accuracy comparison of various regression models

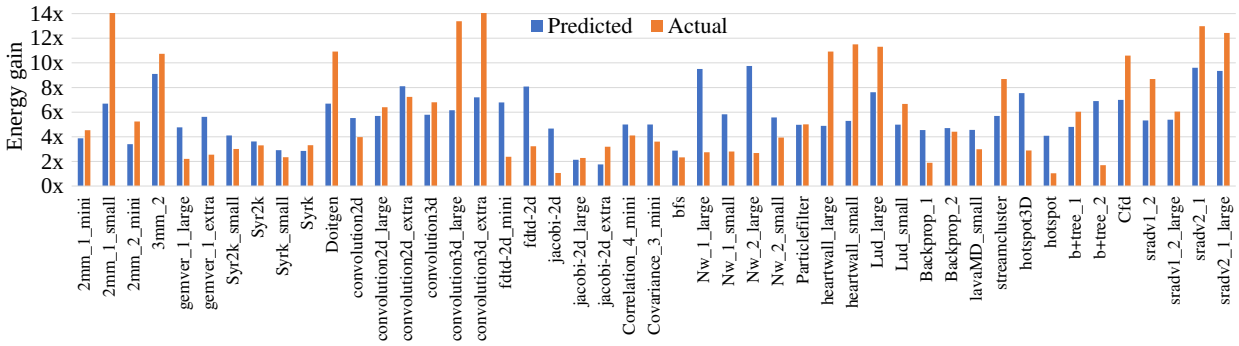


Figure 14: Predicted vs. actual energy gains using regression analysis for hotspots assigned into the "Moderate gains" category (Nvidia TX1)

which shows the mean error of the 7 most accurate models. We selected the Random Forest regression method, which provides the highest accuracy.

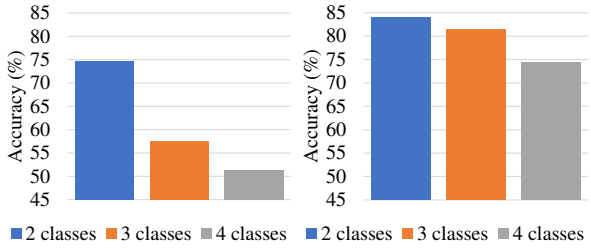
The actual and the predicted energy gains regarding the hotspots (denoted as {*application name*}_{*hotspot id*}_{*input size*}) classified into the "Moderate gains" category are shown in Figure 14. The average error, which is defined as the difference between actual and predicted energy gains gains is 2.6×. However, mispredictions can be more costly for low energy gains, than for higher. As an example, predicting energy gains 4×, while the actual is 2× may affect decision making to a higher degree, than in the scenario of predicting 10×, while the actual is 8×. Based on the results of Figure 14, there are 28 hotspots with actual energy gains below 6×. 22 out of 28 predictions can be considered accurate, since the average error is 1.4× only. However, there are few mispredictions which are attributed to the following reasons:

- Hotspots with relatively small instruction level parallelism may lead to overestimations of energy gains. As an example, *Nw* (for large input) falls into this category: Actual energy gains for *Nw* are around 3×, while the model predicts more than 9×.
- High branch divergence often leads to mispredictions. This was observed for example in *Heartwall*: the model predicts a energy gains below 6×, while the actual gains are more than 10×.

Very few real-world applications and synthetic benchmarks had similar behavior with respect to branch divergence and instruction level parallelism. Therefore, the models are not trained to provide accurate predictions for such cases. However, enhancement of the dataset with more real-life and synthetic applications with behavior similar to the above is expected to improve the energy consumption predictions.

Fine-grain regression analysis is only applied to the estimation of potential energy gains for hotspots classified in the "Moderate Gains" class. As mentioned before, this is a design choice of the proposed framework, as we can consider the expected gains of hotspots classified in the "High gains" class as high enough to convince the developer of the need for acceleration. The quality of a regression process that applies in these cases also lead to higher absolute error as the real values of energy gains are significantly higher.

4.2.3. *Motivation for Combining Dynamic and Static Analysis*



(a) Accuracy of static analysis approach

(b) Accuracy of dynamic instrumentation approach

Figure 15: Energy gains classification accuracy (Nvidia Jetson TX1)

In this paragraph we present our first results that motivated us to select a hybrid approach. If we increase the number of classes in the classification step to three or four, the prediction accuracy results of static and dynamic analysis based approaches are shown in Figure 15 (The classes divide hotspots into equal parts. For example, in the case 3 classes, the boundaries are set so that each 1/3 of the dataset is classified into a different class). According to these results, we might conclude that the classification accuracy of the static analysis approach significantly reduces for more than 2 classes. This shows that the static analysis can be effectively

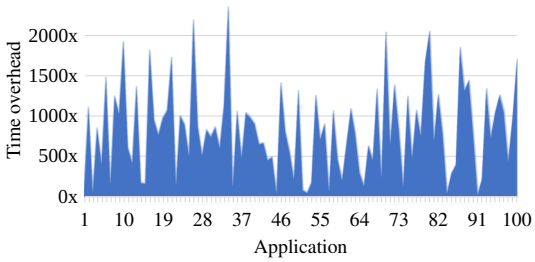


Figure 16: Execution time overhead of the dynamic instrumentation

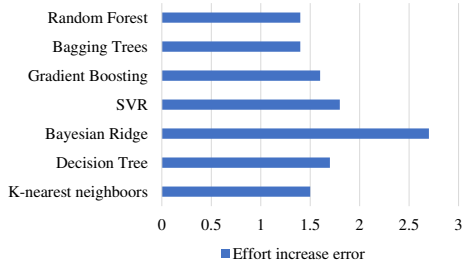


Figure 17: Effort prediction accuracy comparison of various regression models

used for coarse-grained predictions (i.e. up to 2 classes). However, for more fine grained predictions using three or more classes, analysis based on dynamic instrumentation should be employed, as shown in Figure 15b. Indeed, predictions based on the dynamic instrumentation technique provide correct classification with 75% accuracy even for 4 classes.

The overhead of the dynamic instrumentation approach in terms of execution time is shown in Figure 16. The dynamic instrumentation is performed by analyzing each one of the applications of the dataset with Intel Pin tools, (as stated in 3.3.3), in order to extract the features for performing classification. Figure 16 shows that the dynamic instrumentation adds a large time overhead, making the execution more than 2000 times slower in some cases, compared to running the application without dynamic instrumentation profiling. The reason, is the fact that instrumentation process adds extra instructions in application binaries in order to extract the required information. Static analysis, on the other hand, takes only the source code as input without running the application. This means that the time required is fixed, making it easy to implement as part of a software development/analysis toolkit. The static analysis process takes less than 1 second to make a prediction (using a CPU of a personal computer, eg. Intel i7-6400).

These results motivated us to combine the two approaches. The applications hotspots are statically analyzed using text analytic methods, to provide a coarse grained prediction of potential energy gains by acceleration. Applications for which the static approach is not adequate and need a fine-grained classification are further analyzed using dynamic instrumentation.

4.2.4. Programming effort accuracy evaluation

In terms of programming effort, the test dataset includes hotspots from Polybench and Rodinia benchmark suites. We discarded synthetic applications, since only for applications developed by programmers makes sense to evaluate the programming effort. With respect to the selected estimation model, Figure 17 shows that Random forest regression yields the highest accuracy. The prediction accuracy of the programming effort increase of developing CUDA code compared to the corresponding CPU is shown in Figure 18. The average absolute error (i.e. *Effort* increase difference between predicted and actual values) is 1.4×. The accuracy is very high for the applications from the Polybench benchmark suite (about 93%). Few inaccurate predictions are observed in the Rodinia applications, which are much larger and more complex than the Polybench applications. Inaccurate predictions are attributed to the fact that the CPU versions of these applications provide functionality that is not present in the corresponding GPU version. However, we decided to maintain the original structure of the applications, without making any source code modifications.

Figure 19 presents the accuracy of using the proposed models to predict the CUDA code development effort expressed in the more traditional LOC increment. The accuracy is considered high (87.3%). However, as we'll also analyse in Section 4.4, LOC may not be a good estimator of the programming effort.

4.3. Extension - Adding more Devices

The detailed experiments, presented above, were performed on Nvidia Jetson TX1 device, highlighting the accuracy of each component. In this paragraph, we evaluate the extensibility of the proposed tool and the ability to support more (similar) devices. A prerequisite for being able to support the new device is that it must include an energy sensor. The step-by-step guidelines provided below show how to add predictions for new devices:

- Step 1: Download the dataset from git repository
- Step 2: Build the dataset in the new device.
- Step 3: Run the dataset. The developer has to update the paths of power sensors of the new device.
- Step 4: The final dataset is added in the tool. The models are re-trained in the new data (transfer learning).

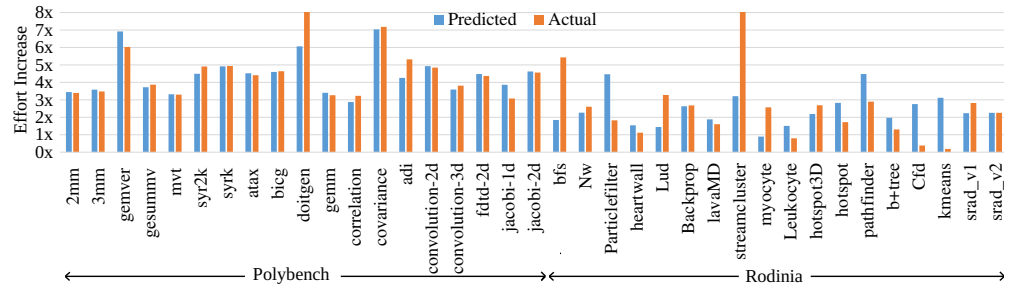


Figure 18: Predicted vs. actual programming effort increase using regression analysis

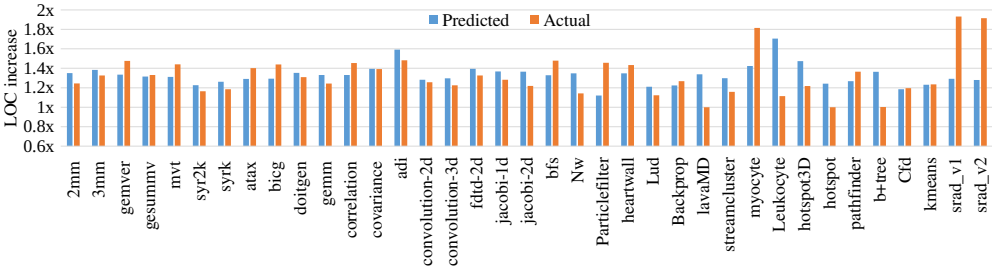


Figure 19: Predicted vs. actual LOC increase using regression analysis

4.3.1. Evaluation on Nvidia Jetson Nano

Nvidia Jetson Nano is very similar to TX1. It incorporates a version of the SoC (Tegra X1) with the same CPU ARM Cortex-A57 at a maximum frequency of 1.48GHz (MaxN power mode is used) instead of 1.73GHz of TX1 and a smaller 128-core Maxwell GPU at 921MHz instead of TX1's 256-core GPU at 994MHz. It should be noticed here, that we used the default power modes for the two platforms (without custom frequency scaling) and the dataset was build with keeping this configuration constant in all measurements, as the methodology aims to use the application software as input.

For Nano, Rodinia/Polybench hotspots are classified slightly differently, as we have 4 more hotspots in the "Moderate gains" category and 1 more in the "No gains" category, while the values of energy gains differ in most cases. The accuracy of each component is similar to the results for TX1. More precisely, the static analysis classification reaches 78% accuracy, while the accuracy of the dynamic analysis classification is around 82%. The results of the fine-grain regression analysis for "Moderate gains" hotspots are presented in Figure 20. Based on these results, we observe a similar performance, as the average error is 2.9×, while the proposed model misspredicts the energy gains of the same benchmarks described in 4.2.2 (eg. the Nw app). In terms of programming effort, the results are the same as shown in 4.2.4, as the developed source code (CUDA) is the same.

4.3.2. Evaluation on Nvidia Jetson Xavier NX

NVIDIA Jetson Xavier NX is a high performance edge device that focuces on AI applications. It incorporates a different architecture (Nvidia Volta) including 384 CUDA cores and a 6-core NVIDIA Carmel ARMv8.2 CPU, with 2-level cache and 8 GB of RAM.

For Xavier NX, Rodinia/Polybench hotspots are classified differently. 28 hotspots show no gains, while 39 hotspots are expected to have high Energy gains (more than 16x). The increased capabilities of this device in both CPU and GPU compared to the Tegra X1 offer better CPU performance and much faster but more more power consuming GPU usage. These characteristics increase the energy gains of some hotspots, such as 2mm and 3mm from the Polybench suite due to the higher efficiency that exceeds the higher power consumption and reduces the energy gains for kernels that do not benefit from the increased number of GPU cores. Classification accuracy decreases to 78% for dynamic analysis and 75% for static analysis. This is due to the fact that we maintain the same boundary between the "Moderate Gains" and "High Gains" classes, which is based on accuracy optimization for TX1. Figure 21 presents the regression results for "Moderate gains" hotspots. Based on these results, we observe a similar performance to the other devices, as the average error is 2.7×. In terms of effort, the results are again the same as the developed source code (CUDA) is the same.

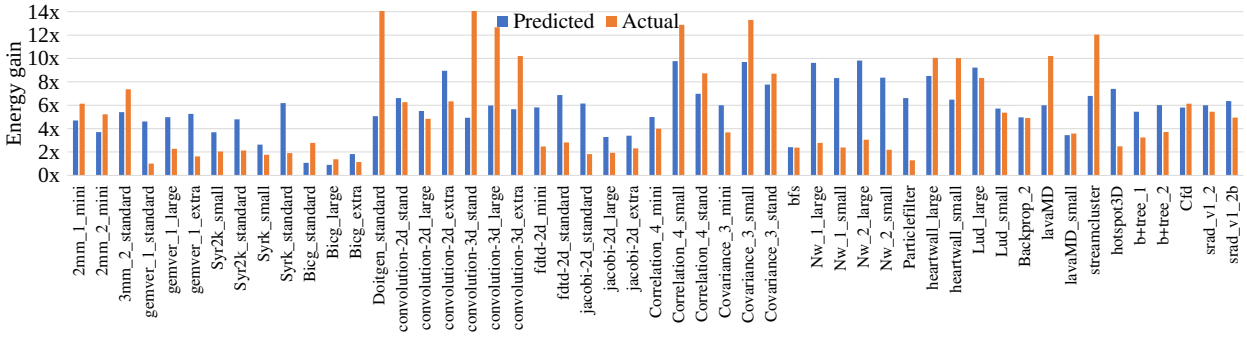


Figure 20: Energy Gains Prediction results for Nvidia Jetson Nano

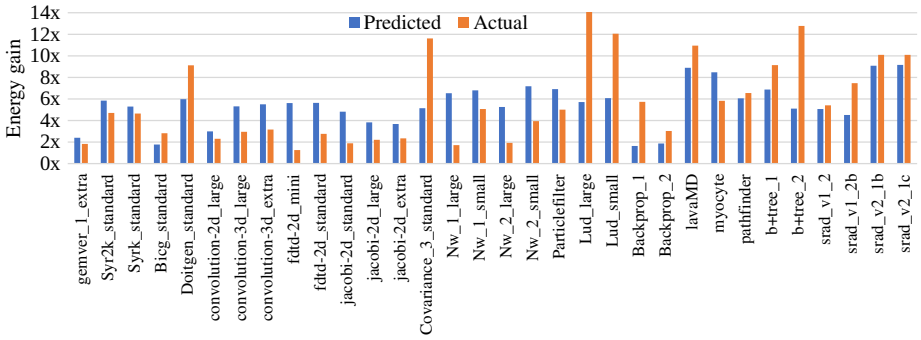


Figure 21: Energy Gains Prediction results for Nvidia Jetson Xavier NX

4.4. Discussion

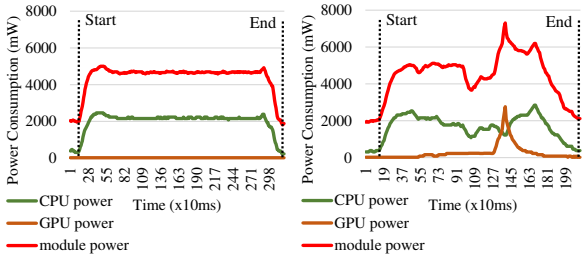
In this subsection, we summarize our observations with respect to the evaluation of the proposed method.

The prediction of energy consumption gain by acceleration in heterogeneous embedded devices is feasible. Existing works focus on the prediction of speedup by offloading a piece of CPU code on GPGPUs [1, 26]. However, in the area of embedded heterogeneous systems, energy efficiency is an equally important quality that affects design decisions. Based on the presented results, we conclude that the existing approaches that analyze CPU code to provide speedup predictions can be extended towards predicting the energy gains by acceleration. One may argue that energy consumption can usually be estimated based on the prediction of execution time by using the power delay product or an analytical model. However, the effort required to design an accurate model that correlates execution time with energy consumption is very high, particularly for heterogeneous systems such as CPU-GPU devices that we target in the context of this study. Indeed, power is not constant because it depends on a wide range of parameters (memory transactions, cache behavior, number of cores used, CPU utilization, etc.), which significantly increases the complexity of designing a relatively accurate analytical model that relates the two metrics. This is also indicated in Figure 22. Figure 22a shows the power consumption of the CPU, GPU, and the entire module unit to run the convolution-3d application from the Polybench benchmark suite, while Figure 22b shows the same power consumption modules for the GPU version. Based on this experiment, we observe that the additional power of the GPU leads to a higher instantaneous module power consumption compared to the CPU-only version, while the power consumption can not be considered stable at all. Based on these results, we can expect that the performance gains should be more than the energy savings in most cases. Therefore, the correlation between the two metrics is not straightforward. Although an analytical model that relates performance with energy consumption for heterogeneous systems would be very useful and practical for application developers, it is beyond the scope of this work.

The overhead of dynamic instrumentation can be avoided in cases in which high energy gains by acceleration are predicted. In contrast with existing works ([1, 26]), the fact that the proposed methodology, predicts the cases for which relatively high gains are ex-

Table 4: Comparison against related tools

	Estimated Metric	Method	Targeted platform	Accuracy	Extensibility/Usability
[1]	Speed-up	Dynamic	CPU-GPGPU	76% - 88% regression	Medium
[26]	Speed-up	Dynamic	CPU-GPGPU	81% regression	Medium
[2]	Speed-up	Partially Static	CPU-GPGPU	62-85% classification	Medium
Proposed	**Energy gain**	**Both Static and Dynamic**	**Embedded**	**Static classification 76% Dynamic classification 85.3% Regression 63% (median)**	**High (guidelines and scripts provided)**
	Effort (LOC, Effort increase)	**Static**	**-**	**Regression 87%**	**High**



(a) CPU version power consumption (b) GPU version power consumption

Figure 22: Convolution3d power consumption on Nvidia TX1

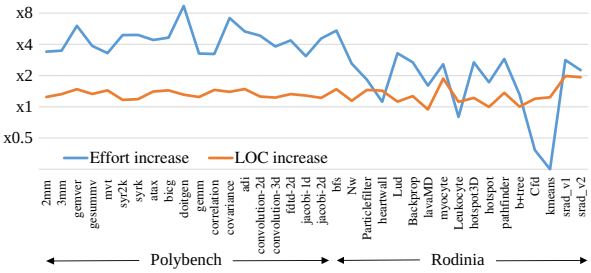


Figure 23: LOC and *Effort* actual increase of GPU version in comparison to CPU for Polybench and Rodinia benchmarks

pected using only static analysis is a significant advantage for application developers. The time required to reach a prediction by using dynamic analysis reached even 10 hours for some applications included in the dataset. However, in the case of applying static analysis, prediction results are instantly generated. Therefore, the combination of both techniques compromises granularity and time overhead to reach a prediction.

The quantification and the prediction of programming effort is important, especially in cases in which the predicted energy gains are relatively low. For such applications, we observe that it is feasible to predict the effort that is required to develop the acceleration-specific code using only the CPU code as input. Such a solution can contribute significantly in assisting developers in deciding whether to invest in acceleration. Using advance metrics, such as the *Effort*, the programming labor required for developing CUDA code is captured more accurately than using LOC. Figure 23 shows the actual LOC and *Effort* increase between the CPU vs. GPU versions for the Polybench and Rodinia applications. We notice that *Effort* ranges from x1 and x8, while LOC ranges from x1 to x2. However, in several cases the increase in LOC can be safely attributed to statements such as definitions and variable initialization, which cannot be directly related to programming effort. On the other hand, *Effort* is mainly affected by the number of operands, operators and function calls. Therefore, programming effort is more accurately expressed through *Effort*. Indeed, in Figure 23 we notice that there are applications in which there is a very small increase in LOC, however the effort increases by more than x4. This observation is inline with Shihab et al. that state that relying in LOC often leads to underestimation of programming effort [24]. Interestingly, there are some applications for which the effort seems to decrease. These applications are from the Rodinia benchmark suite, as shown in Figure 23. The reason, as explained in Sec 4.2.4, is the fact that the CPU versions of these applications provide functionality that is not present in the corresponding GPU version.

Extending the methodology for providing predictions in other embedded platforms is straightforward. There are no constraints in the proposed methodology with respect to portability. When accurate energy consumption measurement is feasible, the proposed methodology can be easily applied not only to typical CPU-GPU architectures, but also to domain specific accelerators, as well as to data flow architectures.

Comparison against related tools: Although we focus on energy instead of performance gains, a comparison against related tools is summarized in Table 4. We should mention that our main purpose is to provide an extensible tool that makes the prediction of energy gains as well as estimating the effort of developing GPU-accelerated code prior development feasible and not to increase the accuracy of models as much as possible.

5. Conclusions

This work proposes a methodology for Energy Consumption and Programming Effort estimation of ac-

celerating CPU applications on heterogeneous devices. It relies on both static and dynamic analysis of CPU source code to provide a compromise between prediction granularity and time overhead. It shows that approaches focusing on speed-up can be extended towards Energy Consumption predictions for heterogeneous embedded systems. The methodology, supported by a tool-flow, aims to assist application developers to decide whether to invest in accelerating CPU applications, considering not only speed-up but Energy Consumption and programming effort for developing CUDA code, as well. Accuracy evaluation performed in well-known (Nvidia GPU based) heterogeneous platforms, where energy measurements are feasible and enabled by sensor, shows that the proposed methodology is able to provide accurate predictions and that a future increase of the size of the dataset has the potential to further improve the accuracy of the models.

6. Acknowledgements

This work is partially funded by the EU Horizon 2020 research and innovation program, under project EVOLVE, grant agreement No 825061.

References

[1] Ardalani, N., Lestourgeon, C., Sankaralingam, K., and Zhu, X. (2015). Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737. ACM.

[2] Ardalani, N., Thakker, U., Albarghouthi, A., and Sankaralingam, K. (2019). A static analysis-based cross-architecture performance prediction using machine learning. *arXiv preprint arXiv:1906.07840*.

[3] Baldini, I., Fink, S. J., and Altman, E. (2014). Predicting gpu performance from cpu runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 254–261. IEEE.

[4] CEVA XM6 (2018). Specifications of CEVA XM6:. `https://www.ceva-dsp.com/wp-content/uploads/2017/01/CEVA-XM6-Product-Note.pdf`.

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.

[6] Chen, Y., Chen, T., Xu, Z., Sun, N., and Temam, O. (2016). Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112.

[7] Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE.

[8] Georgiou, S., Rizou, S., and Spinellis, D. (2019). Software development lifecycle for energy efficiency: techniques and tools. *ACM Computing Surveys (CSUR)*, 52(4):1–33.

[9] Hamm, A., Willner, A., and Schieferdecker, I. (2019). Edge computing: a comprehensive survey of current initiatives and a roadmap for a sustainable edge computing development. *15th International Conference on Wirtschaftsinformatik*.

[10] Helal, A. E., Feng, W.-c., Jung, C., and Hanafy, Y. Y. (2017). Automatch: An automated framework for relative performance estimation and workload distribution on heterogeneous hpc systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 32–42. IEEE.

[11] Intel/Movidius Myriad (2019). Specifications of Intel/Movidius Myriad:. `https://www.movidius.com/myriadx`.

[12] Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

[13] Lachenbruch, P. A. and Mickey, M. R. (1968). Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11.

[14] Lee, S., Meredith, J. S., and Vetter, J. S. (2015). Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 405–414.

[15] Legaux, J., Loulergue, F., and Jubertie, S. (2014). Development effort and performance trade-off in high-level parallel programming. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 162–169. IEEE.

[16] Li, X., Shih, P.-C., Overbey, J., Seals, C., and Lim, A. (2016). Comparing programmer productivity in openacc and cuda: an empirical investigation. *International Journal of Computer Science, Engineering and Applications (IJCSEA)*, 6(5):1–15.

[17] Memeti, S., Li, L., Pllana, S., Kołodziej, J., and Kessler, C. (2017). Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6.

[18] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[19] Nvidia Tegra X1 (2017). Specifications of Nvidia Tegra X1:. `https://shield.nvidia.com/blog/tegra-x1-processor-and-shield`.

[20] Pouchet, L.-N. et al. (2012). Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*.

[21] Ramprasad, B., da Silva Veith, A., Gabel, M., and de Lara, E. (2021). Sustainable computing on the edge: A system dynamics perspective. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 64–70.

[22] Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, pages 22–es.

[23] Riaz, M., Mendes, E., and Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE.

[24] Shihab, E., Kamei, Y., Adams, B., and Hassan, A. E. (2013). Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, 55(11):1981–1993.

[25] Silvano, C., Fornaciari, W., and Villar, E. (2014). *Multi-objective design space exploration of multiprocessor SoC architectures*. Springer.

[26] Wang, S., Zhong, G., and Mitra, T. (2017). Cgpredict: Embedded gpu performance estimation from single-threaded applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):146.