Journal of
Low Power Electronics and Applications

MDPI

Review

Deep Learning Approaches to Source Code Analysis for Optimization of Heterogeneous Systems: Recent Results, Challenges and Opportunities

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

Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi" (DEI), Università di Bologna, Via Zamboni 33, 40126 Bologna, Italy; emanuele.parisi@unibo.it (E.P.); a.bartolini@unibo.it (A.B.); andrea.acquaviva@unibo.it (A.A.)
* Correspondence: francesco.barchi@unibo.it

Abstract: To cope with the increasing complexity of digital systems programming, deep learning techniques have recently been proposed to enhance software deployment by analysing source code for different purposes, ranging from performance and energy improvement to debugging and security assessment. As embedded platforms for cyber-physical systems are characterised by increasing heterogeneity and parallelism, one of the most challenging and specific problems is efficiently allocating computational kernels to available hardware resources. In this field, deep learning applied to source code can be a key enabler to face this complexity. However, due to the rapid development of such techniques, it is not easy to understand which of those are suitable and most promising for this class of systems. For this purpose, we discuss recent developments in deep learning for source code analysis, and focus on techniques for kernel mapping on heterogeneous platforms, highlighting recent results, challenges and opportunities for their applications to cyber-physical systems.

Keywords: cyber-physical systems; heterogeneous device mapping; source code analysis; system optimisation; literature review

Citation: Barchi, F.; Parisi, E.; Bartolini, A.; Acquaviva A. Deep Learning Approaches to Source Code Analysis for Optimization of Heterogeneous Systems: Recent Results, Challenges and Opportunities. *J. Low Power Electron. Appl.* **2022**, *12*, 37. https://doi.org/10.3390/jlpea12030037

Academic Editors: Andreas Peter Burg and Minsu Choi

Received: 4 September 2021
Accepted: 26 May 2022
Published: 5 July 2022

1. Introduction

Cyber-physical system development is restrained by its inherent heterogeneity. Therefore, system-level designers focus on formal methodologies and tools to compose heterogeneous computing resources while achieving system-level goals (performance, energy, cost) [1]. It is then the role of application developers to map the application code to the heterogeneous hardware resources. However, not all the application kernels can run effectively in all the computing architectures [2]. Specialised computing architectures are the most applicable tool left to obtain performance and energy-efficiency growth since Moore's Law is losing steam and the Dennard's scaling has ended [3]. Graphical processor units (GPUs) leverage a large number of simpler cores that share their control units, smaller-sized caches, and lower frequency than CPUs to attain higher efficiency and peak performance in throughput-critical applications. On the contrary, CPUs are more suited to latency-critical applications due to their out-of-order, multi-instruction issue and higher frequency cores. Programming heterogeneous computing systems effectively requires a deep understanding of the static and dynamic characteristics of application kernels. Indeed, application kernels where data transfers dominate execution time or branch divergence do not allow for the uninterrupted execution on all GPU cores, achieving higher performance on CPUs than GPUs [2]. In the domain of CPS, it is thus important to provide the application expert with tools for extracting source code information and easing the challenges of heterogeneous system programming.

For years the development of source code for CPS and its analysis have been activities relegated to domain experts. The difficulty in creating an automated system of analysis and understanding code lies primarily in the structural properties of source code, making

it difficult to process and interpret through simple algorithms. The same difficulties, but at the same time structurally different, were found in other fields such as Computer Vision, Natural Language Processing, and Speech Recognition [4].

In recent years, it has been possible to successfully address and solve several problems (such as identification, reconstruction and generation of data) in these areas thanks to a family of statistical methods capable of learning and extracting information from large amounts of data. We are talking about methods that use Machine Learning (ML) algorithms or, in some applications, Deep Learning (DL). Deep Learning techniques belong to the set of Machine Learning practices [5]. Still, they are characterised by specific properties such as a highly-stratified structure.

The research activity on these learning techniques is still ongoing. Every year, new critiques and improvements to the models are proposed. Within this lively research, the scientific community sees the possibility to exploit these learning techniques to address and solve problems involving source code [6].

It is beyond the scope of this paper to propose a comprehensive survey on the topic of machine learning for code. Surveys on this field, which we explore in Section 2, provide a good way to formalise the problem of source code analysis in a plethora of different application categories [6–8]. The works mentioned above provide a taxonomy to frame the applications and techniques present in the literature, but they do not delve into technical detail and do not provide direct comparisons between the articles they cite. This type of comparison is not feasible in an exploratory context as broad as in the reference surveys. However, they do not target specifically heterogeneous device mapping nor cyber-physical systems. In this paper, we describe the State-of-the-Art (SoA) tools for heterogeneous device mapping (Section 2), and we additionally provide critical reviews and comparisons (Section 4) of available tools to assess their impact on CPS programming. This paper stems from the research questions that drove our works on this topic [9,10]. As a consequence, we pose to the reader the following research questions (RQs) to guide the narrative path of the article:

RQ1 How, historically, have learning techniques been applied to source code?

RQ2 How have machine learning techniques been applied to cyber-physical systems and, in particular, to heterogeneous device mapping?

RQ3 How do the machine learning methods analyse source code in heterogeneous device mapping, and what results have been obtained?

The methodology we applied to answer these questions is composed of three steps, following the paper organisation: Section 2 presents a list of selected papers on source code analysis techniques. In this section, we explore works dealing with ML and DL techniques applied to source code to address the RQ1 and provide a comprehensive overview of the problem for the reader. We report a collection of the most pioneering scientific papers on the topic, and we analyse them along with different directions: (i) Techniques used to analyse the code, (ii) Levels of abstraction at which the analysis is performed and (iii) Problems addressed. Section 3 addresses the RQ2, focusing on a specific problem in the CPS field, heterogeneous device mapping, and allowing the reader to quickly identify the articles that first introduced techniques or first addressed a specific problem in this field. Section 4 addresses the RQ3, describes the problem of heterogeneous device mapping and exposes state-of-the-art works along with a critical review and a comparison of the presented methodologies.

We focus on a restricted set of papers facing the heterogeneous code mapping problem, which is essential to ease the developer burden in CPS. Heterogeneous code mapping is about selecting code fragments (kernels) and deciding the most appropriate execution target. By the term “most appropriate” we mean the execution target that maximises or minimises a metric of interest to the developer, such as execution time or energy consumed by the code fragment. Sections 5 and 6 report current challenges in the field and future

directions to improve available techniques or develop new ones. We report a critical view of current challenges and future directions.

Overall, this work aims to help researchers in this field boost the design and implementation of intelligent software development tools capable of making complex decisions for CPS architectures.

2. Research Timeline

The topics we deal with in this work belong to a relatively young research branch involving machine learning and source code. The field of source code analysis has embraced deep learning techniques, and we want to present a historical and rapid review of research works in this field with, firstly, the aim of answering the RQ1: "How, historically, have learning techniques been applied to source code?". Furthermore, we want to give the reader an overview and historical perception of the improvements made in this research area to better understand the aspects presented in the next section.

2.1. Selection Criteria

In this research timeline, we only considered articles published after the work of Hindle et al. "On the naturalness of software" [11] which we consider to be the seminal work for the whole research field. The main source from which we selected the articles of interest is a corpus of about 390 papers called ml4code [12] collected and selected by the community for the initiative of Allamanis M. [6] and a corpus of 180 other papers obtained using a direct keywords search on Scopus, we will refer to this corpus by the name Corpus 0.

The first phase of the selection process was the following: we defined a set of keywords belonging to machine learning and deep learning techniques. The keywords are listed in the third column of Table 1, and the complete list is reported in Figure 1. For each keyword, we selected only the papers that use it in their title or abstract. In this first phase, we reduced the corpus to 94 documents. We will refer to this corpus by the name of Corpus 1.

Then, we identified pioneering articles as those proposing a new machine-learning technique or tackling a new problem. To achieve this, we selected papers from Corpus 1 using qualitative and quantitative metrics. Regarding the qualitative metrics, the selection criteria was the relevance of techniques used and the problems addressed. After this review, among the resulting papers, we selected the ones referred to as seminal works by more recent papers on the same subject (in our case, the source code analysis using deep learning). These are valid candidates to be labelled as pioneering works as they introduced the application of a new approach or solved a problem for the first time.

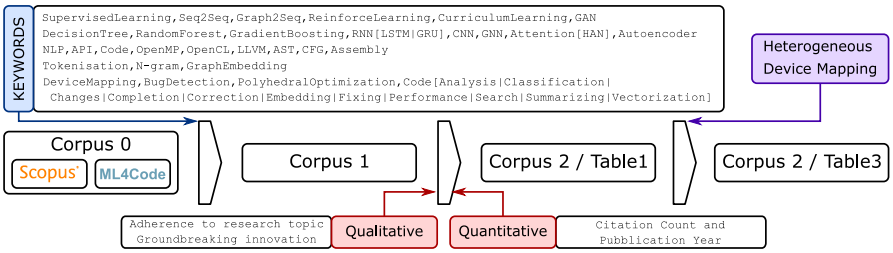

Figure 1. The paper selection pipeline. A Corpus is a list of papers. Corpus 0 is filtered using keywords to obtain Corpus 1. Corpus 1 was selected using both qualitative and quantitative metrics. Results are shown in Table 1. From the line of research started in [13], we selected papers focused on heterogeneous device mapping. We listed them in Table 3.

Table 1. List of selected papers for an in-depth discussion on source code analysis methods. The first part of the table contains the main surveys on the topic. The second part of the table contains the most representative works on source code manipulation and representation.

Reference		Year	Keywords
Key papers and Surveys			
Hindle A. et al.	[11]	2012 *	N-gram, Code properties, Code completion
Allamanis M. et al.	[6]	2018	Survey, Code properties
Ashouri AH. et al.	[7]	2018	Survey, Compiler autotuning
Wang Z. et al.	[8]	2018	Survey, Compiler optimization
Innovative Models, Applications and Techniques			
Grewe D. et al.	[13]	2013	DT, Device mapping
Raychev V. et al.	[14]	2014	RNN, N-gram, Code completion
Zaremba W. et al.	[15]	2014	Seq2Seq, LSTM, Curriculum Learning
Iyer S. et al.	[16]	2016	LSTM, Summarizing, Attention
Bhatia S. et al.	[17]	2016	RNN, LSTM, Seq2Seq, Code fixing
Mou L. et al.	[18]	2016	Tree CNN, AST, Code classification
Gu X. et al.	[19]	2016	Seq2Seq, RNN Encoder-Decoder, NLP to code API
Allamanis M. et al.	[20]	2017	GGNN, GRU, Code analysis, Graph2Seq
Gu X. et al.	[21]	2018	RNN, Code Search, Cosine Similarity, NLP to Code Example
Santos ED. et al.	[22]	2018	LSTM, N-gram, Code Correction
Bavishi R. et al.	[23]	2018	LSTM Autoencoder, Code Analysis
Bui NDQ. et al.	[24]	2018	Tree CNN, AST, Code classification
Alon U. et al.	[25]	2019	AST Paths, Code Classification, Attention
Alon U. et al.	[26]	2019	AST Paths, Code Classification, LSTM Encoder-Decoder
Mendis C. et al.	[27]	2019	Hierarchical LSTM, Code Performance Regression, Assembly
Pradel M. et al.	[28]	2020	AST features, RNNs, Type prediction
Hoang T. et al.	[29]	2020	HAN, Bidirectional GRU, attentions, Code changes
Karampatsis RM. et al.	[30]	2020	Code embedding, BPE, LSTM
Haj-Ali A. et al.	[31]	2020	RL, Code vectorization
Brauckmann A. et al.	[32]	2021	RL, Polyhedral optimisation
Allamanis M. et al.	[33]	2021	GNN, Bug detection, GAN

* Published for Journal in 2016.

We also cross-checked the selection results using quantitative metrics. These quantitative metrics are publication year and citation count. Seminal papers should be earlier in terms of publication year and must be consistently cited by the following papers. It must be noted that the selection procedure makes use of our prior knowledge in this area of research, and it is not a mere application of thresholds on selected metrics. The selected papers, that compose the Corpus 2, will be discussed in Section 2.3. The papers are listed in Tables 1 and 3.

This research timeline is divided into two parts: Relevant related surveys and historical overview of ML and DL techniques for source code. In the first part, Section 2.2, we present an article that we consider the seminal work for the whole research field on source code analysis [11] and three most cited surveys on the subject [6–8]. We will therefore use this first part to outline the contours of the topic and summarise the fundamental concepts proposed by the authors in their works.

In the second part, Section 2.3, we describe in detail the most important techniques and intuitions behind models mentioned by previous surveys. We chose to follow a historical timeline useful to emphasise the research improvements over the years and highlight the insights behind the enhanced performance of recent code analysis methods. We describe the selected works and their categorisation alongside topic keywords. The description is a summary of the work with the aim to highlight three research aspects: (i) The abstraction level for source code analysis (e.g., high-level source code, intermediate representation, assembly); (ii) The code representation technique (e.g., token-based stream, abstract syntax tree, control flow graph); (iii) The problem against which the proposed method has been tested.

2.2. Relevant Related Surveys

The following surveys are useful to depict and explain the research field of machine learning on code. They help to identify how scientific research has evolved over the years and the field's key challenges. The most cited paper in the field that we consider a seminal work is "On the Naturalness of Software" [11], which, to the best of our knowledge, is one of the first approaches to investigate the source code properties in terms of statistical analysis. The authors view the source code as an act of communication and try to answer the following questions:

> *Is it [source code, author's note] driven by the "language instinct"? Do we program as we speak? Is our code largely simple, repetitive, and predictable? Is code natural? ... Programming languages, in theory, are complex, flexible and powerful, but, "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive; thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks [11].*

Authors suggest that source code presents repetitiveness at multiple levels (lexical, syntactic, semantic). This property can be exploited to improve a wide range of applications they list in their article. Along with the paper, the authors prove this claim and report several examples of application problems. They start by introducing the n-gram model, a statistic-based language model. The goal of the model is to estimate the probability of a token (the smaller part of a string with a lexical meaning) to be present after a sequence of n other tokens using conditional probability. Cross-entropy can be used to measure the goodness of a language model. The main problem of this approach is the high number of coefficients to learn, which grows exponentially.

More formally, the n-gram model can be summarised using the Equation (1): we have a model M that gives the N-gram probability $\widetilde{p}_{M,N}(s)$ to a document s. The document consists of a sequence of n tokens a_i and has an entropy $H_M(s)$. Each token a_i has the N-gram probability $\widehat{p}_{M,N}(a_i)$

$$\begin{aligned} \widehat{p}_{M,N}(a_i) &= p(a_i|a_{i-N}\cdots a_{i-1}) = \frac{|a_{i-N}\cdots a_{i-1}a_i|}{|a_{i-N}\cdots a_{i-1}|} \\ \widetilde{p}_{M,N}(s) &= \prod_{i=1}^{n}\widehat{p}_{M,N}(a_i) \\ H_M(s) &= -\frac{1}{N}\sum_{i=1}^{n}\log\left(\widehat{p}_{M,N}(a_i)\right) \end{aligned} \tag{1}$$

Using cross-entropy, the authors compared the naturalness of code with "English texts". Very important in this field is the choice of a dataset. In the following sections, we will see how important and difficult it is to create a good dataset. In [11], the authors used the Brown and the Gutenberg corpus for the English part. For the source code part, they used a collection of 10 Java projects and a collection of C programs from Ubuntu categorised into 10 applications families.

The first test was the computation of the self cross-entropy, for English and Java datasets, of n-gram models using a different n-gram depth: N from 1 to 10. The cross-entropy for the English dataset is about 10 bit for the n-gram model with $N = 1$ and rapidly shrinks to 8 bit when $N \geq 4$. The cross-entropy for the Java dataset is about 7 bit for the n-gram model with $N = 1$ and rapidly shrinks to 3 bit when $N \geq 4$. This gives two important results: (i) Increasing N decreases cross-entropy (ii) Java has a much lower self cross-entropy than English. The first results suggest that both datasets have some regularities better highlighted using more complex models (increasing N). The second results suggest that source code is more regular than English. Using this evidence, they develop a code-typing suggestion engine in Eclipse using a 3-gram model able to beat the Eclipse standard code typing suggestion engine.

Barr ET. and Devanbu P. authors of "On the Naturalness of Software" [11] with Allamanis M. and Sutton C. in 2018 published "A Survey of Machine Learning for Big Code and Naturalness" an extensive literature review on the probabilistic model for programming languages [6]. The aim of [6] is to provide a guide for navigating the literature and to categorise the design principles of models and applications. Their hypothesis, expressed in their introduction, can be summarised in the following statements: Formal and logic-deductive approaches dominate research in programming languages. The advent of open-source, and the availability of a large corpus of source code and metadata ("big code"), opens to new approaches to develop software tools based on statistical distribution properties. Using Amdahl's law, they justify the approach to extract knowledge from "thousands of well-written software projects" on which statistical code analysis techniques are applied. By mediating the information extracted from many codes, it is possible to obtain knowledge that then helps to improve the average case. In this work, the authors define the *Naturalness Hypothesis*:

> *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools [6].*

They view software as the connection between two worlds: The human mind and computers. This bimodal property of the code defines the similarity and the differences between natural languages and programming languages. The main differences are the following properties of code:

Executability A small change in the code produces a big change in the code meaning. Thus, a probabilistic model requires formal constraints to reduce the noise introduced. Moreover, the executability property determines the presence of two forms of code analysis, static and dynamic.

Formality Unlike natural languages, programming languages are not written in stone and do not evolve over centuries. They are built as mathematical models that can change drastically over time. Moreover, the formality property does not avoid the semantic ambiguity of some languages due to some design choices (such as polymorphism and weak typing).

Cross-Channel Interaction The source code has two channels: algorithmic and explanatory. These channels are sometimes fused (e.g., explanatory identifiers). A model can exploit this property to build more robust knowledge.

The authors, then, provide a taxonomy of probabilistic models of code. They define three groups: Code-generating models, Representational models, Pattern mining models. The models of the first group aim to learn a probability distribution of code components and use this knowledge to generate new code. While the goal of the first group is to generate something (code, documentation, etc.), the second group aims to create a representation of the code to provide some information or prediction to the user. Instead, the pattern mining models try to infer the latent structure of the code to expose patterns and other information in an unsupervised fashion.

In the first category, code-generating models, the authors view the models as a probability distribution that describe a stochastic process for generating code. Equation (2) formalises this concept, where P_D is a conditional probability distribution given a dataset D, c is a code representation and $C(c)$ is a code context.

$$P_D(c|C(c)) \tag{2}$$

They distinguish the models based on how the source code is represented and the type of context information provided. The source code can be represented using the following abstraction level: Sequences (Token-level Models), Trees (Syntactic Models), and Graphs (Semantic Models). A large portion of identified works use Token-level Models (character or token). Some works use Syntactic Models or a hybrid composition of Syntactic and

Token-level Models. Furthermore, considering a graph as a natural representation of code, they did not find works using Semantic Models for this category.

The context information can be mathematically described using a context function. The context function identifies three model categories: (i) The model is a pure "language model" when $C(c) = \emptyset$ (no external information is provided) (ii) The model is a "transducer model" when $C(c)$ is itself code (iii) The model is a "code-generative multimodal model" if the context is provided but it is not code.

In the second category, representational models, authors collected methods capable of transforming a code into a representation that describes their properties. Equation (3) formalises this concept, where P_D is a conditional probability distribution, π is a feature vector and $f(c)$ is a function that maps the code c to a representation.

$$P_D(\pi|f(c)) \tag{3}$$

They define two independent categories for the identified models: (i) Models able to obtain distributed representations of the code and (ii) models able to obtain a structured code prediction. A single model may belong to both categories. The first category aims at representing source code in algebraic structures as vectors or matrices. In this way, the information is projected in a multidimensional metric space. In these methods, the function $f(c)$ is defined as $f : C \to \mathbb{R}^d$ where d is the number of dimensions of the representation vector and C is the set of source code. This category consists mainly of deep learning models. In fact, most deep learning models work on algebraic representations of data, on which operators can be easily applied, and the gradients can be conveniently calculated. The structured prediction models instead generalise the classification task. A model that belongs to this category can classify portions of the input, taking into account the inherent structural relations (and structural constraints) of the input data and the desired output, such as during grammatical analysis of the text.

On the other hand, the third category (pattern-mining) includes models that can discover patterns in the code in an unsupervised manner. Equation (4) formalises this concept, where P_D is a conditional probability distribution, $f(c)$ is a function that maps the code c to a representation, $g(c)$ is a function that extracts partial information from the code and X is the latent representation set that the model wants to learn.

$$P_D(f(c)) = \sum_{x \in X} P_D(g(c)|x)P(x) \tag{4}$$

The authors highlight the difficulty in building models belonging to this category, given the unsupervised nature of the task and the requirements of models (belonging eventually to other categories), helping to provide a definition of $f()$ and $g()$.

In conclusion, the authors provided a good classification of statistical models for source code analysis and manipulation. Their taxonomies can be helpful to clarify the advantages and disadvantages of each model and when it can be used, or how it can be adapted for a specific problem or application.

In contrast to the Allamanis et al. survey, the work of Ashouri AH. Killian W. Cavazos J. and Palermo G. entitled "A Survey on Compiler Autotuning using Machine Learning" [7] is more focused on machine learning techniques applied to compiler problems such as auto-tuning. Their aim is not to categorise a broad spectrum of models but to focus on the works facing optimisation-selection and phase-ordering problems of compilers.

Authors introduce the concept of the compiler optimisation phase as a subsequent ordered application of a certain number of optimisation steps at different levels of compiler layers (front-end, intermediate-representation (IR), and backend) to transform the code into an enhanced version of itself. The "enhanced version of code" is such if some performance metric is optimised, for example: Execution time, code size, or power consumption. They highlight key problems in this field; the optimisation steps can be language or architectural specific, an optimisation step can degrade the performance of the code if it is applied with

wrong parameters or in the wrong order. For these motivations, they identify two major problem sets: *Optimisation selecting* (which optimisation step to use and which parameters to use) and *Phase-ordering* (in which order to apply the selected optimisation steps). The first problem has an exploration space $|\Omega_{Selection}| = \{0, 1, \ldots, m\}^n$ where m is the number of parameters of an optimisation step and n is the number of optimisation steps. The second problem has an exploration space $|\Omega_{Phases}| = \sum_{i=0}^{L} n^i$ where n is the number of optimisation steps and L the maximum sequence length desired, with repetitions allowed.

The authors then introduce the concept of an auto-tuning framework. From a corpus of code divided into training-set and test-set, the training-set is compiled using a sequence of optimisations (chosen by an "optimisation design" component) and executed to evaluate the "objective metrics". These metrics are then used to train a machine-learning algorithm, which is fed by the source code features extracted by a feature-extraction procedure. The machine learning algorithm is then tested using the test set. The first step, as already discussed in [6] is to extract a feature vector for the source code. The authors divide the techniques into three categories: Static, Dynamic, Hybrid. The static analysis includes features based on source code information extracted directly from the text or provided from compilers and features based on graph structure of the code (Data Dependency Graph, Control Flow Graph), again extracted using compiler tools. Instead, dynamic features can depend on the execution flow in specific hardware (architecture-dependent) or can be extracted using specific code instrumentation to obtain architecture-independent features.

Then, the authors of [7] categorise some models based on the learning paradigm. In the unsupervised category, they identify two subcategories: Clustering and Evolutionary Algorithms. Of particular interest to the community is the second category, to which Neuro Evolution of Augmenting Topologies (NEAT) and other genetic algorithms (GA) belong [34]. In the supervised category, we can find well-known machine learning models such as Bayesian Networks, Linear Models and SVMs, Decision Trees and Random Forests, and Graph Kernels. Therefore, this work makes it possible to define better the context of code analysis research for compiler optimisations and collects works by target architecture (Embedded, Desktop, HPC) and compiler (GCC, LLVM, ICC, etc.). More details, in this specific field, can be found in [7].

Contemporary to [7], Wang Z. and O'Boyle M. in "Machine learning in compiler optimisation" [8] also provided a survey on the specific area of compiler optimisation. In the introduction of [8], the authors pointed out that the translation carried out by a compiler is a complex task. There are several ways to translate (compile) a code. The objective is to find the optimal translation that maximises evaluation metrics. Here, they emphasise that the term optimal is a misnomer. Indeed, when we refer to an optimal solution, we mean a sub-optimal solution obtained through heuristics. Their primary aims are to demystify the machine learning approach for compilers and demonstrate how this can lead to new and interesting research areas.

The authors summarise in three steps the integration of a machine learning pipeline in a compiler: Feature Engineering, Learning a Model, Deployment. Along with the discussion, they explain that there is not yet the ability to determine which ML model is the most suitable for a specific task. Some models such as Support Vector Machine (SVM) or Artificial Neural Network (ANN) can work with high-dimensional feature space but require a big dataset to work, and deep learning methods require even more data. Instead, more simple models such as Decision Tree (DT) or Gaussian Processes Classifier (GPC) can work with smaller datasets but manage fewer features. This work answers the question: How can machine learning be applied to compilers? What are the main applications? They identified two main categories: Optimising sequential programs and optimising parallel programs. Examples of applications in the first category are optimal loop unroll factor and function inlining, whereas examples in the second category are heterogeneous device mapping, scheduling and thread mapping.

In particular, we are interested in the heterogeneous device mapping problem owing to the relatively extended research paper corpus. Because of the effort required to create, build

and profile different versions of the same algorithm for the corresponding heterogeneous targets, a single dataset is used by almost all of these works, presented for the first time in [35]. On the other hand, the use of a common dataset helps to facilitate comparisons between techniques.

A third category incorporates several works not categorised, such as comment generation and mining API as we have seen in [6]. Please refer to chapter VI of the survey [8] for a complete list of works identified by the authors. In conclusion, in [8] the authors hope that research in this area will lead, in the future, to a better definition of code representation, i.e., one capable of identifying a metric of the distance between programs. The same should be carried out to obtain a representation of the computing capacities of the hardware. In this way, it would be possible to identify relationships between these representations: enabling more effective decisions at the compilation stage.

2.3. Machine Learning and Deep Learning Techniques for Source Code Analysis

This section describes works in Corpus 2 (Table 1) that apply new ML and DL techniques to source code analysis. We briefly describe each work emphasising the technique introduced, the problem addressed and the results obtained. The papers will be presented in chronological order by year of publication.

2.3.1. The Beginning, from NLP to Code Analysis (2013–2015)

In the period 2013–2015, ML techniques were explored on high-level code, starting from a manual definition of features to the first deep-learning applications capable of autonomously extracting features from code. The techniques used are strongly inspired by the background of natural language processing (NLP), including n-grams, decision trees (DT) and recursive neural networks (RNN). RNNs, in particular, have been extensively tested to assess their effectiveness [14,15]. The analysis is carried out on engineered features [13] or directly on the source code: function names [14] or directly on the raw code level, processing the characters that compose it [15]. The problems addressed are, for example, heterogeneous device mapping (the main topic of this paper), code completion and code execution results prediction.

In 2013, Grewe D. et al. in [13] developed a workflow to translate an OpenMP program in OpenCL and then decide the most suitable compute unit between CPU and GPU for each generated OpenCL kernel. There, the authors extract features from the code manually. They used the number of operations of three categories: compute, accesses to local and global memory and data transfer. They also defined some combinations of these raw counters to feed the classifier the communication–computation ratio, the computation–memory ratio and the local–global memory ratio. The classifier (predictor) is a decision tree trained with the C4.5 algorithm [36]. They used the NASA Advanced Supercomputing Parallel Benchmarks dataset (NPB) [37]. The classes are CPU or GPU, depending on what is the best target in terms of performance. In the case of the CPU class, the code uses OpenMP, whereas, in the GPU class, the code exploits OpenCL primitives. Authors obtained encouraging results with an accuracy of 75% when the target is an NVIDIA GPU and 62% when the target is an AMD GPU.

Raychev V. et al. [14] in 2014, proposed a code completion strategy that compares the n-gram model and recurrent neural networks (RNN). They implement their technique in a tool called SLANG with a top-3 accuracy of 90%. The authors first extract sequences of API calls from the dataset and then apply n-gram to these sequences. The RNN instead takes the last word in the sequence (w^i) as input using one-hot-encoding (v_i), and uses it to predict, similar to a classifier, probabilities for the next word ($y^i_{w_{i+1}}$). Specifically, they use a variant of RNN called RNNME. The authors of [14] report that the n-gram technique can discover regularities between the last $n-1$ elements of function-calls sequences. RNN instead can discover relations at longer distances. This work can be seen as an extension of [11] and one of the first works to employ RNN in this field.

Zaremba W. et al. [15], in their paper, try to test the limits of LSTM-based networks with a task considered difficult: given a code fragment, inferring the result as though you were running the code. They defined a simple class of programs (with some constraints such as single left-to-right pass using constant memory) and used an LSTM cell in a sequence-to-sequence model to obtain the result of the code execution. They reached 99% accuracy in this task. The LSTM receives direct input of the characters, thus it was not necessary to develop a complex embedding layer. This work highlights a good example of the capabilities of recurrent neural networks.

2.3.2. Broad Investigations: New Models, New Representations and New Applications (2016–2018)

In 2016–2018, ML techniques for code started to be explored more in-depth by the scientific community. Alongside the RNNs and their variants (LSTM and GRU cells), complex models based on a sequence-to-sequence (Seq2Seq) structure emerge [16,17,19] alongside an innovative adaptation of CNN on tree structures [18,24] and the introduction of a new neural network structure: the Graph Neural Network (GNN) [20,38]. The techniques were applied on Sequences of Tokens (extracted from the source code), Abstract Syntax Tree [18] and API call sequences. The problems addressed span from code description and code comprehension to syntax error recognition [22] and code fixing.

Iyer S. et al. in [16] developed CodeNN, a tool able to describe in English code written in C# and SQL. They used the LSTM model with attention mechanism trained using a dataset gathered from Stack-Overflow comprising more than 900 k post for C# and the same number for SQL. CodeNN is also able to retrieve source code if a code description is provided. CodeNN outperforms the state-of-the-art performance in both tasks, both on code description and code retrieval.

Bhatia S. et al. in [17] instead compared RNN and LSTM performance in correcting small programs written in Python. The authors propose the SYNFIX Algorithm, which operates as follows: it parses a Python program, identifies a syntax error, and tries to generate the sequence of tokens using a Seq2Seq model to correct the error. The model manages the code as a sequence of tokens, and the authors use a dataset of python code produced by students. The model's performance is modest; in many cases (32% of code corpus), it was possible to correct some errors, but in the presence of multiple errors, the percentage of code that presents syntax or runtime errors remains predominant. No substantial differences were found between the RNN and LSTM models.

Mou L. et al. in [18] try to overcome the limitations imposed by analysing the source code as a linear sequence of tokens. They propose a Tree-Based Convolutional Neural Network (TBCNN), a deep learning model that works with source code's Abstract Syntax Tree (AST). The authors adapted the Convolutional Neural Network (CNN) to work with tree structure generalising the convolution operator to work with sub-trees of an AST. The AST node embedding was performed using a custom method to map similar symbols to similar feature vectors. They used this model to solve two problems, functionality classification of a program and code fragment identification that matches some programming pattern. Using a dataset composed of programs submitted by students, their method outperforms other models such as SVM and DNN (based on Bag of Word (BoW) and Bag of Tree (BoT) feature extraction), reaching 94.0% accuracy against the 89.7% reached by the best competitor (DNN+BoT). In the second problem, code fragment identification, the improvements against state of the art is more pronounced: 89.1% against 77.1% (SVM+BoT).

Gu X. et al. in [19] propose DeepAPI, a deep learning method able to generate instructions for the user about the API sequence of functions and calls to accomplish a task. Their approach leverages an RNN-based encoder–decoder. This approach can fit in the "code-generative multimodal model" category of [6]. The encoder transforms the input sequence (the user query or context) into a fixed-size vector of features (latent representation). The decoder takes the latent representation as input and generates a sequence of API calls. The dataset used is composed of 442 k Java projects, from GitHub,

with code descriptions. The API sequences used to train the model are extracted, parsing the AST of code. The goodness of results is evaluated using the BLEU score (Bilingual Evaluation Understudy) [39].

Allamanis M. et al. in [20] propose a radical new method for analysing code: using neural network models with graph structures. They use Gated Graph Neural Networks (GGNN) and an enriched AST code representation. The enriched AST captures the control flow and data dependency properties of the code and the programming language's grammar. The GGNN use a recurrent neural network, more specifically a Gated Recursive Unit (GRU) [40], in each graph node and process the data that flow along the edges. The output of one node (GRU cell) flows along the edges of the graph and is given as input of another graph node (another GRU cell). The first node representation is achieved by combining information from textual node representation and its type. The node type is inferred with a smart code inference scheme described in [20] to section four. The proposed model was tested on two applications: Variable Naming (VarNaming) and Variable Misuse (VarMisuse). The model trained for the first problem (VarNamig) uses a dataset of 29 C# projects. The extended AST is modified, replacing for a variable v a special token $< SLOT >$ whose representation, at the end of the GGNN propagation phase, is fed to another GRU that generates a sequence of symbols to reconstruct the variable name. The second application is more complex as it modifies the extended AST to represent a speculative presence of a variable in the code. In the end, the presence of a misuse variable is detected using the final representation of GGNN nodes and an additional linear layer. This work enables the researchers to create a better code analysis and graph representation of source code.

To solve the code search problem, Gu X. et al. in [21] propose Code-Description Embedding Neural Network (CODEnn, not to be confused with CodeNN [16]) a neural network model, developed by them, able to map both code and English description in the same high-dimensional vector space. The coexistence, within the same space, of the code representation and its description allows easy identification, through a distance metric, of a code given its description. Two distinct models are used to represent the code and its description in the same feature space, CoNN for the code and DeNN for the description. Both models output a feature vector; the cosine similarity is then used to compute the error, minimise, and obtain similar representations. The code is represented in three forms, sequence of method names, API invocations, and tokens. Each representation is individually analysed and results combined in a single result. The method is trained from a corpus of Java Projects, with documentation and comments taken from GitHub. The authors, in the results section, show the good performance of their model. This interesting approach shows how these techniques allow reasoning with a high degree of freedom over the information in two different languages, Java and English.

Bavishi R. et al. in [23] propose Context2Name, an LSTM Autoencoder to create a language representation and to infer meaningful names for variables. This problem was inspired by the minimisation that JavaScript code undergoes before being embedded in a hypertext. The minified JavaScript loses the original variable names, replaced by simple, short names to optimise the transfer of the program. Reversing this process is a challenge addressed by autoencoders based on LSTM recurrent networks fed with a token-based code representation. The model is enriched with a semantic-preserving name recovery functionality to maintain the same variable name along with the whole code fragment. The results section highlighted that Context2Name reached state-of-art (SoA) performance results compared to other tools but dramatically improved the execution time, with a median of 52ms compared to 73ms of JSNice and 20s of JSNaughty (two other SoA tools). Despite the simple code representation method used in this tool, the results are promising. With more sophisticated techniques, it may be possible to improve performance enough to improve the SoA.

2.3.3. Consolidation, Graph Models and Multilevel Code Analysis (2019–2021)

In recent years, the topic has become more mature and more works have tried to exploit the Encoder–Decoder structure [26], the GNN models [33,41] or other models based on hierarchical structures such as DAG-RNN [27]. The techniques were applied on AST, CFG, and other ad hoc compositions of them such as CDFG [41]. The analysis of intermediate representation (such as LLVM-IR) or assembly code becomes a new way to work with code [9,41,42]. The problems addressed are, for example: function name and type inferring (classification) [25,28], performance prediction [27] and loop-optimisation [31,41].

Alon U. et al. in [25] and then in [26] propose to represent source code as paths along with the AST and use this representation to infer function name, a task also called Code Classification. These techniques are not limited to Code Classification. Still, authors show how it is possible to perform reasoning on the learned embedding and solve other tasks as semantic similarities, combinations and analogies. In the first work, code2vec, the authors solve a classification task using a "path attention network". The network used is composed of a fully connected layer and an attention layer. In the second work, code2seq, the authors propose a more complex model, a network able to process the AST paths in an Encoder–Decoder fashion. Each path component is encoded using an LSTM node and enriched with the sub-tokens that constitute the AST nodes. Then, the encoding of all AST paths is given to a decoder, with an attention model to obtain the desired output sequence (e.g., the function name or the description in English). Using a vast Java dataset, the authors show the superiority in performance (F1 score) of code2seq against code2vec and other SoA related works [43–45].

Mendis C. et al. in [27] describe Ithemal, a tool for predicting the throughput of code during its execution. Ithemal considers the source code of a basic block in assembly format and predicts the throughput on a specific computing architecture for which it has been trained. The first step is the canonisation of code statement tokens and their transformation in a vector representation. Then, the subsequent two LSTM layers process the code. The first LSTM layer processes all tokens of a statement and produces a statement vector representation. The second LSTM layer processes the sequence of statements vector representation to obtain a block vector representation. Then, from the block vector representation, a further linear layer produces the Throughput Prediction. They generate a dataset maximising the x86-64 instructions coverage and targeting three Intel architectures: Ivy Bridge, Haswell and Skylake. Moreover, they perform a model exploration and propose another model, DAG-RNN, where the hierarchical-LSTM (the original model) is changed with a graph structure of LSTM cells to satisfy data dependencies along with the basic block statements. The best performance has been obtained using the Hierarchical method, highlighting that, in a basic block, the order of the instructions is more critical than dependency chains, at least for the processors under consideration. Ithemal is faster and more precise of SoA tools such as llvm-mca [46] and IACA [47]. It is also one of the first applications of LSTM directly to assembly language.

Pradel M. et al. in [28] propose Typewriter, a tool based on Deep Learning and natural language properties of code, able to infer the types of arguments and return value of a function. Using a dataset consisting of Python code achieves 0.64 F1 score in return type predictions and 0.57 F1 score in arguments types prediction. The Typewriter comprises three steps: Static analysis, Neural type prediction, and Consistent types search. In the first step, the tool extract data from the AST: tokens, identifiers and comments. In the second step, the previously extracted sequences are processed by different RNNs. A fully connected layer then processes the output of each RNN to obtain the type vector. Then, the third step iteratively tries to obtain a consistent type for each function argument and return value. Typewriter is developed in Facebook and internally used by programmers in the code review domain. This represents a good example of the practical usage of Deep Learning technology applied to source code.

Hoang, T. et al. in [29] proposed CC2Vec, an innovative deep-learning model to learn code representations from software modification and commit descriptions in a versioning

system exploiting attention at multiple levels of abstraction: word, line, hunk. In versioning software jargon, a "hunk" is the area where two files differ. In this context, the hunk is the area of interest where the software was modified. The authors demonstrate the goodness of learned features in CC2Vec, solving three different problems. The network model is called Hierarchical Attention Network (HAN), and is composed of an encoder and an attention layer for each level of abstraction (word, line, hunk). The encoder is a bidirectional RNN implemented with GRU cells. In this phase, the added code and the removed code are analysed independently. Then, a comparison layer computes the vector that summarises the differences between added and removed code features. Then, a word prediction layer associates the vector of differences with a set of words extracted from the log message. The authors highlight in the result section how CC2Vec can improve, with its code representation, and other tools for solving complex problems: (i) Log message generation (ii) Bug fixing patch identification (iii) Defect prediction.

Karampatsis RM. et al. in [30] address the problem of vocabulary representation when building a code embedding. Specifically, they point out that many methods suffer from two main problems: dictionaries that grow excessively large or the need to accept out-of-vocabulary elements. Their approach involves the usage of Byte-Pair Encoding (BPE) [48]. The dictionary starts with only letters and digits but using a fusion operator, the most represented sub-words and words can be formed. In this way, the use of out-of-word tokens can be avoided, and the growth of the dictionary remains controllable (a parameter defines the maximum fusions allowed). Then, the authors used a GRU model trained with the BPE encoding. They tested their model with Java, Python and C datasets. The proposed model is simple, effective, and easily embeddable to other projects that need an embedding technique.

Haj-Ali A. et al. in [31] propose NeuroVectorizer to address the problem of loop optimisations for SIMD capabilities of modern applications. In this field, the loops can be modified to expose vectorisation and operations interleaving opportunities. Another problem is choosing two factors: Vectorisation Factor (VF) and Interleaving Factor (IF). VF defines how many instructions are useful to pack together. IF defines at which distance operations must be placed to relax architecture component pressure. The authors proposed a solution to these problems using deep reinforcement learning (RL). Their tool can inject compiler pragmas in the source code to manage VF and IR values. To perform code embedding authors applied code2vec [25]. An interesting discovery is that for nested loops, where pragmas are applied only to the innermost loop, the performance of NeuroVectorizer increases if the whole cycle is analysed and not only the innermost. Moreover, the authors show how this technique can be embedded in Polly [49], a reference implementation of polyhedral compilers.

Finally, Brauckmann A. et al. in [41] directly address the problem of polyhedral optimisations through the use of RL. The polyhedral model is a way to identify code transformations that can improve performance, exploit computing architecture components, and preserve the semantic of code. The authors propose PolyGym, a Markov Decision Process (MDP), to model the space of loop polytopes boundaries and explore relative legal transformations. PolyGym is composed of two components (both MDP): The schedule-space constructor and the schedule-space explorer. The schedule-space constructor inputs the SCoP identified by Polly and generates schedule-space generators, and then schedule-space explorer generates a schedule for Polly. The choices made by these two components can be tuned with reinforcement learning defining a reward system. In the results section, the authors show the speedup obtainable with this tool in terms of speedup against standard compiler optimisations.

2.4. Final Remarks

The topic touched on by the works presented in this section approaches code analysis with machine learning and deep learning. Specifically, we addressed the RQ1 providing a

list of key works on machine learning and deep learning techniques to solve source code related problems. We summarise the problem categories in Table 2.

Table 2. Application examples per category.

Code Modelling	Code Manipulation	Code Optimisation
API exploration	Code Completion	Heuristic for compilers
Code Conventions	Code Synthesis	Auto-parallelisation
Code Semantic	Code Fixing	Bug localisation
Code Summarising	Comment generation	

Although we have narrowed the scope of this research timeline to only those papers dealing with code analysis by machine learning techniques and take papers within a selected group of papers [12], the spectrum of problems addressed in these areas is still too broad to make a quantitative comparison of the techniques used. However, the presented works highlight the necessity to apply novel ML techniques to approach code related problems; through ML, the works report and achieve superior performance over previous techniques.

In the next section, we introduce the problem of heterogeneous device mapping. It will be discussed extensively in Section 4.

3. Approaches for Heterogeneous Device Mapping in CPS

This section focuses on specific works dealing with applications purely oriented towards heterogeneous platforms as main digital components of cyber-physical systems (CPS), specifically addressing the heterogeneous device mapping problem. In this section, we thus address RQ2: "How have machine learning techniques been applied to cyber-physical systems and, in particular, to heterogeneous device mapping?". We consider the papers belonging to Corpus 2 and listed in Table 3. All these works deal with the heterogeneous device mapping problem (see Figure 1). This topic (Code Optimisation → Heuristic for compilers) addressed in [13] has paved the way for the scientific community to extend this research. This is motivated by the possibility of alleviating the difficult task of programming the heterogeneous architectures that comprise modern CPSs. Source code analysis can help in finding the most suitable hardware unit on which to execute a given computational kernel. Moreover, developing 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.

Table 3. List of selected papers for source code analysis methods with results of interest in the CPS field.

Reference		Year	Keywords
Heterogeneous Device Mapping			
Cummins C. et al.	[35]	2017	LSTM, OpenCL
Ben-Nun T. et al.	[42]	2018	LSTM, LLVM graph embedding
Barchi F. et al.	[9]	2019	LSTM, LLVM tokenisation
Venkata Keerthy S. et al.	[50]	2020	LLVM embedding, Gradient Boosting
Brauckmann A. et al.	[41]	2020	MPNN (GNN), LLVM graph embedding, CDFG, AST
Cummins C. et al.	[51]	2020	MPNN (GNN), LLVM graph embedding, CFG
Parisi E. et al.	[52]	2021	OpenMP, Random Forest, Energy consumption
Barchi F. et al.	[53]	2021	LLVM tokenisation, CNN, LSTM

Recently, a research line exploiting the maturity of deep learning methods has started since the work of Cummins et al. [35], 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 the classification accuracy improves compared to [13].

The methodologies proposed in [13,35] 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. [42] and Barchi et al. [9] introduced the adoption of code analysis at the intermediate representation (IR) level of the LLVM compiler. In [9], 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 [42], the authors propose Inst2Vec, a system to pre-train the embedding layer analysing the Contextual Flow Graph (XFG). 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. At this intermediate level, source code features can be exploited to perform complex compilation decisions, including allocating code fragments to architecture devices.

In Kheerthy et al. [54], the authors propose IR2Vec, a procedure that projects an IR in a continuous metric space directly. In Cummins et al. [51], the authors propose *ProGraML*, an extension of [42] where a complete GNN-based classifier is proposed. Independently, in Brauckmann et al. [41] another end-to-end graph-based classifier was proposed able to learn vertex embeddings by itself. Moreover, in [41] the graph-based classifier 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 mentioned works use RNNs, that have been introduced to process temporal sequences [5,55]. An RNN maintains an internal state, acting as a memory, that summarises the information extracted from the input sequence. As seen in Section 2.3, very successful implementation of RNN is the Long Short-Term Memory (LSTM), a network able to learn when to memorise or forget 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 [9,35,42,54].

However, given the widespread use of CNN in the context of image recognition [56] but also in NLP [57] 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. In [53], authors introduced CNNs in a code classifier. This network model takes a tokenised and filtered code stream as input. 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. The kernel shape has two dimensions in the image classification field, but this technique can also be used in temporal signals using one-dimensional kernels.

Final Remarks

With reference to RQ2, this section showed how machine learning techniques have been used to solve the heterogeneous device mapping and thread coarsening problem. The techniques used are similar to the techniques exploited by the works presented in Section 2.3. The deep learning models based on CNN, LSTM and GNN have been used to classify code fragments using execution time information obtained in real code deployed on different architectures with CPU or GPU capabilities provided by different vendors. The works focused on different approaches to code modelling, from a simple serialisation of code in tokens to complex graph representations. Furthermore, different ways to process contextual information (working data size and computing elements configurations) were considered and introduced in the model.

The majority of the methods discussed [9,35,41,51,53] are trained and evaluated on a common dataset introduced in [35]. The dataset consists of 256 OpenCL kernels sourced from seven benchmark suites on two CPU/GPU pairs combinations. The 256 unique kernels belong to 7 suites and 71 benchmarks as reported in Table 4. Each pair is labelled CPU/GPU

according to the processing element in which it executes faster. Each dataset consists of 680 labelled pairs derived from the 256 unique kernels by varying dynamic inputs. The same pair has been executed in two different heterogeneous 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 pair is characterised by three values: The source code and two auxiliary inputs, namely the payload size and OpenCL Work Group size. In the next section, we delve into the details about the works described here.

Table 4. Dataset composition [35]. The first two columns are the number of benchmarks in suite and the number of unique kernels in suite. In the dataset, composed by the tuple code and meta-information, each suite has a different number of pairs.

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

4. Deep Learning Methods for Heterogeneous Device Mapping

In this section, we delve deeper into the heterogeneous device mapping problem and discuss state-of-the-art methodologies approaching it. In particular, we want to reply to the RQ3: "How do the machine learning methods analyse source code in heterogeneous device mapping, and what results have been obtained?" This section reports a meta-analysis of different techniques applied to the same dataset. As seen in the previous section (Section 3), we discuss in more detail the works in Table 3 that approach the heterogeneous device mapping problem and use a common dataset: More specifically, the dataset identified in [13,35]. The experimental results of identified works are therefore comparable. In this section, we present a critical discussion of the proposed techniques to identify their strengths and weaknesses.

The problem of heterogeneous device mapping is stated as follows. Given a cyber-physical system featuring a set S of computing devices $d_0, \ldots, d_{N-1}$, the source code of a function's kernel and related contextual information, then select the best device $d_i \in S$ to execute the kernel such that some system metric is optimised. Typical interesting metrics for a cyber-physical system are runtime performance, energy consumption or peak power consumption.

Examples of helpful contextual information to predict the best possible computing devices are the amount of data the device d_i handles or the estimate of the amount of parallelism available in the offloading accelerator.

With the end of Dennard's scaling and the slowing down of Moore's law, heterogeneous architectures have become increasingly popular [3]. Currently, it is common to have parallel multi-core systems that take advantage of hardware accelerators such as GPU, FPGA or DSP and choosing where to move computation is not clear and requires effort and resources [51]. As a result, the activity of exploiting static source code features to predict the best executing device in a system-on-chip platform has gained interest in the research community.

All tools that approach the heterogeneous device mapping problem share a similar structure. First, they summarise the source code to be analysed by associating an embedding vector to it. Such an operation is called *language modelling*, and it maps the input kernel to an N-dimensional Euclidean space. Then, program embedding feeds a classifier according to the nature of the problem to be solved. The methodologies that approach

heterogeneous device mapping commonly employ a binary classifier to choose the most promising execution device among CPU and GPU. However, such a scheme easily scales to multi-class classification problems and can serve both supervised and unsupervised learning scenarios.

In Sections 4.1–4.3 we detail how state-of-the-art tools perform language modelling. Section 4.4 provides details about machine learning models to translate kernel embedding vectors into the classification prediction. Furthermore, it gives an insight of how effective the different approaches proposed thus far in the literature are in terms of classification accuracy over both Nvidia and AMD hardware. Figure 2 provides a logical representation of the heterogeneous device mapping tools described in this section. Each methodology is associated with a coloured line that describes the tool's analysis flow, following the legend at the bottom of the figure.

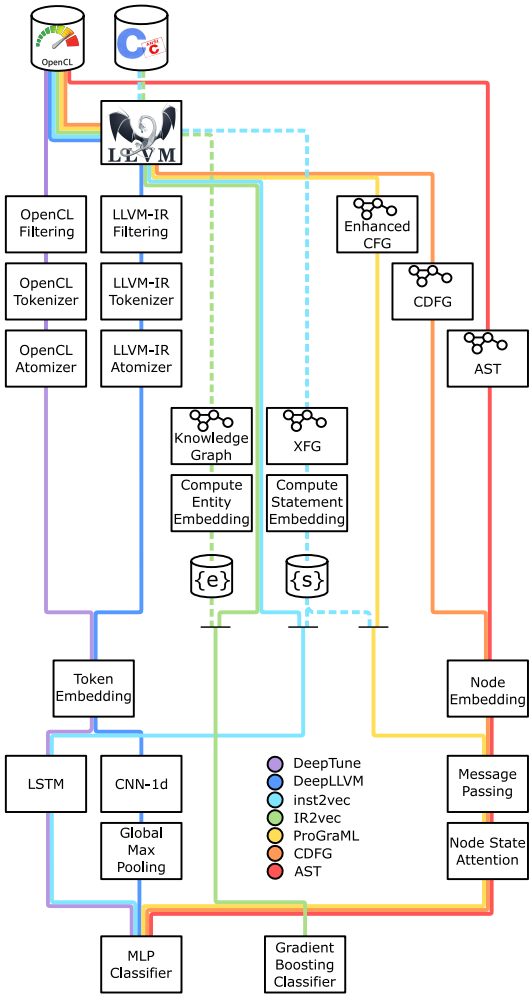

Figure 2. Map of heterogeneous device mapping methodologies described in Section 3. Dashed lines represent the methodology used in [42,50] that exploit unlabelled C source codes to build LLVM statement and entity embedding dictionaries. Whenever continuous and dashed lines join, it means a tool requires a previously built embedding dictionary to continue its analysis.

4.1. Token-Based Language Modelling

Token-based language modelling methodologies analyse source code as a sequence of tokens, as described in Section 2. Relevant literature works that apply such a policy and discuss possible improvements are [9,35,53].

In [35], Cummins et al. propose *DeepTune* (purple line in Figure 2), to build language models from OpenCL sources. After the input source is loaded and filtered from comments and irrelevant metadata, the list of OpenCL tokens is extracted without further processing.

The tokenisation procedure considers the source code a text and scans it from beginning to end. Each word in the text that is either a C keyword or an OpenCL directive is a token. Any word that does not match the previous definition is decomposed into its characters, all of which is a token. The first step of language modelling turns each element of the token list into a numeric array using an embedding layer whose weights are learned during training. The sequence of token embedding is then processed by two recurrent LSTM layers, which output a digested representation of the kernel being analysed. The rationale behind the usage of recurrent cells resides in their ability to learn a relationship over a long sequence of tokens, which is the case of source code analysis, where data and control dependencies may span the whole program. While producing promising results, the presented approach has few inconveniences. The language model is built from OpenCL tokens, limiting its validity to source written in C and exploiting the OpenCL programming model. Furthermore, the token list is not actively filtered to remove tokens with poor meaning, such as punctuation marks or other symbols. Both drawbacks are addressed in [9].

The authors of [9] keep the same language modelling network of [35] while testing the impact of translating OpenCL source code to its LLVM intermediate representation before analysing it. As a result, the source code analysed is decoupled from the high-level programming language used. In fact, differently from [35], a language model trained on LLVM intermediate representation has the potential to represent source codes written in any high-level language with an LLVM back-end. Moreover, the authors of [9] suggests filtering the list of LLVM tokens removing those with poor informative content. After the whole source code corpus is tokenised, each token-source code pair (t, s) is associated with a *Tf-Idf* score, a well-known metric to evaluate the importance of words in document corpora. If the score of the pair is below a given threshold, then token t is removed from the source code s. The authors conclude that the additional effort required to translate the source code to intermediate representation and remove redundant tokens make the classifier more robust, which is corroborated by a gain in classification accuracy.

The authors of [53] further improve the state-of-the-art, proposing a pipeline called *DeepLLVM* (blue line in Figure 2) which substitutes LSTM cells with convolution and pooling layers for language modelling. The sequence of token embedding feeds a set of 1D convolution layers which then exploits global max pooling to extract relevant features from the convolution output. The authors show how such an approach achieves 4–7x shorter training time with respect to LSTM since CNN exposes a high degree of parallelism, making GPU training more agile.

All three methodologies described in this section deserve credit for the evolution of source code modelling state-of-the-art with deep learning-based approaches. However, token-based source code modelling is problematic since code is not a sequence. Major program features, such as reaching definition or branching, imply long-distance relationships between program entities and may be challenging to learn if sources are modelled as sequences of tokens, even using recurrent networks. Furthermore, all previously described works modify token sequences with padding or truncating operations. Indeed, language modelling networks work on fixed-length sequences without taking into account a possibly large difference in size between two functions.

4.2. Graph-Based Methodologies

Graph-based language modelling recently appeared in the literature to overcome the limitations of token-based approaches described in Section 4.1. Relevant works, in this context, for graph-based modelling are [41,51].

Cummins et al. [51] propose a tool called *ProGraML* (yellow line in Figure 2) that compiles input source code into LLVM intermediate representation and builds a graph, referenced as Enhanced-CFG in Figure 2, by the following three steps. First, add a node in the graph for each opcode in the intermediate representation and connect them using edges to represent control flow information. Then, encode data-flow including nodes for variables and constants in the graph along with additional edges that link them to the

opcodes that produce or use them. As the last step, call edges may be included to represents whole programs since control-flow edges do not span functions. The authors of [51] exploit GRU-based message-passing neural networks (MPNN) [58] to build a proper language model from the graph-based source code representation. *ProGraML* uses a modified version of [42] to assign each graph node its hidden state array, which is required to feed the message passing procedure. Once message passing is run for the desired number of time steps, each vertex is assigned to its final hidden array, and is processed by an attention layer that aims at extracting the most relevant features from each node state. At last, the sum of all hidden states provides the embedding vector for the program analysed, which models the analysed kernel.

At the same time, the authors of [41] propose two strategies to represent source code based on (i) Abstract-syntax tree (AST) and (ii) Control data-flow graph (CDFG) summarised by red and orange lines in Figure 2. The first representation exploits the compiler front-end to extract the AST directly from the OpenCL source code. Furthermore, the AST is enhanced with data-flow edges to avoid losing information regarding which identifier is bounded to which data. The second representation proposed is called CDFG and models LLVM intermediate representation functions with a smaller amount of nodes with respect to what is proposed in [51]. It consists of a graph whose nodes are LLVM opcodes while its edges belong to three different classes (i) Control-flow edges, (ii) Data-flow edges and (iii) Call edges to model functions return data. As in [51], both representations feed a GRU-based MPNN. Differently from what is described in [51], external tools are not required to assign to each graph node a hidden state array initialisation value. In fact, each vertex identifier is associated with a one-hot encoded vector whose embedding is computed through a node embedding layer whose weights are learnt as the training proceeds. Once each node is assigned its starting hidden array, message-passing is run for a predefined number of times. The back-end of the [41] language model is similar to that proposed in [51]. After each graph vertex is assigned its final hidden state, an attention mechanism selects the most meaningful features from each node state. Then, the overall program embedding can be computed by summing the contribution of each node.

Both [41,51] employ MPNNs while they differ in kernel modelling approaches. On one hand, *ProGraML* [51] authors prove that their source code representation is so rich that even typical compiler tasks such as finding out live variables or checking statement reachability could be accomplished. On the other hand, ref. [41] still pushes the state-of-the-art proving that (i) AST can be successfully used to model code without compiling it to intermediate representation and (ii) solving the heterogeneous device mapping problem is possible with CDFGs which trade representation richness with graph size when compared with [51].

4.3. Alternative Methodologies

Alternative modelling strategies exist that either do not entirely fall into the token-based or graph-based taxonomy proposed so far [42,50], or exploit Siamese topology, a recently proposed technique [10] which promises to further increase classification accuracy and is applicable to any model previously described.

In [42], Ben Nun et al. describe *inst2vec* (cyan in Figure 2) to compute LLVM intermediate representation statements embedding. It exploits the skip-gram model [59] adopted to compute word embedding by redefining the notion of instruction context. Word embedding relies on the notion of word context, supposing that words with similar context have similar semantics and states that a word's context is the set of terms within a given radius around it. Unfortunately, such a definition is not valid considering intermediate representation instructions. Pair of statements far in the source may have control or data relationships, making it impossible to extend the notion of context from words to statements as it is. Using a directed multi-graph called *conteXtual Flow Graph*, *inst2vec* models LLVM instructions, where graph nodes represent local or global identifiers, while graph edges represent data or execution dependencies. The *conteXtual Flow Graph* dual graph is a graph whose nodes represent LLVM statements, and its outgoing edges define its context. Given the notion

of statement context, the authors exploit the skip-gram model, as originally proposed in word embedding [59], to build statement embedding. Once the kernel to be analysed is reduced to a sequence of embedding, language model is produced feeding the vector list to two recurrent layers similarly to what described in Section 4.1 and detailed in [9,35,53].

While the authors of [42] propose interesting considerations about source code naturalness and profitably adapt the concept of word context to LLVM statements, *inst2vec* is prone to the Out-of-Vocabulary issue. Since the variety of different LLVM statements is extremely large, it is possible to design an LLVM kernel with statements not available in the *inst2vec* dictionary as they are not present in the corpus of C sources that [42] use to train statement embedding. Such an effect is characterised in [50] when comparing *inst2vec* with another source code embedding strategy called *IR2vec*.

Venkata Keerthy et al. [50] propose *IR2vec* (green in Figure 2) as an approach to compute LLVM source code elements embedding hierarchically. First, it learns the embedding of LLVM-IR entities: (i) Data types, (ii) Local variables, and (iii) Opcodes. Then, the tool combines entity embedding vectors linearly to generate the representation of LLVM statements, basic blocks, functions and, eventually, the whole program. *IR2vec* models each statement as a set of relationships between entities. It identifies three kinds of relationships: (i) *NextInstr* to model the sequence of opcodes, (ii) *TypeOf* to model the type of data an opcode works with and produces, and (iii) *Arg* to keep track of which arguments an instruction requires (local variable, pointer, constants, function name, or label). A Knowledge Graph is built to represent the corpus of the target dataset of source codes, where nodes represent entities while edges represent all the relationships each entity is subject to.

Knowledge Graphs are widely studied structures for representing entity relationships, and a plethora of machine learning models are widely used in the literature to compute entity embedding from the knowledge graph. The authors of [50] chose *TransE* [60] to associate embedding vector to each discovered entity. The embedding for an LLVM statement is a weighted linear combination of its entity embedding vectors and the previously computed embedding of LLVM instructions that have reaching-definition to the current statement considered.

The summation of the embedding vector of live statements produces the representation of a basic block, and similarly, the summation of basic block vectors gives the representation of a function. *IR2vec* has three properties that make it particularly interesting and unique when compared with other language modelling tools. First, it is not prone to Out-Of-Vocabulary issues since the amount of LLVM entities program embedding that it is built from is bounded. The authors prove it in the experimental section of their paper [50] when comparing with *inst2vec*. Furthermore, it models LLVM intermediate representation without relying on opaque models learnt through deep recurrent layers or graph-based networks. It works deterministically and employs well-known data-flow analysis such as Reaching-Definitions and Use-Def relationships widely used in compiler optimisations. Finally, being hierarchical it has the potential to further explore currently known methodologies. For example, it could be used to initialise *ProGraML* node hidden state in place of *inst2vec* to avoid Out-Of-Vocabulary issues or to apply the message-passing methodologies detailed in [41,51] to the control-flow graph of a kernel using basic-block embedding to initialise the graph nodes.

The authors of [10] proposed to arrange the token-based CNN model described in [53] in a Siamese topology to classify the most convenient device to execute a computational kernel. A Siamese network requires two ingredients. (i) A machine learning model able to project input data into a N-dimensional Euclidean space and (ii) A loss function able to associate a loss value to a pair of N-dimensional point projections. Furthermore, the dataset needs to be arranged such that pairs of samples are fed into the model at training time. Whenever the training procedure fetches a pair of samples (D_1, D_2) from the dataset, the CNN at the core of the model projects both into the N-dimensional points (S_1, S_2), with $N = 2$ in [10]. The loss function evaluates the distance $d(S_1, S_2)$ between the two sample projections and behaves such that samples belonging to different classes are moved away.

On the contrary, samples with equal labels are penalised proportionally to the distance of their projections.

4.4. Classification Models and Comparative Results

After the language modelling neural network has represented the input source code using an appropriate embedding vector, it can feed a classifier to choose the device where the target kernel should be executed. Because of the properties of the only heterogeneous device mapping dataset publicly available, described in Section 2, all described works employ a binary classifier to predict the best execution platform.

The tools described in [9,35,41,42,51,53] process the embedding produced by the language modelling using a multi-layer perceptron. At first, the auxiliary information is concatenated to kernel embedding and normalised using a batch normalisation layer. Such a choice is inherited from [35] whose authors claim that batch normalisation is necessary given the arbitrarily large values auxiliary inputs have, with respect to source code embedding produced by language models. The output of batch normalisation feeds the two-class multi-layer perceptron, which outputs the best device where the kernel can be executed.

Using a different approach, the authors of [50] concatenate auxiliary information to the kernel embedding produced by *IR2vec* and use the resulting vector to feed a gradient boosting classifier.

Concerning the approach based on Siamese network [10], the way they work does not require a final binary classifier to infer the best kernel offloading device. In fact, once training is over, the network performs inference on every samples in the training set and for each class, the centroid of the projections of samples belonging to that class is computed. At test time, the CNN at the core of the Siamese network is used to compute test samples projections and predict labels depending on the centroid closer to each projected point.

Following RQ3, in this section we compared the techniques used in heterogeneous device mapping, exposing the obtained results and providing the internal components and analysis steps summarised in Figure 2. Table 5 reports comparative results available in the literature for the state-of-the-art methodologies for heterogeneous device mapping. On the one hand, the difference in accuracy between different machine learning models falls below 10%, highlighting how none of the models outperform alternatives, especially considering the size of dataset tests. Moreover, not all work presented in Table 5 performs hyper-parameter tuning to optimise model performance. On the other hand, Siamese [10] and *IR2vec* [50] performance suggests that: (i) Siamese network configuration may be a valuable addition to be further investigated for all available state-of-the-art methodologies and source code modelling techniques that take advantage of compiler analysis, such as *reaching definitions* and (ii) *Use-defs* may constitute a lightweight alternative to deep learning models requiring a large amount of data to be appropriately trained.

Table 5. Comparison with state-of-the-art methodologies. Accuracy of methodologies from the top part of the table comes from [10], while the accuracy of *ir2vec* is reported from [50].

		State-of-the-Art Methodologies		
		AMD	**NVIDIA**	**Mean**
DeepTune	[35]	0.814	0.805	0.810
NCC/*inst2vec*	[42]	0.802	0.810	0.806
CDFG	[41]	0.864	0.814	0.839
ProGraML	[51]	0.866	0.800	0.833
DeepLLVM	[53]	0.853	0.823	0.838
Siamese	[10]	0.917	0.888	0.903
IR2vec	[50]	0.924	0.870	0.897

5. Deep Learning for CPS Programming: What Is Missing?

This section reports some considerations about the current challenges to unlock deep learning for source code analysis in heterogeneous embedded platforms such as those used in a cyber-physical system (CPS).

Considering the characteristics of existing CPS, i.e., the presence of multiple accelerators and shared and contested resources, the solo performance of an isolated task can not be the only metric to evaluate techniques based on neural networks or machine learning in general. Moreover, real-world programs involving CPS may include a wide variety of applications with different levels of complexity and dependence on the hardware for which they are developed. Together with the requirements of supervised learning to work with a large and varied set of labelled data, these considerations make it difficult to assess the level of generalisation guaranteed by the main works we have analysed. We observed from the results of the previous works that among all the considerations listed above, the major limitation is the usage of a limited dataset. From the state-of-the-art analysis conducted in previous sections (RQ1) on Corpus 2, we can conclude that early approaches are inspired by natural language processing: A sequence of tokens are extracted from the source code being analysed and combined to obtain a representation of the whole kernel.

Some tools analyse OpenCL code directly [35], or LLVM intermediate representation with token filtering, which proved to deliver more robust classification results [9]. Both recurrent and convolution layers were compared in their ability to extract features from a sequence of tokens [53] and results show how CNN-based analysis equalises or increases the accuracy of recurrent networks with a cheaper training procedure. Graph-based techniques appeared recently to embed advanced control and data-flow relationships in the language model. In [41,51], various flavours of this representation proved to be effective. Even if model accuracy is similar to the one obtained with token-based techniques, these program representations are so effective that even typical compiler analyses are solved successfully [51]. Among hybrid language-modelling tools, *inst2vec* [42] adapts the definition of statement context to source code and training statement embedding using the skip-gram [59] model. In contrast, *IR2vec* [50] proposes a technique to hierarchically build instruction, basic block, function and program embedding starting from few LLVM entities.

At the end of this analysis, it is evident that the first challenge to adopting deep-learning-based source code analysis techniques for heterogeneous device mapping is not related to the complexity of the deep learning model. Still, it is related to the large amount of labelled data (i.e., source code sequences) required to train models, considering the lack of publicly available labelled (in performance metrics in heterogeneous architectures) datasets. All methodologies described in Section 4 compared with each other using the same dataset detailed in Section 3 that is relatively small and consider only CPU and GPU out of the many kinds of accelerators cyber-physical system may have. Moreover, the relatively small size of the dataset, and the critical issues expressed in [10] concerning its structure and challenging analysis, does not make it possible to fully evaluate the potential of more complex code analysis methods based on graph representations (AST, CFG, and variants) and graph neural networks (GNNs). Additionally, that dataset is labelled considering runtime performance which is not the only metric of interest for cyber-physical system and in some scenarios may not be the most important. Furthermore, current methodologies do not consider that a cyber-physical system may feature multiple accelerators to choose from, which may, in principle, make the problem significantly more challenging. In fact, different devices are likely to have different parallelism models, which may not be easy to learn and discriminate for language modelling networks [52].

Tools discussed in Section 4 focus on kernel mapping. Still, none of them approaches a multi-tasking scenario where multiple active tasks need to synchronise or compete to gain access to the same subset of accelerators. This scenario would require a mapping tool to be aware of execution dependencies and eventually to handle variations in the execution environment to face situations where a task changes priority or is required to be scheduled more frequently.

Additionally, real-time tasks typical in cyber-physical system render the mapping problem more complex. Handling a set of real-time kernels, out of a pool of tasks, to be executed would require extract information about the time required for a kernel to process a given amount of data to ensure offloading a kernel to a device does not preempt other tasks from being executed with the requested timing.

Future directions can be summarised in the following points: (i) Creation or generation of new large datasets; (ii) Using techniques for dealing with scarcity of data; (iii) Improving the embedding techniques and language modelling; (iv) Look at more complex network topologies; (v) Optimising for different metrics at the same time (e.g., performance, timeliness, memory, energy, temperature); (vi) Focus on specific application structure (e.g., stencil computation, communication stack).

6. Conclusions

In recent years, heterogeneous systems have become increasingly relevant since integrating specialised hardware accelerators together with general-purpose CPU to offload computationally intensive tasks proved to be an effective strategy to overcome the shortcoming of the end of Dennard's scaling. From a software perspective, the increase in system performance comes at the cost of handling a more complex execution environment since choosing the best computation device to offload a task is not straightforward and may require advanced system-level knowledge and experience. The problem of heterogeneous device mapping has gained interest in recent years. It was approached by exploiting deep learning architectures to extract code features to predict the best device for a portion of a program.

In this work, we wanted to explore how deep learning methodologies have been applied to the source code (RQ1) through a historical review of specific and innovative works (Corpus 2—Table 1) that proved to introduce new algorithms or address new problems (Section 2). Section 3 analysed a specific set of works (Corpus 2—Table 3) that focused on a problem relevant to cyber-physical systems: the heterogeneous device mapping (RQ2). In Section 4, the works of Table 3 were described in detail, specifically the methods and techniques used. Finally, the results obtained on the same problem and dataset were collected (Table 5) and, in Section 4.4, compared (RQ3).

Despite the promising results shown by state-of-the-art methodologies in Section 2 and, more specifically, in Section 4, there is still a long way to go before considering these tools mature and ready for real-world applications.

All described works in Sections 3 and 4 address the only publicly available dataset with a heterogeneous device mapping problem. The dataset, described in Section 4, is not only small concerning what is usually employed in deep learning, but it considers only CPU and GPU on runtime performance. At the same time, as expressed in Section 5, heterogeneous platforms may feature multiple accelerators and seek energy or peak-power optimisations. An entire optimisation pipeline for the cyber-physical system application is still missing. Production-ready systems consist of multiple tasks where kernel offloading should be orchestrated with data collection and may be subjected to real-time constraints.

Author Contributions: All authors have equally contributed to the work. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Dipartimenti di Eccellenza funding programme of the Italian Ministry of University and Research.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data supporting this work can be found by following the references of the cited works.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract syntax tree
BLEU	Bilingual Evaluation Understudy
BoT	Bag of Tree
BoW	Bag of Word
BPE	Byte-Pair Encoding
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CNN	Convolutional Neural Network
CPS	Cyber-Physical Systems
CV	Computer Vision
DL	Deep Learning
DNN	Deep Neural Network
DT	Decision Tree
GGNN	Gated Graph Neural Networks
GRU	Gated Recurrent Unit (RNN)
GNN	Graph Neural Network
HAN	Hierarchical Attention Network
LSTM	Long short-term memory (RNN)
MDP	Markov Decision Process
ML	Machine Learning
MPNN	Message Passing Neural Network (GNN)
NLP	Natural Language Processing
RL	Reinforcement Learning
RNN	Recurrent Neural Network
Seq2Seq	Sequence to sequence
SIMD	Single instruction multiple data

References

1. Sztipanovits, J.; Koutsoukos, X.; Karsai, G.; Kottenstette, N.; Antsaklis, P.; Gupta, V.; Goodwine, B.; Baras, J.; Wang, S. Toward a Science of Cyber–Physical System Integration. *Proc. IEEE* **2012**, *100*, 29–44. [CrossRef]
2. Mittal, S.; Vetter, J.S. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* **2015**, *47*, 1–35. [CrossRef]
3. Fuchs, A.; Wentzlaff, D. The Accelerator Wall: Limits of Chip Specialization. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019; pp. 1–14. [CrossRef]
4. Pouyanfar, S.; Sadiq, S.; Yan, Y.; Tian, H.; Tao, Y.; Reyes, M.P.; Shyu, M.L.; Chen, S.C.; Iyengar, S.S. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–36. [CrossRef]
5. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
6. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–37. [CrossRef]
7. Ashouri, A.H.; Killian, W.; Cavazos, J.; Palermo, G.; Silvano, C. A survey on compiler autotuning using machine learning. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–42. [CrossRef]
8. Wang, Z.; O'Boyle, M. Machine learning in compiler optimization. *Proc. IEEE* **2018**, *106*, 1879–1901. [CrossRef]
9. Barchi, F.; Urgese, G.; Macii, E.; Acquaviva, A. Code mapping in heterogeneous platforms using deep learning and llvm-ir. In Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6.
10. Parisi, E.; Barchi, F.; Bartolini, A.; Acquaviva, A. Making the Most of Scarce Input Data in Deep Learning-based Source Code Classification for Heterogeneous Device Mapping. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **2021**, *41*, 1636–1648. [CrossRef]
11. Hindle, A.; Barr, E.T.; Gabel, M.; Su, Z.; Devanbu, P. On the naturalness of software. *Commun. ACM* **2016**, *59*, 122–131. [CrossRef]
12. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. Machine Learning on Source Code. Available online: https://ml4code.github.io (accessed on 12 December 2021).
13. Grewe, D.; Wang, Z.; O'Boyle, M.F. 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), Shenzhen, China, 23–27 February 2013; pp. 1–10.

14. Raychev, V.; Vechev, M.; Yahav, E. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; pp. 419–428.
15. Zaremba, W.; Sutskever, I. Learning to execute. *arXiv* **2014**, arXiv:1410.4615.
16. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, 7–12 August 2016; pp. 2073–2083.
17. Bhatia, S.; Singh, R. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv* **2016**, arXiv:1603.06129.
18. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
19. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 631–642.
20. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. *arXiv* **2017**, arXiv:1711.00740.
21. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944.
22. Santos, E.A.; Campbell, J.C.; Patel, D.; Hindle, A.; Amaral, J.N. Syntax and sensibility: Using language models to detect and correct syntax errors. In Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; pp. 311–322.
23. Bavishi, R.; Pradel, M.; Sen, K. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv* **2018**, arXiv:1809.05193.
24. Bui, N.D.; Jiang, L.; Yu, Y. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In Proceedings of the Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018.
25. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
26. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating Sequences from Structured Representations of Code. *arXiv* **2019**, arXiv:1808.01400.
27. Mendis, C.; Renda, A.; Amarasinghe, S.; Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 4505–4515.
28. Pradel, M.; Gousios, G.; Liu, J.; Chandra, S. Typewriter: Neural type prediction with search-based validation. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 8–13 November 2020; pp. 209–220.
29. Hoang, T.; Kang, H.J.; Lo, D.; Lawall, J. CC2vec: Distributed representations of code changes. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea, 6–11 July 2020; pp. 518–529.
30. Karampatsis, R.M.; Babii, H.; Robbes, R.; Sutton, C.; Janes, A. Big code != big vocabulary: Open-vocabulary models for source code. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea, 6–11 July 2020; pp. 1073–1085.
31. Haj-Ali, A.; Ahmed, N.K.; Willke, T.; Shao, Y.S.; Asanovic, K.; Stoica, I. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, 22–26 February 2020; pp. 242–255.
32. Brauckmann, A.; Goens, A.; Castrillon, J. A Reinforcement Learning Environment for Polyhedral Optimizations. *arXiv* **2021**, arXiv:2104.13732.
33. Allamanis, M.; Jackson-Flux, H.; Brockschmidt, M. Self-Supervised Bug Detection and Repair. *arXiv* **2021**, arXiv:2105.12787.
34. Stanley, K.O.; Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.* **2002**, *10*, 99–127. [CrossRef]
35. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. End-to-end deep learning of optimization heuristics. In Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, USA, 9–13 September 2017; pp. 219–232.
36. Quinlan, J.R. *C4. 5: Programs for Machine Learning*; Elsevier: Amsterdam, The Netherlands, 2014.
37. Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Frederickson, P.; Lasinski, T.; Schreiber, R.; et al. The Nas Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.* **1991**, *5*, 63–73. [CrossRef]
38. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE Trans. Neural Netw.* **2008**, *20*, 61–80. [CrossRef]
39. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. Bleu: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 7–12 July 2002; pp. 311–318.
40. Cho, K.; Van Merriënboer, B.; Bahdanau, D.; Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv* **2014**, arXiv:1409.1259.

41. Brauckmann, A.; Goens, A.; Ertel, S.; Castrillon, J. Compiler-Based Graph Representations for Deep Learning Models of Code. In Proceedings of the 29th International Conference on Compiler Construction, San Diego, CA, USA, 22–23 February 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 201–211. [CrossRef]
42. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *arXiv* **2018**, arXiv:1806.07336.
43. Allamanis, M.; Peng, H.; Sutton, C. A convolutional attention network for extreme summarization of source code. In Proceedings of the International Conference on Machine Learning, New York, NY, USA, 20–22 June 2016; pp. 2091–2100.
44. Tai, K.S.; Socher, R.; Manning, C.D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv* **2015**, arXiv:1503.00075.
45. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; pp. 5998–6008.
46. Di Biagio, A. llvm-mca: A Static Performance Analysis Tool. 2018. Available online: https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html (accessed on 20 June 2022).
47. Hirsh, I.; Stupp, G. Intel Architecture Code Analyzer. 2012. Available online: https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html (accessed on 20 June 2022).
48. Gage, P. A new algorithm for data compression. *C Users J.* **1994**, *12*, 23–38.
49. Grosser, T.; Groesslinger, A.; Lengauer, C. Polly–performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* **2012**, *22*, 1250010. [CrossRef]
50. VenkataKeerthy, S.; Aggarwal, R.; Jain, S.; Desarkar, M.S.; Upadrasta, R.; Srikant, Y. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim. (TACO)* **2020**, *17*, 1–27. [CrossRef]
51. Cummins, C.; Fisches, Z.V.; Ben-Nun, T.; Hoefler, T.; Leather, H. ProGraML—Graph-based Deep Learning for Program Optimization and Analysis. *arXiv* **2020**, arXiv:2003.10536.
52. Parisi, E.; Barchi, F.; Bartolini, A.; Tagliavini, G.; Acquaviva, A. Source Code Classification for Energy Efficiency in Parallel Ultra Low-Power Microcontrollers. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 878–883.
53. Barchi, F.; Parisi, E.; Urgese, G.; Ficarra, E.; Acquaviva, A. Exploration of Convolutional Neural Network models for source code classification. *Eng. Appl. Artif. Intell.* **2021**, *97*, 104075. [CrossRef]
54. Keerthy S, V.; Aggarwal, R.; Jain, S.; Desarkar, M.S.; Upadrista, R. IR2Vec: A Flow Analysis based Scalable Infrastructure for Program Encodings. *arXiv* **2019**, arXiv:1909.06228.
55. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. Learning representations by back-propagating errors. *Nature* **1986**, *323*, 533–536. [CrossRef]
56. LeCun, Y. Generalization and network design strategies. *Connect. Perspect.* **1989**, *19*, 143–155.
57. Yin, W.; Kann, K.; Yu, M.; Schütze, H. Comparative study of cnn and rnn for natural language processing. *arXiv* **2017**, arXiv:1702.01923.
58. Zhou, J.; Cui, G.; Hu, S.; Zhang, Z.; Yang, C.; Liu, Z.; Wang, L.; Li, C.; Sun, M. Graph neural networks: A review of methods and applications. *AI Open* **2020**, *1*, 57–81. [CrossRef]
59. Mikolov, T.; Le, Q.V.; Sutskever, I. Exploiting similarities among languages for machine translation. *arXiv* **2013**, arXiv:1309.4168.
60. Bordes, A.; Usunier, N.; Garcia-Duran, A.; Weston, J.; Yakhnenko, O. Translating embeddings for modeling multi-relational data. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 5–8 December 2013; Volume 26.