ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Cyber-Physical Systems Programming

LANGUAGE MODELLING OF SOURCE CODE USING MASKED GRAPH AUTOENCODERS AND GRAPH NEURAL NETWORKS

CANDIDATE
Federico Cichetti

SUPERVISOR
Prof. Andrea Acquaviva

CO-SUPERVISORS
Francesco Barchi, PhD.
Emanuele Parisi

Academic year 2022-2023

Session 3rd

To my 1-hop neighbours

Abstract

In a landscape of constant evolution of software and hardware, the mediating role played by compilers has become increasingly complex. Deep learning (DL)-based source code analysis has proven beneficial in supporting compile-time decisions that impact performance in heterogeneous devices. Graph-based representations of source code are particularly appealing, as they express code properties that would otherwise be challenging to identify. In this thesis, I develop DeepCodeGraph (DCG), a technique for constructing a general graph-based language model (LM), which learns patterns to identify better compilation strategies, optimal hardware configurations and software transformations. DCG includes: i) A large-scale dataset containing over 100 k graph-based representations of compilable source code files. ii) A Graph Neural Network (GNN) implementing a flexible graph-based LM. iii) A self-supervised training procedure based on the framework of Masked Graph AutoEncoding (MGAE), providing general and transferable knowledge to the LM. The performance of DCG is evaluated on two complex tasks: heterogeneous device mapping and thread block size prediction. DCG outperforms previous graph-based state-of-the-art approaches in all tasks, improving previous results by 3% and 5% respectively.

Contents

Chapter 1

Introduction

This work aims at exploring the language of source code from the lens of probabilistic modelling and optimisation techniques. It sets itself to answer three fundamental questions:

1. Is it possible for a learning algorithm to identify useful patterns in the way human programmers write software?

2. Is it possible that alternative representations of software can enhance the learning process?

3. Is it possible to utilise this general understanding of software for optimising the execution of programs?

This introductory chapter is divided into four sections. I introduce the concepts of hardware, software and compilers in Section 1.1. I explore how language modelling can be used for answering the research questions listed above in Section 1.2. I state the objective of this work and my contributions in Section 1.3. Lastly, I present the structure of the rest of the thesis in Section 1.4.

1.1 From Software to Hardware

Software is ubiquitous in modern life, powering smartphone operating systems, social media platform, streaming services, government digital systems and countless other applications that are used every day by millions of people. At its core, software is a sequence of instructions and data that can be executed on some physical hardware [1]. Despite the apparent simplicity of this concept, the interplay between software and hardware is not at all trivial.

The landscape of contemporary computing systems is vast and diverse, ranging from highly specialised embedded devices to versatile general-purpose computers, and from portable devices to high-performance supercomputers. Current hardware is the result of centuries of engineering endeavors and scientific and technological advancements [2].

Moreover, modern computers are frequently characterised as *heterogeneous* devices, featuring diverse hardware modules such as central processing units (CPUs), graphics processing units (GPUs), storage devices and more. Each of these components specialises in executing specific types of instructions, allowing software to leverage their capabilities for accelerated execution. For instance, tasks with high parallelisation potential, such as image rendering for graphical interfaces or complex mathematical calculations on tensors [3, 4], are often offloaded to GPUs thanks to their robust support for extensive parallel processing.

General-purpose hardware typically exposes a set of instructions it can execute (known as the Instruction Set Architecture, ISA), allowing programmers to arrange them freely to perform high-level tasks. However, as hardware has evolved, so too has the language used to express instructions. Over the years, several *programming languages* have been carefully designed and implemented by researchers and programmers alike, contributing to an overall increase in abstraction from the physical device. While early languages like

Assembly and (to some extent) C and C++ tended to have a strong correspondence between the functions they provided to developers and the underlying ISA instructions, contemporary high-level programming language such as Java[1] and Python[2] hide much of that complexity under the hood. Consequently, these languages have witnessed a surge in popularity, simplifying the development process for a broad spectrum of applications.

1.1.1 Compilers and Intermediate Representations

The translation from high-level languages to executable instructions is commonly delegated to a separate program, which, depending on the design of the source language, can take the form of a *compiler*, an *interpreter* or a *hybrid system* (such as Java's JIT compiler). While this work will focus on compiled languages, it's worth noting that many of the presented concepts can be applied more broadly.

Compilers serve not only as translators between source code and computer instructions, but also offer various additional functionalities. These include pre-processing, syntactic and semantic analysis, rewrital of portions of code into more efficient constructs, and more. In essence, they are able to produce an *optimised* sequence of instructions by applying a cascade of algorithms and heuristics to the source code and taking into account the specifics of the underlying hardware architecture.

A preliminary step for most compilers is to translate source code into an *Intermediate Representation* (IR). This is a low-level language that is internally employed by the compiler in order to facilitate further analyses. The IR encapsulates essential structural information about the source code, including syntactic aspects, *data flow* (the flow of usage of variables) and *control flow* (the flow of instructions and their possible branches).

Compilers that employ IRs are often designed around three stages:

[1] https://dev.java/
[2] https://www.python.org/

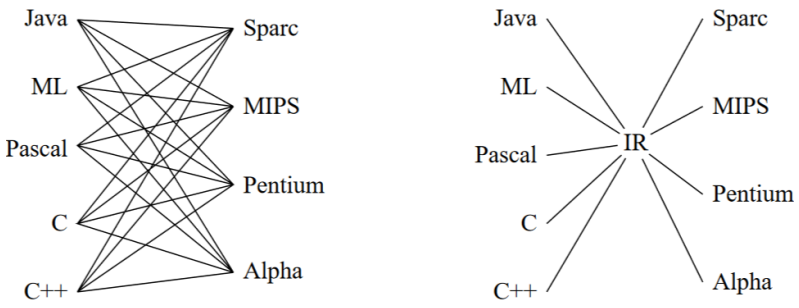


Figure 1.1: The reduction in complexity an IR can introduce in compilers by providing a central representation of source code. Image source: [5]

- The compiler's *front-ends*, managing the transformation from each specific programming language to a common IR.
- The *middle-end*, performing all optimisation steps on the IR code, usually exploiting the additional structural information. As IRs are both language- and device-agnostic, algorithms at this level only need to be implemented once. This characteristic is a compelling advantage of using IRs, as compilers would otherwise require an exponential number of variations to accommodate every possible combination of source languages and target architectures. Figure 1.1 shows the difference in the two paradigms.
- The *back-end*, translating the optimised IR to machine-specific instructions, making use of the properties of the underlying hardware.

Consequently, operating at the level of IRs could provide the advantage of abstracting away language-specific features that might otherwise heavily influence the learned patterns.

1.2 A Language Model of Code

Allamanis et al. [6] describe code as an inherently *bimodal* communication tool. It is the language that programmers employ to communicate with machines, but it can also be seen as a form of *human communication*, expressing

to other developers what one wants the computer to perform at a higher level. Hindle et al. [7] further argue that while programming languages are powerful and flexible tools, developers actually write in a simple and repetitive way, similarly to how people often communicate in a regular and predictable manner in real life scenarios.

The connection between software and natural language is summarised in the *naturalness hypothesis* [6], stating that *software corpora have similar statistical properties to natural language corpora*. This connection justifies the interest in the use of statistical modelling and Machine Learning (ML) or Deep Learning (DL) techniques that were previously deployed in the field of Natural Language Processing (NLP).

Contemporary NLP applications are often powered by Language Models (LMs). These are large parametric systems that learn probability distributions across an extensive collection of language artefacts. They may, for instance, model the likelihood of sentences in the English language: realistic, natural sentences should have a higher probability than unlikely or outright ungrammatical ones [8].

During the learning phase, the model is exposed to numerous examples and autonomously learns to recognise patterns for its designated task through the backpropagation algorithm [9]. The underlying assumption is that knowledge derived from extensive text corpora represents general patterns of natural language, equipping the model to handle uncertainty.

A recent rise in availability of open-source code (often referred to as *Big Code*) offers large corpora of well-engineered software that could be leveraged as training data for language models of source code. Analogously to natural language, the extraction of knowledge from these resources could allow LMs of source code to learn general coding patterns, enabling the implementation of smarter data-driven software engineering tools [10].

However, despite the similarities, the language of source code has rules, properties and patterns that set it apart from natural languages. For instance,

code is *executable* and *formal* [6], meaning that, in order to avoid inconsistencies, unambiguous syntactic and structural rules must be followed in writing. These properties also induce the structural patterns related to data and control flow that compilers are able to exploit. Effectively capturing these patterns while training a LM is the key to a deeper comprehension of code, enabling reasoning on high-level properties such as efficiency, security and maintainability.

In other words, while statistical properties between the two kinds of data are similar, research on LMs of source code should not be limited to the same architectures and representations that have been successful for natural language, as they may fail to fully capture the nuances inherent to code data.

1.3 Objectives and Contributions

In this work, I argue that some of the aforementioned subtleties of source code may be difficult to observe from raw textual representations and could be more easily detected and reasoned upon using different representations of code. The thesis is oriented towards leveraging the additional information introduced by IRs in order to construct graph-based representations. The motivation is that graphs could be better suited to depict structural information in source code and yield more robust and higher-level patterns compared to traditional textual representations.

My exploration in this field brings several contributions:

1. A comprehensive review of the existing literature on graph-based techniques for source code modelling and processing.

2. DeepCodeGraph (DCG), an effective methodology for training a powerful language model of graph representations of source code using the self-supervised learning framework of Masked Graph AutoEncoders (MGAE).

3. The DCG Dataset, an extensive and diverse collection of compilable source code, offering a vast repository of examples for the language model to learn useful patterns from.

Additionally, I discuss several alternative approaches to the components of the DCG methodology, and examine a range of potential options and configurations for the implemented models.

1.4 Thesis Structure

This work is structured as follows.

Chapter 2 provides a comprehensive overview of the relevant literature, establishing the theoretical foundation for this thesis. I review NLP-based approaches for problems related to processing of raw source code, and examine the possible alternative representations with an emphasis on graph-based methods. I also present a family of deep learning models that are designed for processing graph data. Subsequently, I discuss self-supervised learning techniques for graph-based models and describe some supervised tasks that are employed to evaluate the quality of this pipeline.

Chapter 3 outlines the set of tools and techniques that were utilised to accomplish the objectives of this research. I also introduce the DeepCode-Graph methodology and its components, which include a graph-based language model for graph representations of source code and the DCG Dataset. I provide a high-level overview of the entire pipeline, illustrating how all components interconnect, but I also delve into the implementation details to highlight the challenges that have been overcome.

Chapter 4 explores the results of my experiments, encompassing both the language model and task-specific trainings. Additionally, I provide a thorough analysis of the DCG dataset and of the experimental configurations.

Lastly, Chapter 5 concludes the work with some final considerations and proposes directions for future research in this field.

Chapter 2

Background and Related Works

The goal of this thesis is to create a language model of graph-based representations of source code. The LM should satisfy the following properties:

1. It should naturally incorporate reasoning on the structural aspects of source code.

2. Its knowledge should be general and high-level, independent from the syntax of specific source languages.

3. It should be capable of solving complex tasks that require deep comprehension of source code patterns.

The first requirement is satisfied by modelling source code using a *graph-based* representation. In this way, structural aspects of code, such as data and control dependencies, are explicitly modelled as connections between nodes. After a brief review of how traditional methods from NLP have been employed in previous works in Section 2.1, I explore alternative representations of source code in Section 2.2, focusing on graph-based representation in Section 2.2.2. Subsequently, I review contemporary methodologies for encoding and processing graph data with Deep Learning (DL) in Section 2.3.

The graph-based representations are built from the information contained at the level of compiler *intermediate representations* (IRs). In this way, the

syntax of high-level programming languages is avoided. A broadly popular IR is presented in Section 2.2.1, together with a selection of works that employed it successfully in previous research.

Furthermore, the LM is trained directly on the graph representations through the mechanism of Masked Graph AutoEncoders (MGAEs), learning general and high-level patterns of code and satisfying the second requirement. MGAEs and the broader family of self-supervised learning techniques on graph-based models are presented in Section 2.4.

Finally, in Section 2.5 I describe two tasks that should empirically validate the quality of the LM and that are commonly used as reference benchmarks: i) mapping computational kernels to compute units (heterogeneous device mapping) and ii) determining the most efficient thread block size for a GPU kernel (thread block size prediction).

2.1 Learning from Big Code

Big Code has sparked interest in many AI researchers. The expectation is that a large corpus of varied and well-engineered software should contain useful patterns that could be extracted with ML techniques in order to create data-driven models of code. More intelligent and effective software engineering tools could be achieved by employing the knowledge learnt from these models, rather than from complex hand-engineered rules and heuristics [10].

Common problems that can be tackled through the lens of Big Code are extensively reviewed in [6] and include:

- Code completion, which aims at suggesting the most likely continuation of a portion of code by predicting the developer's intent from preceding context.
- Code translation, whose goal is that of porting source code written in a source language to a different target language.

- Code-to-text and text-to-code tasks, which involve either creating a natural language description from a section of code or generating a code segment from a natural language comment.

The rise in popularity of machine learning and deep learning techniques has significantly impacted this branch of research, now oriented towards the development of techniques to convert software into semantically-rich representations and to learn effective patterns from them.

2.1.1 Natural Language Processing Techniques

As explained in Section 1.2, there is a connection between software and natural language motivating the application of traditional NLP techniques for the processing of source code. NLP techniques often include a pre-processing step where text is transformed into a sequence of elements that are manageable by a LM. Pre-processing involves the utilisation of normalisation techniques (e.g. stemming, stopwords removal, etc.), followed by tokenisation, the process of separating a text into smaller sub-units called *tokens*. Depending on the model design, tokens can be words, punctuation marks and, in some cases, sub-words or even single characters [11].

By applying a consistent pre-processing algorithm on a large corpus of text, a set containing all the distinct tokens found in the dataset can be extracted. This set is often referred to as the *vocabulary* of the corpus, and it induces a mapping from tokens to numerical indexes, such that any portion of text can be represented as a numerical sequence and manipulated by the model. Particularly rare tokens are usually filtered out to reduce the size of the vocabulary, and a special "unknown" token is used as a default option to map all text components that are not found in the vocabulary.

The indices attributed to tokens transport no information about their semantic value, so it's common to further map tokens to *feature vectors*, or *embeddings*. These vectors can express information of statistical relevance

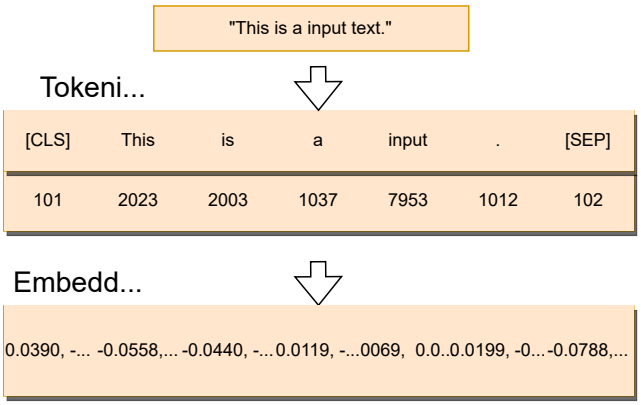


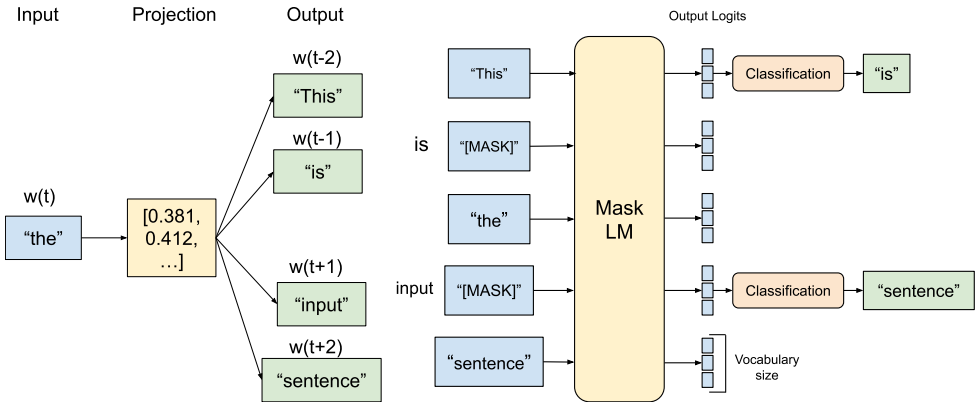
Figure 2.1: Representation of the tokenisation and embedding processes. Image source: vaclavkosar.com

for the token and allow the model to extract stronger patterns from data. A representation of this process can be seen in Fig. 2.1.

Traditionally, the main assumption for creating feature vectors is that tokens that appear often in similar contexts should be semantically "close" to each other. Older methods used *sparse* representations for words, resulting in large matrices that tallied the token instances in each document or their co-occurrences within a specified range. In contrast, modern approaches use Deep Learning architectures to learn *dense* parametric spaces representing token semantics.

These methodologies often belong to the *self-supervised* learning category, where the lack of additional labels forces models to learn directly from text itself. Word2Vec [12] is an influential approach where the embedding space is learned by predicting the probability distribution of tokens neighbouring a specific "context" token through a simple neural network. This is often referred to as *skip-gram* architecture (see Fig. 2.2a).

The prediction task is only needed to progressively build stronger and more expressive embeddings. As the representations start to encode more and more valuable knowledge derived from real-world usage of language, the model improves at predicting the contexts of these words. This framework is powerful enough to induce analogies in terms of vector arithmetic within the



(a) Skip-gram methodology. (b) Masked Language Modelling methodology.

Figure 2.2: Representation of common training methodologies in NLP (Skip-gram and MLM).

embedding space (e.g., “Rome is to Italy as Paris is to France”).

Over the years, the standard language modelling task has evolved towards predicting the appropriate subsequent token given a set, or all preceding ones. The expected output is a probability distribution over the vocabulary: $P(w_i|w_{i-k},\ldots,w_{i-1},\theta)$. The problem with skip-gram is that it only allows limited context, but architectures working on sequential data, such as Long Short Term Memory cells (LSTMs) [13], Convolutional Neural Networks (CNNs) or Transformers [14], can effectively model a long input sequence and generate an embedding of context to condition the probability distribution for the next token.

This methodology is used, for instance, by Radford et al. [15], or by Peters et al. [16], which employ bi-directional language modelling predicting token distributions also on the right-to-left sequence.

One technique that particularly stands out is Masked Language Modelling (MLM) popularised by the Transformer-based architecture BERT [17]. Transformers inherently operate bi-directionally, evaluating the importance of connections between all possible pairs of tokens in the sequence. For this reason, the full potential of Transformers cannot be exploited with traditional left-to-right or right-to-left language models, as they would immediately have

knowledge of the following words.

Instead, in MLM a percentage of the tokens are randomly masked out (e.g. replaced with a `[MASK]` token) and the model is tasked to predict the original tokens as a classification task. Since the `[MASK]` token is not present during inference, to avoid a mismatch between training and testing, in a small percentage of cases the chosen tokens are not masked, but replaced with a random token from the vocabulary or kept the same. Fig. 2.2b shows a representation of MLM.

The MLM framework is the basis for this work, and Section 2.4 will elaborate on how masking can be adapted for graph data.

2.1.2 Raw Source Code Processing

Several works have applied NLP techniques to source code, employing pre-processing methods to transform code elements into sequences of tokens and embeddings, as previously described. Pioneering works by Hindle et al. [7] and Allamanis et al. [10] first applied language modelling techniques to large collections of source code, although the introduction of deep learning brought significant improvements to this field.

More recently, Cummins et al. [18] proposed a pipeline for normalising OpenCL files, which includes the removal of macros and comments, as well as the replacement of user-defined variable and function identifiers with letters coming from a sequential alphabetical series. A coding style is also enforced for all files, so that there is no variance in white-spaces and parenthesis placement. This process of code normalisation aims at eliminating trivial semantic differences in source code, such as the choice of variable and function names. After the normalisation, a LSTM-based network [13] is trained on the sequence of characters, effectively creating a character-level LM of OpenCL code.

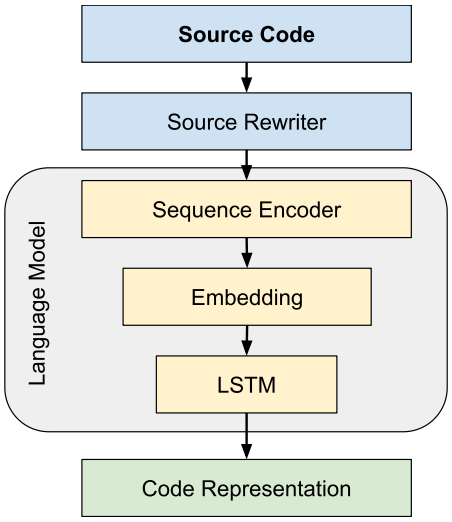


Figure 2.3: Diagram of the DeepTune model [19].

An extension of this work [19] observes that the extraction of character-based sequences reduces the size of vocabularies, but generates very long sequences and complicates the pattern extraction task. Therefore, the authors propose to construct a vocabulary of OpenCL *tokens* by enhancing the above pre-processing phase:

- The source code is normalised and fed to the LLVM compiler, which, as part of its processing pipeline, builds a tree-based representation of the code syntax: the Abstract Syntax Tree (AST). This structure will be described in detail in Section 2.2.2.
- The code is re-written from scratch by parsing the AST and enforcing a specific code style and identifier naming scheme.
- A hybrid tokenizer is utilised to represent common multi-character sequences (e.g. `float`, `if`) as single tokens and infrequent words as character-level sequences of tokens.

The vocabulary for this approach is composed of 128 symbols that are found to fully encode 45k-lines of OpenCL code. Each of these tokens is connected through an Embedding Layer to a dense representation that is learnt jointly with the rest of the LM. A neural network composed of 2 LSTM layers is employed

to extract a final dense semantic representation of the whole sequence. A diagram for this model can be seen in Fig. 2.3.

This simple architecture (also known as *DeepTune*) has become a widely popular LM of OpenCL code and was recently enhanced by Vavaroutsos et al. [20] to correct some of its criticalities. First, they add a Convolutional Layer (CNN) on top of the initial embeddings, enabling the model to take into account the adjacency relations between tokens which would otherwise be unused information. Then, they propose to use Bidirectional LSTM layers (Bi-LSTM) so that both past and future information can be taken into account, since the flow of imperative programming often has cases where instantiated variables and functions are used much later in the code. Finally, they argue that while DeepTune gives the same importance to all tokens, some semantic aspects are more critical than others (e.g. a variable declaration is not as important as the beginning of a `for` cycle): therefore they append an Attention layer after the Bi-LSTM which learns to emphasise and reweigh the given information before extracting a global contextual representation.

Drawbacks of Learning from Raw Source Code

The presented approaches work at the level of source code and build a representation of programs that is directly based on how developers write software. However, these models cannot exploit the additional information provided by IRs built during the compilation process. Additionally, they are often trained on code written in a single programming language (e.g. Java or OpenCL) and are therefore not able to generalise to general properties of source code. In order to build a general model in this fashion, either an extremely large vocabulary that represents tokens of all languages, or several language-specific trainings would be necessary.

Instead, human developers are often polyglot and able to switch between languages with relative ease. This is due to the fact that programming languages, being *formal* languages, stem from the common ground of *logic*.

Apart from language-specific syntax and structures, there are underlying patterns that are common to all programming languages and can be exploited for building a more general model of code. In the following section, I will explore various alternative methods for representing code that inherently integrate such patterns.

2.2 Alternative Representations of Code

During the compilation and optimisation process, raw source code is transformed multiple times into both textual and non-textual representations. Some of the algorithms used by compilers may indeed exploit additional information that is only possible to represent in a custom intermediate language, or even through graph representations. These are often designed to let rules, properties and patterns of raw source code emerge in an explicit manner. In this section, I will analyse some of these representations, as well as some alternatives which have been built specifically for ML algorithms.

2.2.1 LLVM Intermediate Representation

LLVM began as a research project [21] at the University of Illinois and has expanded into a full code optimisation suite with sub-projects such as the popular C/C++ compiler Clang [22] and many other related tools. At the core of the project lies the powerful LLVM Intermediate Representation (LLVM-IR), which is a lightweight, low-level but flexible and human-readable IR.

The representation is language-agnostic and designed to work in a complementary way to the high-level front-ends of specific languages [21]. Many popular compilers such as GCC[1] and NVCC[2] optionally output LLVM IR code that can be later optimised within the LLVM framework. The compiler

[1] https://gcc.gnu.org/

[2] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

```c
int fib(int n){
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
; Function Attrs: nofree nosync nounwind readnone uwtable
define dso_local i32 @fib(i32 noundef %0) local_unnamed_addr #0 {
  %2 = icmp slt i32 %0, 2
  br i1 %2, label %11, label %3

3:                                                ; preds = %1, %3
  %4 = phi i32 [ %8, %3 ], [ %0, %1 ]
  %5 = phi i32 [ %9, %3 ], [ 0, %1 ]
  %6 = add nsw i32 %4, -1
  %7 = tail call i32 @fib(i32 noundef %6)
  %8 = add nsw i32 %4, -2
  %9 = add nsw i32 %7, %5
  %10 = icmp ult i32 %4, 4
  br i1 %10, label %11, label %3

11:                                               ; preds = %3, %1
  %12 = phi i32 [ 0, %1 ], [ %9, %3 ]
  %13 = phi i32 [ %0, %1 ], [ %8, %3 ]
  %14 = add nsw i32 %13, %12
  ret i32 %14
}
```

Figure 2.4: Snippets of code for a simple function written in C and its LLVM-IR output (compiled with optimisation level 3).

of modern programming language Rust[3] is also a front-end for LLVM and can produce LLVM-IR code for whole projects. Many other languages, including Scala and Python, have ongoing projects aimed at producing LLVM-IR code for portions of code or full projects[4][5].

The LLVM-IR instruction set is aimed at capturing the key operations of ordinary processors, but avoids machine-specific constraints such as physical registers and low-level calling conventions [21]. Instead, it provides an infinite set of *typed* virtual registers, supporting a language-independent type system, including a void type, booleans, integers and single and double precision floating-points. The IR also includes derived types such as pointers, structures, arrays and functions. The virtual registers are assigned in Static Single Assignment (SSA) form [23], which essentially ensures that every variable is assigned only once, making it easy to track the *data flow*. LLVM-IR also defines ϕ-expressions, enumerating all possible outcomes leading to a variable. This information is useful for *control flow* analysis and optimisation.

The instruction set includes type-dependent arithmetic operations, instructions for pointer arithmetics, as well as explicit memory-related operations

[3]https://www.rust-lang.org/
[4]https://days2011.scala-lang.org/sites/days2011/files/ws3-2-scalallvm.pdf
[5]https://github.com/numba/llvmlite

(e.g. `load`, `store`, `alloca`) operations for tracking memory accesses and allocation [21].

In conclusion, LLVM-IR is particularly fit for program analysis. Not only it defines a rich and language-independent instruction set, making it easy to represent high-level patterns that are common to different source languages, but the addition of explicit data and control information elevates its representative power despite its purely textual form. Figure 2.4 shows a simple C program and its respective LLVM-IR output after compilation.

NLP Techniques on LLVM-IR

Given the increased representative power and the possibility to create LMs on source code that do not depend on a specific language, several works started to process LLVM-IR code rather than high-level languages, following the trend described in Section 2.1.2.

Barchi et al. [24] use a dataset of OpenCL code compiled with *clang* [22]. They define a complex input processing step, which is employed to clean and standardise the LLVM-IR code before the tokenisation phase (see Fig. 2.5). The model they use in this work bears many similarities with DeepTune, but by using LLVM-IR it can analyse code that has already been partially optimised and is therefore less sensible to noise (e.g. unused variables, unreachable code fragments, ...). They also point out that LLVM-IR is often more verbose than high-level code, so tokenised sequences can be further reduced by selecting a blacklist of tokens with low informative power in the vocabulary (e.g. through tf-idf [25]) that are simply discarded. The same processing technique is used in one of their subsequent works, DeepLLVM [26], where the LSTM layers are replaced with a 1D CNN followed by a max-pooling layer, greatly reducing the number of parameters and thus improving training and inference speed.

Ben-Nun et al. [27] train an embedding space of LLVM-IR instructions. The vocabulary is extracted after pre-processing a large corpus of LLVM-IR code compiled from a set of C, C++, FORTRAN and OpenCL benchmarks.

LLVM-IR Code Fragment

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

Tokenization

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

Figure 2.5: Possible tokenisation of LLVM-IR source code. To reduce the vocabulary size, vectors, arrays and float constants are replaced with placeholders. Image source: [24].

Pre-processing steps are similar to those shown in Fig. 2.5, including replacement of identifiers and immediate values with a placeholder, or with tokens reflecting variable types. A special graph structure (XFGs, described in Section 2.2.2) is extracted from LLVM-IR files in order to represent the data and control dependencies within instructions. Following the *skip-gram* methodology [12], they subsequently train an embedding space, where a model learns to produce expressive representations such that pairs of statements that frequently occur in the same context (i.e., tend to be connected by a dependency) are assigned a greater score than unrelated pairs. The embedding space is called *inst2vec* and exhibits promising qualities, such as strong clustering properties for instructions and the ability to satisfy analogies based on data types and instruction semantics.

2.2.2 Graph-Based Representations

The direct application of NLP-based techniques on LLVM-IR code has been successful. However, recent works related to graph-based representations of code have achieved promising results. In the context of code probabilistic modelling, graphs offer the unique opportunity to provide an alternative view of programs, where high-level properties such as the aforementioned data and

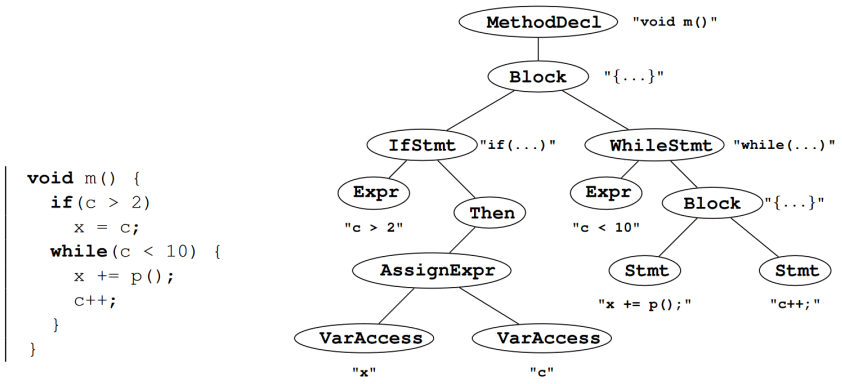


Figure 2.6: Java code sample with its corresponding AST representation. Image source: [28].

control flows are rendered in a clear, explicit manner as relations between nodes.

ML algorithms can greatly benefit from these representations. For instance, a common problem in dealing with sequential data is that of long-term dependencies. It is particularly important to be able to represent these kinds of dependencies for source code, to avoid problems such as losing track of variables defined long before their usage, or not being able to correctly match cases of a large `switch`. Graph representations instead allow the explicit representation of the connections between variables and instructions, or `switch` and `case`s, independently from the distance in source code.

In the rest of this Section, I will review promising graph-based representations that have been employed in recent research works.

Abstract Syntax Trees (ASTs)

Abstract Syntax Trees (ASTs) are tree representations of the syntactic structure of text written in a formal language. In compilers, they are often employed for syntax analysis: nodes correspond to code constructs such as statements, declarations, expressions, types and so on. Fig. 2.6 shows a sample method written in Java and a possible AST representation.

The advantage of using ASTs is that they do not require complex semantic

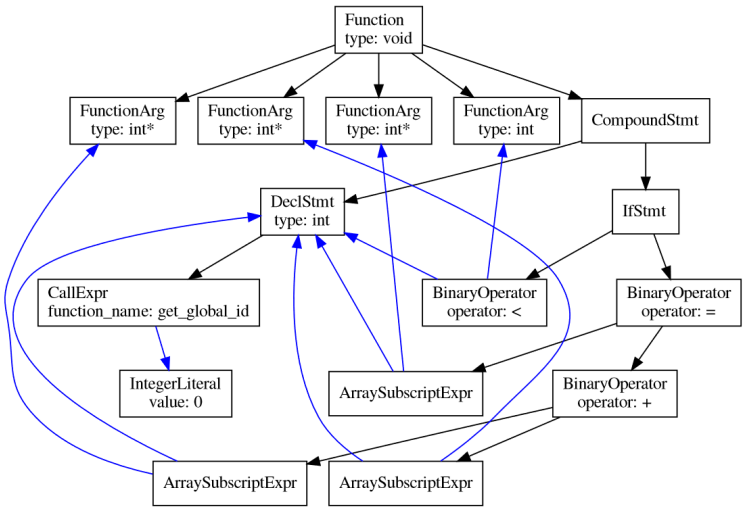


Figure 2.7: Dataflow-enhanced ASTs following the method of Brauckmann et al. Image source: [30].

analyses from the compiler, as they purely examine programs based on their grammar. However, many works have managed to build semantic representations of code snippets using regularities within the AST representation of large amounts of programs. For instance, Alon et al. built *code2vec* [29], a system that produces dense embeddings of entire source code snippets by decomposing code to a collection of paths in its AST. Paths are sequences of nodes from one leaf node to another. As a first processing phase, over 1M frequent paths were extracted from the training set to build a vocabulary; then, at training time, embedding vectors for random AST paths sampled from the training functions are learnt through a word2vec-like methodology. An attention mechanism is used to aggregate the different path embeddings, assigning different weights to the sampled paths.

Allamanis et al. [31] and Brauckmann et al. [30] avoid word2vec-like approaches and directly employ Graph Neural Networks (GNNs), special models that can explicitly process graphs and will be presented in Section 2.3. Both works use GNNs to build semantic representations of code from ASTs extensions, where all nodes referring to the same datum are connected by labelled edges (ASTs do not otherwise have identifier strings, so all arguments

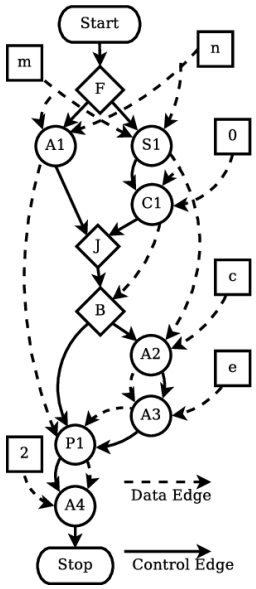


Figure 2.8: CDFG representation of a portion of code. Image source: [32].

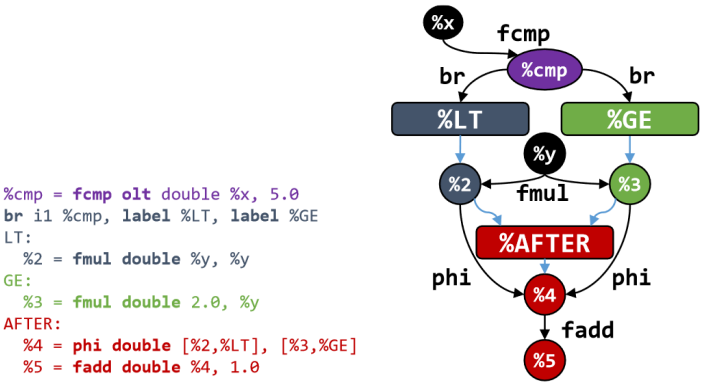
to a function are indistinguishable).

The final AST representing the code is therefore a *labelled directed graph*, where nodes are tagged as either Declarations, Statements or Types, while edges represent the syntactic *child-of* relationships of ASTs or the *use-def* relationships of data flow. Figure 2.7 shows a dataflow-enhanced AST, with the blue edges being the additional edges with respect to the classic structure.

Control and Data Flow Graphs (CDFGs)

Control and Data Flow Graphs (CDFGs) model both data and control operations. Code segments with no conditionals (only having one entry and one exit point) are known as *basic blocks*. They can be represented with a simple *data flow graph*, connecting variables to instructions that use them. Adding control structures, linking basic blocks and expressing the flow of execution and the dependencies of decision points, create a full CDFG. Data flow graphs can be seen as sub-graphs within the full CDFG: therefore it can be said that this structure is a hierarchical representation of code [33]. Figure 2.8 shows a CDFG, highlighting the difference between data and control edges.

Brauckmann et al. [30] used an enhanced version of CDFGs. Nodes



(a) LLVM-IR sample. (b) Contextual Flow Graph

Figure 2.9: LLVM-IR code sample with its corresponding XFG representation. Images source: [27].

correspond to LLVM-IR instructions and can be linked by *Control* and *Data* edges, representing the two kinds of dependencies of standard CDFGs. The authors add *Call* edges, representing function calls and return values, and *Mem* edges for store-load memory accesses.

ConteXtual Flow Graphs (XFGs)

ConteXtual Flow Graphs (XFGs) were introduced by Ben-Nun et al. [27], where the authors re-interpret the Distributional Hypothesis of language (*words that occur in the same contexts tend to have similar meanings*) [34] in terms of code: *statements* that occur in the same *contexts* tend to have *similar semantics*. In their interpretation, *statements* are LLVM-IR instructions, *similarity* has to do with modifying the system in similar ways or consuming similar resources, while *context* has to do with sets of statements whose execution depends directly on each other. This dependence is identified in data and control dependencies, leading to the XFG formulation being the chosen expression of "context".

In XFGs, nodes either represent *variables* or *label identifiers*, where a label can be a function or an entire basic block. Edges represent either data dependencies (labelled with the LLVM-IR instruction they carry) or control

dependencies. Figure 2.9 shows an example XFG.

IR2Vec

Similarly to inst2vec and code2vec, IR2Vec [35] is an encoding algorithm that aims at representing programs in an embedding space. The core difference with respect to prior work is that the representation is *hierarchical*. The authors learn *seed embeddings*, representing the semantics of the core entities that make up LLVM-IR instructions. These are aggregated to compose *instruction embeddings*, which in turn influence the construction of *function* and then *program-level embeddings*. The hierarchical aggregation operation is the result of flow analysis and heuristics rather than a learning process.

To describe an instruction, IR2Vec uses a collection of knowledge graph triplets in the form $\langle h, r, t \rangle$, where h and t are entities and r the relation connecting them. IR2Vec has a limited set of 64 possible entities that include all LLVM-IR opcodes, basic types (int, float, ...) and represent variables, pointers, functions and basic blocks as placeholders. The relations include: i) `TypeOf`, connecting each entity to its type. ii) `NextInst`, connecting each instruction to the following one. iii) $\texttt{Arg}_{1,\ldots,n}$, linking the instruction arguments to the opcodes and defining their order.

ProGraML

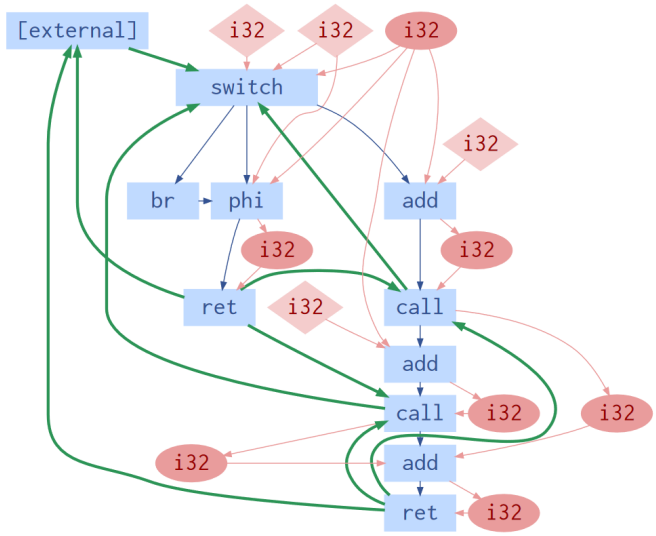
ProGraML [36] is a graph-based and compiler-agnostic representation of programs that models instructions, constants and variables as nodes and is able to capture control, data and call relationships among them.

A ProGraML graph is built in three stages:

1. *Control Flow*: all nodes representing instructions (e.g. in LLVM-IR or XLA [37]) are added into the graph and connected by edges of type "control". These edges contain information about the numeric position of successors (e.g. indexing the different branches in a `switch`

```
define i32 @Fib(i32) #0 {
   switch i32 %0, label %3 [
     i32 0, label %9
     i32 1, label %2
   ]
; <label>:2:
   br label %9
; <label>:3:
   %4 = add nsw i32 %0, -1
   %5 = tail call i32 @Fib(i32 %4)
   %6 = add nsw i32 %0, -2
   %7 = tail call i32 @Fib(i32 %6)
   %8 = add nsw i32 %7, %5
   ret i32 %8
; <label>:9:
   %10 = phi i32 [1, %2], [%0, %1]
   ret i32 %10
}
```

(a) LLVM-IR sample.



(b) ProGraML representation.

Figure 2.10: LLVM-IR sample with its corresponding ProGraML representation. Red edges represent data flow, blue edges represent control flow, green edges represent call flow. Image source: [36].

statement).

2. *Data Flow*: constant values and variables are added as nodes, with edges of type "data" connecting them to the instructions that use them as operands, or linking the instructions to the variables they produce. Data edges are augmented with positional information expressing the order of operands in an instruction. For instance, this mechanism allows to distinguish the first operand from the second in a subtraction operation.

3. *Call Flow*: edges of type "call" are added between function calls and the entry instruction for that function, or between return instructions and the caller. External calls are represented by adding a single new vertex `[external]`: this can be seen as a critical point, as externally allocated resources and instructions are not fully tracked, potentially hiding much of the complexity of a program.

In contrast to XFGs, which lack control edges, and CDFGs, which omit variable and constant types while also disregarding operand ordering, ProGraML

incorporates all of this additional information. Figure 2.10 shows an example transformation from a LLVM-IR sample to its ProGraML graph.

2.3 Deep Learning on Graphs

Graphs are a general concept: they naturally model many real-world phenomena that can be expressed as a matter of *relations* between *entities*. For instance, the KONECT project [38] collects over $1,300$ graph datasets in 24 different categories (connections in social networks, user-to-movie or user-to-product ratings, word occurrences in sets of text, ...). However, unlike other types of data, graphs require specialised methods and representations that put connectivity information at the forefront and allow spatial-based reasoning.

2.3.1 Graphs Representation

Formally, $G = (V, E, u)$ denotes a graph as a tuple of sets:

- V is the set of vertices (or nodes). If node v is represented by a feature vector h_v (e.g. a list of properties or a dense embedding that represents its semantic aspect), then V can be expressed as a matrix by stacking all node features on $|V|$ rows.
- E is the set of edges (or links). It could also be represented as a $|V| \times |V|$ *adjacency matrix* containing binary adjacency information: for each $v, w \in V$, $E_{v,w} = 1$ if there exist an edge connecting node v to node w; $E_{v,w} = 0$ otherwise.
- u is an optional global context, containing useful graph-level information such as the number of nodes, the length of the longest path, or even a learnable representation of high-level graph properties.

While it's common to represent E as a binary adjacency matrix in formal representations of graphs, storing information for all of the disconnected nodes

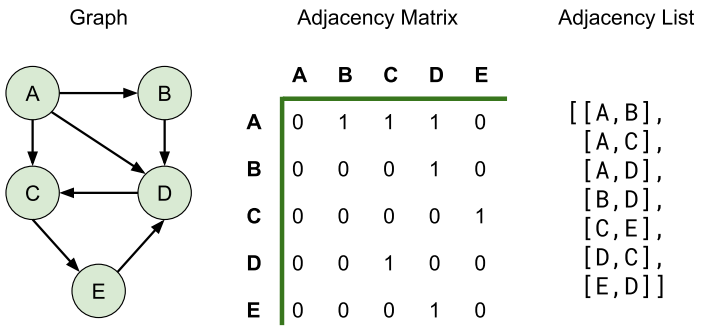


Figure 2.11: Example of a graph, its adjacency matrix and adjacency list.

(i.e. the zeroes in the matrix) is inefficient in practical tasks. A subtler problem is that in many real-world applications, nodes do not have an inherent order, so any permutation of rows and columns in an adjacency matrix is valid and functionally equivalent. However, there is no guarantee that deep neural networks would be *permutation invariant* (i.e. produce the same results on permutations of the same adjacency matrix) [39].

Since forcing one among the many possible permutation matrices may lead to suboptimal results, it's generally preferred to use representations that preserve *graph symmetries*, such as *adjacency lists*. Rather than rows in a matrix, each edge e is encoded as a tuple containing the start and end nodes of the connection: $e = (v, w)$. E thus becomes a *set of tuples*. In the worst-case scenario (i.e. complete graphs with self-loops) the adjacency list has as many entries as the adjacency matrix. However, this representation often saves space in memory and, most importantly, makes the adjacency information *permutation invariant*, since edges directly reference the related nodes as *nodes* rather than referencing them as elements within an ordered structure. The difference between the two connectivity representations can be seen in Fig. 2.11.

Edges may also carry attributes expressing properties of the connections. A matrix of *edge attributes* J having $|E|$ rows can be constructed by stacking edge features $j_{v,w}$ for each edge $e = (v, w)$, analogously to node features in matrix V.

2.3.2 Graph Neural Networks

This formalisation of graphs lends itself particularly well to deep learning algorithms. In terms of modern neural networks, both node and edge attributes can trivially be represented using 2D tensors. However, designing a mechanism to take into account the connectivity information is non-trivial.

Message Passing Neural Networks

Most GNNs are based on the Message Passing Neural Network (MPNN) framework [40], a paradigm that iteratively builds graph representations by local aggregation. In this configuration, GNNs are "*graph-in, graph-out*" models [39], meaning that V, J and u are progressively transformed without ever changing the connectivity information.

Under the MPNN framework, there exist 2 distinct processing steps: i) the *message passing phase*, where node embeddings are updated through "messages" coming from their neighbourhoods, and ii) the *readout phase*, where a global output is generated by aggregating all the available information. The message passing phase is repeated for T time steps. Three operations are executed at each time step t:

1. *Message creation*: The existence of a function $N(v)$ that returns the neighbours of a node is assumed. A function M_t is used on each node v to create a message based on the current state of the node h_v^t, the current states of its neighbours and the features of the edge between them.

2. *Message aggregation*: Messages are aggregated from all neighbours (e.g. by sum), creating a unified representation of all messages sent to the node. Operations 1 and 2 can be formalised as follows:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t \left(h_v^t, h_w^t, j_{v,w} \right). \tag{2.1}$$

3. *Update*: A function U_t is employed to update the current state of the

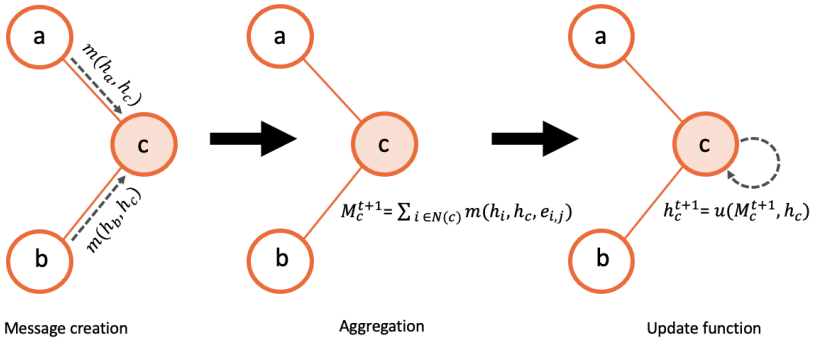


Figure 2.12: An example of the message passing mechanism. Image source: [41].

node h_v^t using the aggregated message:

$$h_v^{t+1} = U_t\left(h_v^t, m_v^{t+1}\right). \tag{2.2}$$

After T message passing cycles, a *read-out function* R computes a representation of the whole graph, if needed:

$$\hat{G} = R\left(\left\{h_v^T | v \in V\right\}\right). \tag{2.3}$$

An example of the message passing mechanism can be seen in Fig. 2.1.

An important observation to be made about this mechanism is that stacking T message passing layers allows information to propagate at most T steps away [39]. For instance, at time step $t = 2$ the neighbours of node v will have received information about their own neighbours, meaning that new messages directed to v will also include information from its 2-hop neighbours, and so on.

MPNN Variants

The general MPNN framework has been extended in multiple ways depending on the task, the type of graph at hand and the kind of communication that is needed. One of many examples is the Graph Nets Layer by Battaglia et

al. [42], that also updates edge and global features during a message passing iteration.

Furthermore, some variants are commonly employed components in the design of neural solutions. In Graph Attention Networks (GATs) [43], message aggregation becomes a self-attention operation over the features of each node's neighbours:

$$h_v^{t+1} = \Big\|_{k=1}^{K} \sigma\left(\sum_{w \in N(v)} \alpha_{v,w} W^k h_w^t\right) \tag{2.4}$$

where $k = 1, \ldots, K$ are multiple heads performing the attention operation in parallel, w is a neighbour of v and $\alpha_{v,w}$ is the attention weight given from v to the features of w, computed as:

$$\alpha_{v,w} = \text{softmax}_{w \in N(v)} \left(\text{MLP}\left(Wh_v, Wh_w\right)\right)$$

Another popular choice is that of Graph Isomorphism Networks (GINs) [44]. The inspiration for this network comes from the Weisfeiler-Lehman test [45], an efficient heuristic that checks whether two graphs are *isomorphic* (have identical connections but a permutation of nodes). Indeed, an ideal property of GNNs is that isomorphic graphs should be mapped to the same representation, while non-isomorphic graphs should be mapped to different ones. GINs theoretically satisfy this property and update node representations as follows:

$$h_v^{t+1} = \text{MLP}\left((1 + \epsilon) \cdot h_v^t + \sum_{w \in N(v)} h_w^t\right) \tag{2.5}$$

where ϵ is a learnable parameter determining the importance of the target node compared to its neighbours. Graph-level embeddings are then composed by concatenating the node embedding sums at different levels of processing:

$$h_G = \sum_{v \in V} h_v^0 \,\Big\|\, \ldots \,\Big\|\, \sum_{v \in V} h_v^T.$$

2.4 Self-Supervised Learning on Graphs

Self-Supervised Learning (SSL) allows the extraction of informative knowledge from data without the need for expensive manual labelling. Carefully constructed *pretext* learning tasks are instead employed to obtain feedback signals from the input data itself [46]. Models in the field of NLP and Computer Vision (CV) have achieved outstanding results thanks to self-supervised pre-training on a large amount of data followed by fine-tuning on smaller labelled datasets for specific tasks [17, 47].

Recently, self-supervision has also been employed on graph data with positive results [48, 49, 50, 51]. However, while language and images are structurally regular data (e.g. language is *always* presented as a sequence of tokens and images are *always* presented as a grid-like composition of pixel values), structure in graphs brings an additional source of complexity. Therefore, pre-training could prepare models to handle this complexity and to better capture the intrinsic knowledge of relationships.

2.4.1 Self-Supervised Learning Methods

Self-supervised learning is a versatile concept applicable in various ways based on input data and tasks. A recent review of SSL methods for graph learning [46] categorizes them into three groups (also represented in Fig. 2.13):

- *Contrastive methods*: At their essence, these methods revolve around the generation of multiple *views* for each graph, each created by meticulously designed *data augmentation* techniques. Pairs of views are then processed by the model, with the goal of maximising the agreement between positive pairs (e.g. originating from the same graph instance) while minimising the agreement between negative pairs (e.g. originating from different graphs). These methods tend to focus on *inter-data similarities*, enhancing both correspondences and differences among graphs.

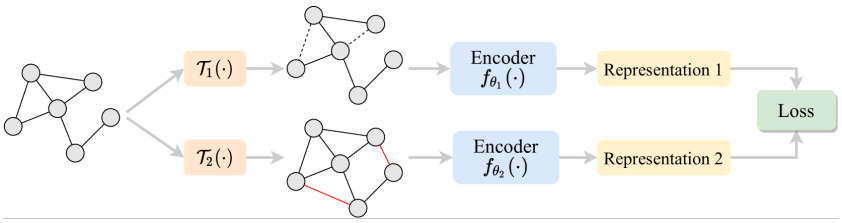
- *Generative methods*: These approaches typically employ encoder-decoder architectures, where the encoder maps input data into an embedding dimension, and the decoder attempts to reconstruct the original data from this embedding. Autoencoders and autoregressors are often the architectural choices for these methods. Their emphasis lies on *intra-data information*, optimising collaboration within neighbourhoods and the flow of information throughout processing steps.

- *Predictive methods*: These methods self-generate informative labels for supervision using objective node properties (e.g. node degrees), local or contextual information (e.g. shortest path between pairs of nodes), pseudo-labels (e.g. obtained from clustering) or knowledge obtained from specialised tools and algorithms. The self-supervised learning task then becomes a supervised task.

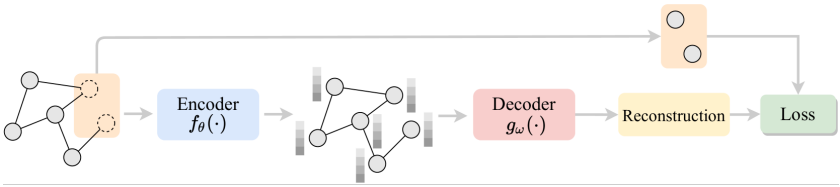
A common critique of contrastive methods is that the pre-training quality is greatly dependent on the design of data augmentations and negative sampling techniques, which can be time and resource-consuming [50, 46, 48]. Furthermore, the choice of which augmentations to apply (node or edge sampling, attribute shuffling and masking, subgraph extraction, ...) is usually a heuristic that takes into account what kind of information the graphs contain and what the final task is, leading to poor generalisation [52, 46].

Generative models can instead be further classified into graph *autoregressive* models or Graph *AutoEncoders* (GAEs). Autoregressive models try to generate a graph iteratively by decomposing the learnt distribution of graphs $p_\theta(G)$ into a product of temporal conditional distribution $p_\theta\left(g_t|G_{<t}\right)$ for time step t [53]. However, autoregressive modelling requires an arbitrary ordering of graph elements to be defined, while most graphs do not inherently have a canonical ordering.

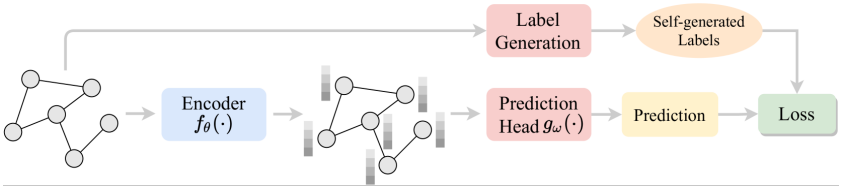
On the other hand, GAEs do not require an ordering and are based on the sound learning framework of autoencoders. Autoencoders are neural networks



(a) Contrastive method: maximisation of the agreement between two transformed views of the same graph.



(b) Generative method: the decoder learns to reconstruct parts of the input graph.



(c) Predictive method: some external component is able to generate a label for the input graph that is used as ground truth.

Figure 2.13: Differences between graph self-supervised learning methods. Image source: [46].

that are trained to attempt to copy their inputs to their outputs [54]. This methodology revolves around the utilisation of encoder-decoder architectures. The encoder $e(x)$ compresses input graphs into a latent dimension, producing an *hidden representation* h, while the decoder $d(h)$ tries to reconstruct the original graph from this embedding. The model is trained to maximise the similarity between $d(e(x))$ and x. Figure 2.14 shows a simple example of this framework.

2.4.2 Masked Graph Autoencoders

Early GAEs (e.g. Kipf et al. [55]) had several issues that prevented them to outperform contrastive methods. They have been shown to over-emphasise proximity information at the expense of structural information [50], excelling

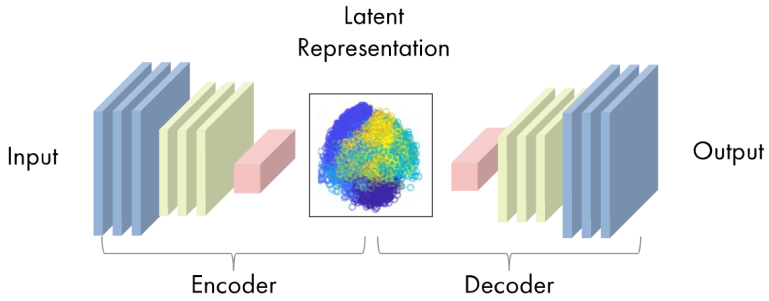


Figure 2.14: Example of an autoencoder network. Image source: [URL]

at tasks such as link prediction or node clustering, but with unsatisfactory results on higher-level tasks like graph classification. Furthermore, while autoencoders on their own can be powerful frameworks, there is a risk of learning trivial mappings (e.g. identity functions for both the encoder and the decoder).

For these reasons, some degree of *regularisation* is often employed during training in order to constrain encoders and decoders to focus on the key features that distinguish each input from the others, in turn creating stronger latent space representations. For instance, the latent dimensions might have a lower dimensionality than the original data, forcing the encoder to learn an optimal compression algorithm. Another popular approach is that of Denoising AutoEncoders (DAEs) [56], where some *noise* is introduced in the original data and the decoder is tasked with reconstructing the uncorrupted signal.

One particular form of noise is *masking*, the substitution of parts of the input with a special "mask signal" that hides the original elements. Masked AutoEncoders (MAEs) are special networks where an encoder computes representations of masked inputs and the decoder aims at reconstructing the original data. The framework achieved remarkable results in NLP [17], CV [57] and more recently even in GNN models.

While language tokens or image patches are typically the targets of masking and reconstruction in NLP and CV models, in graph data several different masking techniques can be employed. Some training strategies mask node

attributes [58, 48, 49, 51] while others prefer to mask parts of the graph structure [50, 59, 60], but both nodes and edges could be masked at the same time [53, 61] and additional information can even be used as an auxiliary target to stabilise the training.

Edge Masking Architectures

Edge masking GAEs partition E into two sets: the set of masked edges E_{mask} and the set of visible edges E_{vis}. A random process decides which edges are masked, often a Bernoulli trial where each node has the same probability of being chosen. Then, a graph encoder (e.g. a GNN) processes the graph $G = (V, E_{\text{vis}}, u)$ (i.e. ignores the masked edges). Finally, a decoder takes into account the representations of a source node h_v^T and a destination node h_w^T, predicting whether an edge connects them or not. The problem is often framed as a binary classification: edges in E_{mask} are the positive samples (i.e. pairs of nodes that the network should predict to be linked), while negative samples are chosen randomly among the set of non-connected pairs of nodes.

MaskGAE [50] adds some novel elements to this procedure. Firstly, the authors propose to replace the naïve Bernoulli process for the selection of E_{mask} with a *path-wise* random masking, where all edges in a set of sampled *random walks* are masked. The authors claim that this masking paradigm breaks the short-range connection between nodes, forcing the model to better exploit patterns in the graph structure and capture more meaningful proximity information. Secondly, they add an auxiliary decoder that predicts the degree (number of incident edges) of each node. This component balances the local information needed for degree prediction with the higher-order information needed for path reconstruction. Figure 2.15 shows the components of the framework.

MGAE [59] works in a similar manner, but introduces an enhanced edge reconstruction decoder. Since the structure of graphs is partially masked, h_v^T and h_w^T will inevitably be noisy representations and therefore hard to use for

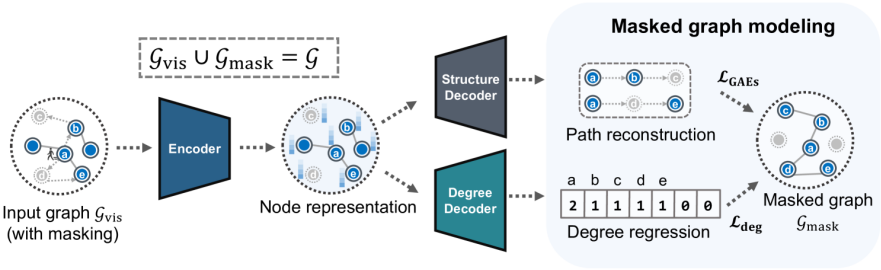


Figure 2.15: Overview of MaskGAE framework. Image source: [50]

reconstruction. Instead, by letting the decoder capture the cross-correlation of node embeddings at different granularities, the decoder could focus on common properties and discard discrepant information. In practice, the decoder receives all h_v^t and h_w^t for all time steps t, multiplies each combination together and concatenates the results:

$$h_{e_{v,w}} = \|_{i,j}^{T} h_v^i \times h_w^j.$$

S2GAE [60] further expands on this concept, reporting that when information is sparse (e.g. graphs having few connections between nodes), *directional edge masking* can greatly improve results. In other words, when E contains both (v, w) and (w, v) but degrees are generally low, masking one edge should not automatically imply masking the other.

Node Masking Architectures

Similarly to edge masking, the selection of masked nodes involves a random process. However, rather than partitioning matrix V, the feature vectors of masked nodes are replaced with the learnable embedding of a virtual "mask token" that is introduced into the vocabulary. This replacement process results in the creation of matrix V_{masked}, and the input graph provided to the encoder is $G = (V_{\text{masked}}, E, u)$.

In the decoding phase, the decoder aims to predict the original information of the masked nodes from the final representations. This prediction task can

be formulated either as classification (e.g., predicting the original token from the entire vocabulary) or as regression (e.g., predicting the original feature vector for each of the nodes).

GraphMAE and the subsequent GraphMAE2 [48, 49] follow this general procedure. Similarly to BERT [17], they corrupt the masking procedure, giving a small probability to nodes that have been selected for masking to either remain unchanged or be replaced with another random token from the vocabulary.

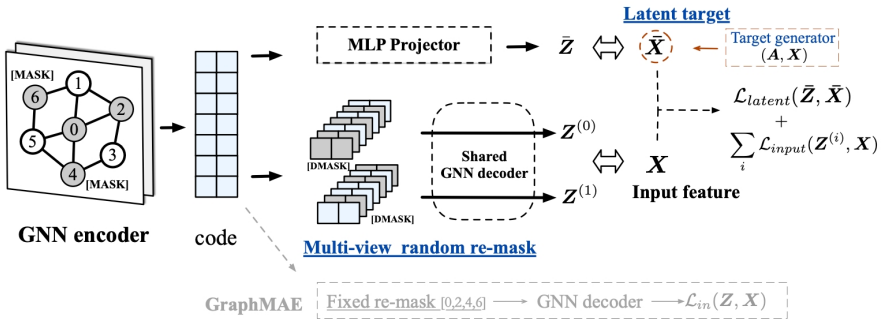
GraphMAE further proposes to re-apply node masking at the level of encoder embeddings. In other words, not only h_v^0, but also h_v^T can be subject to masking, with latent representations being replaced by another specialised "mask" token. This idea is extended in GraphMAE2 [49], where the re-masking operation is performed on multiple copies of the latent representation. The decoder is then tasked to predict the original information for each of the masked views. The reconstruction task is framed as an attribute regression. Scaled Cosine Error (SCE) in Eq. 2.6 is employed as a loss function, penalising the mismatch between predicted (z) and target (x) feature vectors:

$$\mathrm{SCE}(z, x) = \frac{1}{N} \sum_{i}^{N} \left(1 - \frac{z_i^\top x_i}{\|z_i\| \|x_i\|}\right)^\gamma . \tag{2.6}$$

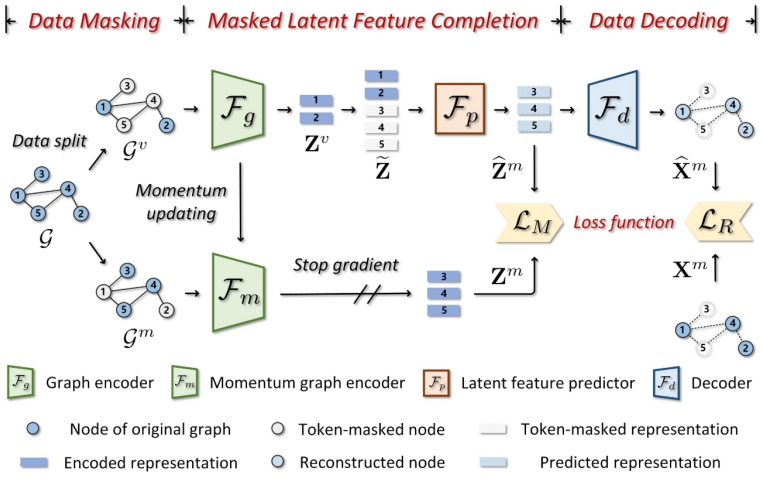
GraphMAE2 also implements an additional module called *target generator*, serving as additional regularisation during training. Architecturally, the module is equivalent to the encoder, and its parameters ξ are initialised in the same manner. However, these parameters are not trained jointly with the rest of the network. Instead, the module is updated using an exponential moving average of the parameters from the encoder (θ), given by the following equation:

$$\xi \leftarrow \tau\xi + (1 - \tau)\theta \tag{2.7}$$

where τ is a momentum factor that balances the integration of old and new



(a) GraphMAE2



(b) RARE

(c) Overview of GraphMAE2 and RARE frameworks. Image sources: [49, 51].

knowledge. The module receives as input the full, unmasked graph and produces its own embeddings for the nodes. An auxiliary loss (computed with SCE) is generated by confronting these embeddings with those produced by the encoder. An overview of the model is provided in Fig. 2.16a.

A similar methodology is also applied in RARE [51], whose architecture is showcased in Fig. 2.16b. In RARE, the masking procedure generates two complementary views of the same graph. The nodes that are masked in one graph are instead visible in the other one, and vice-versa. One of the graphs is passed to the GNN encoder, while the other is processed by a "momentum graph encoder", which has the same properties as the target generator described for GraphMAE2.

The target generator produces its own encodings for the visible nodes of the complementary view (the masked nodes in the other branch) and an auxiliary inner self-supervision signal is produced by comparing the two sets of node features by euclidean distance. For the node reconstruction loss, RARE defines an enhanced version of the SCE used in GraphMAE:

$$\mathrm{SCE}_{\mathrm{RARE}}(z, x) = \frac{1}{N} \sum_{i}^{N} \log \left(\frac{1}{2} + \frac{z_i^\top x_i}{2\|z_i\|\|x_i\|} \right)^{\gamma}. \tag{2.8}$$

Hybrid Masking Architectures

Node masking models have been reported to exhibit subpar performance in tasks where a deeper understanding of the network's structure is required (e.g. edge reconstruction) [60]. Hybrid masking architectures have therefore been introduced, incorporating both node and edge masking strategies to address the shortcomings of using only one of these techniques at a time.

Hu et al. [62] observe that pre-training GNNs using naïve techniques may lead to *negative transfer*. This phenomenon occurs when fine-tuning a pre-trained model results in worse performance compared to training the same model solely on the data of the downstream task, usually due to the incompatibility between pre-training and the downstream tasks [63].

They propose a multi-step pre-training approach for GNNs. The first step is a local-level pre-training, while the second step is based on graph-level properties prediction[6]. During the first step, the model is trained to simultaneously solve node and edge reconstruction, as well as a novel *context prediction* task. In this task, two subgraphs are built for each node: i) the subgraph of its K-hop neighbourhood, and ii) the subgraph of nodes that are between r_1 and r_2 hops away, with $r_1 < K < r_2$. An encoder GNN receives the neighbourhood subgraph, while a parallel *context GNN* computes an embedding for the context. Then, the network is trained to predict whether

[6]Assuming that a global graph-level label can be obtained for the data type.

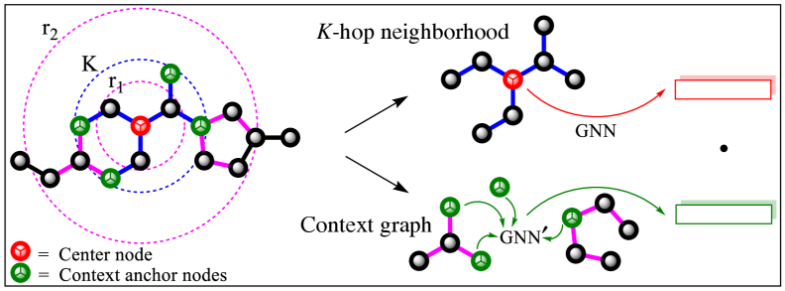


Figure 2.17: Overview of the context prediction task. Image source: [62].

the context and neighbourhood encodings belong to the same node. The process is shown in Fig. 2.17.

The idea of masking multiple levels of graph information is also used in HGMAE [61] . The encoder is trained to simultaneously reconstruct masked adjacency matrices and to re-create the raw attributes of masked nodes. In particular, a *dynamic mask rate* is employed for node masking, as the model should progressively learn from simpler problems before tackling more challenging ones. Consequently, the mask rate is incrementally raised during the epochs, reaching a predefined maximum.

Node and edge masking have also found application in graph generative models. Mahmood et al. [53] built a model focused on the generation of realistic molecules, represented as graphs. The approach iteratively replaces elements of a random molecule until convergence. Notably, it masks out a random group of nodes and edges η and samples new elements from a learned conditional distribution of masked components given the visible part of the graph $p(\eta|G_{\setminus\eta})$, modelled as a GNN. To allow the generation of new edges, prospective edges are considered to exist between all possible node pairs, but with a special "no-bond" type that inhibits message passing. Therefore, dynamic adjacency matrices are permitted by simply changing the type of edges.

Alternative Architectures

While masked graph autoencoding represents a popular approach for pre-training GNNs, it has been observed that the message passing mechanism may cause an over-smoothing problem, resulting in embeddings that become excessively similar to each other [58]. Reducing the number of message passing layers is a potential solution, but this adjustment comes as a trade-off as it diminishes the model's capacity to learn global information.

The introduction of self-attention mechanisms may also bring benefits, as the model would learn what information it should value the most. To this end, Graphormers [64] have been proposed as a way to adapt the Transformer architecture [14] to graph data. Graphormers process sequences of node embeddings, updating their information on each layer based on the relative importance that they learn to assign to all other nodes in the graph. Structural information is injected into the network by redefining the positional embeddings and by summing representations of spatial and edge information into the attention matrices.

GMAE [58] replaces the traditional GNNs in the masked graph autoencoder framework with a 16-layers deep Graphormer encoder, followed by a shallow decoder. Graphs undergo a node masking procedure, but, to reduce memory consumption, the encoder only processes non-masked nodes. "Mask embeddings" are then re-introduced into the encoder's output. The extended sequence is fed to the decoder, which is responsible for reconstructing the original information through a classification task.

2.5 High-Level Tasks

The elements discussed in the previous sections provide a foundation for constructing a comprehensive language model of graph representations of source code. In this section, I therefore present two high-level tasks that are a good fit for testing the capabilities of such a model, since they require a deep

comprehension of code structures to be solved. I also highlight how these tasks have been solved in literature, in order to set up a direct comparison with the methodology of this work.

2.5.1 Heterogeneous Device Mapping

In Section 1.1, I referred to contemporary computers as *heterogeneous* machines, as they often feature specialised hardware modules, like GPUs and Tensor Processing Units (TPUs), that can accelerate the execution of certain instructions. OpenCL [65] is an open-source framework designed for programming applications in heterogeneous environments, enabling developers to execute code portions (*kernels*) on various computing devices. However, in order to achieve optimal performance on the target devices, fine-tuning of source code and framework parameters is often essential.

Furthermore, the effort for tuning software to the target architectures may not always be worth it. The time required for transferring inputs and outputs onto and from the device, as well as the memory access patterns, may render execution on specialised hardware slower than parallel execution on multi-threaded CPUs [66]. The task of Heterogeneous Device Mapping (DevMap) is therefore aimed at learning *where* a given portion of code should be allocated based on an optimisable metric (e.g. lowest execution time, lowest energy consumption, ...).

Datasets

A dataset introduced by Cummins et al. [18, 19] is often employed for training and evaluating models for DevMap. It contains a total of 256 OpenCL kernels from a collection of benchmark suites, 47 of which had previously been used by Grewe et al. [66]. Each kernel is paired with two dynamic values: the *workgroup size* (affecting the amount of kernel parallelism) and the *data* or *payload size* (affecting the transfer time of data from host to device). Different

Suite	Cit.	In [66]	V.	Benchmarks	Kernels	Samples
AMD-SDK	[67]	✓	3.0	12	16	16
NPB	[68]		3.3	7	114	527
NVIDIA-SDK	[69]	✓	4.2	6	12	12
Parboil	[70]	✓	0.2	6	8	19
Polybench	[71]		1.0	14	27	27
Rodinia	[72]		3.1	14	31	31
SHOC	[73]	✓	1.1.5	12	48	48
Total				71	256	680

Table 2.1: DevMap dataset composition. The first column contains a checkmark if some of the kernels from the benchmark were used in the work of Grewe et al. [66]. Partially adapted from [74].

values of workgroup/data size are used for some of the kernels, bringing the total number of samples to 680. The full list of kernels and their sources is provided in Table 2.1.

Each kernel was executed on 2 separate machines, both equipped with an Intel Core i7-3820 CPU but one using an AMD Tahiti 7970 GPU, while the other an NVIDIA GTX 970 GPU. The execution times are collected for the 4 machine-device pairs and split by machine, obtaining two separate sets (NVIDIA and AMD). For each sample in a set, the label is the type of device (CPU or GPU) that managed to execute the program faster, on average.

Models for Heterogeneous Device Mapping

A large number of works have dealt with the problem of device mapping. One of the first solutions was developed in 2013 by Grewe et al. [66]. Programs are represented using a small set of 4 hand-designed features, mostly related to memory accesses, communication and computation operations. Then, a *decision tree classifier* is employed to decide whether it's better to compile a given program as OpenMP parallel code on CPU or to automatically translate it to OpenCL and run it on GPU. Cummins et al. [18] reported poor generalisation results when adding more diverse source code examples: they therefore improved the representation increasing the number of features to 11.

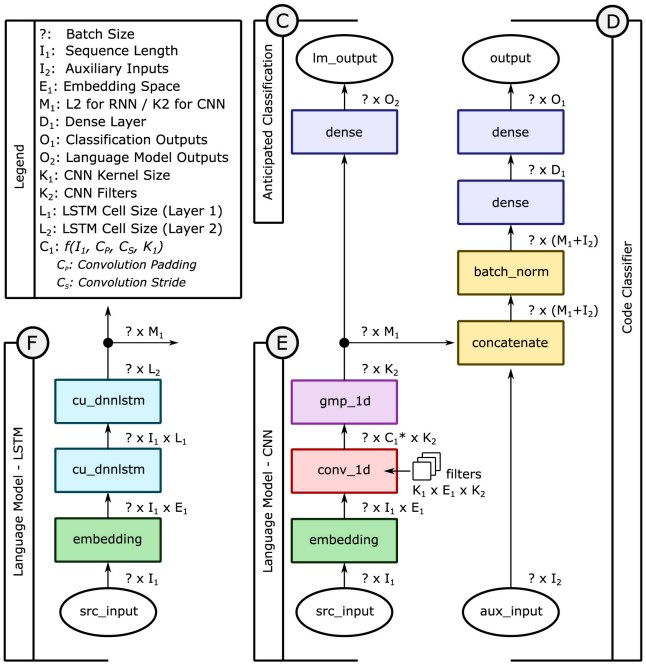


Figure 2.18: Architectures for DeepTune-based (on the left) and DeepLLVM-based (in the middle) heterogeneous device mapping. Elements (E) and (F) are the language models, and their output is concatenated with the auxiliary inputs (on the right). The rest of branch (D) performs device classification. Image source: [26].

Later solutions are based on neural architectures. Their general approach consists in employing a deep LM to analyse the source code and produce a final representation for the whole program, followed by a lightweight classification head specialised for DevMap. The final layer of this classification head is usually a MLP with 2 output neurons and a final softmax activation, producing a probability distribution between the two classes.

Works based on the DeepTune architecture [19, 20, 24] concatenate the raw auxiliary inputs to the output of the LM, before applying a batch normalisation operation [75], followed by a couple of dense layers and the final classification. DeepLLVM [26] changes the traditional LSTM-based LM of DeepTune with a convolutional layer, followed by a global max pooling. The differences between the two architectures can be seen in Fig. 2.18

Ben-Nun et al. [27] and VenkataKeerthy et al. [35] also test the quality of their inst2vec and IR2Vec embedding spaces on DevMap. The former

use DeepTune as a base architecture, but replace the pre-processing pipeline, injecting their own vocabulary and pre-trained embeddings. The latter use a different approach, producing a single embedding for each kernel using their tailored pipeline and employing a simple gradient boosting classifier for the final prediction.

Brauckmann et al. [30] introduce graph-based LMs for the task, represented in Fig. 2.19. They create a vocabulary of 92 and 140 node types for the AST and the CDFG-based representations respectively, then map each node to one-hot encoded representations and use an MLP f_{init} to create the initial embeddings h_v^0. The message passing procedure is repeated for $T = 4$ times:

- The message creation and aggregation steps are similar to those presented in Section 2.3.2, but different learnable parameters are employed depending on the type of edge through which the message is sent. Assuming that $j_{v,w}$ contains edge type information, function M_t of Eq. 2.1 would be defined as:

$$M_t(h_v^t, j_{v,w}) = A_{j_{v,w}} \times h_v^t + b_{j_{v,w}}$$

  where $A_{j_{v,w}}$ and $b_{j_{v,w}}$ are matrices of learnable parameters that are specialised for each edge type.

- The update function U_t in Eq. 2.2 is then implemented as a Gated Recurrent Unit (GRU) cell, which is similar to an LSTM but with fewer parameters [76].

- The readout function R in Eq. 2.3 is implemented by mapping all node states h_v^T to a higher dimensionality through two distinct MLPs. Then, a simple attention mechanism is applied to weigh the relative importance

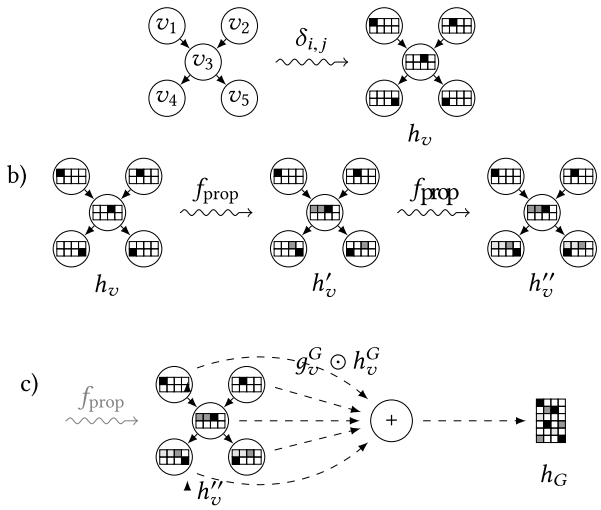


Figure 2.19: Architecture of a GNN-based LM. The initial embeddings are produced by a learnable process. Then, the message passing procedure is repeated for a certain amount of time steps. Finally, the output representation is obtained by aggregation of node features. Image source: [30].

that each node state holds for the final global state:

$$h_v^G = f_m(h_v^T)$$
$$g_v^G = g_m(h_v^T)$$
$$\hat{G} = \sum_{v \in V} g_v^G \times h_v^G.$$

The final embedding $\hat{G}$ is concatenated with the auxiliary inputs, processed by another MLP and given as input to the same classifier head of DeepTune.

ProGraML [36] uses a very similar GNN as Brauckmann's. The main differences are:

- Edge embeddings $j_{v,w}$ contain not only the edge type information $\text{Type}(j_{v,w})$, but also the positional information (argument or branch order) $\text{POS}(j_{v,w})$. The latter is encoded with a sinusoidal positional embedding, following Transformers [14]. This information is multiplied with the current node state at message creation:

$$M_t(h_v^t, j_{v,w}) = A_{\text{Type}(j_{v,w})}(h_v^t \times \text{POS}(j_{v,w})). \tag{2.9}$$

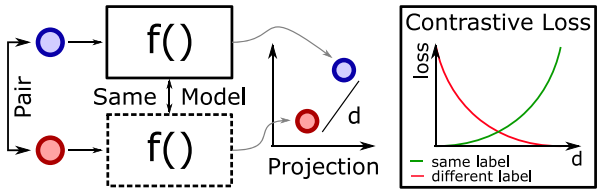


Figure 2.20: The siamese architecture employed by Parisi et al. Image source: [74]

- The readout function is implemented as:

$$\hat{G} = \sum_{v \in V} \sigma\left(g_m(h_v^0, h_v^T)\right) \times f_m(h_v^T) \tag{2.10}$$

  where σ is the sigmoid function and g_m works on the concatenation of h_v^T and h_v^0.

The work of Parisi et al. [74] is the current state of the art on the DevMap dataset. The authors implement a *siamese architecture*, made of 2 equivalent CNNs projecting pairs of samples into a 2D space, as seen in Fig. 2.20. The loss function evaluates the distance between the projections: if the samples have the same label, the model should map them as close as possible in the 2D space and is therefore penalised proportionally to their distance. If, instead, they have different labels, the model is rewarded for pushing the projections away from each other, up until a margin m when there is no need to enforce further separation. This configuration allows the model to learn from each possible pair of kernels in the dataset, considerably increasing the number of training samples.

2.5.2 Thread Block Size Prediction

Another popular framework for GPU programming is CUDA[7], a proprietary library for C, C++, Fortran and Python developed by NVIDIA and specifically targeting NVIDIA GPUs.

[7]https://developer.nvidia.com/cuda-toolkit

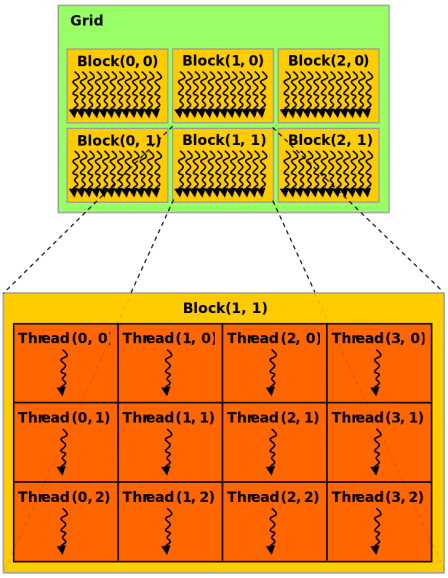


Figure 2.21: An example of the block-thread hierarchy in the CUDA framework. Image source: NVIDIA CUDA Programming Guide version 3.0

CUDA allows programming GPUs for large-scale multi-threading by abstracting the underlying hardware architecture with the high-level concepts of *blocks* and *threads* [77]. Each block can be considered as a work unit that can be executed independently from the others. A block scheduler within the framework takes care of mapping each block to available Streaming Processors (SPs) on the device, so that they can be executed in parallel.

Logically, blocks are organised in a 3D grid and are composed of *threads*, minimal units of work using the same on-device shared memory. The number of threads in a block must be divisible by 32 as threads are organised in *warps*, minimal units of scheduling that allow simultaneous execution of 32 threads at a time. Any multiple of 32 is a valid block size, up to a maximum of 1024. Figure 2.21 shows an example of this hierarchy.

A problem that is usually left to CUDA programmers is that of choosing an appropriate thread block size for a given program. The optimal choice depends on many factors, such as the size of the input and the number of accesses to registers and shared memory within blocks [77].

Developers can be guided in this choice by human expertise or heuristic tools (e.g. `cudaOccupancyMaxPotentialBlockSize`, offered by the API). However, as in the case of DevMap, a ML model trained on enough data might learn which patterns and code structures are actually helpful for the decision and exploit this knowledge to make better proposals.

Datasets

LS-CAT [78] is a large-scale dataset of over 20K CUDA kernels collected mainly for this purpose from larger GitHub projects. The processing pipeline isolated kernels from their repositories, but re-introduced the `#include` directives from the original source, resolved all iterative dependencies and generated parametric main files that act as simple entry points for launching the kernels. In this way, all kernels are actually compilable and executable.

The parameters that the main files require are the *input matrix size* and *thread block size*. Specifically, the authors propose to test kernels on a set of 7 input matrix sizes (240, 496, 784, 1016, 1232, 1680, 2024) and 20 thread block sizes (1D array of threads of all dimensions multiple of 64 up to 1024 and 2D grid of threads of sizes 8×8, 16×16, 24×24 and 32×32). Once the input and thread block size have been chosen, variables with names related to these parameters (e.g. `h`, `w` and `n`) are given their appropriate values and the program is executed tracking the total execution time. The mean time of a large number of experiments is inserted in a database. The authors already provide run times for a machine equipped with a Tesla T4 GPU.

Models for Thread Block Size Prediction

The problem is tackled by Bjertnes et al. [79] (see Fig. 2.22), where LS-CAT is used for training NLP-based ML algorithms. The authors compile kernels into LLVM-IR, apply a tokenisation process and produce token representations using fastText [80]. This algorithm internally represents words as sums of character n-gram representations, allowing tokens that were not observed

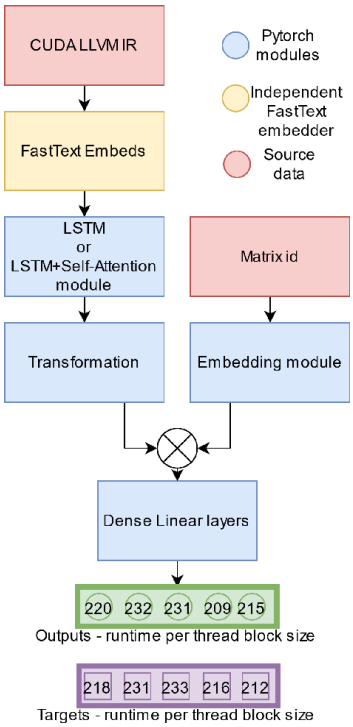


Figure 2.22: Thread block size prediction model by Bjertnes et al. Image source: [79].

during training to also have representative features.

The sequence of token embeddings, together with an embedding of the chosen input matrix size, are passed to the model, composed of a simple LSTM-based network and an optional self-attention module on top. A final Dense layer outputs a score for each of the possible thread block sizes.

Chapter 3

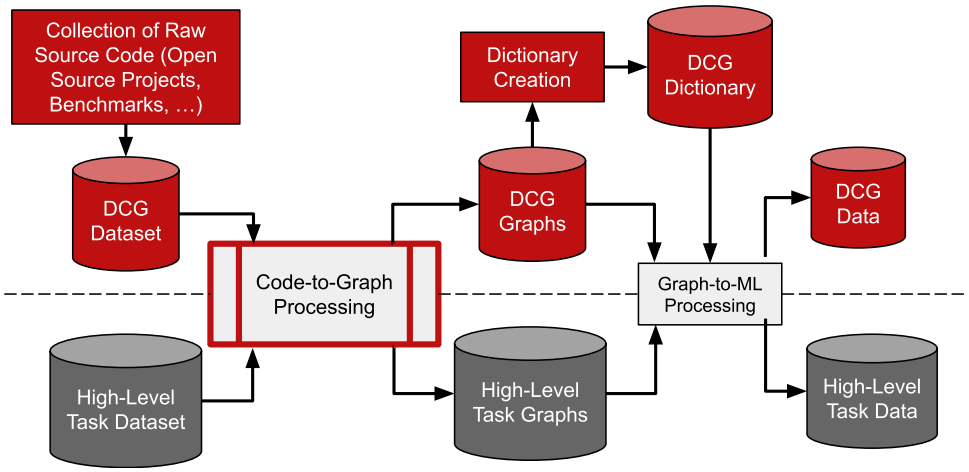
Methods

DeepCodeGraph (DCG) is the workflow that was developed for this thesis in order to tackle the problems presented in the previous chapter. It consists of several co-operating components:

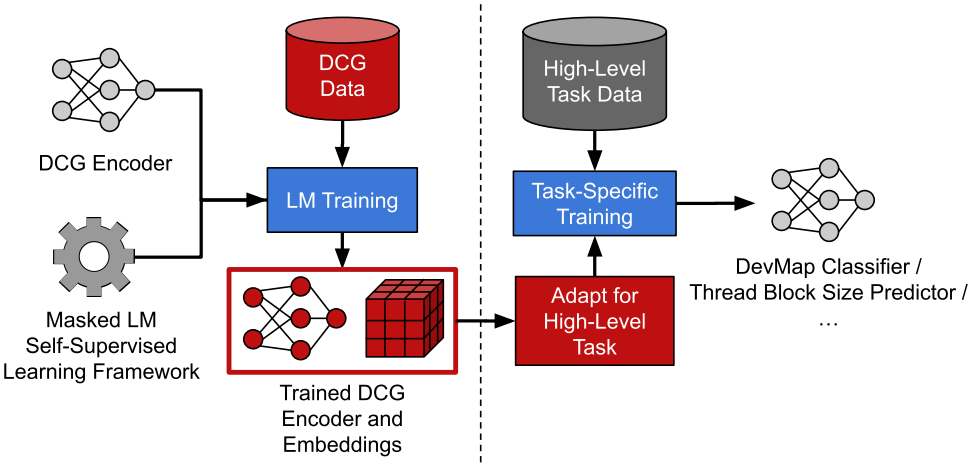
- The **DCG Encoder**, which represents the core of the methodology. It serves as a LM of graph representations of source code. It captures the knowledge present in the patterns of communication between instructions and data and produces dense embeddings expressing the semantic value of the source code inputs. It can easily be plugged into models for solving high-level tasks, transferring general knowledge into more specific scenarios.

- A **self-supervised learning framework**, which is employed for training both the **DCG Encoder** and **DCG Embeddings**. The objective of this training is to enhance the *generality* of the encoder, enabling it to acquire valuable knowledge on how to effectively leverage the representation patterns. This acquired knowledge is intended to be transferable across a diverse range of high-level tasks, offering the potential for substantial benefits.

- The **DCG Dataset**, a large collection of raw source code from a wide



(a) First part of the DeepCodeGraph methodology, from raw code to graph encodings.



(b) Second part of the DeepCodeGraph methodology, where a self-supervised learning framework, the DCG encoder and data are combined in order to train the LM. Then, task-specific trainings are performed.

Figure 3.1: Visual representation of the DeepCodeGraph workflow and its parts. Red elements represent the contributions of this thesis with respect to established workflows in literature. Blue elements imply training/validation experiments. The dotted lines represent the limit between LM and high-level task trainings.

variety of repositories. The dataset is composed by over 100 k compilable files incorporating commonly utilised benchmarks within the field, as well as newly introduced open-source projects. Its primary purpose is to provide a sizeable number of samples for the LM training phase, enabling the DCG encoder to acquire, through learning, diverse knowledge pertaining to high-level coding patterns.

- A **data processing procedure** that converts raw source code files into graph-based representations, facilitating the emergence of explicit patterns of dependency between data and instructions.

- The **DCG Vocabulary**, a collection of tokens extracted through statistical analysis from the graphs of the DCG dataset. It maps the most common instructions and data types to integer values, enabling the LM to manipulate the graphs through mathematical operations. The learnable **DCG Embeddings** transform each token in the vocabulary into a dense representation containing semantic information about the content of nodes, facilitating the work of the encoder.

- **Task-specific trainings**, allowing the LM to be tested on the high-level tasks presented in Section 2.5. These include personalised data processing for the DevMap and LS-CAT datasets, as well as custom adaptations of the DCG encoder for both tasks.

The steps of the methodology are visually represented in Fig. 3.1.

In the rest of this chapter, I will describe each of these components in detail. Section 3.1 presents the tools that were used for the implementation of models and data processing operations. Section 3.2 focuses on the DCG encoder and the self-supervised framework, also providing a description of the input representation. Section 3.3 delves into the creation process behind the DCG dataset. Finally, Section 3.4 shows how the DCG methodology can be adapted to solve high-level tasks.

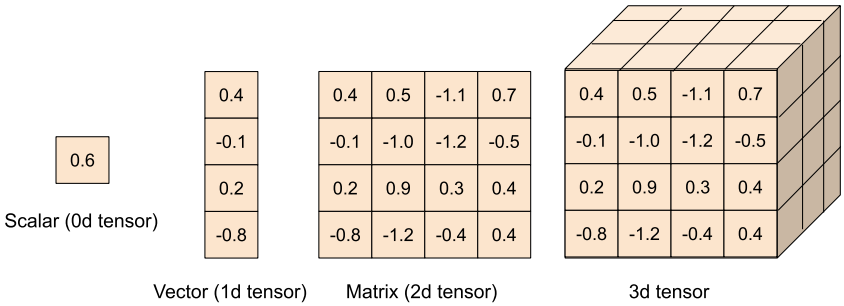


Figure 3.2: Representation of tensors across dimensions (0 to 3).

3.1 Tools for Graph Neural Networks

All of the experiments presented in this thesis were implemented using Python 3.11 and PyTorch 2.0 [4] as the main framework for ML and DL-related procedures. PyTorch Geometric [81] was also employed for developing GNNs and deal with graph data efficiently.

PyTorch

PyTorch is a flexible tool for the development and execution of learning algorithms. It exposes a straightforward API, enabling access to many pre-made modular components and taking care of low-level aspects, such as gradient computation and the handling of multiple devices in heterogeneous environments.

Tensors are the fundamental data structure in PyTorch. They are multi-dimensional matrices containing numerical values of a specific data type (see Fig. 3.2). They can also hold information about the chain of operations that produced them, so that the framework can easily manage automatic backpropagation. In PyTorch, model inputs, weights and outputs are all expected to be tensors.

On the other hand, *modules* are the core building blocks for all neural models. They encapsulate the learnable parameters of a component and enable developers to define the sequence of operations that should be performed on their inputs, streamlining the development of neural networks. PyTorch

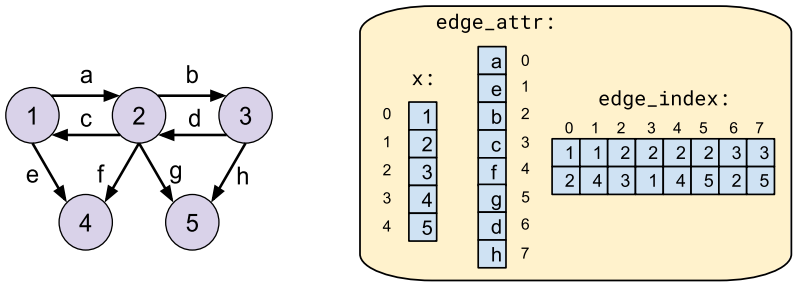


Figure 3.3: Transformation of a graph (on the left) to a PyG `Data` object (on the right).

implements a wide variety of efficient low-level tensor operations, allowing rapid prototyping and experimentation of neural designs.

However, while several general-purpose modules, such as linear and convolutional layers, are pre-implemented, PyTorch does not offer out-of-the-box tools for the processing of graph data.

PyTorch Geometric

PyTorch Geometric (PyG) is a library built upon PyTorch that contains useful utilities for building GNNs and data pipelines for graphs. It is fully integrated with pure PyTorch, enabling access to the operations and components provided by both libraries at the same time.

Data objects Within PyG, graphs are represented as `Data` objects, containers of tensors related to the structure of graphs. Common properties of these objects are:

- `x`, a matrix containing node embeddings. It has size $[N_n, D_n]$, with N_n being the number of nodes of a graph and D_n being the embedding dimension for nodes.
- `edge_index`, a matrix representing the adjacency list of the graph. It has size $[2, N_e]$, where N_e is the number of edges and the first dimension contains the source and target node indices respectively.

- `edge_attr`, a matrix containing edge embeddings, with size $[N_e, D_e]$, where D_e is the dimensionality of edge features.

If a ground truth label for a task is present (e.g. in classification tasks), the `y` property can be used. Any other kind of information can also be stored in custom-named properties. An example of a `Data` object can be seen in Fig. 3.3.

Datasets and Dataloaders A collection of `Data` objects can be handled by instances of the `Dataset` and `Dataloader` classes. Analogously to plain PyTorch, the `Dataset` class is responsible for defining how a single sample can be retrieved from the collection given an index. If the collection is small enough to be entirely stored in RAM (as is the case for the DevMap dataset), this may simply amount to retrieving the corresponding pre-processed `Data` element from a list. On the other hand, when the collection is large and it has to partially be stored on disk, the `Dataset` class may describe how samples are loaded and processed.

`DataLoaders` are a high-level interface for efficiently iterating over a dataset, offering automatic batching and shuffling at the end of each epoch, as well as providing parallel data retrieval and processing from the underlying `Dataset` objects. In particular, batching in PyG is performed by automatically creating a single large graph composed of as many isolated sub-graphs as there are samples in the mini-batch.

The resulting `Data` object contains the concatenation in the node dimension of the property tensors for the graphs that are part of the mini-batch. The `edge_index` property is automatically incremented to reflect the increased number of nodes and to avoid connections between nodes from different sub-graphs. When iterating over the `DataLoader`, the additional property `batch` is added in the `Data` object, containing a sequence of indices mapping each node in the graph to its corresponding sub-graph index. This property can be employed to perform operations on individual samples: for instance, it is

leveraged in the readout layers of GNNs to ensure that global pooling does not affect nodes that are part of other graphs.

Message Passing Neural Networks PyG also provides a `MessagePassing` class that implements the mechanism described in Section 2.3.2. A model that extends `MessagePassing` has access to additional functions:

- `model.message(x_j, ...)`, which defines how a message is constructed employing the features of neighbouring nodes `x_j` and possibly other information. Messages for a receiving node are aggregated automatically, either with element-wise addition, mean or max. These operations are analogous to Eq. 2.1.
- `model.update(...)`, which implements node updates given their current state and the incoming aggregated message. This operation is analogous to Eq. 2.2.

Finally, the `propagate(...)` function can be used to run a whole message passing cycle: it generates all messages and updates node states according to the graph adjacency matrix.

3.2 Architectures Implementation

The beating heart of this project is the DCG encoder, a language model for graph representations of code. A self-supervised graph-based learning framework trains the LM, enabling it to craft global dense representations expressing the semantics of source code samples. The underlying assumption is that if the encoder is able to produce expressive and versatile representations, it could improve the results on high-level tasks, including but not limited to the ones presented in Sections 2.5.

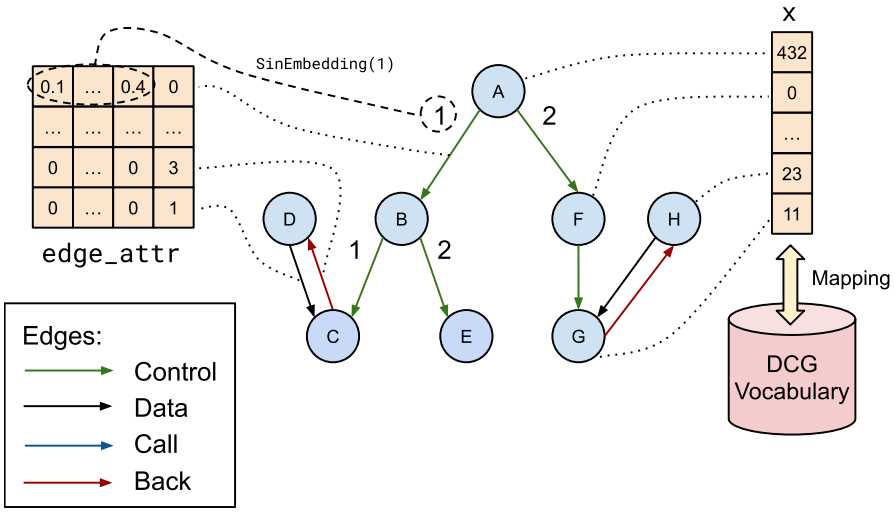


Figure 3.4: Snippet of a possible transformation from a pseudo-ProGraML graph into a PyG `Data` object. The information of nodes is mapped to tokens in the DCG Vocabulary, creating the feature vector `x` (on the right). Edges are instead processed by concatenating their edge types to a sinusoidal embedding of their positional information, creating matrix `edge_attr` (on the left).

3.2.1 Input Graph Representation

The DCG pipeline is general enough to be easily adaptable to different graph representations, but the one that is employed in this work is that of ProGraML [36], presented in Section 2.2.2. As a short summary, ProGraML represents programs as *directed multigraphs*. Nodes represent *instructions*, *variables* and *constants* and are connected by edges with a specific type (*control*, *data* or *call*). Edges also contain features that help distinguish the order of different operands or branching operations.

I augmented the ProGraML representation by adding a fourth type of edge: *back edges*. These are optional links connecting some or all neighbouring pairs of nodes in the opposite direction, increasing the amount of communication in the message passing process. An important feature of back edges is that they enable an information flow (e.g. message passing) towards nodes that would otherwise have no incoming connections.

The graphs are implemented as PyG `Data` objects. As seen in Fig. 3.4, the attributes are created as follows:

```
    # D_n is the node dimensionality
    # D_h is the hidden dimensionality
    # D_e is the edge attribute dimensionality

    def __init__(self, ...):
        ...
        self.embs = Embedding(vocab_size, D_n)
        self.init_n_trans = Linear(D_n, D_h)
        self.init_e_trans = Linear(D_e, D_h)
        ...

    def forward(self, x, edge_index, edge_attr):
        ...
        x = self.init_n_trans(self.embs(x.squeeze()))
        e_pos_embs = self.init_e_trans(edge_attr[:, :D_e])
        e_types = edge_attr[:, -1].reshape(-1, 1).repeat(1,
D_h)
        ...
```

Listing 3.1: Code for the initial transformation of the DCG encoder.

- `x`, the matrix of node features, is a $[N_n, 1]$ matrix of integer values, representing indexes of tokens within the DCG vocabulary. The transformation process from raw source code to tokens will be presented in detail in Section 3.3.2.

- `edge_index` contains the connections between nodes, as usual. The number of edges is naturally affected by the use of back edges.

- `edge_attr` is a $[N_e, D_e + 1]$ matrix containing for each edge a sinusoidal encoding [14] of the positional information with dimension D_e, concatenated with a numerical value representing the edge type (control edge: 0, data edge: 1, call edge: 2, back edge: 3).

3.2.2 DCG Encoder Architecture

The architecture of the DCG encoder closely follows that of the GNN used in the ProGraML paper [36]. The full implementation can be found at the link in the footnote[1]: here only some snippets are shown for brevity.

[1]https://gist.github.com/volpepe/2c9bff2533193af0c27a3dca4a182f31

```
    def __init__(self, ...):
        ...
        self.lin_e_ctrl = Linear(D_h, D_h, bias=True)
        self.lin_e_data = Linear(D_h, D_h, bias=True)
        self.lin_e_call = Linear(D_h, D_h, bias=True)
        self.lin_e_back = Linear(D_h, D_h, bias=True)
        self.gru_cell   = GRUCell(D_h, D_h)
        self.bns        = ModuleList([BatchNorm1d(D_h)
            for _ in range(MSG_PSS_ITERATIONS)])
        ...

    def forward(self, x, edge_index, edge_attr):
        ...
        for i in range(MSG_PSS_ITERATIONS):
            x = self.propagate(edge_index, e_pos_embs,
                               e_types, x)
            x = self.bns[i](x)
            x = F.elu(x)
        ...

    def message(self, x_j, e_pos_embs, e_types):
        msg         = x_j * e_pos_embs
        ctrl_msg    = self.lin_e_ctrl(msg)
        data_msg    = self.lin_e_data(msg)
        call_msg    = self.lin_e_call(msg)
        back_msg    = self.lin_e_back(msg)
        msg = torch.where(e_types == 0, ctrl_msg,
              torch.where(e_types == 1, data_msg,
              torch.where(e_types == 2, call_msg, back_msg)
        ))
        return F.dropout(msg, p=DROPOUT_RATE)

    def update(self, msg, x):
        return self.gru_cell(msg, x)
```

Listing 3.2: Code for the message passing section of the DCG encoder.

Initial processing Firstly, an embedding matrix maps node tokens to dense representations, increasing the dimensionality of the input. The embedding matrix can be loaded from a pre-trained checkpoint or randomly initialised. In both cases, it can either be trained in tandem with the model or left *frozen*.

The edge features tensor is split, separating the matrix of edge positional embeddings from the vector of edge types. They are subsequently expanded into tensors of the same dimensionality, respectively through a dense layer and by repeating columns. Listing 3.1 shows a snippet of code for this initial processing step.

```
    def __init__(self, ...):
        ...
        self.lin_i = Linear(2*D_h, D_h)
        self.lin_j = Linear(D_h, D_h)
        ...

    def forward(self, x, edge_index, edge_attr):
        start_x = x.clone()
        ...
        alignment = torch.cat([start_x, x], axis=-1)
        gate = F.sigmoid(self.lin_i(alignment))
        node_embs = gate * self.lin_j(x)
        graph_embs = global_add_pool(node_embs, batch)
        return node_embs, graph_embs
```

Listing 3.3: Code for the readout section of the DCG encoder.

Message Passing The message propagation process is implemented as a loop of calls to the `propagate(...)` function, followed by batch normalisation and a non-linearity. The `message(...)` function firstly modulates the node embeddings of neighbours using the positional information. It then applies 4 different linear layers on the result, each specialised for one of the edge types. Finally, partial results are aggregated from the outputs of the dense layers based on the edge type information. The final messages undergo a dropout procedure, where a small fraction of their values are zeroed-out, leading the network towards producing more robust representations. This sequence of operations corresponds to Eq. 2.9.

Messages are then aggregated by sum on the receiving nodes, and a GRU cell is employed to update their current states. A snippet of the implementation of this process is shown in Listing 3.2.

Readout After the message passing cycle, a readout operation is performed, implementing Eq. 2.10. Final node features are weighted with a self-attention mechanism that takes into account the alignment between initial and final node embeddings.

Specifically, the node representations are concatenated, processed by a linear layer and transformed into scores in the 0-1 range by a sigmoid function.

These values are used to re-weigh each of the final node features, which are then aggregated with a global add pooling operation, obtaining a unique graph representation. The pooling operation automatically manages batches and avoids summing features from nodes belonging to different graphs.

The model returns both the matrix of final node embeddings and the pooled graph embeddings, so that both graph-level and node-level tasks can be performed. A snippet of code for this final step can be seen in Listing 3.3.

3.2.3 Self-Supervised Learning Framework

In order to train the LM, I implemented a self-supervised learning model based on the framework of masked graph autoencoders. Specifically, the implemented model is based on RARE [51], but several key ideas from other works have also been included. The additional components are optional and were investigated in the hope that they would prove beneficial for the learning task. Due to this set of extensions, the MGAE framework employed in this work is named *GraphMask*.

Node Masking or Edge Masking?

In the process of transforming a LLVM-IR program into the graph representation described in Section 3.2.1, variables, constants and *immediate values* (e.g. hardcoded numerical values) are reduced to their type. For instance, different `int32` variables are transformed into different nodes in V, but their initial information is the same. This reduction serves to maintain a manageable dictionary size, as it would be impossible to allocate tokens for each possible value of variables and constants.

Edge masking GAEs randomly mask edges out of the graph, possibly causing some nodes to become isolated. As a consequence, these nodes will not receive any updates during the message passing cycle, and their final representations will correspond to their starting features. However, if two

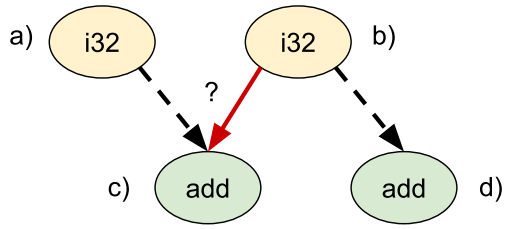


Figure 3.5: The edge reconstruction task on ProGraML-like representations is an ill-posed problem. Nodes a) and b) contain the same information, which is never updated due to the absence of incoming edges. Dashed lines represent true masked edges. During training, the model receives a positive feedback if it correctly predicts an edge between a) and c). However, it receives a negative feedback if it predicts an edge between b) and c), even though the information contained in the nodes is exactly the same.

isolated nodes are variables or constants with the same type (e.g. `int32`), the edge reconstruction task may become exceedingly challenging. Fig. 3.5 illustrates an example scenario that exemplifies one of the problems associated with this approach.

In the DCG dataset, about 6.1% of nodes have no incoming connections, so an aggressive edge masking policy may lead to several of these cases. Therefore, I believe that node masking could be a better fit for learning effective patterns on the representation of source code employed in this work.

GraphMask Implementation

An overview of the GraphMask framework can be seen in Fig. 3.6.

Overview The model receives a graph representation as input and masks a portion of its nodes, as explained in Section 2.4.2. The DCG encoder processes the masked graph and produces representations for each of the nodes. The decoder (a MLP) is tasked to reconstruct the original node embeddings from the encodings, producing a *node reconstruction loss* (Eq. 2.8).

Concurrently, a *target generator* with the same architecture computes a representation for a complementary view of the graph, built by masking the opposite set of nodes. The target generator uses a set of parameters that remains

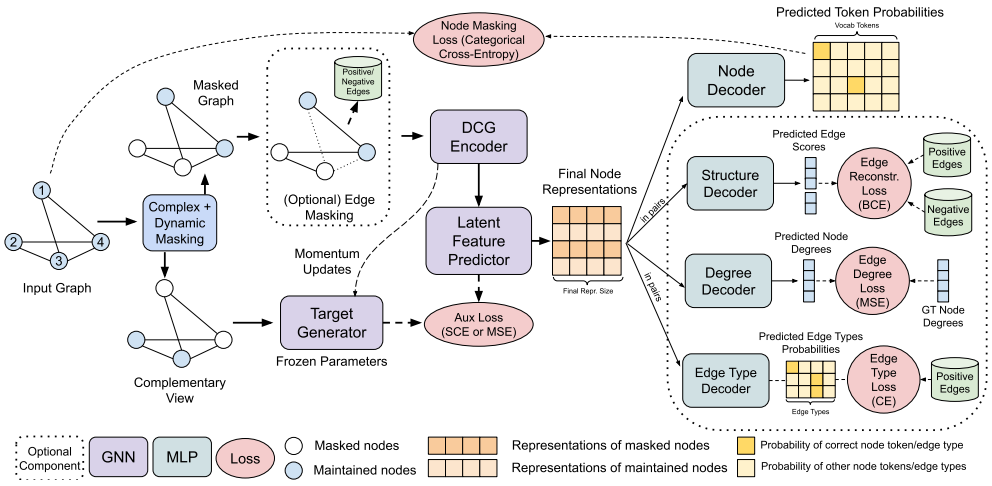


Figure 3.6: An overview of the MGAE framework employed in this work, GraphMask. While the main inspiration is RARE [51], it includes many key components from other works. The complex/dynamic masking module increases the complexity of the node reconstruction task during training, while optional edge masking decoders allow hybrid node/edge masking techniques to be employed. All losses contribute to the final feedback for the learning task.

untouched by standard backpropagation; instead, it only receives updates from the DCG encoder every N training steps, following Eq. 2.7. An *auxiliary loss* (mean squared error, MSE) is produced by comparing the representation of the target generator and that of the encoder.

Node Classification In practice, initial experimentation with this network yielded exceedingly poor training outcomes, resulting in negative transfer. Upon analysing the issue, it became evident that the parameters of the DCG encoder were being pushed close to 0, impairing the network's capability to generate expressive encodings. The likely root cause of this problem was that the training targets, the DCG Embeddings, were being trained simultaneously with the rest of the model.

Indeed, if all DCG embeddings are pushed towards a singular point in the embedding space (e.g. 0), distance-based losses could be minimised by simply having the decoder consistently output values close to this central point. In this scenario, the loss decreases, but the model is not actually learning an

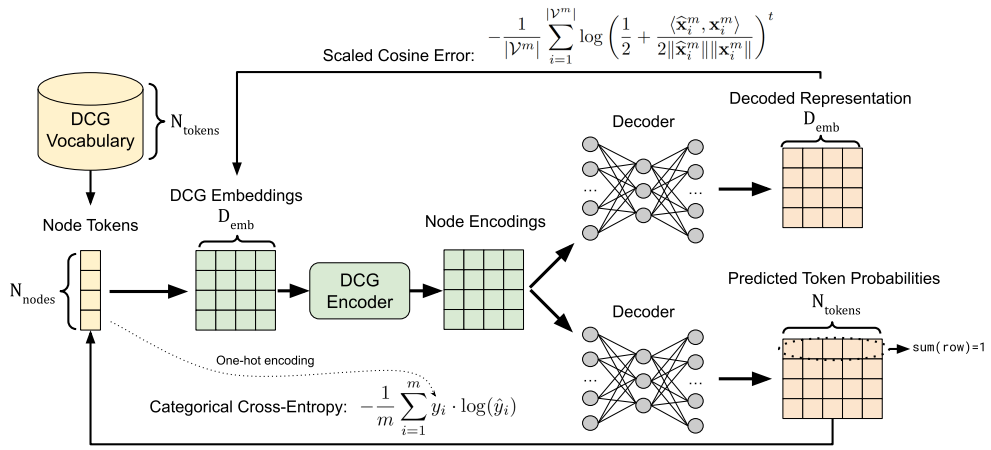


Figure 3.7: Shift in task for GraphMask training, from feature reconstruction (above) to token classification (below).

effective embedding space.

A possible solution could be to frame the problem as a node token *classification*, similarly to the approach used in large language models (e.g. BERT) [17]. Through classification, the model is driven to learn discriminative embeddings in order to facilitate the identification of the correct option between all tokens in the vocabulary. Furthermore, as the targets are one-hot encoded indices from the DCG vocabulary, they cannot change during training.

The final layers and reconstruction loss were adapted to reflect the new task. The decoder needs to produce probability distributions over the vocabulary, while the loss becomes a standard cross-entropy, where the target is a one-hot encoding of the original token. The task shift is represented in Fig. 3.7

Additional Components Apart from re-framing the task as a classification problem, I introduced several additional components:

- *Dynamic Masking*: inspired by HGMAE [61], this component increases the node masking rate during the training epochs, raising the difficulty of the task as the model learns better patterns.

- *Complex Masking*: inspired by BERT [17], GraphMAE and GraphMAE2 [48, 49], tokens selected for masking are, with a small probability, replaced by another random token from the vocabulary or kept intact. When combined with dynamic masking, the replace rates can also change during training.

- *DCG Encoder*: The DCG encoder replaces the encoder and target generator of the original network (GATs or GINs). Additionally, RARE features a shallow 1-layer GNN that further processes the encodings of the LM and produces *latent features* to be compared with those of the target generator for the auxiliary loss. I also replace this network with a 1-layer DCG encoder to better exploit the additional edge information that is included in the custom graph representation.

Optional Edge Reconstruction Inspired by hybrid masking architectures, the model was adapted in order to optionally provide edge reconstruction as an auxiliary task during training. When the module is active, the process explained in Section 2.4.2 is utilised, partitioning the set of edges and employing additional classifiers. The DCG encoder does not receive the masked edges, while the target generator processes the whole structure.

When the negative sampling process is run on a mini-batch of samples from a PyG Dataloader, it is crucial to ensure that negative edges are only sampled *within* sub-graphs and don't link disconnected samples. To this end, PyG provides the utility function `batched_negative_sampling` that automatically filters valid pairs.

As for the decoding part, the two decoders employed in MaskGAE [50] (see Fig. 2.15) are implemented as follows:

- The *Structure Decoder* receives pairs of node representations from the encoder, aggregates them through an element-wise product and produces a score (through a MLP) representing the likelihood of an edge between

the two nodes. The score is in the range 0-1, while the target of this classification is 1 for positive pairs and 0 for negative pairs.

- The *Degree Decoder* processes single node representations, predicting the number of incoming edges (degree) for each of the nodes. Rather than producing a probability, it employs ReLU as a final activation function to avoid negative values.

Finally, I introduce a third decoder named *Edge Type Decoder*, specialising on predicting the ProGraML type of positive edges. The aim is to produce representations of nodes that are more aware of the possible kind of relations towards their neighbourhood. Structurally, it is implemented in the same way as the Structure Decoder, but it produces a vector of scores for each of the 4 possible types (control, data, call, back).

The Structure, Degree and Edge Type Decoders respectively employ Binary Cross-Entropy (BCE), Mean Squared Error (MSE) and Categorical Cross-Entropy as loss functions. The final loss is computed as a weighted sum of the losses for the active decoders.

3.3 The DCG Dataset

To obtain a comprehensive LM, the implementation of an effective learning framework is just one half of the equation. An equally crucial aspect is to expose the model to a vast and diverse set of samples. The DevMap dataset is not fit for the task, due to its limited size. LS-CAT, on the other hand, has a diversity problem: the samples should be able to represent the general population of source code, while this dataset has a strong focus on scientific computation.

The internet provides convenient access to an immense quantity of source code samples, be it from public repositories and open-source projects hosted

Name	Cit.	From	Files	Compiled	Graphs
Blas	[82]	NCC	300	300	300
bowtie2	[83]	SRC	57	57	39
bwa-mem	[84]	SRC	24	24	24
cBench	[85]	GYM	23	23	22
clang	[22]	SRC	711	711	568
CLGen	[18]	GYM	996	996	996
eigen	[86]	NCC	4,998	4,998	4,802
gemm_synth	[27]	NCC	3,700	3,700	3,700
Gromacs	[87]	SRC	1,249	1,205	1,196
JotaiBench	[88]	GYM	5,535	5,535	5,535
Linux	[89]	NCC	13,920	13,920	13,918
LLVM	[21]	SRC	21,371	21,371	19,604
MiBench	[90]	GYM	40	40	40
OpenCV	[91]	NCC	442	442	442
POJ104	[92]	GYM	49,816	49,815	49,815
stencil_synth	[27]	NCC	12,800	12,800	12,800
Tensorflow	[3]	NCC	1,985	1,985	1,966
Total			117,967	117,922	115,767

Table 3.1: DCG Dataset description. The source of each included dataset is expressed with a symbol: NCC means inclusion in the NCC dataset, GYM means that the dataset files were downloaded using CompilerGym [93] and SRC means that the files were built from scratch. The second half of the Table contains the number of total `.c` or `.cpp` files, of LLVM-IR files and of graphs.

on websites like GitHub[2] or from snippets of code that solve specific user-requested problems found in forums and communities. However, as the representation relies on LLVM-IR, the collected samples must also be compilable using tools from the LLVM toolchain. To this end, I introduce the DCG Dataset, a large, heterogeneous collection of compilable C/C++ and LLVM-IR code sourced from a wide range of open-source projects and publicly available benchmarks. Table 3.1 shows the composition of the DCG dataset.

[2]https://github.com/

3.3.1 Data Collection

Code snippets are frequently incomplete, while source files in large projects often rely on external libraries and components that are logically defined in other files. To correctly link all files, the compiler needs knowledge about the location of the external portions. However, the interdependence between files may become quite intricate, and writing build files by hand is often unfeasible in these cases.

Many large projects use build automation tools such as CMake[3] to automate the generation of build files by exploiting definitions of the project's structure. However, this adds another layer of complexity for extracting the intermediate LLVM representations. Therefore, I employed a mix of different approaches to gather data.

Code From NCC There exist several solutions to this problem in literature. The authors of inst2vec [27] trained their embeddings on a large dataset of LLVM-IR code (NCC) which has been made publicly available on their GitHub repository[4]. It is worth noting that there is an overlap between the kernels used in the DevMap dataset (see Table 2.1) and those of NCC. While only a small portion of the dataset is shared (256 kernels over 24,030), training an encoder on the same samples that are used for a high-level task may lead to an overestimation of the results. Therefore, the DCG dataset only employs the subset of benchmarks that are not in common between the two datasets.

Code from CompilerGym Another valuable source of LLVM-IR code was the CompilerGym library [93]. The project is an experimental infrastructure for compiler optimisation, supporting different tasks and compilers. However, it also provides several pre-compiled datasets and benchmarks, whose download links are hard-coded in the library source code[5].

[3] https://cmake.org/
[4] https://www.github.com/spcl/ncc
[5] https://compilergym.com/_modules/compiler_gym/envs/llvm/datasets.html

These datasets are often distributed as large repositories of LLVM-IR bitcode (`.bc`) files that can be easily converted back to bytecode (`.ll`) using the LLVM disassembler (`llvm-dis`[6]). Some are instead retrieved as C/C++ sources and compiled internally. By examining the library's source code, I reconstructed the compilation pipeline for those collections, obtaining LLVM-IR source code for several benchmarks.

Code Compiled from Scratch The DCG dataset also contains a collection of source code files that, to the best of my knowledge, had never been utilised by previous works. I collected open-source projects from the internet and compiled them, following their official build instructions.

For relatively small projects with compilation instructions written in a simple Makefile, obtaining LLVM-IR code simply involved adjusting the compilation commands for each `.c` file, changing the compiler to `clang`, removing incompatible flags and setting up the appropriate flags and output extensions for the production of LLVM-IR files.

This approach is unfeasible for larger projects, due to the complexity of the build automation tools that are often employed. For projects that use CMake, I utilised the `DCMAKE_VERBOSE_MAKEFILE` flag to force the generated Makefiles to print all executed instructions while running. Projects were then compiled as normal, but the full compilation logs were captured and processed, filtering out unrelated commands and modifying the calls to `clang` as explained above. The process resulted in sequences of compilation commands for generating LLVM-IR samples from each compilable file, maintaining all defined flags, paths, and libraries from the real compilation commands.

Benchmarks Description

The DCG dataset is a heterogeneous collection of source code. It contains several libraries for scientific computation (Tensorflow, Blas, eigen), a version

[6]https://llvm.org/docs/CommandGuide/llvm-dis.html

of the Linux kernel and some biological-oriented projects (bowtie2, bwa-mem) [3, 82, 86, 89, 83, 84]. Synthetic datasets are also utilised: gemm_synth and stencil_synth contain procedurally generated linear algebra operations, while CLGen [18] consists of synthetic OpenCL kernels sampled from a LSTM-based LM trained on over 9 k real kernel functions.

POJ104 [92] is a noteworthy collection of code, both for its size (totalling over 47% of the whole DCG dataset) and for its peculiar structure. It collects almost 50 k C++ files written by students as solutions to 104 simple programming problems. In other words, a large number of programs are slight variations of each other and solve the same exact task.

Finally, the Jotai benchmark suite [88] consists of 18, 761 programs sampled from a larger dataset known as AnghaBench [94]. This is a massive collection of over a million C files mined from popular public repositories on GitHub, with the characteristic that all files have been processed by employing a type inference engine to address any missing component and ensure compilability. The subset utilised in Jotai is also rendered executable through a specialised library capable of generating random inputs for the programs and a driver that safely executes them.

Alternative sources

Several alternative paths were explored during the process of collecting LLVM-IR code for the dataset. These options were not explored further for this thesis due to time constraints or other concerns, but they could be valid directions for expanding this work. For instance, AnghaBench [94] could have been used as the only component of the DCG dataset thanks to its size and variety. However, it would have increased training times excessively.

Another potential option for expanding the variety of sources would be the utilisation of LLVM interfaces for other programming languages. For instance, Numba[7] is a LLVM-based compiler capable of transforming Python functions

[7] https://numba.pydata.org/

into high-performance machine code. Similarly, Codon[8] is a LLVM-based compiled language maintaining many syntactic aspects of standard Python. While these options would allow the inclusion of several additional code-bases into the DCG dataset, they both require a substantial effort in terms of partial rewriting of the original Python source code for alignment with the frameworks.

Projects written in Rust were also considered, as Rust compiler `rustc` is a front-end to LLVM. Furthermore, Rust projects are often built using Cargo, a build system and package manager that deals with dependencies and code structure automatically. Obtaining LLVM-IR code for a Rust project simply amounts to running a command analogous to `cargo rustc --emit=llvm-ir`. However, this approach has some problems:

- Rust LLVM-IR files are considerably larger than those obtained by C/C++ files, because *compilation units* in Rust are *crates*, collections of files and possibly entire projects, as opposed to single source files in C/C++. Excessively large files cause several problems during the code-to-graph transformation.

- As of December 2023, the ProGraML library[9] supports at most LLVM version 10.0 and cannot parse later versions. The latest Rust version based on LLVM 10.0 is version 1.45[10]. This version is still under the 2018 edition of Rust idioms, but most contemporary projects use the 2021 edition and cannot be directly compiled with the older version. This could be solved by re-building the ProGraML library adding support to a newer edition of LLVM, or by cloning earlier versions of Rust projects and manually dealing with dependency versions. However, both solutions would be very time-consuming.

[8] https://github.com/exaloop/codon
[9] https://github.com/ChrisCummins/ProGraML
[10] https://github.com/rust-lang/rust/releases/tag/1.45.0

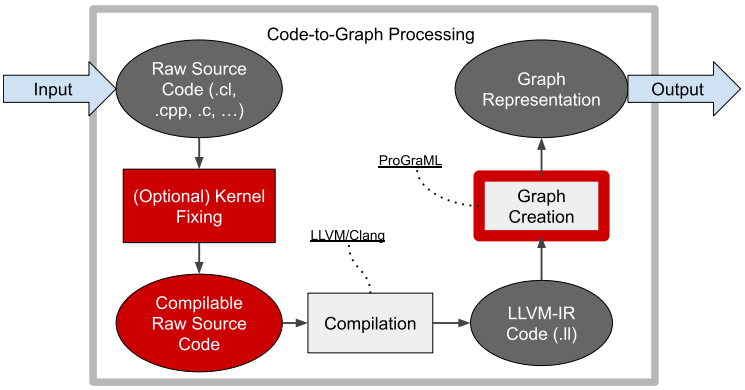


Figure 3.8: Detail of the data processing procedure transforming raw source code into graph encodings. The same colouring rules of Fig. 3.1 apply. The use of external libraries and software tools is explicitly marked with underlined elements connected to their relative components.

3.3.2 Data Processing

Figure 3.8 outlines the processing steps that are required to translate the collected raw source code files into the graph representation described in Section 3.2.1. The process involves:

- Optionally rewriting source code in order to make it compilable (e.g. dealing with undefined elements or fixing imports).
- Compiling the source code into valid LLVM-IR code with a LLVM-based compiler (e.g. Clang [22] for C and C++)
- Parsing the LLVM-IR code to extract variables, constants, instructions, function calls and the relations between them, iteratively composing the graph representation.

Compilation: From Raw Source Code to LLVM-IR

Compilation instructions for the raw source code files of the DCG dataset were mined as explained in Paragraph 3.3.1. These are usually in the form:

```
clang -S -emit-llvm ... -o out.ll in.c/cpp/cl
```

where the flags `-S` and `-emit-llvm` inform the compiler that only the preprocessing and compilation steps should be run and that LLVM-IR code should be produced in the `out.ll` file. Variations of this base command have been used depending on the specific dependencies and characteristics of each sample.

Graph Extraction: From LLVM-IR to Graphs

The ProGraML Python library[11] was employed for producing graph representations from LLVM-IR files. In particular, the library's API exposes functions that autonomously parse LLVM-IR code and output `ProgramGraph` objects. These representations contain lists of nodes, edges and other features.

Nodes contain the `type` attribute (`INSTRUCTION`, `VARIABLE`, `CONSTANT`, `TYPE`), the original snippet of LLVM-IR code that produced it, an ID representing the enclosing basic block and other features. Edges instead contain a `flow` attribute (`CONTROL`, `DATA`, `CALL`, `TYPE`), an optional `position` information and indices for the source and target nodes. The 4th `flow` type is unused in the rest of the program, so it is employed to represent `BACK` edges. These features are subsequently parsed in order to construct the PyG `Data` objects that represent the models' inputs, as explained in Section 3.2.1.

For particularly large LLVM-IR files, the graph creation process may take a long time or break: therefore I set up a timeout of 10 seconds per sample, discarding graphs that excessively slow down this process.

As a side note, I personally contributed to the library development by finding and correcting a breaking bug in the code. My pull request has been merged into the main branch of the library[12].

[11] https://github.com/ChrisCummins/ProGraML
[12] https://github.com/ChrisCummins/ProGraML/pull/212

3.4 Methods for High-Level Tasks

The general knowledge learnt by the DCG encoder through the masked language modelling task can be re-used in order to achieve better generalisation in more specific problems, such as those proposed in Section 2.5. The data processing pipeline, loss functions and model structure have to be slightly re-adapted for solving the novel tasks, but the core engine of the DCG encoder remains the same.

3.4.1 DCG for Heterogeneous Device Mapping

Heterogeneous device mapping on the DevMap dataset is a binary classification task. The objective is to generate a probability distribution over two options (CPU or GPU). The ground truth option is represented as a one-hot encoded vector. Binary cross-entropy is employed as a loss function, training the model to maximise the probability of the correct option for each sample.

Architecture The model for DevMap consists of 2 consecutive modules: the DCG encoder and a binary classifier. The former handles the input graphs and produces dense embeddings by aggregating the node features at the end of message passing. The auxiliary inputs (workgroup and payload size) are concatenated to this final representation. The resulting vector is then processed by the classifier, a sequence of Dense layers with dropout and ReLU activations in between, that produces a probability distribution on two outputs through a final softmax activation function.

Input graphs contain two additional properties: one representing the ground truth label for the sample (CPU: 0, GPU: 1) and the other holding the auxiliary input information (two values expressing the OpenCL transfer and work group sizes).

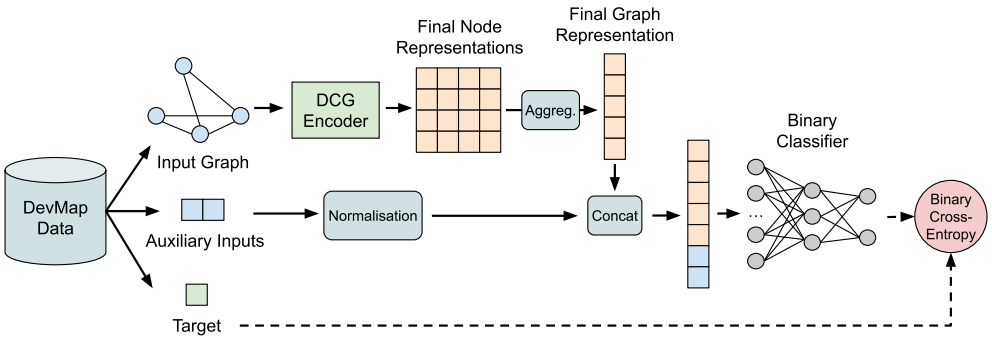


Figure 3.9: Adaptation of the DCG encoder to the task of heterogeneous device mapping.

Auxiliary Inputs The scale of auxiliary values is often large (more than 10^9 for payload size, up to 10^3 for workgroup size), while most DL techniques assume that the range of data lies within $[0, 1]$ or $[-1, 1]$ to avoid issues related to numerical stability [95]. Therefore, the raw auxiliary values are normalised before concatenation following a procedure introduced by Parisi et al. [74]. The normalisation pipeline includes three steps:

- First, a Power Transform makes the distribution more Gaussian-like [96].
- Then, Standard Scaling imposes 0-mean and unary variance to the distribution:

$$z = \frac{(x - \mu)}{\sigma}$$

  where μ and σ are respectively the mean and standard deviation of the distribution.

- Finally, Min-Max Scaling adjusts the values into the range $[-1, 1]$:

$$o = 2\frac{z - z_{min}}{z_{max} - z_{min}} - 1$$

The parameters of these transformations are learnt on the training set and applied on the validation and test set.

Compilation and Source Code Fixing The OpenCL code of the DevMap dataset was compiled with additional flags[13] specifying the language standard of OpenCL 2.0 and including default header files. However, a portion of the DevMap kernels did not compile right away due to undefined external functions and constants being used. Without *defining* the body of said functions, a large part of the computation in those kernels was being dropped.

To capture the full complexity of kernels, the missing elements were reconstructed using the source code from the original benchmarks. Firstly, `extern` elements in the DevMap files were matched to real references in the benchmarks code. Function bodies and constants were copied and adapted into the dataset files. For conditional constant definitions in the original benchmarks, the raw values present in the code from the DevMap files were used.

Thanks to this process, all kernels were successfully compiled and represented in their full complexity. I am not aware of other studies that have applied a similar correction to the source code of this dataset, but I believe it is necessary for an accurate estimation of the results for this task.

3.4.2 DCG for Thread Block Size Prediction

The objective of thread block size prediction on LS-CAT is to assign scores to the 20 available options of thread block sizes, given a kernel in graph representation and the input matrix size. It can be framed as a multi-class classification task (e.g. finding the optimal configuration), but previous works proposed a multi-label setup instead (e.g. assigning a score to the quality of each configuration). The loss function is therefore a sum of per-class binary cross-entropies, where the target is a score representing how close each option is to being optimal, with the score for the optimal thread block size being 1.

[13] `-xcl -cl-std=CL2.0 -Xclang -finclude-default-header`

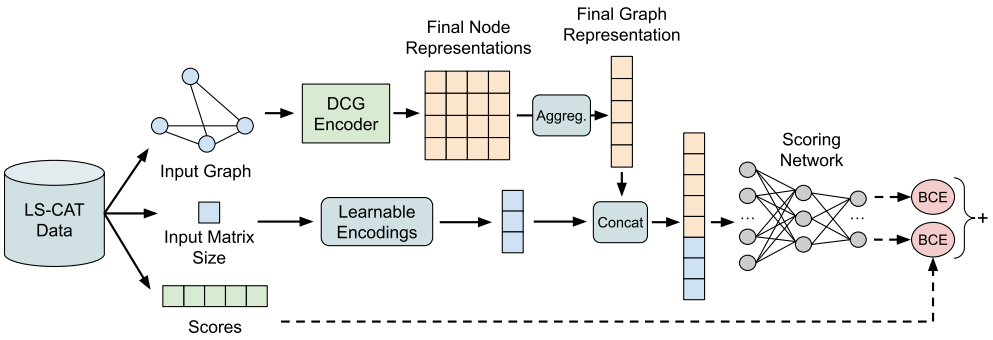


Figure 3.10: Adaptation of the DCG encoder to the task of thread size prediction.

Compilation CUDA files of LS-CAT are compiled with `clang++` and additional flags[14] that allow optimisations and set up compilation only for the kernel functions that are run on the CUDA device.

I managed to compile more kernels (20,043) than those reported by the authors (19,683, as in [79]).

Architecture Similarly to the model used for heterogeneous device mapping, the DCG encoder is employed to produce graph representations that are subsequently processed by the classification layer. The matrix size is processed analogously to the auxiliary inputs in the DevMap model, but this time the values are mapped to learnable embeddings before concatenation.

Graphs for the thread block size prediction task hold the 20 runtimes for each of the thread choices as an additional property. They also contain information about the matrix size that was chosen for the sample.

[14] `-pthread -std=c++11 -stdlib=libstdc++ -O3 -cuda-device-only`

Chapter 4

Results

This chapter describes the experiments that were conducted in order to assess the efficacy of the DCG methodology.

Section 4.1 offers a comprehensive analysis of the newly introduced DCG dataset, while Section 4.1.2 delves into the creation of the DCG vocabulary, an essential component for the processing of all graphs.

Section 4.2 describes the experiments related to the language model training, aimed at evaluating the impact of the optional components that were incorporated into the general GraphMask framework.

Lastly, Section 4.3 shifts the focus to the outcomes achieved by the language model on the high-level tasks outlined in Section 2.5.

4.1 DCG Dataset Analysis

The DCG dataset is a comprehensive and diverse collection of source code samples. A thorough exploration is crucial for attaining optimal results during the language model training phase. Therefore, in this Section, its characteristics will be investigated through statistical analyses and graph-related metrics.

Name	Filtering		Splitting	
	Pre	Post	Train	Val+Test
Blas	300	216	164	52
bowtie2	39	25	19	6
bwa-mem	24	15	11	4
cBench	22	9	8	1
clang	568	66	54	12
CLGen	996	996	788	208
eigen	4,802	3,464	2,744	720
gemm_synth	3,700	3,168	2,488	680
Gromacs	1,196	833	641	192
JotaiBench	5,535	5,535	4,422	1,113
Linux	13,918	8,622	6,819	1,803
LLVM	19,604	18,532	14,728	3,804
MiBench	40	38	29	9
OpenCV	442	254	207	47
POJ104⋆	49,815	49,804	40,213	9,591
stencil_synth	12,800	12,800	10,171	2,629
Tensorflow	1,966	687	546	141
Total	115,767	105,064	84,052	21,012

Table 4.1: Composition of the DCG dataset before and after filtering, including per-subset train/val/test splits distribution. ⋆ 20 problems of POJ104 have been randomly selected for the test-val set.

4.1.1 Dataset Properties

The DCG dataset contains 117,967 raw source code files. However, 45 programs did not compile successfully during processing and 2,155 LLVM-IR files were not parsed into valid graphs. The total number of samples after pre-processing is therefore 115,767, as shown in Table 3.1.

Filtering

The distribution of the number of nodes per graph (see Fig. 4.1) reveals that the pre-processed graphs range from a minimum of 1 to a maximum of 206,103 nodes. The mean of the distribution is 1,342, while the median is 267 and the 90th percentile is 2,117. These statistics highlight the presence of a long tail of exceptionally large graphs.

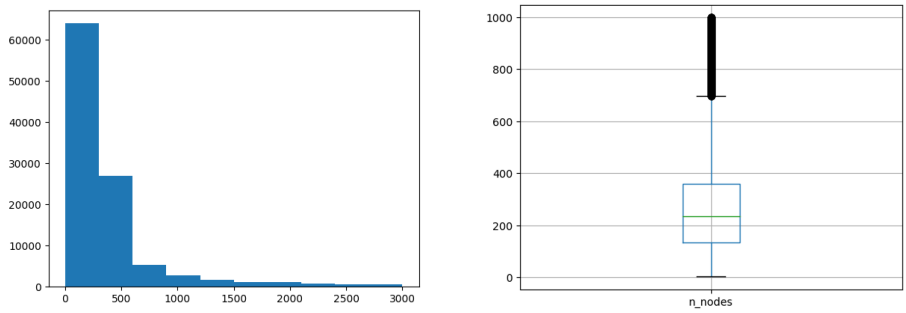
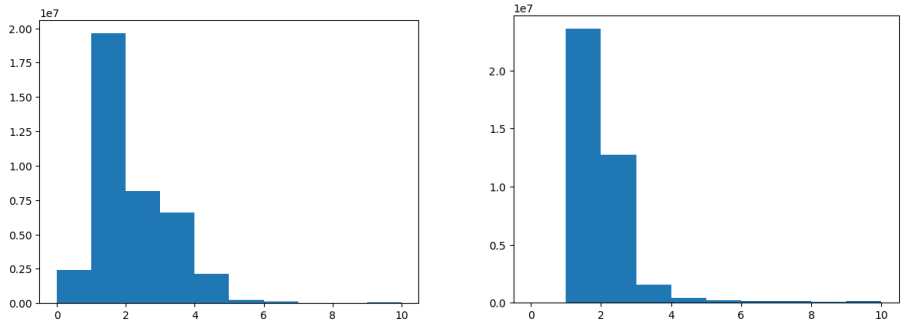


Figure 4.1: Histogram and boxplot of the distribution of the number of nodes per pre-processed graph in the DCG dataset. To better show the distributions, the plots are cut to 3,000 nodes.



(a) Incoming edges.

(b) Outgoing edges.

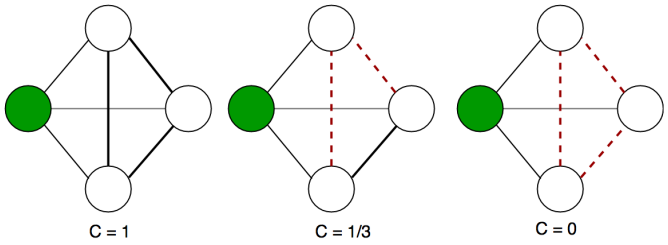
Figure 4.2: Histograms of the distributions of incoming and outgoing edges in the DCG dataset (after pruning).

To prevent Out-of-Memory (OOM) errors, an upper limit on the number of nodes per graph was set: graphs with more than 3,000 nodes were removed from the dataset. Additionally, graphs with a single node were also removed, as they don't represent meaningful cases. These criteria resulted in a dataset of 105,064 samples, constituting 90.75% of the original collection.

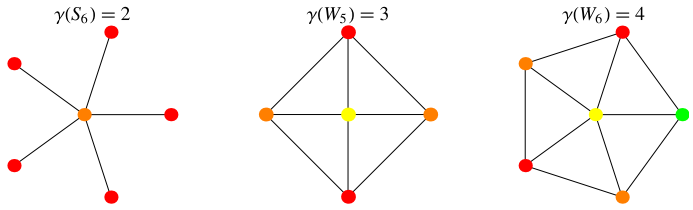
The remaining set of graphs is much more manageable, with an average number of nodes per graph of 375. The per-subset distribution can be seen in Table 4.1.

Edge Analysis

After filtering, the average number of edges per graph is 647. 27.8% of edges are of type `CONTROL`, 65.8% are of type `DATA` and only 6.4% are of type `CALL`.



(a) Average Clustering. Image source: [URL].



(b) Chromatic Number. Image source: [URL].

Figure 4.3: Examples of some classic graph metrics.

As can be seen in Fig. 4.2, most nodes only have 1 or 2 outward connections, while it's typical for nodes to receive up to 4 incoming edges.

It's also not uncommon for nodes to have no incoming connections at all, suggesting that a considerable portion of the graphs may never be influenced by the message passing mechanism. This situation affects 6.1% of nodes, 99.9% of which only have outgoing `DATA` relations.

Other Metrics

There exist a wide number of metrics that can be employed to characterise graphs. For instance, *average clustering* (or *clustering coefficient*) measures the tendency of a graph to create densely connected neighbourhoods (clusters), where higher values indicate the presence of many local cliques (groups of vertices where all nodes are adjacent, see Fig. 4.3a). Graphs in the DCG dataset have an average clustering coefficient of 0.1491, with a standard deviation of 0.042 and a maximum of 0.774.

Another related measure is the *chromatic number*, the minimum number of colours that can be assigned to nodes in a colouring problem, where no two adjacent nodes can have the same colour (see Fig. 4.3b). A graph colouring

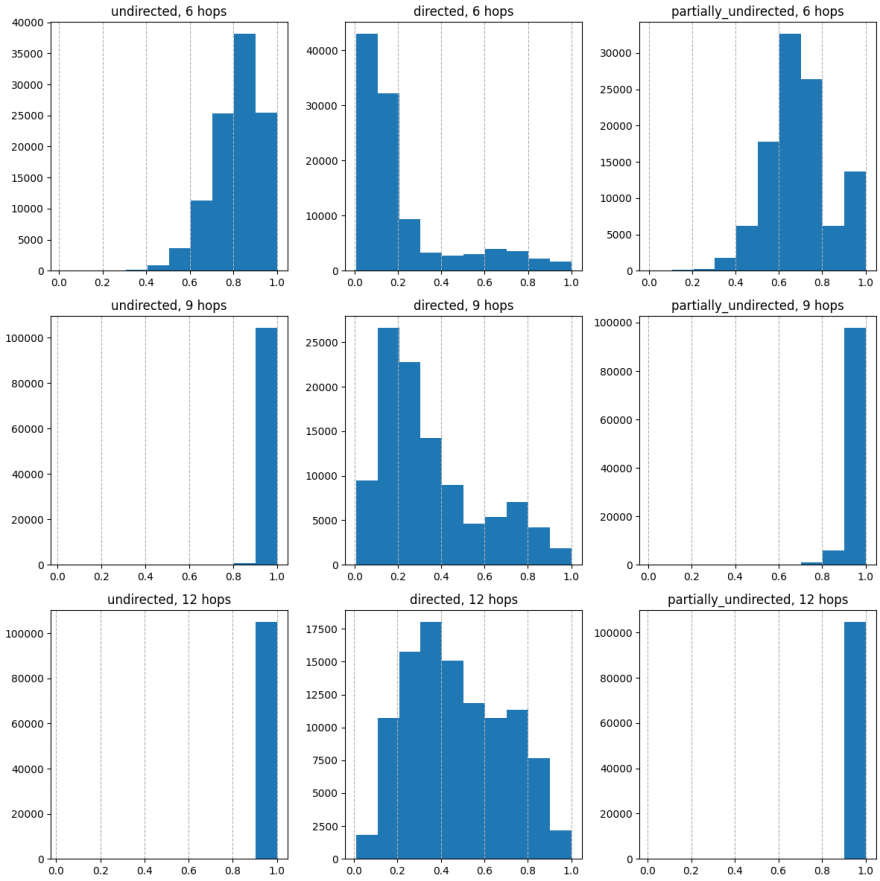


Figure 4.4: Graph expansion distribution in the DCG dataset, expressing the amount of communication in graphs at different levels of message passing cycles ($h = [6, 8, 12]$) and backward edges configurations (undirected, directed and partially undirected)

problem in the DCG dataset can be solved with an average of 2.781 colours, reflecting the statistics of the edge analysis and highlighting the low degree of connectivity among nodes.

A particularly interesting metric is that of *graph expansion* [97], which is defined as the average fraction of nodes that fall within a circle of radius h centred at a random node x. By starting from a large set of random nodes and averaging over the values, this measure can be used to study the amount of information propagation in graphs under different conditions.

In this work, it was used to study the level of communication at various values of h (e.g. message passing iterations) and for different back edges

configurations. Fig. 4.4 shows the distribution of the metric at $h = [6, 9, 12]$ and when no back edges are used (*directed* case), back edges are instantiated only for edges of type `DATA` (*partially undirected* case), or when back edges are instantiated for all standard edges (*undirected* case). The configuration that was chosen as a "standard" in this work is the one in the top-right of the Figure, which represents a balanced amount of communication.

Train/Test Splits

The dataset was split into 2 sets: the training set, containing approximately 80% of the dataset (84,052 graphs), and the validation-test set, covering the remaining 21,012 samples.

Notably, extra care was taken for the splitting of POJ104, in order to avoid including solutions from the same problem in the different sets. If the model observes very similar programs during training, validation and test, final results could be overestimated. To this end, I selected the sources of 20 random problems for the validation and test sets, while reserving the rest for training. Then, programs from the other subsets were randomly sampled up until an 80-20 proportion between sets was reached. Table 4.1 shows the distribution of samples for the different sets.

4.1.2 The DCG Vocabulary

The authors of ProGraML [36] initially utilised inst2vec's embeddings for nodes in their experiments. Nodes for statements were represented as the respective tokens within inst2vec's vocabulary, adding placeholder nodes for input variables and outputs, as shown in Figure 4.5.

Inst2vec's vocabulary has 8,565 tokens, but since they represent entire instructions with their typed inputs and outputs, the actual coverage in programs may be low. For instance, Cummins et al. [98] assembled a dataset of 461 k LLVM-IR files (although almost 400 k of the samples are generated by

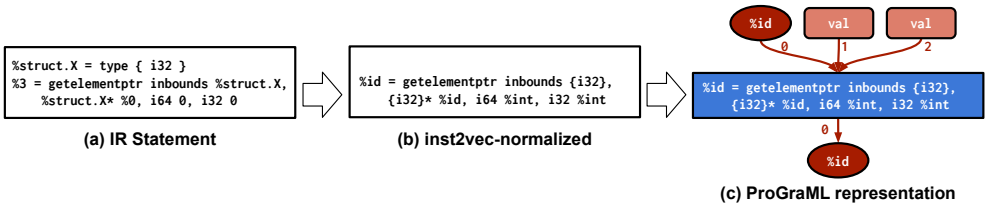


Figure 4.5: Usage of inst2vec embeddings in ProGraML. Image source: [36]

compiling POJ104 repeatedly with different optimisation flags), and observed that the tokens from inst2vec's vocabulary cover a mere 34% of their test set. Instead, by building their own vocabulary on a set of training graphs, the authors are able to separate instruction and data types representations and to achieve a 98.3% coverage using only 2,230 tokens.

Possibly due to the introduction of new, more varied code, the vocabulary of Cummins et al. manages to cover only 94.315% of node tokens from the test set of the DCG dataset. In order to achieve a larger coverage, I followed a similar procedure to extract my own set of tokens from the dataset: the *DCG Vocabulary*.

I started by gathering all tokens that were created by the graph extraction procedure from the DCG dataset's train set. I then filtered out tokens that were only encountered once. However, as the vocabulary still contained over 9 k elements, I also applied an upper bound. I sorted tokens by their frequency within the training set and computed the *cumulative frequency* by progressively adding the values up to 1. A threshold on the cumulative frequency was decided after careful analysis, evaluating the trade-off between vocabulary size and representation capabilities.

A significant result is that 99.5% of the DCG train dataset can be covered with a small vocabulary of only 341 tokens. This finding suggests that the vocabulary used by Cummins et al. [98] may be too large, containing a long tail of infrequent and extremely specific tokens. However, the threshold was set in order to obtain a vocabulary of comparable size. I chose a coverage of 99.87% on the training dataset, resulting in a vocabulary of **2,221** tokens.

Name	Masking			Loss Type	Latent Predictor
	Node	Edge	Advanced		
Vanilla RARE	✓			SCE	GAT
GraphMask-A	✓			CE	GAT
GraphMask-B	✓		✓	CE	GAT
GraphMask-C	✓	✓	✓	CE	GAT
GraphMask-D	✓			CE	DCG
GraphMask-E	✓		✓	CE	DCG

Table 4.2: Proposed configuration studies on the GraphMask model. From left to right, columns indicate the use of Node, Edge and "Advanced" (Complex and Dynamic) Masking (see Section 3.2.3), followed by the loss function (Scaled Cosine Error, SCE, or CrossEntropy, CE) and the Latent Predictor type (either a GAT or the DCG encoder).

The vocabulary only leaves 0.3% of nodes in the test set unmatched: these are mapped to the additional token `[UNK]`.

4.2 Language Model Training Results

As explained in Section 3.2.3, the GraphMask architecture contains several optional components that distinguish it from a standard re-implementation of RARE [51]. Table 4.2 shows the proposed configuration experiments for exploring the various additions that have been implemented. In this Section, the impact of these components on the overall performance of the trained DCG encoder is evaluated in terms of metrics on the LM training task.

4.2.1 Experimental Setup

Dimensionality In order to compare the various configurations of GraphMask, the hyperparameters that are not involved in the choices are kept fixed across experiments. The DCG encoder has an initial node embedding size of 200 and edge embedding size of 16, while the hidden dimensionality is 256.

The network contains $t = 6$ Message Passing layers, with a Dropout of 0.3 between them. The target generator is architecturally equivalent to the DCG encoder, while the latent predictor is implemented as a single-layer GAT or DCG encoder and uses the same hidden dimensionality. The decoder is a simple Dense layer reducing the dimensionality to 128.

Masking The base node masking rate is 0.5. If *complex masking* is active, masked nodes have a probability of 15% to remain intact and a probability of 5% to be replaced by another random token in the vocabulary. When *advanced masking* is used, the node masking ratio is gradually increased from 0.5 to 0.7 with a step of 0.005 during epochs. The replace and keep-same rates are also increased, respectively from 0.15 to 0.2 and from 0.05 to 0.1. When the *edge masking* task is active, the edge mask rate is 0.2. The Degree Decoder is never employed, while the Structure and Edge Type Decoders are active and have an equal weight of 0.2 on the loss.

Learning Rate During preliminary trials, a bug caused the direction of momentum updates to be inverted. In this way, the target generator would never be updated, and its parameters would always remain an exact copy of the DCG encoder's initial weights. The inversion caused updates from gradient descent to be less effective, as the actual updates would be slightly shifted towards the initial parameters of the network, as per Equation 2.7. Despite this problem, the model achieved considerably better results on the DevMap dataset than models trained without the bug with an equal learning rate. Correcting each update towards the initial weights can be compared to using a reduced learning rate to avoid exploring distant areas in the loss landscape. For this reason, all LM training experiments use a learning rate of 3×10^{-7}, decreasing by a factor of 0.05 every 7 epochs of non-improvements on the validation loss.

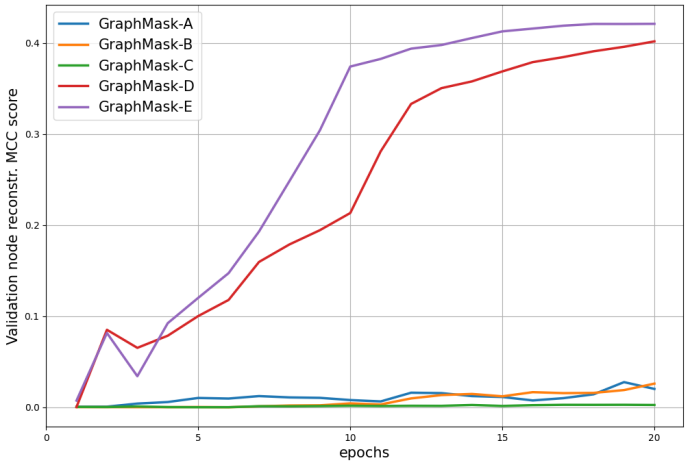


Figure 4.6: Validation MCC score on the task of node reconstruction for the different flavours of GraphMask.

Training Scheme GraphMask is trained for 20 epochs using the Adam optimiser [99], gradient norm clipping of 1.0 and weight decay of 0.001. The training set of the DCG dataset is used as-is, while the "val-test" split is divided equally into 2 parts, establishing different validation (10%) and test (10%) sets for each experiment.

4.2.2 Metrics

For all GraphMask configurations, standard classification metrics such as accuracy, MCC score and macro averages of precision, recall and F1 scores are computed on the validation set of the DCG dataset. In the "Vanilla RARE" configuration, the task is based on regression, so only the validation loss is tracked. However, the loss cannot be compared to the GraphMask experiments, as a different function is employed. Therefore, this configuration will be compared in terms of results on the high-level tasks in Section 4.3.1.

At the end of training, the same metrics are also computed on the test set to obtain definitive results. I log and track all metrics on Weights & Biases [100]. Fig. 4.6 shows the difference in terms of MCC score among the presented GraphMask configuration experiments.

GraphMask-D and E yield the best results, with a rapidly increasing validation MCC score. As the main difference with respect to other configurations is the use of the DCG encoder as a latent predictor, I hypothesise that this component is particularly important for learning. Indeed, it enables the model to utilise the positional information within graph edges. Advanced masking also seems to give an edge to GraphMask-E: solving a harder problem may lead the model to better generalisation. GraphMask-E was therefore chosen as the "standard configuration" for the language model training procedure.

4.3 High-Level Tasks Results

At the end of the language model training, the parameters of the DCG Encoder and the matrix containing the DCG Embeddings are extracted from GraphMask. Depending on how these pre-trained components are handled, I define five tests that aim to explorea the quality of this methodology.

The tests are summarised in Table 4.3 and can be described as follows:

- In *Test A*, the initial features of all nodes originate from the trained DCG embeddings. For Test A1, I also initialise the parameters of the DCG encoder from the LM trained with the GraphMask-E configuration. Conversely, in Test A2, the DCG encoder starts with random weights.

- In *Test B*, the same methodology of Test A1 is employed, but the language model is trained on a best-effort reconstruction of the NCC dataset [27]. A novel vocabulary was also created for this experiment, using the same approach as the DCG vocabulary. The LM trained using the DCG dataset is expected to yield better validation loss and accuracy metrics during training, as it is approximately 4.5 times larger than NCC [27]. The larger dataset size correlates with increased exposure to a broader spectrum of programming patterns and reduced overfitting risks.

	LM Training			Task-Specific Model Init	
Test	Dataset	Vocabulary	Model	Embedding	Encoder
A1	DCG	DCG	DCG	PreTrained	PreTrained
A2				PreTrained	Random
B	NCC	DCG*	DCG	PreTrained	PreTrained
C	NCC	NCC	NCC	PreTrained	Random
R	n.a.	NCC	n.a.	Random	Random

Table 4.3: Possible configurations for task-specific trainings. NCC refers to the dataset and methodology used for training the inst2vec embedding space [27]. DCG components, as introduced in this work, include the DCG dataset (Sec. 3.3), model (Sec. 3.2.2) and vocabulary (Sec. 4.1.2). The DCG* vocabulary in Test B is constructed following the methodology of this work, but differs from the standard DCG vocabulary due to a different code collection. The portion on the right pertains to the usage of pre-trained components in the initialisation of models for high-level tasks.

- In *Test C*, I utilise the inst2vec vocabulary and embeddings as external components, following the methodology outlined by Cummins et al. [36]. This test does not employ any of the components from the LM training phase. Therefore, its primary objective is to assess the enhancements that the DCG methodology offers compared to the processing method of inst2vec [27].

- Finally, *Test R* offers a baseline for all previous experiments. Similarly to test C, the inst2vec vocabulary is utilised to map graph elements to tokens. However, the initial node embeddings are completely random and are trained solely through the high-level task.

To ensure a fair comparison with other tests, the DCG encoder in Test B was trained with the same configuration of Test A1, but for 4.5 times the epochs (90 instead of 20), maintaining approximately the same number of training steps.

4.3.1 Heterogeneous Device Mapping

Dataset Analysis

The DevMap dataset is notoriously small, and a naïve utilisation can lead to an over- or underestimation of training techniques. Parisi et al. [74] conducted an in-depth examination of the dataset, revealing several types of imbalance issues. As illustrated in Table 2.1, the distribution of samples across benchmarks is significantly imbalanced, with NPB accounting for over 75% of the total amount.

Moreover, some of the benchmarks exhibit a particularly imbalanced label distribution, with a clear majority of samples belonging to a single class. The authors also investigated the self-similarity of kernels within suites, observing that the NPB, Polybench and SHOC benchmarks all contain kernels with a high degree of similarity.

ProGraML graphs constructed on the DevMap dataset contain, on average, 929.8 nodes, albeit with a substantial degree of variation, as indicated by the standard deviation of 1607.5. Approximately 3% of the graph nodes are assigned to the `[UNK]` token by the DCG vocabulary, although in almost half (331) of the total samples, all nodes are covered by the dictionary.

Regarding graph metrics, the average clustering coefficient distribution mirrors that of the DCG dataset, averaging at 0.144, but with a lower maximum of 0.231. Nearly all graph colouring problems can be solved using precisely 3 colours, while the node expansion metric distribution for the $h = 6$ case bears a close resemblance to that of the DCG dataset (in Fig. 4.4).

Experimental Setup

Dimensionality The DCG encoder uses the same dimensionalities of the language model training experiments. The normalised auxiliary features are transformed into a 32-dimensional tensor by a Dense layer before concatenation with the graph encodings, producing an input for the classifier with size

288. The classifier has a single hidden layer with an output size of 64, followed by the output layer producing a probability distribution on two outputs.

Training Scheme The small size of the DevMap dataset can lead to large fluctuations of metrics, due to the training and testing sets not properly representing the distribution of source code. A common training strategy for dealing with the problem is *Stratified K-Fold Cross-Validation* (SKF). SKF randomly partitions the dataset into K folds and, in turn, the model is trained on $K - 1$ folds while the other one is left out for evaluation. In particular, the imbalance in the dataset labels is maintained across the folds. The final result is then computed as the mean among the K iterations.

In all DevMap experiments, SKF with 10 folds is employed for evaluation. The results are also averaged over 10 random trials with different random seeds, totalling 100 separate experiments per training. The models are trained with an Adam optimiser and a starting LR of 2.5×10^{-4} reduced by a factor of 0.05 on loss plateaus. A label smoothing with factor 0.05 is applied for regularisation.

Furthermore, class weights are assigned to address the imbalance between the two classes. Given a function $N(c_i)$, returning the number of samples for class c_i in the training set, the weight assigned to samples of that class in the loss function is:

$$w_{c_i} = \frac{max_i(N(c_i))}{N(c_i)}$$

Metrics

Device mapping solutions are commonly evaluated using binary classification metrics. GPU and CPU labels are designated as “positive” and “negative” respectively. Predictions are then categorised into distinct groups: *True Positives* (TP) and *True Negatives* (TN) encompass predictions that accurately match the respective labels. *False Positives* (FP) represent CPU-labelled samples erroneously classified as GPU, while *False Negatives* (FN) comprise the

opposite cases. Several metrics can be derived based on the cardinalities of these groups (respectively TP, TN, FP, FN):

- Accuracy, the fraction of correct predictions over all samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

- Precision, the fraction of positive predictions that are correct:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

- Recall, the fraction of positives that were correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

- F1 score, the harmonic mean of precision and recall.

Parisi et al. [74] suggest to also use the Matthews Correlation Coefficient (MCC), as it is considered to be more effective in the case of imbalanced datasets:

$$\text{MCC} = \frac{TN \times TP - FN \times FP}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.1}$$

Results

Table 4.4 provides accuracy results for the five training configurations. An additional decision point was investigated: whether to maintain the initial embeddings without further training (*Embeddings frozen*) or to allow embeddings to change along with the rest of the model parameters (*Embeddings not frozen*).

The results suggest that it is preferable to fix the initial embeddings rather than to enable further tuning for the high-level task. This aligns with observations by Cummins et al. [36], who reported a similar behaviour, blaming the

	Embeddings not frozen		Embeddings frozen	
Test	NVIDIA	AMD	NVIDIA	AMD
A1	.8529 ± .0029	.8879 ± .0029	.8524 ± .0020	**.8901 ± .0018**
A2	.8494 ± .0033	.8846 ± .0018	**.8553 ± .0042**	.8856 ± .0018
B	n.a.	n.a.	.8500 ± .0028	.8865 ± .0042
C	.8262 ± 0031.	.8685 ± .0022	.8146 ± .0034	.8591 ± .0019
R	.8472 ± 0019.	.8847 ± .0017	.8524 ± .0033	.8831 ± .0030

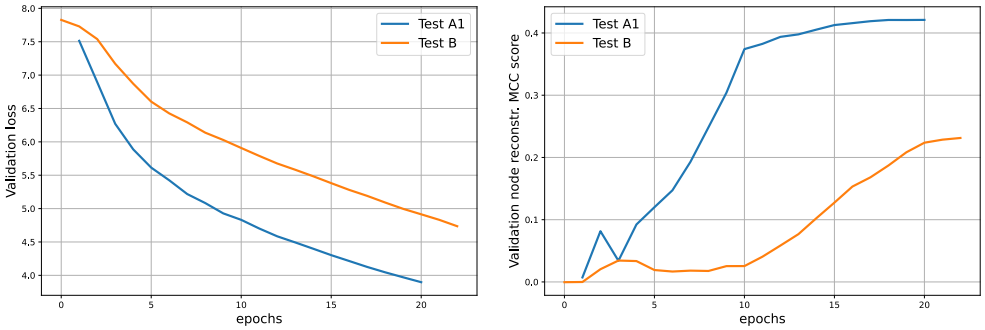
Table 4.4: Test accuracy scores (mean and 95%-confidence interval) for the different task-specific training configurations on DevMap.

increase in number of parameters and, consequently, training complexity. Notably, the embedding matrix comprises 444 k parameters, constituting nearly one-third of the language model's total weights (1,358,520). This, paired with the small size of the DevMap dataset, could lead to overfitting.

The results are all very similar and fall within the same confidence interval, except for Test C, which exhibits a considerable drop in performance. In particular, Test A1 yields outstanding results on the AMD subset of DevMap. The same configuration also results in a slightly better average performance (.8713) than that of Test A2 (.8705) and Test B (.8683).

Test R (where weights of the model and node embeddings are initially random) is also very similar to Tests A and B, with an average accuracy over the two dataset splits of .8678. Therefore, it cannot be definitively concluded that a large-scale LM pre-training through the DCG methodology followed by task-specific fine-tuning would provide a considerable improvement over the sole task-specific training.

Despite this, further investigating the results of tests A1 and B demonstrates that the DCG dataset empowers the LM to produce more expressive representations, as can be observed by the lower decoding loss and higher node classification MCC score in Fig. 4.7.



(a) Node reconstruction loss. (b) Node reconstruction MCC.

Figure 4.7: Validation MCC and loss for tests A1 and B during language model training. Epochs are divided by 4 for Test B.

Moreover, the LM trained on the DCG dataset exhibits enhanced generalisation capabilities, as the NCC dataset partially overlaps with the DevMap dataset yet still produces a lower score. In contrast, the DCG dataset avoids by design any intersection with the DevMap dataset.

These findings, along with the poor results achieved by Test C, support the hypothesis that the deployed MGAE approach provides an edge to the model with respect to previous state-of-the-art graph-based approaches.

Comparison with Literature In Table 4.5, I compare the results obtained by the DCG methodology to previous solutions in terms of average test accuracy. The best configuration (Test A1) improves upon all previous graph-based source code analysis models by more than 3%, setting a new state-of-the-art in heterogeneous device mapping with no substantial hyperparameters tuning.

Model	Source	NVIDIA	AMD	Mean
DeepTune	[19]	.805	.814	.810
Inst2vec	[27]	.820	.828	.824
ProGraML	[36]	.800	.866	.833
DeepLLVM	[26]	.823	.853	.838
CDFG	[30]	.814	.864	.839
Enhanced-DeepTune	[20]	.815	.874	.844
DCG-A1 (Ours)		**.852**	**.890**	**.871**

Table 4.5: Test accuracy results on DevMap comparing the best configuration for my DCG methodology against previous works in literature.

	DevMap Test Results	
Name	Acc.±	MCC
Vanilla RARE	.8449 ± .0019	.6862 ± .0040
GraphMask-A	.8569 ± .0030	.7114 ± .0058
GraphMask-B	.8515 ± .0017	.7008 ± .0032
GraphMask-C	.8494 ± .0035	.6959 ± .0070
GraphMask-D	**.8582 ± .0032**	**.7141 ± .0062**
GraphMask-E	.8488 ± .0028	.6945 ± .0056

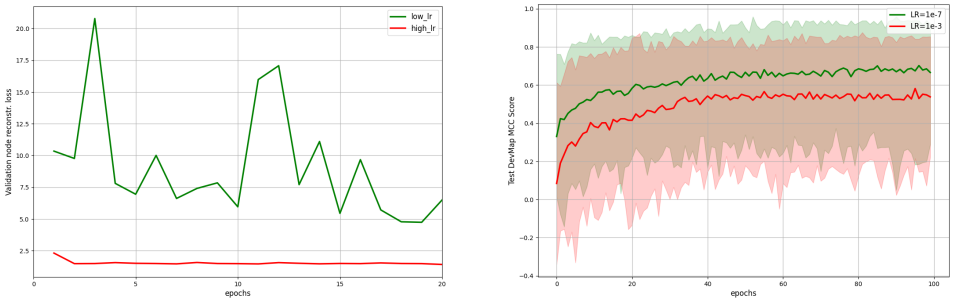
Table 4.6: Average accuracy and MCC score (with 95%-confidence interval) obtained on the DevMap test experiments after LM training with the different variants of GraphMask. The row in **bold** represent the configuration yielding the best results, while the underlined row indicates the default LM configuration.

It should be noted that, as explained in Paragraph 3.4.1, some of the files in the DevMap dataset have been partially modified. It is impossible to know for certain how other research works have dealt with external dependencies, so it is unclear whether some of these comparisons are fair or not. However, the changes that were implemented for my tests provide realistic results for the task.

Encoder Configuration Studies

The different architectural choices in GraphMask can also be assessed based on the benefits they confer to the performance of high-level tasks. The embeddings and encoder weights derived from each of the LM training configurations detailed in Table 4.2, were utilised as starting checkpoints for the DevMap training.

The results are reported in Table 4.6. They are largely similar, but nonetheless highlight a divergence between the LM and task-specific training phases. An optimal solution for the masked node reconstruction task may not necessarily translate to an ideal configuration for high-level tasks.



(a) Validation loss during pre-training (lower is better).

(b) Test MCC score on DevMap task (higher is better).

Figure 4.8: LM training and task-specific progressions with different learning rates. The only difference between the 2 experiments is the pre-training LR: the red model was pre-trained with a LR of 1×10^{-3}, while the green model with 1×10^{-7}. Multiclass MCC score is defined in [101].

This discrepancy becomes particularly evident when considering the training Learning Rate (LR). Figure 4.8a illustrates the difference between using a lower and a higher LR (3×10^{-7} and 1×10^{-3}) in terms of LM training loss on the GraphMask-C experiment. Interestingly, using a higher LR guarantees quicker convergence during this phase. However, these benefits do not transfer to the high-level task, where encoder weights trained with a lower LR yield substantially better results.

To determine whether a genuine difference exists between the means of the presented scores or if their variations are merely coincidental, T-tests are conducted on each pair of ablation experiments. A T-test considers two groups of results and yields a confidence level for the *null hypothesis*, the assumption that their mean is exactly the same and variations are only due to chance. Consequently, a low confidence (e.g. 0.05) is an indicator of a meaningful difference between the two populations.

Figure 4.9 presents the outcome of this statistical analysis. It can be observed that, with different degrees of confidence, all distributions significantly deviate from the results of Vanilla RARE. This confirms the effectiveness of casting the pre-training problem as a classification task.

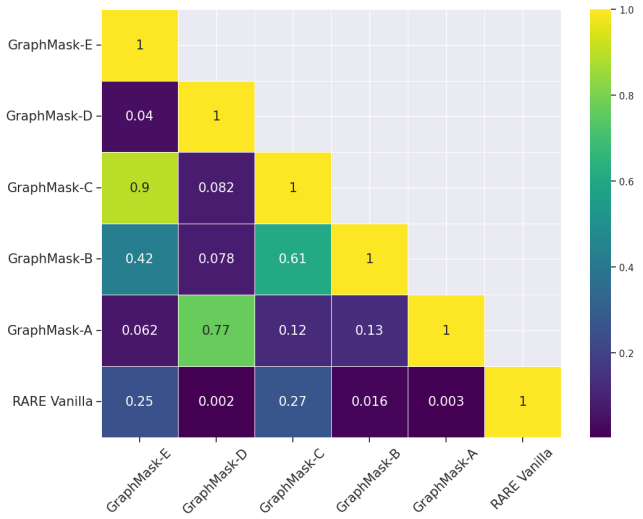


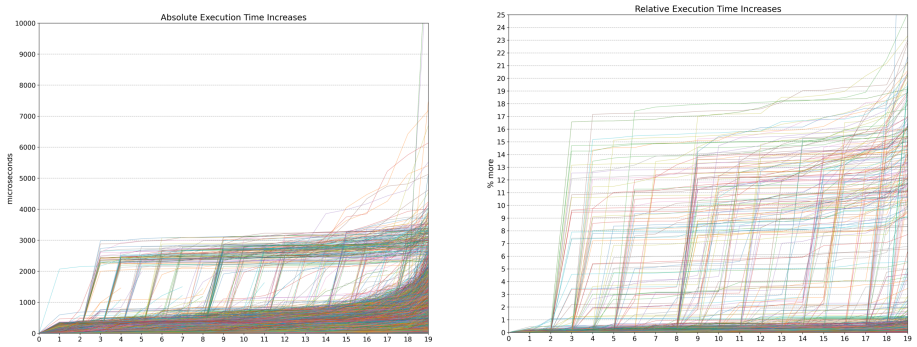
Figure 4.9: Null hypothesis confidence scores for all combinations of GraphMask ablation experiments. T-tests were conducted on groups of test accuracy scores resulting from 10 different experiments with each model.

4.3.2 Thread Size Prediction

Dataset Analysis

The LS-CAT dataset labels each combination of kernels, input matrix sizes and thread block size with an average execution time in microseconds. The majority of these combinations exhibit execution times that are closely aligned with the optimal configuration, with a 3-4 milliseconds difference, as depicted in Fig. 4.10. Nonetheless, when considering relative time improvements, certain choices can lead to a reduction in execution times by as much as 25%.

The graphs produced by the samples in LS-CAT are smaller than those of the DCG dataset. They contain, on average, 266.5 nodes, with a standard deviation of 560.65. Less than one node per graph is mapped to an `[UNK]` token by the DCG Vocabulary (0.8837, with a standard deviation of 6.25). On the other hand, inst2vec's vocabulary would leave about 20.3 nodes per graph as `[UNK]` tokens, with an even larger variability (standard deviation of 55.6). In both cases, the code elements that are assigned `[UNK]` tokens all have highly specific signatures, often representing arrays of specific dimensions or structs. This analysis proves the superiority of the DCG Vocabulary for this task.



(a) Absolute increases (μs). (b) Relative increases (%).

Figure 4.10: The increase in average execution times by changing the thread block size for 10,000 random kernel-matrix size pairs. The 20 available times are sorted in increasing order and compared against the best option, which is always at the origin of the plot.

The average clustering coefficient has a similar distribution to that of the DCG Dataset, with a mean of 0.134 and a standard deviation of 0.036. The maximum (0.267) is considerably inferior to that of the DCG Dataset. The reduced number of nodes also has an impact on the graph expansion distribution, which has generally higher values. This can be seen in Fig. 4.11, where the graph connectivity is measured with lower hs than in Fig. 4.4, but the metric is nonetheless higher.

Experimental Setup

Dimensionality Several architectural decisions from the DevMap architecture are retained: i) The DCG encoder maintains the same dimensionalities of the language model training experiments. ii) The "auxiliary" input matrix sizes are mapped to 32-dimensional tensors via an embedding layer before concatenation. iii) The classifier employs a single hidden layer with an output size of 64 before the output layer. However, in this task, the final layer of the classifier yields 20 outputs.

Training Scheme The model is trained for 40 epochs using 10-fold cross-validation with an Adam optimiser, starting from a LR of 2.5×10^{-4} that is

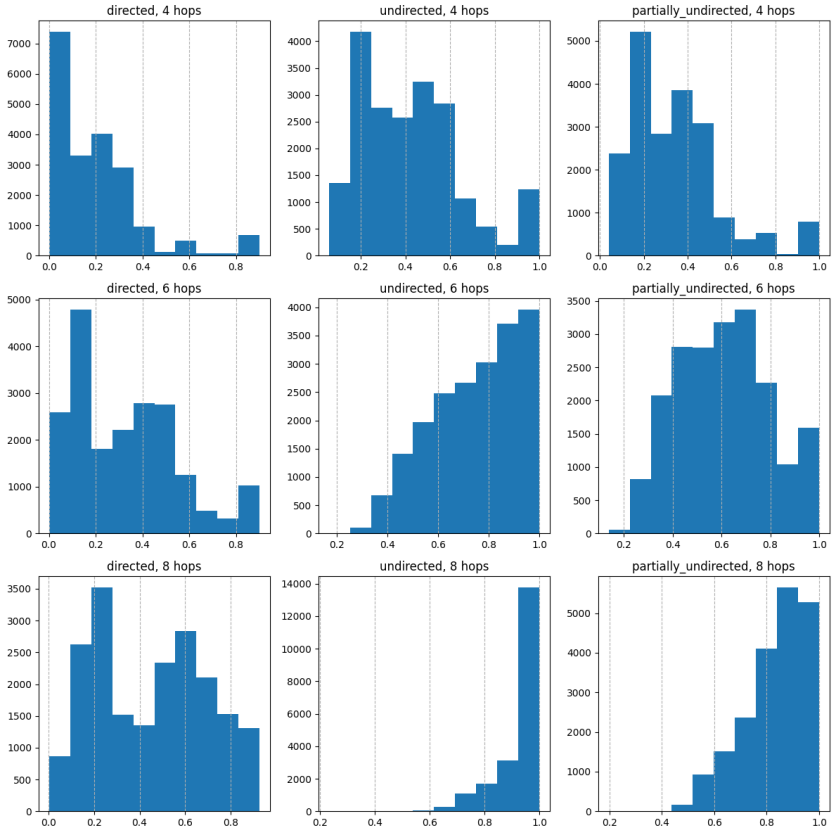


Figure 4.11: Graph expansion distribution in LS-CAT ($h = [4, 6, 8]$, backward edges configurations: undirected, directed and partially undirected)

manually reduced over time.

Metrics The problem is framed as a multi-label classification task, recognising that non-optimal choices may still achieve remarkable execution times relative to the average case scenario. In this approach, the model receives penalties based on a quantitative evaluation of their deviation from the optimal class, providing stronger negative feedback for larger errors.

Given a combination of kernel k and matrix size i, the quality of a specific block size b is measured by dividing by the minimum time for the kernel-matrix size pair by the execution time T of the whole combination:

$$p_{(k,i,b)} = \frac{\min_x T_{(k,i,x)}}{T_{(k,i,b)}}.$$

Name	Top-1 Acc.	Top-3 Acc.	Perf.
Random Selection	0.050	0.015	0.923
Fixed Selection (1D, 1024)	0.056	0.016	0.933
fastText+LSTM	n.a.	n.a.	0.948
DCG-A1	**0.477**	**0.751**	**0.997**

Table 4.7: Test metrics for thread size prediction on LS-CAT.

This metric is named *performance* [79] and expresses how close each choice is to being optimal on a 0-1 scale. The global performance metric is computed by averaging the performance values of the thread blocks predicted as best scoring.

While useful for evaluation, this metric tends to have high values for almost all choices. For instance, the reported performance metric for a random classifier is 0.94 [79]. For this reason, these scores are further processed with a transformation aimed at raising the entropy in the distribution:

$$m_{(k,i,b)} = \frac{e^{Cp_{(k,i,b)}}}{\sum_x e^{Cp_{(k,i,x)}}}$$
$$l_{(k,i,b)} = \frac{m_{(k,i,b)}}{\max_x m_{(k,i,x)}}$$

where C is a constant set to 12 and l represents the actual training labels.

Results

I compare a model trained following the DCG-A1 configuration against the model by Bjertnes et al. [79] (described in Section 2.5.2). The top-1 and top-3 accuracies, as well as the performance results, are obtained by averaging over five different experiments with varying random seeds and are reported in Table 4.7. The same set of metrics is also reported for random and fixed selection (i.e. always choosing the largest thread block size). DCG achieves state-of-the-art results on the task, with a performance of 0.997 and a top-3 accuracy of over 75%. This represents an improvement in performance of almost 5% with respect to the previous solution.

Chapter 5

Conclusions

To facilitate compile-time decisions in a setting of growing complexity and diversity of modern computing systems, recent studies have employed machine learning and neural networks for source code analysis. Some of these solutions come from the field of NLP, such as language models, large parametric systems that produce semantically rich representations of the input by extracting patterns of language that are relevant for the tasks. Self-supervised learning techniques enable LMs to acquire general knowledge from vast collections of language artefacts, enhancing their ability to generalise across different tasks.

In the domain of source code, the learned patterns may encapsulate knowledge about software engineering practices and properties such as efficiency and complexity. However, certain properties are intrinsically linked to data-instruction dependencies or execution sequences. To effectively capture these patterns, graph-based representations have recently been adopted, offering enhanced expressiveness for high-level code concepts.

In this thesis, I explored several graph-based representations, focusing on ProGraML [36], one of the most comprehensive, flexible, and easy-to-use. I extended ProGraML by introducing *back edges*, strengthening communication within the graphs. I then presented Graph Neural Networks, a family of deep learning models that enable learning on graph data. GNNs refine node information through the "message passing" mechanism, iteratively aggregating

local knowledge based on graph structure.

Similar to NLP-based LMs, a wide range of self-supervised techniques exist for training GNNs as general language models of graph data. In particular, I focused on the framework of Masked Graph AutoEncoders (MGAEs), a simple but powerful technique with many analogies to Masked Language Modelling (MLM) in NLP. I then developed "DeepCodeGraph" (DCG), a procedure that combines masked graph autoencoders and graph-based representations of source code. DCG constructs a general language model that is naturally capable of reasoning on the high-level patterns of software. To the best of my knowledge, DCG is the first methodology to utilise MGAE for self-supervised training in source code analysis. Both the language model and the self-supervised framework were comprehensively described from both design and technical perspectives. Additionally, an extensive analysis was conducted to determine the optimal configuration for utilising all modules effectively.

I also collected and thoroughly analysed a novel large-scale collection of graph-based source code representations, encompassing approximately 100 k samples from a variety of open-source projects and benchmarks: the DCG dataset. This dataset provides a vast and diversified repository for effective LM training, and empirical findings suggest that the derived information empowers the language model to a greater extent compared to previous datasets, such as NCC [27]. The data collection and processing pipeline are extensively discussed as well, highlighting several alternative options that could enhance the results in this work even further.

The DCG methodology is evaluated on two high-level tasks that are commonly employed as reference benchmarks: i) mapping computational kernels to compute units (heterogeneous device mapping) and ii) determining the most efficient thread block size for a GPU kernel (thread block size prediction). Datasets for both tasks are analysed, and criticalities are identified and addressed. For instance, the source code for the task of DevMap was extended, as it did not fully capture the complexity of programs.

The DCG methodology surpasses previous graph-based techniques designed for code optimisation. On heterogeneous device mapping, DCG achieves an average accuracy of over 87%, improving upon previous graph-based models by 3.2%. In thread block size prediction, DCG achieves a 4.9% enhancement, reaching a score of 0.997 according to the LS-CAT benchmark performance metric.

In conclusion, the questions presented in the Introduction of this thesis have been addressed extensively. It has been demonstrated that a learning algorithm is capable of identifying valuable patterns in software. Moreover, the utilisation of alternative representations has proven beneficial by explicitly representing particularly useful patterns. Additionally, the general knowledge of source code learned by the DCG LM has been successfully applied to program execution optimisation, improving upon previous works.

5.1 Future Works

One potential enhancement for this work involves expanding the dataset by incorporating additional samples, leveraging comprehensive compilable source code datasets such as Anghabench [94].

Another option for refinement could be the optimisation of the DCG vocabulary. I highlighted how almost the entirety of nodes in the current version of the dataset can be covered by a subset of the DCG vocabulary which is more than 6 times smaller. The choice of employing a larger vocabulary was made for comparative purposes with similar works [98], but utilising a constrained vocabulary could yield substantial advantages, including a reduction in parameters, complexity, and training times.

Moreover, investigating the misalignment between the language model and high-level tasks is crucial. Exploring alternative self-supervised learning approaches or introducing novel auxiliary tasks may enhance the coherence between the two training phases, thereby leading to improved overall results.

Bibliography

[1] D. R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[2] M. Davis. *The Universal Computer: The Road from Leibniz to Turing, Third Edition*. CRC Press, Inc., USA, 3rd edition, 2018. ISBN: 1138502081.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL: `https://www.tensorflow.org/`. Software available from tensorflow.org.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR*, abs/1912.01703,

2019. arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[5] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, USA, 2004. ISBN: 0521607647.

[6] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, 2018. DOI: 10.1145/3212695. URL: https://doi.org/10.1145/3212695.

[7] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012. DOI: 10.1109/ICSE.2012.6227135.

[8] D. Hiemstra. *Language Models*. In *Encyclopedia of Database Systems*. L. LIU and M. T. ÖZSU, editors. Springer US, Boston, MA, 2009, pages 1591–1594. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_923. URL: https://doi.org/10.1007/978-0-387-39940-9_923.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[10] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, 2013. DOI: 10.1109/MSR.2013.6624029.

[11] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, and S. Tan. Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP. *CoRR*, abs/2112.10508, 2021. arXiv: 2112.10508. URL: https://arxiv.org/abs/2112.10508.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space, 2013. arXiv: 1301.3781 [cs.CL].

[13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *CoRR*, abs/1706.03762, 2017. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[15] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training, 2018.

[16] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations, 2018. arXiv: 1802.05365 [cs.CL].

[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805, 2018. arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[18] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99, 2017. DOI: 10.1109/CGO.2017.7863731.

[19] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-End Deep Learning of Optimization Heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, pages 219–232. IEEE Computer Society, 2017. DOI: 10.1109/PACT.2017.24. URL: https://doi.org/10.1109/PACT.2017.24.

[20] P. Vavaroutsos, I. Oroutzoglou, D. Masouros, and D. Soudris. Towards making the most of NLP-based device mapping optimization for OpenCL kernels. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6, 2022. DOI: 10.1109/COINS54846.2022.9855002.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[22] Clang. URL: https://clang.llvm.org/ (visited on 08/11/2023).

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[24] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva. Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, Las Vegas, NV, USA. Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3317789. URL: https://doi.org/10.1145/3316781.3317789.

[25] *TF–IDF*. In *Encyclopedia of Machine Learning*. C. Sammut and G. I. Webb, editors. Springer US, Boston, MA, 2010, pages 986–987. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_832. URL: https://doi.org/10.1007/978-0-387-30164-8_832.

[26] F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva. Exploration of Convolutional Neural Network models for source code classification. *Eng. Appl. Artif. Intell.*, 97:104075, 2021. DOI: 10.

1016/j.engappai.2020.104075. URL: https://doi.org/10.1016/j.engappai.2020.104075.

[27] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3589–3601, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html.

[28] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.*, 78:1809–1827, 2013. URL: https://api.semanticscholar.org/CorpusID:12013393.

[29] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning Distributed Representations of Code, 2018. arXiv: 1803.09473 [cs.LG].

[30] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillón. Compiler-based graph representations for deep learning models of code. In L. Pouchet and A. Jimborean, editors, *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, pages 201–211. ACM, 2020. DOI: 10.1145/3377555.3377894. URL: https://doi.org/10.1145/3377555.3377894.

[31] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. *CoRR*, abs/1711.00740, 2017. arXiv: 1711.00740. URL: http://arxiv.org/abs/1711.00740.

[32] K. Arya, M. Desai, and P. Desai. A competitive pathway from high-level programs to hardware specifications, August 2023.

[33] M. Wolf. Chapter 5.3 - Models of Programs. In M. Wolf, editor, *Computers as Components (Fifth Edition)*, The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, fifth edition edition, 2023. DOI: https://doi.org/10.1016/B978-0-323-85128-2.00005-0. URL: https://www.sciencedirect.com/science/article/pii/B9780323851282000050.

[34] Z. S. Harris. Distributional Structure. *WORD*, 10(2-3):146–162, 1954. DOI: 10.1080/00437956.1954.11659520. eprint: https://doi.org/10.1080/00437956.1954.11659520. URL: https://doi.org/10.1080/00437956.1954.11659520.

[35] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.*, 17(4):32:1–32:27, 2020. DOI: 10.1145/3418463. URL: https://doi.org/10.1145/3418463.

[36] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, June 2021. URL: https://proceedings.mlr.press/v139/cummins21a.html.

[37] XLA. URL: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html (visited on 08/22/2023).

[38] J. Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013. URL: http://dl.acm.org/citation.cfm?id=2488173.

[39] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko. A Gentle Introduction to Graph Neural Networks. *Distill*, 2021. DOI: 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

[40] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural Message Passing for Quantum Chemistry. *CoRR*, abs/1704.01212, 2017. arXiv: 1704.01212. URL: http://arxiv.org/abs/1704.01212.

[41] D. Pujol-Perich, J. Suárez-Varela, M. Ferriol, S. Xiao, B. Wu, A. Cabellos-Aparicio, and P. Barlet-Ros. IGNNITION: Bridging the Gap between Graph Neural Networks and Networking Systems. *IEEE Network*, 35(6):171–177, 2021. DOI: 10.1109/MNET.001.2100266.

[42] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018. arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261.

[43] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks, 2018. arXiv: 1710.10903 [stat.ML].

[44] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks?, 2019. arXiv: 1810.00826 [cs.LG].

[45] B. Weisfeiler and A. Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

[46] L. Wu, H. Lin, Z. Gao, C. Tan, and S. Z. Li. Self-supervised Learning on Graphs: Contrastive, Generative,or Predictive, 2021. arXiv: 2105.07342 [cs.LG].

[47] M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski. DINOv2: Learning Robust Visual Features without Supervision, 2023. arXiv: 2304.07193 [cs.CV].

[48] Z. Hou, X. Liu, Y. Cen, Y. Dong, H. Yang, C. Wang, and J. Tang. GraphMAE: Self-Supervised Masked Graph Autoencoders, 2022. arXiv: 2205.10803 [cs.LG].

[49] Z. Hou, Y. He, Y. Cen, X. Liu, Y. Dong, E. Kharlamov, and J. Tang. GraphMAE2: A Decoding-Enhanced Masked Self-Supervised Graph Learner, 2023. arXiv: 2304.04779 [cs.LG].

[50] J. Li, R. Wu, W. Sun, L. Chen, S. Tian, L. Zhu, C. Meng, Z. Zheng, and W. Wang. What's Behind the Mask: Understanding Masked Graph Modeling for Graph Autoencoders, 2023. arXiv: 2205.10053 [cs.LG].

[51] W. Tu, Q. Liao, S. Zhou, X. Peng, C. Ma, Z. Liu, X. Liu, and Z. Cai. RARE: Robust Masked Graph Autoencoder, 2023. arXiv: 2304.01507 [cs.LG].

[52] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen. Graph Contrastive Learning with Augmentations, 2021. arXiv: 2010.13902 [cs.LG].

[53] O. Mahmood, E. Mansimov, R. Bonneau, and K. Cho. Masked graph modeling for molecule generation. *Nature Communications*, 12, May 2021. DOI: 10.1038/s41467-021-23415-2.

[54] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[55] T. N. Kipf and M. Welling. Variational Graph Auto-Encoders, 2016. arXiv: 1611.07308 [stat.ML].

[56] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, Helsinki, Finland. Association for Computing Machinery, 2008. ISBN: 9781605582054. DOI: 10.1145/1390156.1390294. URL: https://doi.org/10.1145/1390156.1390294.

[57] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. Masked Autoencoders Are Scalable Vision Learners, 2021. arXiv: 2111.06377 [cs.CV].

[58] S. Zhang, H. Chen, H. Yang, X. Sun, P. S. Yu, and G. Xu. Graph Masked Autoencoders with Transformers, 2022. arXiv: 2202.08391 [cs.LG].

[59] Q. Tan, N. Liu, X. Huang, R. Chen, S. Choi, and X. Hu. MGAE: Masked Autoencoders for Self-Supervised Learning on Graphs. *CoRR*, abs/2201.02534, 2022. arXiv: 2201.02534. URL: https://arxiv.org/abs/2201.02534.

[60] Q. Tan, N. Liu, X. Huang, S.-H. Choi, L. Li, R. Chen, and X. Hu. S2GAE: Self-Supervised Graph Autoencoders Are Generalizable Learners with Graph Masking. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*, WSDM '23, pages 787–795, Singapore, Singapore. Association for Computing Machinery, 2023. ISBN: 9781450394079. DOI: 10.1145/3539597.3570404. URL: https://doi.org/10.1145/3539597.3570404.

[61] Y. Tian, K. Dong, C. Zhang, C. Zhang, and N. V. Chawla. Heterogeneous Graph Masked Autoencoders. In B. Williams, Y. Chen, and J. Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 9997–10005. AAAI Press, 2023. DOI: 10.1609/aaai.v37i8.26192. URL: https://doi.org/10.1609/aaai.v37i8.26192.

[62] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec. Strategies for Pre-training Graph Neural Networks, 2020. arXiv: 1905.12265 [cs.LG].

[63] M. Rosenstein, Z. Marx, L. Kaelbling, and T. Dietterich. To Transfer or Not To Transfer, January 2005.

[64] C. Ying, T. Cai, S. Luo, S. Zheng, G. Ke, D. He, Y. Shen, and T. Liu. Do Transformers Really Perform Bad for Graph Representation? *CoRR*, abs/2106.05234, 2021. arXiv: 2106.05234. URL: https://arxiv.org/abs/2106.05234.

[65] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010. DOI: 10.1109/MCSE.2010.69.

[66] D. Grewe, Z. Wang, and M. F. P. O'Boyle. 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 2013, Shenzhen, China, February 23-27, 2013*, 22:1–22:10. IEEE Computer Society, 2013. DOI: 10.1109/CGO.2013.6494993. URL: https://doi.org/10.1109/CGO.2013.6494993.

[67] AMD OpenCL Accelerated Parallel Processing SDK. https://web.archive.org/web/20170628060105/http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/. Retrieved from Internet Archive. Original link: http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/.

[68] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL, November 2011. DOI: 10.1109/IISWC.2011.6114174.

[69] CUDA Toolkit. URL: https://developer.nvidia.com/cuda-zone (visited on 10/20/2023).

[70] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. In 2012. URL: https://api.semanticscholar.org/CorpusID:497928.

[71] Polybench benchmark. URL: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 (visited on 10/21/2023).

[72] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. DOI: 10.1109/IISWC.2009.5306797.

[73] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, Pittsburgh, Pennsylvania, USA. Association for Computing Machinery, 2010. ISBN: 9781605589350. DOI: 10.1145/1735688.1735702. URL: https://doi.org/10.1145/1735688.1735702.

[74] E. Parisi, F. Barchi, A. Bartolini, and A. Acquaviva. Making the Most of Scarce Input Data in Deep Learning-Based Source Code Classification for Heterogeneous Device Mapping. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(6):1636–1648, 2022. DOI: 10.1109/TCAD.2021.3114617. URL: https://doi.org/10.1109/TCAD.2021.3114617.

[75] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015. arXiv: 1502.03167 [cs.LG].

[76] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, abs/1406.1078, 2014. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[77] CUDA C++ Programming Guide. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 09/13/2023).

[78] L. Bjertnes, J. O. Tørring, and A. C. Elster. LS-CAT: A Large-Scale CUDA AutoTuning Dataset. *CoRR*, abs/2103.14409, 2021. arXiv: 2103.14409. URL: https://arxiv.org/abs/2103.14409.

[79] L. Bjertnes, J. O. Tørring, and A. C. Elster. Autotuning CUDA: Applying NLP Techniques to LS-CAT. In *Norsk IKT-konferanse for forskning og utdanning*, number 1, pages 72–85, 2021.

[80] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching Word Vectors with Subword Information, 2017. arXiv: 1607.04606 [cs.CL].

[81] M. Fey and J. E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[82] LAPACK GitHub Repository. URL: https://github.com/Reference-LAPACK/lapack (visited on 10/22/2023).

[83] B. Langmead and S. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9:357–9, March 2012. DOI: 10.1038/nmeth.1923.

[84] M. Vasimuddin, S. Misra, H. Li, and S. Aluru. Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 314–324, 2019. DOI: 10.1109/IPDPS.2019.00041.

[85] G. Fursin. Collective Tuning Initiative: Automating and accelerating development and optimization of computing systems, July 2014.

[86] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[87] H. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1):43–56, 1995. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(95)00042-E. URL: https://www.sciencedirect.com/science/article/pii/001046559500042E.

[88] C. C. Kind, M. Canesche, and F. M. Q. Pereira. Jotai: a Methodology for the Generation of Executable C Benchmarks. Technical report 02-2022, Universidade Federal de Minas Gerais, 2022.

[89] Linux kernel archives. URL: https://www.kernel.org/ (visited on 10/20/2023).

[90] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International*

Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pages 3–14, 2001. DOI: 10.1109/WWC.2001.990739.

[91] OpenCV, Open-Source Computer Vision Library. URL: https://github.com/opencv/opencv (visited on 10/20/2023).

[92] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing, 2015. arXiv: 1409.5718 [cs.LG].

[93] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. *CoRR*, abs/2109.08267, 2021. arXiv: 2109.08267. URL: https://arxiv.org/abs/2109.08267.

[94] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimarães, and F. M. Q. Pereira. ANGHABENCH: A Suite with One Million Compilable C benchmarks for code-size reduction. In J. W. Lee, M. L. Soffa, and A. Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 378–390. IEEE, 2021. DOI: 10.1109/CGO51591.2021.9370322. URL: https://doi.org/10.1109/CGO51591.2021.9370322.

[95] F. Chollet. Deep Learning with Python, 2017.

[96] I.-K. Yeo and R. A. Johnson. A new family of power transformations to improve normality or symmetry. *Biometrika*, 87(4):954–959, December 2000. ISSN: 0006-3444. DOI: 10.1093/biomet/87.4.954. eprint: https://academic.oup.com/biomet/article-pdf/87/4/954/633221/870954.pdf. URL: https://doi.org/10.1093/biomet/87.4.954.

[97] J. M. Hernández and P. V. Mieghem. Classification of graph metrics. In 2015. URL: https://api.semanticscholar.org/CorpusID:37136216.

[98] C. Cummins, H. Leather, Z. Fisches, T. Ben-Nun, T. Hoefler, and M. O'Boyle. Deep Data Flow Analysis, 2020. arXiv: 2012.01470 [cs.PL].

[99] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: http://arxiv.org/abs/1412.6980.

[100] L. Biewald. Experiment Tracking with Weights and Biases, 2020. URL: https://www.wandb.com/. Software available from wandb.com.

[101] G. Jurman, S. Riccadonna, and C. Furlanello. A comparison of mcc and cen error measures in multi-class prediction. *PLOS ONE*, 7(8):1–8, August 2012. DOI: 10.1371/journal.pone.0041882. URL: https://doi.org/10.1371/journal.pone.0041882.

Acknowledgements

I would like to thank Prof. Andrea Acquaviva and my co-supervisors Francesco Barchi and Emanuele Parisi, for their support and guidance throughout my journey in this fascinating field of research. I have learned a lot from them, and I am truly grateful for their insights and encouragement. Considering the topic of this work, I would also like to thank the open-source community, whose tireless efforts have made life easier for so many people.

The strongest thanks go to my family and my boyfriend. They always believed in me and have given me the strength and motivation to continue this university path, even in the toughest moments.

I also thank Marcello and Alessandro for being loyal companions throughout the projects we have collaborated on during these years.

At last, I extend my heartfelt thanks to all my friends who have enriched every single day of this experience. In particular, I thank the group of friends from the extended "Cesena macro-area" - as someone would call it. Despite all the changes we have gone through, they are still a fixed point in my life.

I am eternally grateful for being surrounded by so many wonderful folks. Thank you.