Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR

Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR

Francesco Barchi, Gianvito Urgese
DAUIN Dept.
Politecnico di Torino, Italy
{firstname.lastname}@polito.it

Enrico Macii, Andrea Acquaviva
DIST Dept.
Politecnico di Torino, Italy
{firstname.lastname}@polito.it

ABSTRACT

Modern heterogeneous platforms require compilers capable of choosing the appropriate device for the execution of program portions. This paper presents a machine learning method designed for supporting mapping decisions through the analysis of the program source code represented in LLVM assembly language (IR) for exploiting the advantages offered by this generalised and optimised representation. To evaluate our solution, we trained an LSTM neural network on OpenCL kernels compiled in LLVM-IR and processed with our tokenizer capable of filtering less-informative tokens. We tested the network that reaches an accuracy of 85% in distinguishing the best computational unit.

Keywords

LLVM-IR; Deep Learning; Code Mapping; Heterogeneous Platforms; Embedded Platforms

1. INTRODUCTION

Currently, the process of code mapping on the hardware units available on heterogeneous multi-core embedded system is one of the main challenges for a software developer. Despite the availability of machine independent languages (like LLVM [**?**]) and interfaces supporting the offloading to selected accelerators (such GPUs, DSPs and FPGA) the execution of code fragments (kernels) [**?**, **?**, **?**] a consistent amount of research is still in progress for defining automatic mapping techniques aimed at improving the available computational power and avoiding the effort of manual profiling.

While methods have been developed performing machine learning based code analysis for high-level languages such as OpenCL [**?**], the potential of LLVM has not been fully exploited to this purpose. At this level, the code has undergone a preliminary optimisation pass during high-level code transformation, such as the removal of unnecessary elements. From the other side, concerning machine dependent assembly code, LLVM representation is more hardware independent. Finally, LLVM-IR is a general representation that can be reached from different high-level languages. For this reason, developing a code classifier at this level would be more generally applicable and robust. However, the possibility to keep the same expressiveness as for the higher level language is questionable.

In this work, we addressed this issue, and we designed a method able to identify, select, and encode the syntactic language elements (tokens) of a source code of a kernel compiled in the LLVM - Intermediate Representation (LLVM-IR).

We used the generated sequence of tokens as input for training a Deep Neural Network (DNN) in recognising which is the most appropriate architectural component for each piece of code evaluated.

We trained our network using a dataset of OpenCL kernels profiled on the kernel execution time executed on CPU and GPU [**?**]. Then, we tested the kernel-to-device allocation performance demonstrating that our LLVM-based classifier achieves an accuracy of 85% in selecting the best kernel allocation. We compared our approach based on LLVM-IR with the state-of-art solution based on native source code (e.g. OpenCL) showing that our solution produces a more accurate mapping, with the advantage of working on a layer decoupled from the code-language.

Our results show that LLVM-IR keeps the informative content needed to perform an effective classification making possible the application of our classifier to any source code for which an LLVM compiler exists.

2. RELATED WORKS

Recently, compiler designers started considering the adoption of machine learning techniques to obtain heuristic compilers capable of learning from the data rather than relying on experience and manual effort [**?**, **?**]. These techniques come into play to cope with the complexity of the code allocation and optimisation for heterogeneous multicore embedded systems.

In this paper, we work 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

ACM ISBN xxx.
DOI: xxx

LLVM-IR Code Fragment

```
%9 = and i64 %8, 4294967295
%10 = getelementptr inbounds <4 x float>, <4 x ↩
    float>* %1, i64 %9
%12 = fsub <4 x float> <float 1.0e+00, float 1.0e↩
    +00, float 1.0e+00, float 1.0e+00>, %11
%13 = fmul <4 x float> %11, <float 3.0e+01, float ↩
    3.0e+01, float 3.0e+01, float 3.0e+01>
```

Tokenization

```
% 9 = and i64 % 8 , _integer_constant
% 1 0 = getelementptr inbounds _float_4 , _float_4 ↩
    * % 1 , i64 % 9
% 1 2 = fsub _float_4 _vector_constant , % 1 1
% 1 3 = fmul _float_4 % 1 1 , _vector_constant
```

Atomization

```
10 9 11 13 14 10 8 12 15
10 1 0 11 16 17 18 12 18 19 10 1 12 14 10 9
10 1 2 11 20 18 21 12 10 1 1
10 1 3 11 22 18 10 1 1 12 21
```

Figure 1: **Example of code transformations:** The code in the top pane is an LLVM-IR code fragment. The code in the middle pane contains the result of the transformations applied in the tokenisation phases. The tokens sequence in the bottom pane is the network input, the result obtained after the atomization phase.

fragments. Several techniques 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. 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 their machine learning algorithms. These include static code structures extracted from the source code or the compiler intermediate representation [**?**] and dynamic profiling information obtained through runtime profiling of the program execution [**?**]. Compiler optimisation methods based on supervised learning have been proposed using Bayesian Networks [**?**], Support Vector Machines [**?**,**?**], Decision Trees [**?**,**?**], Graph Kernels [**?**], and Deep Neural Network [**?**].

In this research area, we designed a method that applies a deep learning approach to the LLVM intermediate representation of code fragments making the methodology suitable for a wide range of languages.

3. METHOD

The *Low-Level Virtual Machine* (LLVM) is a compilation framework that allows decoupling a programming language from the target architecture. Identifying features within a *LLVM-IR* program allows decoupling the classifier from the programming language. We chose to build an LLVM-IR machine learning algorithm using a supervised learning method belonging to the deep learning category. Given the availability of code written in several high-level languages (C, C++, OpenCL, Python), we can easily obtain a dataset of code fragments compiled in LLVM-IR. Then, we analyse the LLVM-IR code and filter the most significant syntactic elements (tokens). Reducing the tokens to a sequential list of integers we can bring them in a deep neural network (composed of LSTM layers) for performing the code analysis. In the following sections, we will give details about the code conversion and the structure of the deep neural network.

3.1 Tokenisation and Atomisation

The first step for using deep-learning techniques is the conversion of the code-fragments dataset in a form suitable to be processed by the input layer of the machine-learning model.

To build a dataset of code fragments in LLVM-IR we can use any dataset written in a high-level language that has a front end compiler for LLVM. The *clang* compiler, for example, allows compiling the main C-Like languages (C, C++, Objective C). Using version 6 of clang and llvm is also possible to compile OpenCL code for *nvptx* (NVidia), *amdgcn* (AMD) and *spir* (Standard Portable Intermediate Representation) architectures. The generated bytecode can then be sent to the OpenCL Platform Runtime and executed.

The LLVM-IR code obtained after the compilation is rich in metadata useful for the back-end compiler. It is necessary to clean and normalise the code before inserting the code fragments in the neural network. During this phase, our pipeline removes many of the redundant information by transforming the kernels from a sequence of characters to a series of integers.

The ***Tokenisation*** procedure identifies the most significant language syntactic elements (tokens) within the character sequences. All the tokens are catalogued and placed in a dictionary. The ***Atomisation*** procedure transforms code sequences replacing the characters that compose a token with the position of the token in the dictionary.

We implemented the *Tokenisation* procedure in two steps: the *pre-tokenisation* phase and the *post-tokenisation* phase. The *pre-tokenisation* phase act on each line of a kernel and performs the following four operations:

- Remove empty lines, comments and extract functions body.
- Simplify vectors and arrays data-types.
- Replace vectors, arrays, float constants with a placeholder.
- Expand the symbols " ( ) [ ] { } < > = * : , " by inserting a space before and after the character.

During this phase, the procedure makes a significant reduction of the code fragment length, through the simplification of complex data types and by replacing constants with placeholders. In LLVM, for example, real constants can be expressed in 3 different ways: i) standard decimal notation (e.g. 123.421), ii) exponential notation (e.g. 1.23421e+2), or a iii) hexadecimal notation (e.g. 0x432ff973cafa8000). Each of these representations is identified and replaced with a placeholder. After this step, it is possible to identify as a token every sequence of characters separated by spaces.

The *post-tokenisation* transformations act directly on the tokens for applying the following higher level generalisations:

- Remove tokens starting with ! (unnamed meta-data) or # (attribute groups).
- Replace variable and function names with a placeholder.
- Identification of some special labels starting with "phi", "pre", "in", "preheader" and "loopexit".
- Identification and transformation of integer constants.
- Identification and character expansion of global and local unnamed identifiers (e.g. %54 → % ␣ 5 ␣ 4)

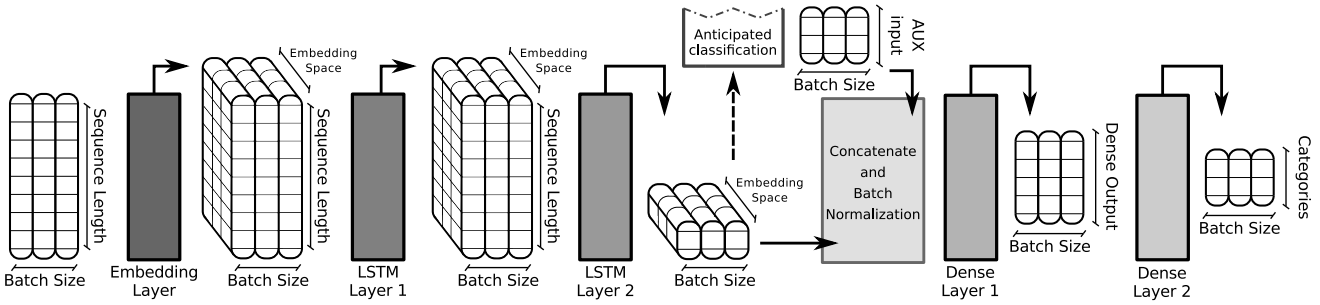

Figure 2: Deep Neural Network architecture. The data flow (tensors exchanged between the network layers) was highlighted.

Then, in the *Atomisation* step, all the tokens are replaced with their integer indexes of the token dictionary forming a sequence of integers to be used as input for the network.

An example of the results of code transformation can be appreciated in Figure 1. In the literature, the performance of a text-based deep-learning system (e.g. in the sentimental analysis) depends heavily on the dictionary chosen to transform the input into a numerical sequence [**?**].

The technique used to build the dictionary impatcs the length of the resulting sequences. Long sequences require complex models capable of storing and correlating information for more extended periods, such models need to train many weights; this is feasible only using a high number of training samples that are not always available. In the Natural Language Processing (NLP) field is a common practice to remove less-informative tokens called *Stop Words* [**?**]. Section 4 shows how, by carefully selecting the tokens to be inserted in the dictionary, it is possible to filter out less-informative tokens for keeping shorter sequences and increase the classification capability of the model.

3.2 Deep Learning Model

Figure 2 shows the network architecture used in this work. It is based on a standard network able to manage multiple and heterogeneous inputs [**?**], and it is composed of six principal layers:

- Embedding Layer: This layer takes a sequence of η symbols and returns a sequence of η vectors, each κ elements long.
- RNN Layer 1: This first LSTM layer transforms the sequence coming from the Embedded Layer in another sequence that correlates the symbols between them.
- RNN Layer 2: This second LSTM layer performs a second correlation among the symbols leaving the previous layer. The output of this layer is a single vector of κ elements. It represents the whole sequence.
- Concatenation Layer: The auxiliary inputs and the LSTM output are concatenated in a single vector.
- Dense Layer 1: This layer, with a RELU activation function, transforms the auxiliary inputs and the LSTM output into a vector of υ elements.
- Dense Layer 2: The last dense layer, with a sigmoidal activation function, transforms the output of the previous layer in a vector with ω elements (the number of the target classes).

At each iteration of training, the sequences that are part of the training set can be introduced into the network one at a time or in groups (in this case we speak of batch insertion). After the introduction of the input, the output emitted by the network is compared with the expected output (supervised learning). The difference between the network output and the expected output is the classification error. Through the classification error, it is possible to modify the weights of the network and to reduce the error at the next iteration. The training was performed using the optimiser Adam (Adaptive moment estimation) a variant of the SGD (Stochastic Gradient Descent) that through the backpropagation modifies the weights of the network layers and minimises the classification error [**?**].

The network input is a *tensor* composed of *batch-size* sequences each one composed of *sequence-length* elements. We will refer to the input elements with the term *token-indexes*, since each component represents the position of a token inside the token dictionary.

Machine learning algorithms need to work on comparable data, and since the token indexes do not have this property because we cannot define a distance metric between two indices, the sequence of *token-indexes* must be projected into a metric space.

Embedding Layer is the first layer of the network that receives sequences of *token-indexes* and projects each element into an *embedding space* $\mathbb{R}^{\kappa}$. The output of the Embedding Layer is, therefore, a list of sequences each one composed of vectors belonging to the *embedding space*. The weights of the Embedding Layer determine how the *token-indexes* are projected in the *embedding space*. At the beginning of the training, the weights are initialised randomly with the consequence that the projection of the *embedding space* starts in a random condition. They change during the training phase of the network.

The sequence in output from the *embedding space*, the *token-points*, is now ready to be introduced into the first LSTM layer. The LSTM is a neural network with memory, which makes it suitable for processing data sequences [**?**]. The *token-points* are introduced within the layer one by one (the entire batch is used to maximise the efficiency of the GPU) producing a change in the internal state of the network. The internal state of the network is used to process the next *token-point*. The sequence in output from the LSTM layer is recomposed element by element, and once the last *token-point* has been received the resulting sequence is ready to be introduced in the second LSTM layer.

The second LSTM layer performs the same procedure as the first one, but it does not recompose the sequence into the output. At the end of the processing, when the last *token-point* is issued, it is used to represent the features extracted from the code sequence. The features will first be linked with μ auxiliary inputs (used to introduce others data) and then classified with a sequence of two *Dense Layers*.

The first Dense Layer reduces the number of features from

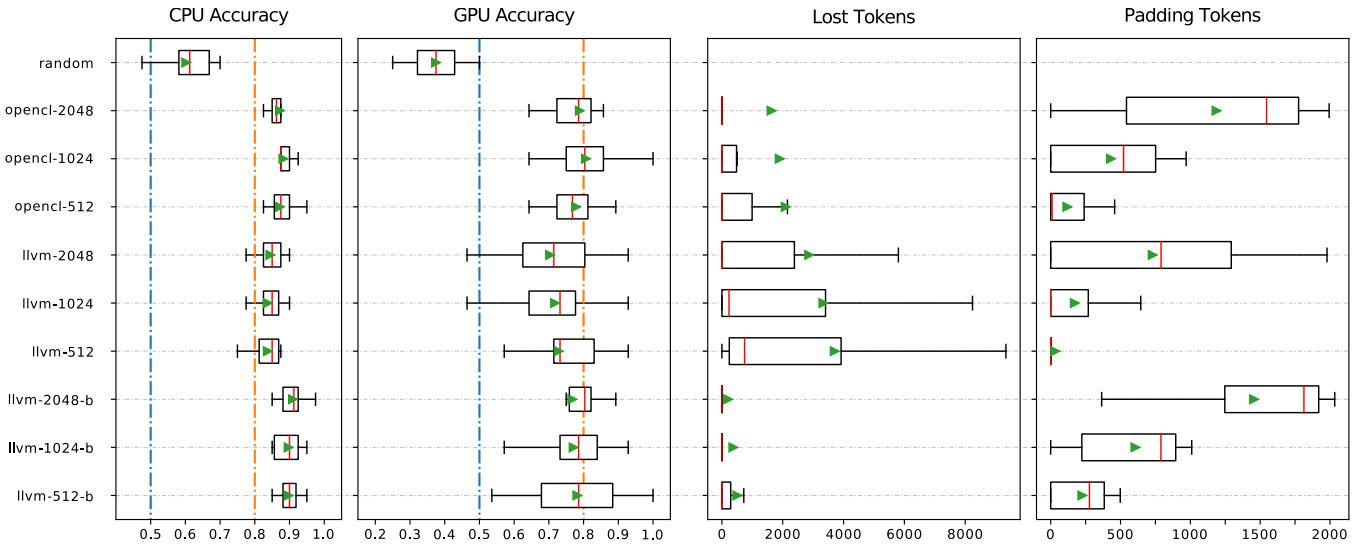

Figure 3: For each combination of datasets (OpenCL, LLVM), sequence lengths (2048, 1024, 512) and token-blacklist (used or not used): the first two box-plots show the distribution of the classification accuracy of the ten classifiers in cross-validation, the last two box-plots show the distribution of the lost-tokens (truncation) or added-tokens (padding) in the classifier input sequences.

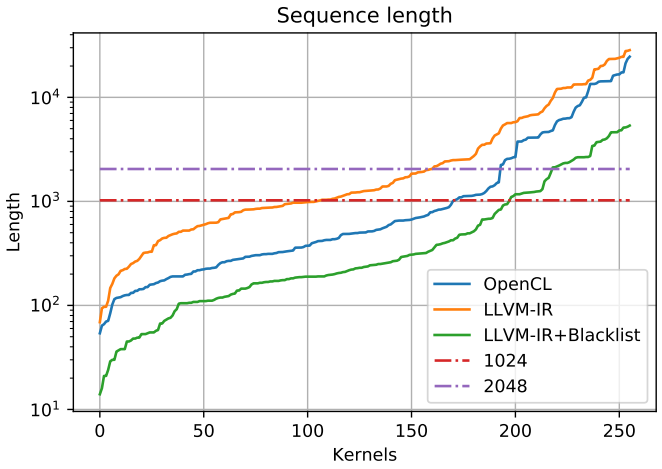

Figure 4: Length of code sequences in the three datasets: OpenCL, LLVM and LLVM with blacklist.

$\kappa + \mu$ to υ and the second layer brings them further to ω, the number of desired classes.

In this work we use: $\kappa = 64$, $\mu = 2$, $\upsilon = 32$ and $\omega = 2$ since each element of the final output represents one of the two target devices considered: CPU and GPU.

4. RESULTS

Since an adequately labelled dataset is needed to build a classifier using deep learning we have used a dataset composed of OpenCL kernels where each element has a label that denotes the better performing computation unit between a multicore CPU (AMD) and a GPU (ATI or NVidia). The dataset is a composition of six collections of code [**?**]: i) AMD and NVidia OpenCL examples and benchmarks ii) NPB, the NASA Advanced Supercomputing Parallel Benchmarks iii) Parboil, computing applications for studying the performance of computing architecture and compilers. iv) PolyBench/GPU v) Rodinia, the University of Virginia Rodinia benchmark suite vi) SHOC, Scalable HeterOgeneous Computing benchmark suite. The authors of the dataset executed each kernel belonging to the code collection using different loads (*byte transfer*) and a different level of parallelism (*work group size*).

Each triple composed by *kernel*, *byte transfer* and *work group size* was labelled with the computing unit (CPU or GPU) which proved to be the fastest. The full dataset is composed of 680 triple, and 256 different kernels with an unbalance factor of about 60/40 in favour of the CPU class.

Using *clang* and *llvm* we have compiled in LLVM-IR all the kernels of the dataset. The *clang* compiler can emit LLVM-IR code using the *"-emit-llvm"* parameter, set the desired OpenCL version using *"-cl-std=CL2.0"* and import OpenCL headers using the *"-Xclang -finclude-default-header"* parameters. The compilation of some kernels ended with the presence of errors. We then proceeded to manually fix the broken OpenCL kernels and use the entire dataset.

The construction of the token dictionary can be done in three ways: i) Using a *pure-character dictionary*, it considers only the characters with the advantage of avoiding complex analysis for token construction, but it can be used mainly for short sequences. ii) Using a *hybrid dictionary*, it allows a reduction of the length of the sequences by encoding the most common words with a single symbol and processing as single characters all the letters not recognised as dictionary words. iii) Using a *pure-token dictionary*, it allows a substantial reduction of the length of the sequences through a transformation of the code where complex syntax artefacts are encoded with single symbols.

In this work, we used the *pure-token dictionary* because LLVM-IR sequences tend to be much longer than their counterparts in a C-like language. The use of a hybrid dictionary would have required a more extensive dataset for feeding a much more complex network (concerning trainable weights).

Even by simplifying the LLVM code through the pre and post tokenisation phases and using a pure-token dictionary,

Table 1: Comparison of results.

	Seq.Len.	Median	Average
DeepTune [**?**]	1024	80.9%	82.2%
OpenCL	2048	83.1%	83.8%
	1024	84.6%	85.1%
	512	83.1%	83.4%
LLVM	2048	78.7%	78.7%
	1024	80.9%	78.8%
	512	80.1%	79.3%
LLVM-B	2048	86.0%	85.1%
	1024	86.8%	84.6%
	512	85.3%	85.0%

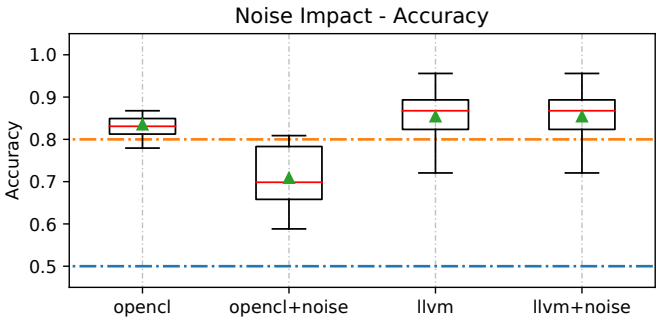

Figure 5: The graph show the classification accuracy distribution of ten classifiers in cross-validation for OpenCL and LLVM dataset in presence of noise.

the sequences were too long to be correctly analysed by the network (Figure 4). It was, therefore, necessary to generate a *token-blacklist* for removing redundant tokens (like the *stop words* in NLP) keeping only the most significant. We used a *Term Frequency - Inverse Document Frequency* (*TF-IDF*) analysis for eliminating tokens that did not provide a high level of information [**?**]. The Figure 4 shows the length of the kernels in OpenCL (hybrid dictionary) that are always shorter than LLVM (pure-token dictionary) while kernels elaborated with the LLVM-B (pure-token dictionary with blacklist) reach a reasonable size.

We built and trained our network using Tensorflow, configured with LSTM layers optimised for GPU execution (via the CuDNN library). The optimised LSTM layers allow a substantial time reduction (from 10 hours to 50 minutes) of the training time using a GeForce Titan Xp (NVidia Pascal Architecture). The reduced training time allowed us to experiment with some parameters including the batch-size (affects the Optimiser) and the length of the sequences (trade-off between information loss and network simplicity). The length of the sequences, in particular, has proved to be a critical factor because long sequences require more complex models and large datasets to be adequately processed. Moreover, the presence of very long and very short sequences within the dataset forces the user to add padding to the short sequences and eliminate elements from the longer sequences. It is possible to configure variable batch sizes (n, 2n, 3n) and to insert the sequences into the appropriate batches for minimising both padding and truncation. This method overcomes the problem but, for small datasets, it has the disadvantage of preventing the introduction of sequences into the network in a random-way (the reshuffling is performed within each batch and not on the entire training set) and carries the risk of introducing biases during training. Given the small dataset: train-set of 612 elements and 256 different kernels, we chose fixed size sequences of 512, 1024 and 2048 elements. Then we performed k-fold cross-validation with k=10 to have a better statistical basis of the values of classification accuracy. The Figure 4 highlight the lengths (2048, 1024) of the sequences introduced into the neural network during the experiments. The line-plot highlight the amount of padding and the number of sequences to be truncated for each dataset used.

We trained the network using three different datasets and three different sequence lengths. The first two box plots in Figure 3 show the accuracy distributions for each class (CPU, GPU) of the ten classifiers that were built to perform the cross-validation. We trained the model also using a dataset in which we replaced the code sequences with a random sequence (only the contribution of the auxiliary input remains). We immediately notice the contribution of the code sequences compared to the random sequences and the difference in accuracy between the CPU and GPU classes due mainly to the unbalance of the dataset.

In the third and fourth box-plots of Figure 3 we can note the distributions of the number of padding tokens and tokens deleted for each dataset. The LLVM sequences are too long, and they are strongly disadvantaged by the truncation of more than 60% of the kernels. The introduction of the token blacklist has drastically reduced the length of the sequences, and the accuracy of the classifier has returned to the levels of the OpenCL dataset, with an improvement in the classification of the CPU class while maintaining, on average, the same levels of accuracy in the GPU class. The higher variance in GPU class accuracy disappears as the sequence length increases (llvm-b-2048). This behaviour is an indication that LLVM input is more difficult to be classified than OpenCL code. The difficulty depends on the highly rigid structure of an assembly-like language that requires a longer-term memory of the LSTM layers as the information is distributed over more extended sequences. The modest unbalance of the dataset contributes to creating difficulties for the classification of the disadvantaged class.

Table 1 shows the average results of each classifier in which we can see the improvement over DeepTune [**?**], and the growth of performances of the LLVM classifier when the TF-IDF token-blacklist is applied. The LLVM-B slightly exceeded the performance of the OpenCL classifier.

The main advantage of using LLVM-IR is that it is built after a code compilation and multiple optimisation phases. The code is analysed and cleaned from unused variables and unreachable code fragments, the cycles are examined and simplified, and other optimisations are performed that generally allow reducing the variability of code (noise) introduced by the programmer compared to the database used in the learning phase.

To highlight the impact on classification accuracy we introduced unreachable code in both OpenCL and LLVM test sets. Results are reported in Figure 5 where the LLVM classifier is not affected by the introduction of the noise code-fragment (the compilation step eliminates the noise introduced) while the classifier using a high-level language degrades its performance by about 10%.

5. CONCLUSIONS

In this work, we presented a framework designed for evaluating the performance of an LLVM-IR source code classifier based on deep neural networks. The possibility to apply a deep-learning methodology directly on the intermediate representation used by LLVM allows building a more robust and generally applicable (more supported programming languages) source code classifier.

Using an LLVM compiler, we obtained a general and optimised low-level representation (IR) of a source code written in a high-level programming language. At the LLVM-IR level, the code can be manipulated and filtered for condensing complex syntactic language elements in a restricted set of keywords (tokenisation procedure) that once translated in sequences of numbers (atomization procedure) are suitable for being used as input for the deep neural network classifier.

We evaluated the performances of our LLVM-based classifier using a dataset of OpenCL kernels properly manipulated with our tokenisation - atomization strategy. Furthermore, through a TF-IDF weight analysis, we remove less informative tokens thus reducing the input dimension and being able to obtain an accuracy of the classifier with a median value of 86% (5% better compared to 81% achieved by the state-of-the-art code classifier DeepTune, Table 1).

These results show that, while analysing LLVM code requires an additional effort than working on the high-level language, it allows achieving a more general and more robust classifier.

In future works, we will investigate deeper network models that require more massive datasets.