Making the Most of Scarce Input Data in Deep Learning-based Source Code Classification for Heterogeneous Device Mapping

Making the Most of Scarce Input Data in Deep Learning-based Source Code Classification for Heterogeneous Device Mapping

Emanuele Parisi, Francesco Barchi, Andrea Bartolini, and Andrea Acquaviva

***Abstract*—Despite its relatively recent history, Deep Learning (DL) based source code analysis is already a cornerstone in machine learning for compiler optimization. When applied to the classification of pieces of code to identify the best computation unit in a heterogeneous Systems-on-Chip, it can be effective in supporting decisions that a programmer has otherwise to take manually. Several techniques have been proposed exploiting different networks and input information, prominently sequence-based and graph-based representations, complemented by auxiliary information typically related to payload and device configuration. While the accuracy of DL methods strongly depends on the training and test datasets, so far no exhaustive and statistically meaningful analysis has been done on its impact on the results and on how to effectively extract the available information. This is relevant also considering the scarce availability of source code datasets that can be labelled by profiling on heterogeneous compute units. In this paper, we first present such study, that leads us to devise the contribution of code sequences and auxiliary inputs separately. Starting from this analysis, we then demonstrate that by using normalization of auxiliary information it is possible to improve state-of-art results in terms of accuracy. Finally, we propose a novel approach exploiting Siamese networks that further improve mapping accuracy by increasing the cardinality of the dataset, thus compensating for its relatively small size.**

***Index Terms*—Heterogeneous platform, Deep learning, Machine learning, Source Code analysis, Computation mapping.**

I. Introduction

Augmenting the intelligence of compilers to solve the problem of efficiently mapping programs to heterogenous platforms is the subject of very recent and rich research [1], [2]. In the last five years, new approaches based on supervised Deep Learning (DL) models have been proposed to analyse and classify source code to decide the best compute unit or allocation configuration using performance or energy metrics [3], [4], [5]. It is known that DL algorithms are very powerful and able to automatically extract relevant features from input data, however they require a representative dataset for train and test to achieve high classification accuracy and avoid overfitting. In case of source code analysis, a dataset is represented by fragments of source code that are compiled and profiled for all the possible compute units in the target platform. Depending on the selected metric (e.g. execution time, power or energy), a label is assigned indicating the best compute unit where each fragment has to be allocated. In literature, basically all state-of-art papers dealing with DL for source code analysis targeting heterogeneous CPU/GPU platforms make use of the dataset designed by Cummins et al. in [5]. This is composed by 256 kernels (belonging to different benchmark suites) written in C with OpenCL libraries and profiled for CPU and GPU. As additional input to the network, besides code sequences, auxiliary information (we also refer to it as meta-information) representing payload and work group size is provided. Auxiliary inputs increase the size of the dataset creating additional combinations. Typically, they enter the network directly in the final classification layer, that takes as another input the output of a DL language model stage, whose job is to extract meaningful information from code sequences.

All authors: Department of Electrical and Computer Engineering, Università di Bologna, Bologna, Italy e-mail: emanuele.parisi@unibo.it.

The dataset used [5] has become a de-facto reference for any new techniques proposed. However, the impact of the relatively limited size of the dataset and of auxiliary inputs has never been discussed nor analysed. The large majority of literature papers use this dataset as it is without making a careful analysis of the information content and how to maximize its exploitation for DL algorithms training and test. This is in true for source code datasets in general.

Conversely, such analysis it is of uttermost importance not only to understand the contribution to classification accuracy of auxiliary information and DL-based language models separately, but also to devise techniques able to better exploit both type of information.

In this paper we first present an in-depth statistical analysis of the dataset introduced in [5]. In particular we study the similarity between code sequences and discuss their subdivision in train and test set, revisiting the conclusion provided in [6] and showing a more fair and effective subdivision. Apart from its importance because of its impact on this widespread used reference, the analysis we present provides insight for evaluation or creation of benchmarks for source code classification.

Second, we evaluate the impact of the meta-information on the classification and we show how this can be enhanced by a normalization technique, increasing the accuracy of kernel mapping with respect to state-of-art DL networks.

Finally, we propose a novel approach to exploit the information contained in the dataset through Siamese networks [7]. The dataset size is a general issue related to code benchmarks for language modelling with the purpose of classification. Since the kernels have to be compiled and executed on different targets, using code collections downloaded from on-line repositories

as they are is not viable in general. From the other side, synthetic code generation based on predefined models lacks generality and significance. This is the reason it is so hard to find alternative datasets in this research line.

Siamese networks have been recently proposed to overcome the issues lead by small datasets. Indeed, they increase the cardinality of the dataset by creating couples of dataset points and using the information of their distance for classification. They are called Siamese because they implement two equal networks each one working on a single element of a dataset pair. Through a contrastive loss function and a subsequent projection to a bidimensional space the final classification is performed. In this work Siamese are exploited to increase the dataset cardinality (the considered one is made of 680 samples of which 256 are code sequences). The resulting classification accuracy of CPU/GPU mapping reaches around 92% on AMD and 89% on NVIDIA datasets, leading to an improvement of around 6% with respect to the best state of art technique. By introducing and demonstrating the effectiveness of Siamese networks for information extraction from source code datasets, this work opens the way to the application of the proposed method to code datasets designed for purposes beyond device mapping.

The rest of the paper is organised as follows: In Section II, we provide background on source code analysis and Siamese networks, we describe the relevant related works and the reference dataset. In Section III, we describe our methodology, focusing on source code, meta-information and dataset evaluation. In Section IV, we evaluate the accuracy of the proposed methods compared with state-of-art DL approaches. Finally, in Section V we draw some conclusions and we discuss possible future works.

II. Background and Related Works

As the effect of the end of Dennard's scaling and the slow down of Moore's law, heterogeneous architectures composed of general-purpose processors tightly coupled with HW accelerators have become prominent in the full-spectrum of computing systems. Often, to decide if a given source code can effectively take advantage of the HW accelerator requires to port the source code to the target accelerator. Moreover, the accelerators' performance does not depend only on the source code but also on auxiliary parameters, which depends on the dataset/payload [5].

A. Source Code Analysis

To ease the programmer from this daunting task several works in the recent years aim at exploiting source code features for predicting the device with the shortest runtime where to execute a computation.

To do so researchers tried to answer the question *can we analyse the code like text?* exploiting natural language code modelling, translation, and classification techniques for code quality assessment [8], plagiarised source code detection [9], classification for execution on a certain hardware target [5], [10], [11], [12], [13]. Source code analysis could also be applied in optimisations for power energy and thermal resources [14] and in the context of code execution and memory access patterns optimization [15]. As shown in the survey of Allamanis [1], the source code maintains some properties of natural languages (it can be considered, like text, a human communication form) but it has profound differences. Some of the code properties like executability, formality and structure make it more complex to analyse than text. Compiler designers started considering the adoption of machine learning techniques to obtain heuristic compilers capable of learning from the data [2], [16] for code optimisation. Research in the field of Natural Language Processing (NLP) has evolved considerably in recent years. The current SOTA is composed by the language models BERT [17], XLNet [18] and GPT [19]. While BERT uses Autoencoders and token-masking to generalise model knowledge, XLNet and GPT use self-regressive models based on transformer networks [20]. Different versions of GPT-3 composed of a variable number of weights (from 3B to 175B) are able to solve problems of text generation, question answering, reading comprehension in zero-shot, one-shot and few-shot mode. Recently the ability of GPT-3 to generate JavaScript XML (JSX) code in few-shot (two sample context) mode has been shown. These techniques are the cutting edge of research on generic text understanding models, but their size does not make them easily applicable in domain-specific contexts. Several domain-specific techniques, instead, have been proposed in the literature to represent programs using a set of quantifiable properties or features compatible with the inputs of the learning module [21]. Standard machine learning algorithms typically work on fixed-length inputs, so the selected properties shall be transformed into a fixed-length vector of features (boolean, integer, or real values). Compiler researchers have designed, during the years, various forms of program features for machine learning algorithms. These include static code structures extracted from the source code or the compiler intermediate representation [22] and dynamic profiling information obtained through run-time profiling of the program execution [23]. Compiler optimisation methods based on supervised learning have been proposed using Bayesian Networks [24], Support Vector Machines [25], [26], Decision Trees [21], [27] and Graph Kernels [28]. In the last decade, the problem of deciding the most suitable hardware unit on which to execute a given computational kernel raised with the increasing complexity and heterogeneity of digital platforms. Source code analysis can help this decision avoiding to make useless porting efforts. Moreover, a profiling approach based on source code analysis without the need for the final target hardware or an accurate virtual (simulation or emulation) platform can speed up the embedded systems development process. Along this direction, in 2013, Grewe [29] developed a workflow to translate an OpenMP program in OpenCL and to decide for each generated OpenCL kernel the most suitable compute unit between CPU and GPU. There, the authors defined metrics to extract from the code (like the number of calculation operations or local and global memory access) to make decisions based on a probabilistic method (i.e. a decision tree classifier). In last

years, a research line exploiting the maturity of deep learning methods has started since the work of Cummins et al. [5], where the decision tree classifier was replaced with a deep learning model based on a RNN. Thanks to deep learning, it is no longer needed to extract the features manually since they are inferred automatically during the training phase and improves classification accuracy compared to [29]. The methodologies proposed in [29] and [5] were developed and customised for kernels implemented in OpenCL, thus constraining the methodology to work with a given source programming language. To overcome this limitation, Ben-Nun et al. [11] and Barchi et al. [10] introduced the adoption of code analysis at the intermediate representation (IR) level of the LLVM compiler. LLVM is increasingly adopted in the embedded system world, because it is capable of decoupling the front-end compiler from the target architecture, in this way many optimisation steps can be performed at the IR level before generating the binary machine code. Source code features, at this intermediate level, can be exploited to perform complex compilation decisions, including allocating code fragments to architecture devices. Machine learning techniques can be applied to learn these characteristics by creating a learning model based on training code fragments. The LLVM based methods presented in [10] and [11] differ for the strategy for the projection of source code in the continuous metric space. In [10] the code stream is filtered and then introduced directly into the network, relying on the Embedding Layer for the learning of the best token projection. On the other side, in [11], the authors propose Inst2Vec a system to pre-train the embedding layer analysing the Contextual Flow Graph (XFG). Further contributions to this projection problem have been devised later. In Kheerthy et al. [12], a procedure to project an IR in a continuous metric space directly, called IR2Vec is proposed. In Cummins et al. [13], the authors propose *ProGraML*, an extension of [11] where a GNN-based classifier is proposed for the first time. Independently, in Brauckmann et al. [6] another GNN-based classifier was proposed able to learn vertex embeddings by itself. Moreover, in [6] the GNN is used to analyse both an LLVM-IR Control and Data Flow Graph (GNN-CDFG) and a Clang Abstract Syntax Tree (GNN-AST). Concerning the deep neural network model, all previous work use RNNs, that have been introduced to process temporal sequences [30], [31]. An RNN maintains an internal state, acting as a memory, that summarises the information extracted from the input sequence. Very successful implementation of RNN is the Long Short-Term Memory (LSTM), a network able to learn when to memorise or forgot information of the input sequence and correlate together elements at different times. For this reason, LSTM is the model adopted in state-of-art papers [5], [11], [12]. However, given the widespread use of CNN in the context of image recognition [32] but also in NLP [33] as well as fast learning time and the maturity of network design and configuration tools, it is worth exploring their application to source code classification. Behind the success of this type of network, there is the assumption of information locality in the input data. All data inside a region called "kernel" are considered correlated, and this correlation is weighed by a filter, identical for any region considered in the input. For image classification, the kernel shape has two dimensions, but this technique can also be used in temporal signals using one-dimensional kernels. This is the approach we follow in this work, where we give as an input to the CNN a tokenised and filtered code stream directly without using additional information.

Suite	Version	Benchmarks	Kernels	Samples
amd-sdk	3.0	12	16	16
npb	3.3	7	114	527
nvidia-sdk	4.2	6	12	12
parboil	0.2	6	8	19
polybench	1.0	14	27	27
rodinia	3.1	14	31	31
shoc	1.1.5	12	48	48
Total		71	256	680

Table I: Dataset composition [5]. The first two columns are the number of benchmarks in suite (Benchmarks) and the number of unique kernels in suite (Kernels). In the complete dataset, composed by the tuple Code and Meta-information, each suite has a different number of pairs (Pairs).

In [3] authors introduced for the first time in a code classifier method CNNs, by giving as an input to the network a tokenised and filtered code stream directly without using additional information. Authors compared the CNN-based proposed approach with the RNN-based network used in [5] and [10] outperforming previously proposed methods. Results from [3] confirm that features extraction from IR is a valuable strategy for analysing sources without dealing with complex high-level constructs, and it can be done keeping all the information required for performing classification tasks in the context kernel-device mapping.

B. Reference Dataset

The majority of the methods discussed [6], [13], [5], [10], [3] are trained and evaluated on a common dataset introduced in [5]. The dataset consists of 256 OpenCL kernels sourced from seven benchmark suites on two combinations of CPU/GPU pairs. Each pair is labeled CPU/GPU accordingly to the processing element in which it executes faster. The same pair has been executed in two different heterogeneus system configurations: The AMD set uses an Intel Core i7-3820 CPU and AMD Tahiti 7970 GPU; the NVIDIA set uses an Intel Core i7-3820 CPU and an NVIDIA GTX 970 GPU. Each dataset consists of 680 labeled pairs derived from the 256 unique kernels by varying dynamic inputs. Each pair is characterized by three values: the code, and two auxilliary inputs, namely the payload size and OpenCL Work Group size. The 256 unique kernels belong to 7 suites and 71 benchmarks as reported in the Table I.

In [6], [13], [5], [10], [3] the proposed models' performance evaluation has been conducted with k-fold cross-validation. Authors of [6] introduce a k-groups-split methodology to stress out the generalization capability of the proposed methods. In the latter, the dataset is split into parts along the benchmark suites. Then models are trained in all the pairs belonging to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3114617, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

k-1 and evaluated in the k-th part. Authors of [6] report poor code generalization performance on the k-groups-split. In this manuscript we will tackle this issue by doing further analysing the issue. Table I shows that the number of pairs in each suite is strongly inbalanced. With the NPB benchmark suite accounting for more than three quarters of the total pairs composing the dataset.

In this manuscript we propose an in depth analysis of the code generalization capability of the models by studying in isolation the relative impact of the code and of the auxiliary inputs on the models accuracy. We focus our analysis on the CNN-based approach presented in [3] as it outperforms previous methods[13], [6].

Our results show that previous methods fail in extracting usefull information from the auxiliary inputs. To do so we propose a pipeline of value normalization capable of reprensenting in a correct way the information presents in auxiliary input. Our result show an increase of the $10-20\%$ of accuracy w.r.t previous methods when only auxiliary inputs are considered. When both the code and auxiliary inputs are considered the proposed method achieves an increase of the $4-6\%$ in accuracy w.r.t previous methods.

To overcome the limitation of k-groups-split validation we propose a new partitioning based on the benchmarks rather than on the suite.

We than propose a new training method based on siamese networks to increase the dataset size without adding new pairs. Siamese networks increase the cardinality of the dataset by training the parameters of the model on the distance between each pair in the dataset.

III. Methodology

The proposed methodology starts from the dataset analysis. Each dataset element is composed of a source-code component and a meta-information component. We will refer to these elements with the following notation: $D : (D_C, D_M)$ where D_C is the code component and D_M is the meta-information component. In [5] the dataset structure is composed of a triple of values. The meta information in this case is $D_M : (A^1, A^2)$ where A^1 and A^2 (also called auxiliary inputs) represent the amount of data processed (Payload) and the device configuration (OpenCL Work Group size) respectively. The third element is the kernel sequence that is associated to these inputs. The payload impacts the transfer time of data towards the accelerator, and the workgroup size affects the kernel parallelism. Because of its structure, using the dataset D for train and test, the overall classification depends both on source-code analysis capabilities and impact of auxiliary information. However, to distinguish these contributions is relevant to better design deep learning models. This is even more important considering the small size of the datasets available [5]. In that case, the strategy, used in split data to build training-set and test-set, must be analyzed in detail to avoid creating sets with insufficient coverage in the feature space.

This section will explore the dataset shape in terms of meta-information, source-code and dataset division strategies. We will show how this information can be used to lay the foundations for a fair comparison between classifiers techniques. In section III-A we will explore D_C in its hierarchical structure as benchmark suites. Moreover, the code serialization of kernels can be compared using string distance algorithm; this provides a way to explore the benchmark suite similarity. In section III-B we will explore three different ways to build training-set and test-set; stratified k-fold, benchmark-folding and suite-folding. Finally, in section III-C we will explore D_M in its feature space using Decision Trees (DT) and a Multi-Layer Perceptron (MLP).

Suite	Samples	D - AMD			D - NVD		
		CPU	GPU	Imb.	CPU	GPU	Imb.
amd-sdk	16	10	6	62/38	1	15	6/94
npb	527	326	201	62/38	199	328	38/62
nvidia-sdk	12	1	11	8/92	5	7	42/58
parboil	19	9	10	47/53	13	6	68/32
polybench	27	2	25	7/93	12	15	44/56
rodinia	31	17	14	55/45	19	12	61/39
shoc	48	35	13	73/27	44	4	92/8
Total	680	400	280	59/41	293	387	43/57

Table II: Class imbalance in the complete datasets. For each suite we point out the number of samples and the percentage of label imbalance.

Suite	$\widetilde{D}_C$ - AMD				$\widetilde{D}_C$ - NVD			
	K.	CPU	GPU	Imb.	K.	CPU	GPU	Imb.
amd-sdk	16	10	6	62/38	16	1	15	6/94
npb	45	37	8	82/18	68	22	46	32/68
nvidia-sdk	12	1	11	8/92	12	5	7	42/58
parboil	6	2	4	33/67	6	4	2	67/33
polybench	27	2	25	7/93	27	12	15	44/56
rodinia	31	17	14	55/45	31	19	12	61/39
shoc	48	35	13	73/27	48	44	4	92/8
Total	185	104	81	56/44	208	107	101	51/49

Table III: Class imbalance in the datasets with no auxiliary inputs. For each suite we point out the number of samples and the percentage of label imbalance.

A. Source Code Impact Analysis

The source code can be grouped in a three level hierarchy: suite, benchmark, kernel. The dataset has seven suites, and each suite is composed of a variable number of benchmarks. Each benchmark, in turn, is composed of one or more kernels. Our goal is to explore the dataset D_C to identify the best way to analyze the code and evaluate language models based on deep learning.

The source code is provided in OpenCL. As described in [3] we transform the source code in a sequence of tokens after a compilation step in LLVM-IR. In [10], a detailed discussion about the sequence length and the padding procedure explains how to reduce the code in a limited range of 2048 items.

Given a dataset $D : (D_C, D_M)$ with 680 entries, if we remove the meta-information component the resulting dataset D_C is composed of 680 sequences of code. Still, only 256 of them are uniques (see Table I).

When we try to reduce this dataset (removing the duplicates) to obtain a dataset composed of unique sequences we face

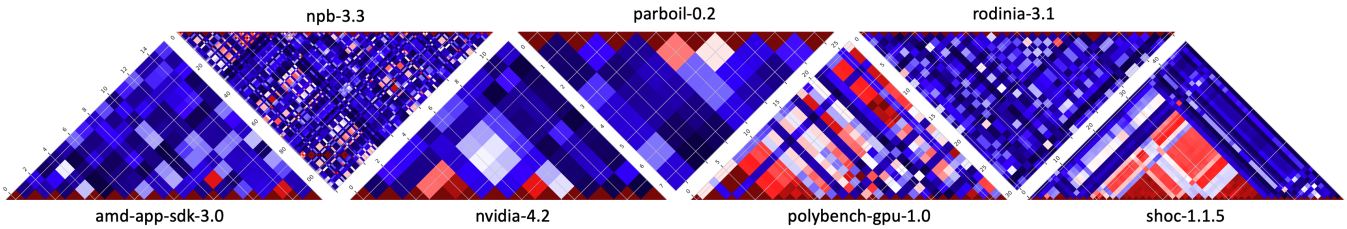

Figure 1: Each triangle rapresents the suite self-similarity computed using Needleman–Wunsch. The color code represents the dissimilar pairs with cold tones (blue) and the most similar pairs with warm tones (red).

the following problem. Some kernels present label incoherency, meaning that the same kernel has been assigned with different labels depending on the meta-information. Since our goal is to decouple code and meta-information impact, in this phase we only consider code sequences that have a single label match.

If a sequence of code has the same label independently from the meta-information value for which it has been evaluated, then it will be assigned that label. If, on the other hand, the code sequence has, for instance, label A if coupled with a given meta-information value and label B if coupled with other meta-information values, then it will not be considered for the construction of the dataset. After this procedure, we obtain a new dataset; we will refer to it with the following symbol $\widetilde{D}_C$. The usage of this dataset is only for analyzing the training-set and test-set construction methods and for evaluating the language analysis models directly on the code. In Section IV, to make a comparison with the state of the art, we will use the complete dataset, which includes the meta-information. In section III-B, we present the classification performance of $\widetilde{D}_C$, using different strategies to build test-set and training-set.

Small datasets are subject to high accuracy fluctuations due to incomplete manifolds considered in training and testing. The ideal condition for making fair comparisons is that both training-set and test-set embrace the same feature space of the entire population we consider (the dataset we have). Given the limited number of sequences in our dataset, it is useful to build a strategy to identify similarities between code sequences to evaluate the classification results and training-set and test-set split strategies.

As specified in Schölkopf et al. [34], when a statistical learning method is used (i.e. DL), if the data do not satisfy the condition of "independent and identically distributed (i.i.d.)", it is not possible to obtain an adequate level of generalization and therefore learning. So, we can say that if the test and train sets are not "statistically" similar, then a model cannot be learned. In this subsection, we study the code similarity in the dataset. In the following subsection, this analysis provides the foundation to study dataset split methods that mitigate code dissimilarities during train/test set creation.

We use Needleman–Wunsch (NW), a global alignment algorithm belong to family of algorithms used in compilers in order to reduce the program size choosing how to fuse functions [35]. It is also used in other fields: from optimizing the configuration time of neuromorphic hardware, [36], [37], [38] to genetic sequence and amino acid sequence alignment [39], [40] in bioinformatics. We apply NW using naive scores for Match, Mismatch and Insertion or Deletion cases. We assign a positive score of $+1$ for a token match and -1 for all other cases. Considering the token sequences, we have sequences of a fixed length of 2048 tokens. Sequences shorter than 2048 tokens fill the remaining positions with a special token (padding). When we consider the similarity between two sequences, we want to avoid to take into account the padding as match-cases; we remove all the padding from the sequence with less padding and reduce the length of the sequence with more padding until we get two sequences of the same size.

Preserving padding in only the shortest sequence allows us to penalize the similarity score in the unlucky case that a short sequence A is very similar to a portion of a longer sequence B. It also simplifies the normalization of the similarity score.

A pair of two sequences that share the same tokens in the same order (identical sequences) has a similarity-score equal to the number of tokens. For this reason, the similarity-score is normalized by the number of tokens, obtaining a value in a [-1, +1] range.

Given a set of sequences the mean of the distribution of the similarities of all pairs in the set is the *self-similarity* of the set. Given two sets of sequences, say A and B, the mean of the distribution of similarities of all pairs (a, b) where a is an element of A and b an element of B is the *cross-similarity* of the sets.

In Figure 1 we show the matrix of similarity-score for each suite in the dataset. The colour coding uses cold tones for negative similarity (prevalence of mismatch, insertions and deletions in the code-sequence pair) and warm tones for positive similarity (prevalence of token match in the code-sequence pair). It is possible to qualitatively evaluate the high similarity of polybench, shoc and npb suites.

B. Dataset Creation Strategies

Deep learning methods need huge amount of data to learn useful features and to provide reliable accuracy scores. When the amount of data is limited, due to the difficulty of finding or labeling large quantities of samples, the DL methods are evaluated using a different strategy to increase the data coverage. Mainly, this is performed by training the model different times using each time a different way to split the dataset in training-set and test-set.

One of these procedures is called *k-fold cross-validation*. In k-fold cross-validation the dataset is split in K folds, in turn each fold is used to test a classifier trained using the remaining $K-1$ folds. As result the entire dataset is valuated

in test. A variant, called "stratified", try to preserve the dataset imbalance in labels for each fold. We will refer to this variant, the "stratified k-fold cross-validation", using the acronym *SKF*. If the dataset is very small the cross-validation procedure can be performed multiple times in order to evaluate a statistic.

Other techniques to split the dataset in training-set and test-set can be used. For example, we can use the membership of a kernel in a suite or in a benchmark in order to define a dataset splitting policy. The most radical choice in this case is to use the suite membership as done in [6]. With this splitting policy we divide the dataset in seven folds, each containing all the kernels of a suite. In tables II and III are shown the compositions of these suite for both the full dataset D and the code-only dataset $\widetilde{D}_C$.

Another way to create folds can be the usage of benchmark membership. The following procedure create a valid fold composed by kernels belonging to seven benchmarks.

```
function GET_FOLD(len_min, len_max, imbalance_max)
    iter ← 0
    repeat
        fold ← new list
        for all suite in dataset do
            benchmark ← get_random_benchmark(suite)
            for all kernel in benchmark do
                fold ← add kernel
            end for
        end for
        f_size ← len_min ≤ len(fold) ≤ len_max
        f_imbalance ← imbalance(fold) ≤ imbalance_max
        iter ← iter + 1
    until (!f_size or !f_imbalance) and (iter≤max_iter)
    return fold
end function
```

A valid fold must have a number of kernels in a range between the 15% of the dataset size and a label imbalance less than 70/30.

To summarize, we have three different methodologies to build the training set: i) SKF, ii) Benchmark Folding iii) Suite Folding. In order to evaluate these dataset splitting policies, we trained a DeepLLVM model [4] on $\widetilde{D}_C$ dataset, which, in these experiments, will therefore be fed using only code sequences.

Table IV describes the performance of the CNN model on the dataset $\widetilde{D}_C$. Experiments based on Benchmark Folding perform always better than Suite Folding. In particular, Benchmark Folding causes the CNN to have accuracies higher than 70% and MCC better than 0.4. On the contrary, Suite Folding accuracy never reaches 50% and a MCC around zero.

Such results can be explained by referring to Figure 2, where the Suite Folding has the larger self-similarity in the test-set. This shows that the set of samples used to train the model may not be sufficiently representative of the kernel space. This splitting policy is unfair to evaluate how good a model is at learning program representation.

[1]The Matthews correlation coefficient is a fair metrics for unbalanced dataset, its values range is in [-1, +1]

		AMD					
		ACC			MCC		
		SKF	Bench.	Suite	SKF	Bench.	Suite
CNN	μ	.808	.730	.492	.611	.471	-.007
	σ	.020	.124	.047	.041	.250	.089
	CI_{95}	.014	.025	.033	.010	.051	.064

(a) Accuracy and MCC obtained with $\widetilde{D}_C$ - AMD Dataset

		NVIDIA					
		ACC			MCC		
		SKF	Bench.	Suite	SKF	Bench.	Suite
CNN	μ	.774	.756	.443	.548	.521	-.137
	σ	.026	.116	.049	.053	.225	.107
	CI_{95}	.019	.024	.035	.038	.046	.076

(b) Accuracy and MCC obtained with $\widetilde{D}_C$ - NVIDIA Dataset

Table IV: Benchmark and Suite splitting methodologies experiments. We consider as metrics the well-known accuracy and the Matthews correlation coefficient (MCC[1]).

More specifically, in Figure 2 we compare the properties of the three techniques used to split the dataset in training-set and test-set, namely *SKF* (Red dots), *Benchmark Folding* (Green dots) and *Suite Folding* (Blue dots). In the Figure, each dot represents one fold, and each star represents the centroid of the folds. The left plot in Figure 2 shows, in the x-axis, the cross-similarity between the training-set and test-set. In the y-axis, it shows the average accuracy, averaged among the repetitions of each fold. The size of each dot represents the min-max variation in the model accuracy. The right plot in Figure 2 shows, in the x-axis, the self-similarity of the training-set. In the y-axis, it shows the self-similarity of the test-set.

All the similarity-scores have been normalized with respect to the self-similarity of the whole dataset. In this way, we can observe the deviation of training-set and test-set from the mean of the whole dataset when the three different fold construction techniques are used.

From the plot on the left, it is apparent that, on average, the *SKF* has a cross-similarity equal to the self-similarity of the entire dataset. In contrast, the *Benchmark Folding* has on average the lowest cross-similarity. It is important to notice that even if the *Benchmark Folding* has a lower cross-similarity between the test-set and training-set, the models trained with this policy has an higher accuracy than the *Suite Folding*. This can be explained by looking at the right plot in Figure 2. The plot reports in the x-axis the average self-similarity for the training-set and in the y-axis the average self-similarity for the test-set. From it, we can notice that the *Suite Folding* has on average the highest self-similarity in the test-set. This information, when combined with the standard deviation in the accuracy, which is the highest for the *Suite Folding*, suggests that the resulting folds have test-set populated with more similar codes samples. It is likely that the trained model between the different repetition is subject to overfitting and local minima (high min-max variability) and achieves poor performance on average when tested in relatively similar codes. This does not happen in the *SKF* and *Benchmark Folding* which is by

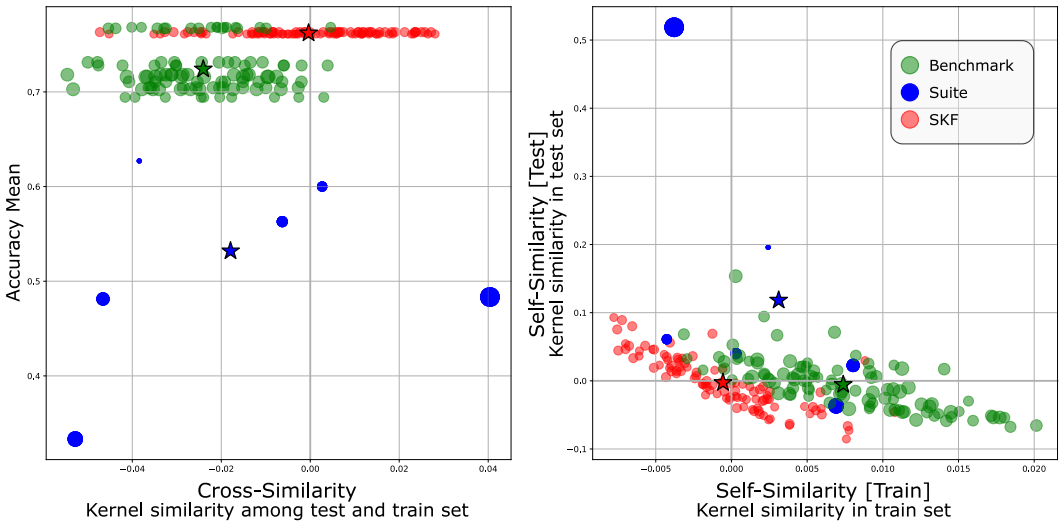

Figure 2: Results of CNN analysis in $\widetilde{D}_C$ - AMD using different train-test divisions: In red the Stratified 10-Fold, in green the Benchmark division, in blue the Suite division.

construction more representative of the entire dataset in each fold.

It is also important to note that *SKF* has the lowest self-similarity in the training-set and thus is characterized by a more "rich" training-set. As expected, this split leads to the best model results. We must notice that being the dataset pruned by kernels repetitions training-set and test-set by construction cannot contain the same kernels. Notice that this does exclude the case in which some kernels in the training set happen to be highly similar to one or more kernels in the test set. We conclude that, in this evaluation, the *SKF* split has the best characteristics in terms of separation of the test and train set and fidelity with the original dataset content.

C. Meta-information Impact Analysis

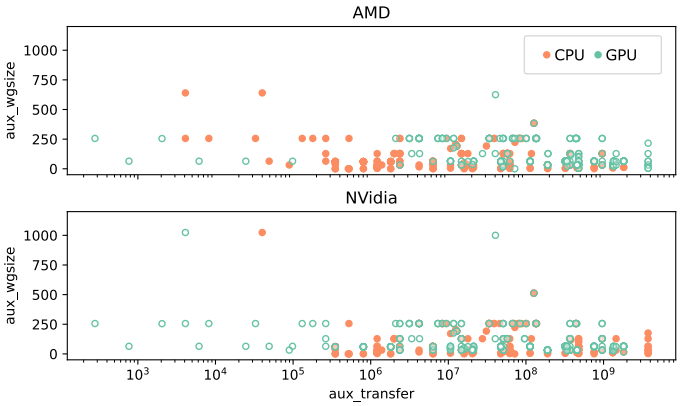

Figure 3: Distribution of auxiliary input features for the two datasets considered. Each point is coloured in green or orange depending on its label.

Meta-information defines the application context such as payload and device configuration (such as GPU work group size) For the heterogeneous-mapping problem used in [5] the meta-information space is shown in Figure 3. As done in section III-A, in order to analyze this dataset, we start from

	AMD			
	ACC		MCC	
	DT	MLP	DT	MLP
$\widetilde{D}_M$ Raw	0.738	0.465	0.488	-0.012
$\widetilde{D}_M$ Normalized	0.711	0.731	0.422	0.472

(a) Accuracy and MCC obtained with AMD Dataset

	NVIDIA			
	ACC		MCC	
	DT	MLP	DT	MLP
$\widetilde{D}_M$ Raw	0.727	0.535	0.456	0.074
$\widetilde{D}_M$ Normalized	0.724	0.636	0.453	0.248

(b) Accuracy and MCC obtained with NVIDIA Dataset

Table V: Results of meta-information dataset classification using decision tree (DT) and multi layers perceptron (MLP)

$D : \{D_C, D_M\}$ and remove the source code component. The resulting dataset $D_M : \{A^1, A^2\}$ is composed of auxiliary information pairs. A_1 is the kernel payload in byte and affects the data movement phases. A_2 is the work-group size, an OpenCL platform parameter that changes the device parallelism.

We want to analyze this dataset in order to evaluate the classification accuracy of two classifier models: a decision tree (DT), a machine learning technique used in [4], and a multi-layer perceptron (MLP), the baseline of all deep learning techniques. This analysis can give insight in how manage this part of dataset. For the same reasons discussed in section III-A about the incoherency in labels, from the D_M dataset we removed duplicated elements with incoherent labels. The resulting dataset $\widetilde{D}_M$ is then analyzed in Stratified 10-Fold Cross Validation (SKF) using both DT and MLP. The evaluation was repeated ten times in order to provide more stable results (random initialization of model parameters and training-set division lead to slightly different results)

In Table Va we can observe a 73% of accuracy in the decision

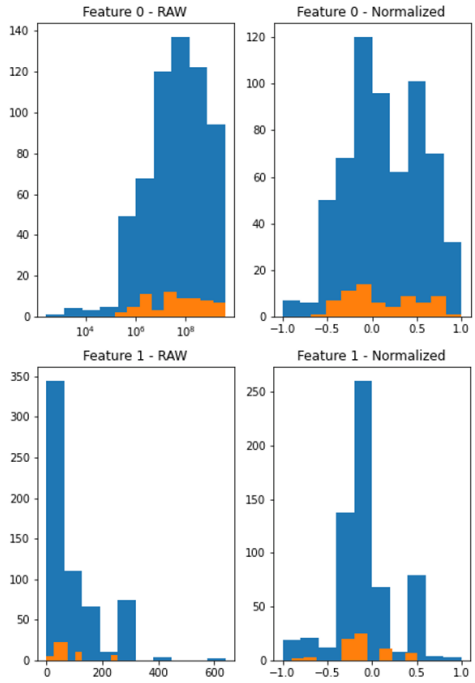

Figure 4: Distribution of the meta-information before and after normalization. In blue the train-set distribution, in orange the test-set distribution.

tree model, which is acceptable considering the use of meta-information. On the other side, the MLP model is unable to learn giving accuracy of around 50% (and around 0 for MCC. The problem is that Deep-learning techniques assume well formed data, where the range lies in [0, 1] or [-1, 1] [41]. This is necessary in order to compute correct gradients and avoid issues related to the numerical stability of the learning procedure. This is not the case for the considered auxiliary inputs.

We address this problem indroducing a pre-processing procedure of data in the training-set. Specifically we used three different normalization steps in cascade: A power transform, a standard scaler, and a min-max scaler. The power transform [42] implement a parametric monotonic transformations to make data more Gaussian-like. The standard scaler removes the mean and scales data to obtain unitary variance. Finally, the min-max scaler scales the data values in a [-1,1] range. In Figure 4 an example of pre-processing procedure is depicted. The parameters of the pre-processing procedure was learned from the training-set (in blue) and applied in the test-set (orange). The data coverage in enough to avoid out-of-range values in test-set. In this way, the huge features range is transformed and can be processd by the MLP.

The results obtained with this procedure are presented in Table Vb. The MLP now is able to learn in both dataset flavors (AMD and NVIDIA), and reaches the performance of the decision tree classifier. Although the decision tree performs reasonably well, we are interested in the use of MLP as it is embeddable in the code analysis model.

The CNN model, as described in [3], merges the aux

Parameters		CNN	Siamese
	Epochs	70	15
	Batch size	32	64
Optimizer	Learning rate	1e-3	1e-3
	Weight decay	5e-4	5e-4
LR scheduler	Factor	5e-1	-
	Threshold	1e-4	-
	Patience	5	-
Contrastive loss	Margin	-	2

Table VI: Training and callbacks hyper-parameters used to train the proposed machine learning models.

inputs with the source code features, then performs a batch-normalization step in the same way as in previous works [5], [11], [10], [6], [13]. The proposed techniques, input pre-processing and MLP, can be applied jointly with batch-normalization and improve its performance. In section IV we will show that pre-processing step and the embedded MLP layer improve the classification accuracy of the CNN based classifier. Furthermore, we will show that the proposed technique also improves the results of other methods found in literature originally using batch normalization without input pre-processing.

IV. Results

This section reports a set of experiments we designed to validate the methodologies described in Section III. Subsection IV-A provides the details about the structure of the machine learning model tested. We also provide information about the hyper-parameters we used to setup the gradient descend optimizer and the learning-rate scheduler. Subsection IV-B reports the impact of auxiliary input pre-processing on classification performance. At last, Subsection IV-C compares the performance of our models with alternative state-of-the-art methodologies. The source code of our implementation can be found online in a public Git repository[2].

A. Machine learning models and training hyper-parameters

We conducted a set of experiments using two machine learning models, which from now on will be referenced as CNN and Siamese. The CNN model is designed taking inspiration from the network presented in [3]. It exploits a 1D convolutional layer and a global max pooling filter to extract the most relevant features from a stream of LLVM-IR tokens. Such features are concatenated with the auxiliary inputs and classified using a multi-layer perceptron. As discussed in Subsection III-C, we enhance the previous CNN topology adding a fully-connected layer of neurons before joining auxiliary input features with the output of the language modelling sub-network. Subsection IV-B shows how auxiliary inputs pre-processing and the additional dense layer introduced in their processing path impact the classification performance of the CNN.

Siamese refers to the siamese network we implemented. It has the CNN described in the previous paragraph at its core tuned to project every dataset sample in a two-dimensional

[2]https://gitlab.com/ecs-lab/deepllvm

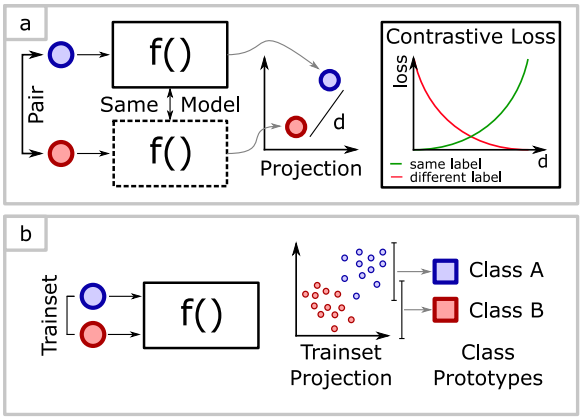

Figure 5: Siamese network training. First, the weight of the core network are trainined using the contrastive loss computed on the projections of the points in the train folds. Then, centroids of same class samples are computed and the a label is assigned to each sample in the test set, depending on the closer centroid.

space. It means that the final layer of the multi-layer perceptron has two neurons without activation function. The procedure to train siamese networks is depicted in Figure 5. For each combination of sample pairs in the train folds, each element S_1 and S_2 of the pair is separately fed into the core of the siamese network which projects it into a 2-dimensional space. Then, the distance between the projections of S_1 and S_2 is computed and the weights of the siamese core network are updated according to the following function, called *constrastive loss*

$$loss = \begin{cases} d(S_1, S_2)^2, & \text{if } label(S_1) = label(S_2) \\ \max(0, m - d(S_1, S_2))^2 & \text{otherwise} \end{cases}$$

where m is a function hyper-parameter. The loss function behaves such that samples with different labels are moved away, while samples with equal labels are penalized proportionally to the distance of their projections. At the end of training epochs, all projections of train samples are collected and for each class a centroid is computed by averaging all the projections of samples of that class. Finally, the siamese network is tested by computing the projections of all points in the test fold and assigning a label depending on the closer centroid.

Table VI details the training framework we set up to evaluate our models. The CNN was trained with a batch size of 32 for 70 epochs, while the siamese training used 64 samples batch and lasts for 15 epochs. Both models are trained using the Adam optimizer with a learning rate of 0.001. Since the CNN model is trained for a larger number of epochs, it takes advantage of a learning-rate scheduler which pregressively reduces the learning rate the optimizer uses as the model advance trough the epochs. The learning rate scheduler we use is controlled by three parameters: *patience*, *threshold* and *factor*. After each training epoch the train loss of the model is compared with the one of previous epochs. If it does not improve more than *threshold* for a number of epochs (*patience*), then the learning rate of the optimizer is multiplied by *factor*. Such a technique may prevent the gradient descend algorithm to get stuck in some local minima of the features space and it is known to

Experiment	Additional model techniques	
	input pre-processing	input dense-layer
A	No	No
B	Yes	No
C	No	Yes
D	Yes	Yes

(a) Experiments composition.

		AMD Experiments							
		ACC				MCC			
		A	B	C	D	A	B	C	D
CNN	μ	.853	.882	.868	**.890**	.695	.758	.726	**.775**
	σ	.013	.008	.008	**.006**	.027	.015	.017	**.012**
	CI_{95}	.009	.006	.006	**.004**	.020	.011	.012	**.008**
Siam.	μ	.882	.910	.873	**.917**	.757	.816	.738	**.829**
	σ	**.006**	.008	.009	.007	**.012**	.015	.019	.014
	CI_{95}	**.004**	.005	.007	.005	**.009**	.011	.014	.010

(b) AMD Experiments

		NVIDIA Experiments							
		ACC				MCC			
		A	B	C	D	A	B	C	D
CNN	μ	.823	.843	.830	**.873**	.638	.678	.653	**.767**
	σ	.010	**.008**	.009	.009	.021	**.017**	.017	.018
	CI_{95}	.007	**.006**	.006	.006	.015	**.012**	.012	.013
Siam.	μ	.859	.885	.832	**.888**	.713	.765	.657	**.771**
	σ	.010	**.008**	.010	.009	.021	**.017**	.020	.018
	CI_{95}	.007	**.006**	.007	.006	.015	**.012**	.014	.013

(c) NVIDIA Experiments

Table VII: Impact of auxiliary input pre-processing on model accuracy.

be beneficial to improve deep learning models performance. We set the margin (m) hyper-parameter of the contrastive loss equal to 2.

Experiment outcomes are evaluated using two metrics: accuracy (ACC) and Matthews correlation coefficient (MCC). We consider MCC since it is proven to be more effective than accuracy and F1 score to evaluate the effectiveness of binary classifiers when classes are imbalanced [43]. All experiments are repeated 10 times with different fold splits and the results are described providing mean (μ), standard deviation (σ) and 95% confidence interval (CI_{95}). All experiments were run on a server equipped with a 24 GB RAM NVIDIA Quadro RTX 6000 GPU.

B. Impact of auxiliary input pre-processing

The impact of pre-processing and the presence of the additional dense layer on the auxiliary inputs processing path is validated checking the classification performance of the CNN and the Siamese model. We propose two modifications over the traditional way auxiliary inputs are treated in other literature works on the topic.

First, the raw auxiliary input values present in the dataset are pre-processed using three successive scalers:

- A power transformer based on the Yeo-Johnson method [42] is used to make the shape of the data distribution more Gaussian-like.

		Test on state-of-the-art models					
		AMD			NVD		
		A	B	Δ	A	B	Δ
DeepT.	μ	.814	**.855**	.041	.805	**.839**	.034
	σ	.020	**.015**	-.005	.008	**.007**	-.007
	CI_{95}	.014	**.007**	-.001	.006	**.005**	-.001
CDFG	μ	.864	**.889**	.025	.814	**.853**	.039
	σ	.010	**.007**	-.003	**.006**	.009	.003
	CI_{95}	.007	**.005**	-.002	**.004**	.006	.002

Table VIII: Impact of auxiliary input pre-processing on DeepTune [5] and CDFG [6] methodologies.

- A standard scaler removes the mean from the samples and scales them to unit variance.
- As a last step, the data are scaled to a fixed range, between -1 and +1.

Additionally, we add a single-layer fully-connected perceptron in the auxiliary input analysis path, in order to give the network more degree of freedom for reshaping the features space at training time. The two modifications we propose are orthogonal, and one does not imply the other. We run four experiments, namely A,B,C,D, to evaluate them separately. The composition of each experiment is reported in Table VIIa. Experiment A reproduces the results shown in [3] for the CNN, and provides a baseline for the evaluation of the performance of the siamese network.

Tables VIIb and VIIc describe the outcome of the four experiments. For each metric and statistics computed, the best result is bolded. Experiment D consistently maximizes both accuracy and MCC scores in all experiments for both datasets. It proves that not only auxiliary input normalization is beneficial for classification performance but also adding an additional dense layer is a promising strategy. The CNN reaches a classification accuracy of 89.0% in the AMD dataset, showing 3.7% better accuracy than the baseline depicted in experiment A. Considering the NVIDIA dataset, the two modifications we propose deliter a boost of 5% of accuracy which increases from 82.3% to 87.3%.

We also observe that using the CNN in the siamese framework still increase classification performance. The siamese networks reach a top classification accuracy of 91.7% on the AMD dataset and of 88.8% on the NVIDIA dataset, providing higher performance with respect to alternative methodologies for source code mapping on heterogeneous platforms.

Additionally, we prove that auxiliary input pre-processing is beneficial when applied to other state-of-the-art methodologies for source code device mapping. Table VIII shows the impact of the proposed pre-processing pipeline on DeepTune [5] and CDFG [6]. Experiments highlight that adding auxiliary input pre-processing leads to an increase in accuracy of at least 2.5%, with a peak increase of 4.1% in the case of DeepTune tested on the AMD dataset.

Table IX gives an idea of how much auxiliary inputs impact the performance of a source code classifier trained on the available dataset. If $\widetilde{D}_C$ is used in place of D classification accuracy experience a drop of approximately 8.2% in AMD dataset and 9.9% in NVIDIA. Interestingly, the mean performance does not drop drammatically, which gives a hint that the neural networks are able to extract sufficient information from the source code alone. We can also comment on how the presence of auxiliary inputs make the outcome of the models more stable. In facts, removing them make the standard deviation and the confidence interval larger in the general case, with respect to the standard experiment.

C. Comparative results

Table X compares the CNN and the Siamese with other state-of-the-art tools that solve the problem of heterogeneous device mapping. To be fair, we limit the selection of alternative methodologies to the one which focus on LLVM-IR analysis (we include DeepTune [5] for historical reasons.).

All state-of-the-art works considered, including the proposed one, use Stratified K-Fold Cross Validation (SKF). Depending on the authors, the methodology used to build training-set and test-set is referred to by different names (e.g. fixed split, random split). The work in [5] shows the results of only one experiment, while [6] reports the average of 10 experiments. In this work, we report results on an average of 10 experiments, where each one has a different assignment of the samples in the folds. Considering the small size of the dataset, the composition of the folds may lead to significant performance variations for a given model. To obtain a fair comparison between the different techniques and machine learning models, it is thus preferable to consider the statistics of the results obtained by varying the order of the samples within the folds across different experiments, which is achieved by re-running SKF for each experiment. To this purpose, we have replicated the techniques used for comparison, where each experiment was performed using the same setup used for our models. Specifically, we used "SKF Training Set with 9/10 folds - 10 Repetitions - Rebuild Folds". [3] To obtain a fair comparison between the proposed technique and ProGraML [13], which uses a smaller training set (80%), and therefore a less favourable condition in training, we re-trained CNN and Siamese using a splitting rule with the same fraction of the dataset used for training as in [13]. Specifically, we used "SKF Training Set with 8/10 folds - 10 Repetitions - Rebuild Folds". Looking at results reported in Table X, it can be observed that the proposed methods show a significant accuracy improvement with respect to state-of-the-art.

The two auxiliary input processing techniques described in the present work (normalization and additional dense layer) boost classification accuracy of the baseline model described in [3] from 85.5% to 88.2% on average. Using such model configured for siamese training provides a further performance improvement, significantly outperforming currently available methodologies. Furthermore, an interesting feature of auxiliary inputs normalization and siamese framework is that they can be applied to all of the methods reported in Table X.

[3]When designing the testbed architecture to compare our methodology to state-of-the-art, we chose to rebuild the SKF folds for each methodology tested. The reason is that most state-of-the-art tools have custom test code and do not allow to force training on a specific set of samples without jacking with their implementation.

		AMD			
		ACC		MCC	
		D	$\widetilde{D}_C$	D	$\widetilde{D}_C$
CNN	μ	**.890**	.808	**.775**	.611
	σ	**.006**	.020	**.012**	.041
	CI_{95}	**.004**	.014	**.008**	.010
Siam.	μ	**.917**	.789	**.829**	.578
	σ	**.007**	.019	**.014**	.038
	CI_{95}	**.005**	.013	.029	**.027**

(a) AMD dataset

		NVIDIA			
		ACC		MCC	
		D	$\widetilde{D}_C$	D	$\widetilde{D}_C$
CNN	μ	**.873**	.774	**.741**	.548
	σ	**.009**	.026	**.018**	.053
	CI_{95}	**.006**	.019	**.013**	.038
Siam.	μ	**.888**	.769	**.771**	.539
	σ	**.009**	.024	**.018**	.048
	CI_{95}	**.006**	.017	**.013**	.035

(b) NVIDIA dataset

Table IX: Stratified 10-folds cross-validation experiments.

State-of-the-art methodologies		AMD	NVIDIA	Mean
DeepTune	[5]	.814	.805	.810
NCC/inst2vec	[11]	.802	.810	.806
CDFG	[6]	.864	.814	.839
DeepLLVM	[3]	.853	.823	.838
CNN	this work	.890	.873	.882
Siamese	this work	**.917**	**.888**	**.903**

(a) State-of-the-art methods were re-evaluated in this work using SKF and Training Set with 9/10 folds.

State-of-the-art methodologies		AMD	NVIDIA	Mean
ProGraML	[13]	.866	.800	.833
CNN	this work	.894	.877	.886
Siamese	this work	**.908**	**.879**	**.894**

(b) CNN and Siamese methods were re-evaluated in this work using SKF and Training Set with 8/10 folds.

Table X: Comparison with state-of-the-art methodologies.

V. Conclusions

In this work, we presented a strategy for making the most of a source code dataset used in heterogeneous device mapping by using a deep learning classifier. We analysed the dataset in its components: the source code and the meta-information. For the source code dataset, we analysed the code repetitions and label incoherence, and we define a procedure to obtain a similarity measure between two kernels (code sequences). Moreover, we explored three different strategies to define training-set and test-set. Stratified K-Fold (SKF), Benchmark Folding and Suite Folding was evaluated training a classifier based on the CNN model. The classification performance and similarity metrics were used to investigate the best strategy to split the dataset to evaluate a DL model. The Stratified K-Fold proved to keep stable cross and self-similarities of the test and training-sets. We adopted it for the experimental results section (Section IV), where we compare the accuracy of state-of-the-art models.

We also analysed the meta-information dataset using two different techniques, a Decision Tree (DT) and a Multi-Layer Perceptron (MLP). While DT was generally more performant, integrating MLP with code analysis makes this methodology more attractive for our purposes. Using a normalisation pipeline, we were able to obtain good results also using MLP. Finally, we introduced the Siamese Network a new training paradigm that uses contrastive loss to learn by similarities between samples belonging to the same class. Using the new MLP and Meta-information normalisation procedure and a Siamese Network, we obtained a classification accuracy in the heterogeneous device mapping of 91.7% and 88.8% in the AMD and NVIDIA variants of the dataset, respectively. In future works, we will explore new techniques to improve classification accuracy by enhancing the language model and consider the dataset label incoherency. Moreover, we will explore new models able to deal with statistically different training and test-sets, like the one generated by the suite split.

References

[1] M. Allamanis *et al.*, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[2] Z. Wang *et al.*, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[3] F. Barchi *et al.*, "Exploration of convolutional neural network models for source code classification," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104075, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197620303353

[4] E. Parisi *et al.*, "Source code classification for energy efficiency in parallel ultra low-power microcontrollers," *arXiv preprint arXiv:2012.06836v1*, 2014.

[5] C. Cummins *et al.*, "End-to-end deep learning of optimization heuristics," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 2017, pp. 219–232.

[6] A. Brauckmann *et al.*, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 201–211. [Online]. Available: https://doi.org/10.1145/3377555.3377894

[7] D. Chicco, "Siamese neural networks: An overview," *Artificial Neural Networks*, pp. 73–94, 2021.

[8] T. Sharma *et al.*, "On the feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019.

[9] J.-W. Son *et al.*, "An application for plagiarized source code detection based on a parse tree kernel," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 8, pp. 1911–1918, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197613001085

[10] F. Barchi *et al.*, "Code mapping in heterogeneous platforms using deep learning and llvm-ir," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[11] T. Ben-Nun *et al.*, "Neural code comprehension: a learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, 2018, pp. 3585–3597.

[12] V. K. S *et al.*, "Ir2vec: A flow analysis based scalable infrastructure for program encodings," 2019.

[13] C. Cummins *et al.*, "Programl: Graph-based deep learning for program optimization and analysis," 2020.

[14] G. O. Ganfure *et al.*, "Deepprefetcher: A deep learning framework for data prefetching in flash storage devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3311–3322, 2020.

[15] S. Pagani *et al.*, "Machine learning for power, energy, and thermal management on multicore processors: A survey," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 101–116, 2018.
[16] A. H. Ashouri *et al.*, "A survey on compiler autotuning using machine learning," *arXiv preprint arXiv:1801.04405*, 2018.
[17] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[18] Z. Yang *et al.*, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in neural information processing systems*, 2019, pp. 5753–5763.
[19] T. B. Brown *et al.*, "Language models are few-shot learners," 2020.
[20] A. Vaswani *et al.*, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[21] A. Monsifrot *et al.*, "A machine learning approach to automatic production of compiler heuristics," in *International conference on artificial intelligence: methodology, systems, and applications*. Springer, 2002, pp. 41–50.
[22] Y. Jiang *et al.*, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 248–256.
[23] J. Cavazos *et al.*, "Rapidly selecting good compiler optimizations using performance counters," in *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE, 2007, pp. 185–197.
[24] A. H. Ashouri *et al.*, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, p. 21, 2016.
[25] M. Stephenson *et al.*, "Predicting unroll factors using supervised classification," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 123–134.
[26] E. Park *et al.*, "Predictive modeling in a polyhedral optimization space," *International journal of parallel programming*, vol. 41, no. 5, pp. 704–750, 2013.
[27] Y. Ding *et al.*, "Autotuning algorithmic choice for input sensitivity," in *ACM SIGPLAN Notices*, vol. 50. ACM, 2015, pp. 379–390.
[28] E. Park *et al.*, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 196–206.
[29] D. Grewe *et al.*, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.
[30] D. E. Rumelhart *et al.*, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[31] I. Goodfellow *et al.*, *Deep learning*. MIT press, 2016.
[32] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, vol. 19, pp. 143–155, 1989.
[33] W. Yin *et al.*, "Comparative study of cnn and rnn for natural language processing," *arXiv preprint arXiv:1702.01923*, 2017.
[34] B. Schölkopf *et al.*, "Toward causal representation learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 612–634, 2021.
[35] R. C. Rocha *et al.*, "Function merging by sequence alignment," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE.
[36] A. Siino *et al.*, "Data and commands communication protocol for neuromorphic platform configuration," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE, 2016, pp. 23–30.
[37] G. Urgese *et al.*, "Work-in-progress: Multiple alignment of packet sequences for efficient communication in a many-core neuromorphic system," in *2018 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. IEEE, 2018, pp. 1–2.
[38] F. Barchi *et al.*, "Flexible on-line reconfiguration of multi-core neuromorphic platforms," *IEEE Transactions on Emerging Topics in Computing*, 2019.
[39] S. B. Needleman *et al.*, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.
[40] G. Urgese *et al.*, "Benchmarking a many-core neuromorphic platform with an mpi-based dna sequence matching algorithm," *Electronics*.
[41] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.
[42] I.-K. Yeo *et al.*, "A new family of power transformations to improve normality or symmetry," *Biometrika*, 2000.
[43] D. Chicco *et al.*, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, 2020.

Emanuele Parisi is Ph.D. student at the Department of Electrical, Electronic, and Information Engineering at Alma Mater Studiorum - Università di Bologna. He received his M.Sc. degree (summa cum laude) in Computer Engineering - Embedded Systems, at Politecnico di Torino. His research activity mainly focuses on compiler optimizations for cyber-physical systems and low-power embedded systems.

Francesco Barchi is Research Assistant at the Department of Electrical, Electronic, and Information Engineering at Alma Mater Studiorum - Università di Bologna. He received his PhD in Computer Engineering at Politecnico di Torino. During his PhD, he worked on Spiking Neural Network mapping on the SpiNNaker neuromorphic platform and developed an optimised communication middleware and MPI implementation for the same architecture. Moreover, his studies focused on deep learning and compilers for heterogeneous architectures. His research interests focus on machine learning, compilers and optimisation problems for Cyber-Physical Systems (CPS) and heterogeneous architectures.

Andrea Bartolini received a Ph.D. degree in Electrical Engineering from the University of Bologna, Italy, in 2011. He is currently Assistant Professor in the Department of Electrical, Electronic and Information Engineering (DEI) at the University of Bologna. Before, he was Post-Doctoral researcher in the Integrated Systems Laboratory at ETH Zurich. Since 2007 Dr. Bartolini has published more than 80 papers in peerreviewed international journals and conferences with focus on dynamic resource management for embedded and HPC systems.

Andrea Acquaviva is Full Professor at the Department of Electrical, Electronic, and Information Engineering at Alma Mater Studiorum - Università di Bologna. He received the Ph.D. degree in electrical engineering from the University of Bologna, Italy, in 2003. In 2003, he became an Assistant Professor with the Computer Science Department, University of Urbino, Italy. From 2005 to 2007, he was a Visiting Researcher with the Ecole Polytechnique Federale de Lausanne, Switzerland. In 2006, he joined the Department of Computer Science, University of Verona, Italy. He has been with the Department of Computer Engineering and Automation, Politecnico di Torino. His research interests focus mainly on parallel computing for distributed embedded systems such as multi-core and sensor networks and simulation and analysis of biological systems using parallel architectures. In the fields above, he has authored over 140 scientific publications.