Universidad Zaragoza

1542

Master thesis

Towards autonomous resource management: Deep learning prediction of CPU-GPU load balancing.

Author

Iñigo Gabirondo López

Supervisors

Darío Suárez Gracia

Rubén Gran Tejero

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2024

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

D./Dª. Iñigo Gabirondo López,

con nº de DNI 44573346J en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster, (Título del Trabajo) Towards autonomous resource management: Deep Learning prediction of CPU-GPU load balancing.,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 22 de junio del 2024.

IÑIGO
GABIRONDO
LOPEZ

Firmado digitalmente
por IÑIGO GABIRONDO
LOPEZ
Fecha: 2024.06.22
12:58:19 +02'00'

Fdo:

Acknowledgements

I would like to thank Darío Suárez, Rubén Gran and Alejandro Valero for their support, guidance and advice during this project. They decided to believe in me for opening this new research line, and without them it would have been impossible to finish this work.

Apart from that, thanks to all my classmates and colleagues for their help, good moments and support throughout these two years. Coming to Zaragoza to study this master's degree has been a challenge for me, and thanks to them everything has been much easier.

Last but not least, thanks to my family and friends for always being by my side. You will always be the main reason behind all this hard work.

Towards autonomous resource management: Deep learning prediction of CPU-GPU load balancing.

SUMMARY

The demand of data centers has increased due to the latest improvements of Artificial Intelligence. These data centers are composed of thousands of servers with cooling systems that consume high amounts of energy.

The servers usually contain more than one processing unit that can work together for solving computational tasks. When making a proper partitioning of the entire workload among the processing units of the same machine, the total execution time and power consumption of the server is highly decreased. Hence, creating load balancing algorithms that create proper workload partitions can improve the energy efficiency.

This work presents a deep learning based load balancer that is thought for CPU-GPU heterogeneous systems. The load balancer takes as input an OpenCL kernel, the work group size of the kernel, and the input size in bytes, and it outputs the amount of work to assign to the CPU.

The load balancer is based on the heterogeneous device mapping model presented in ProGraML. ProGraML exhibited a very poor performance in our setup, which made it impossible to do any kind of experiment. Hence, in order to solve this performance issue, we decided to migrate the full ProGraML project to the pytorch-geometric library. After that, we adapted the heterogeneous device mapping model for the load balancing task.

Experimental results show that the load balancer is able to predict accurate workload partitions, even if the testing setup and the setup used for labelling the datasets differ. The final model has been able to predict 6 of the 8 tested kernels with a difference of less than 20% with respect to the theoretical work partition. Among those 6 kernels, 3 of them were predicted with an error less than 10%. In addition, our new ProGraML implementation removes the performance issue of the original work, it is publicly available and it is based on a library that will allow performing new experiments more easily.

Index

Chapter 1

Introduction

1.1 Motivation

Data centers are infrastructures that consume high amounts of energy, mainly due to the servers that perform the computations and the required cooling systems. Moreover, they have become one of the main driving forces behind the development of the latest Artificial Intelligence (AI) tools. The improvements made on this field are mostly linked to an increase of the size and complexity of the neural networks they rely on, which as a consequence, have also increased the hardware usage and power consumption of data centers [1, 2].

The demand of data centers is expected to grow even more in the following years, making their energy consumption a major concern. As an example, according to the International Energy Agency, almost one-third of the total energy consumption of Ireland could come from data centers by 2026 [3]. Consequently, creating efficient methods for managing data centers is crucial for a sustainable promotion of AI.

Data centers consist of servers that have different Processing Units (from now on PUs). PUs, also known as heterogeneous systems, are usually composed of a CPU and a GPU, but can also be extended to more specific ones such as FPGAs or ASICs [4]. These PUs can cooperate together for solving different computational tasks, achieving better performance than working on their own. However, using heterogeneous systems imposes two main challenges: creating unified code for all the PUs and partitioning the total work in the system.

The PUs of a system can have different instruction sets, capabilities and architectures, which forces the programmer to write a different piece of code for every PU. In order to alleviate this burden, the OpenCL framework offers a unified application programming interface (API) and a C-like programming language for executing programs on the different PUs of an heterogeneous system [5]. Although OpenCL solves the unification problem, it does not provide any type of support for balancing the workload among PUs.

When solving computational tasks, such as training a deep learning model, properly partitioning the entire workload among the PUs leverages the computational performance of the server and reduces its power consumption. Nevertheless, these PUs have distinct characteristics and are better suited for different computational tasks. Therefore, establishing an inappropriate work partition may create bottlenecks and inefficiencies during the computation. As a consequence, researchers have proposed load balancing techniques so that every PU has a proportional amount of work given its characteristics [6, 7, 8, 9].

The classical load balancing techniques often involve a profiling effort that imposes an overhead to the computation, increasing the execution time of a given task. Moreover, these methods are usually based on intricate heuristics, which also complicates the process of creating new methods. Because of that, researchers have explored machine learning methods for solving this task. When using code as input for machine learning tasks, it is important to have an encoding that correctly represents all the semantics of the code. For this reason, Cummins *et al.* proposed ProGraML, a framework that represents the input code as heterogeneous graphs [10].

This master's thesis proposes a deep learning based load balancer for heterogeneous systems consisting of a CPU and a GPU. Taking as input an OpenCL program in the ProGraML graph format and runtime information of the program, the model is able to predict how much workload should be assigned to the CPU.

1.2 Context and Starting Point

This project has been developed in the Computer Architecture Group of Zaragoza (gaZ). It represents a new research line in the group, also being the first deep learning project of the supervisors. In addition, we intend to publish the outcomes of this work in form of a research paper.

This work started by setting up the ProGraML work, where the initial tests showed a very poor performance in our setup. With the purpose of solving this performance issue, we decided to do a full migration of the ProGraML work, including the deep learning model and the training datasets. Once the migration was done, we adapted these models and datasets for the load balancing task.

The following section describes the goals for carrying out the migration process and the adaptation for load balancing.

1.3 Goals

The main objective of this work is to create a deep learning based load balancer for CPU-GPU heterogeneous systems. The goals of this project are the following (Appendix A shows a summary of the temporal distribution of the tasks):

- Study the state-of-the-art in the field of deep learning for heterogeneous systems.
- Creating a working environment based on the ProGraML representation for dealing with deep learning tasks in the field of heterogeneous systems, leaving publicly available the used models and datasets.
- Adapting the heterogeneous device mapping model proposed by Cummins *et al.* for predicting load balancing in a CPU-GPU system [10].
- Ablation study of the load balancing model.
- Testing the load balancing model against data examples not seen in the training phase.

1.4 Scope

All the initial goals of this project have been achieved. After studying the state-of-the-art, we migrated the models and datasets of the ProGraML work to the pytorch-geometric library. The new implementation does not have any performance issue, is ready to be used on any machine with a GPU, and the training times of the models have been improved without hurting their accuracy. In addition, some of the implemented code has been merged with the official ProGraML repository and the author of the original ProGraML work has considered our repository as a valid alternative for performing the deep learning experiments. In the near future, the official ProGraML repository will have a link to our repository, allowing other users to access the created datasets and models.

Regarding the load balancing model, the heterogeneous device mapping model and dataset have been successfully adapted. We have considered the load balancing problem as regression and classification tasks. The ablation study covers different configurations of Graph Neural Networks trained with the same methodology, and the best model has achieved an accuracy of 0.782 and a mean absolute error of 7.324 in the new dataset. In addition, when testing this model against OpenCL programs not seen during the training process, the model has been able to perform accurate workload distributions, where the regression model has shown the strongest results: among the 8 programs used for testing, the model predicted 6 of them with a difference of less than 20% to the theoretical work partition, and among those 6 kernels, 3 of them were predicted with an error less than 10%.

The repository of this project can be found in the following link:
`https://github.com/igabirondo16/pyg_programl.git`

1.5 Structure of the Thesis

The remainder of this document is structured as follows: Chapter 2 presents the state-of-the-art and the related works. Chapter 3 contains the methodology followed in this project. Chapter 4 explains the migration process of the ProGraML deep learning model and the DeepDataFlow and Heterogeneous Device Mapping datasets to the pytorch-geometric libraries. Chapter 5 introduces the load balancing model, as well as the different tests carried out. Finally, chapter 6 summarizes the main conclusions of this work and proposes future research directions.

Chapter 2

State-of-the-Art

This chapter describes the state of the art of deep learning based scheduling techniques for heterogeneous systems. First, an overall explanation about heterogeneous systems is given, showing their characteristics and challenges, including the main focus of this project: load balancing and device mapping. Next, we describe previous works that use machine and deep learning methods for the device mapping problem. Finally, the last section gives a more detailed explanation of ProGraML [10].

2.1 Heterogeneous Systems

Heterogeneous systems consist of different types of processing units (PUs) such as: CPUs, GPUs, or FPGAs [11]. These types of PUs have different characteristics and each one is better suited for specific types of computational problems[1] [12]. Establishing a task distribution that takes into account the characteristics of each PU, can potentially improve the overall performance and energy consumption of the whole system. Due to the latest improvements in AI and the gaming industry in the last years, the CPU+GPU combination has gained a high popularity. Nevertheless, many combinations exist depending on the problem to be solved.

Regarding the different PUs, Figure 2.1 shows an heterogeneous system with a CPU, a GPU, and an FPGA. The PUs have different underlying hardware designs, which makes them have distinct qualities, and consequently, to be better suited for different types of tasks. Hence, a sound knowledge of the tackled problem and available PUs is mandatory to maximize the efficiency of the computation. For instance, CPUs are designed to execute general-purpose programs. GPUs offer a greater data parallelism opportunity than CPUs, while not being so efficient on tasks that have branch divergences (this is because of their smaller number of transistors devoted to control units). Finally, PUs like FPGAs or ASICs can address more specific tasks with

[1] Hereafter these computational problems are referred as tasks.

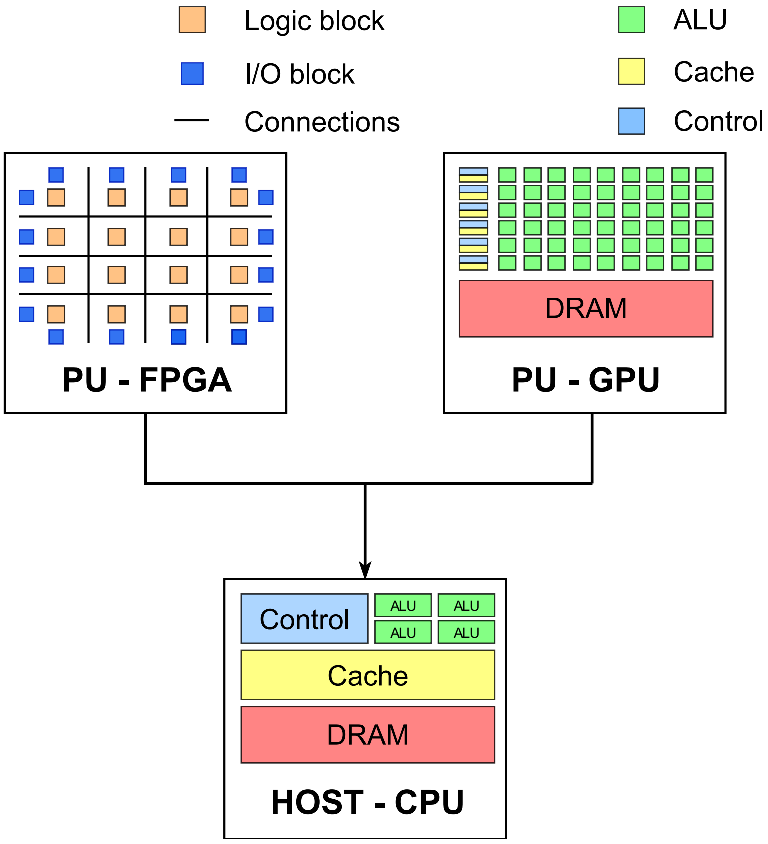


Figure 2.1: Diagram representing an heterogeneous system composed of a CPU, a GPU, and an FPGA.

a specialized design, making them more efficient than CPUs or GPUs for a given task, but also adding more complexity to the design, implementation, and programability of them (see Appendix B for more details about the mentioned accelerators). In summary, specificity in the hardware design allows achieving a better efficiency when solving a particular tasks at the cost of making the PUs more complex to manipulate, expensive, and harder to adapt to other problems.

On the software side, hardware heterogeneity brings additional complexity for the runtime and to the programmer because in many cases, she/he has to provide a different code for each device. In addition, each PU inside an heterogeneous system can have its own instruction set, capabilities, and architecture. In order to alleviate all these shortcomings, the OpenCL framework offers a unified application programming interface (API) and a programming language for writing and executing programs across the different accelerators of an heterogeneous system [5]. The programming language is a C-like language, where the functions are commonly known as *kernels*. Employing OpenCL, developers can create parallel applications using data and task-based parallelism.

From the OpenCL perspective, the different PUs of an heterogeneous system are composed of several compute units, and each compute units has multiple work items. When executing a kernel in a PU, the work items are divided into work groups. All the work items inside a work group are executed in parallel inside a compute unit, while the work groups run in batches based on the available compute units of a PU. Moreover, OpenCL allows synchronization among the work items of the same work group, but not among work items that belong to different work groups. Developers can choose the total number of work items to be launched, and the size of the work groups. These two parameters are known as *global work size* and *local work size* or *work-group size*, respectively, and correctly adapting them to the needs of each case maximizes the throughput of the application.

While OpenCL solves the problem of code portability across devices and enables parallel execution using different types of accelerators, it does not provide any type of support for load balancing in a scenario with several PUs. Not performing a proper load balancing distribution can lead to potential performance and energy inefficiencies. Many research works have tried to tackle this issue [7, 13, 14].

2.2 Device Mapping and Load Balancing on Heterogeneous Systems

Using heterogeneous systems for solving computational problems intrinsically brings to the table deciding which PUs will be involved in the computation process. As mentioned in Section 2.1, PUs are more suitable (computationally speaking) for different types of tasks, thus making an appropriate partition of the work among the available PUs can highly leverage the overall computational and energy performance of the system. This work partitioning can be done by assigning the whole workload to a single PU, or by dividing it among some (or all) the available PUs of the system.

When a single PU performs the whole computation, deciding which PU will be the one to execute the kernel is crucial for performance. The problem of mapping a kernel to a PU taking into account their characteristics is known in the literature as the device mapping problem [15, 16, 8]. For example, an application that is parallelizable will achieve a better performance by running it on a GPU rather than on a CPU. Hence, mapping each kernel to the appropriate PU is a decision that can have a huge impact on the execution time of a kernel. Nevertheless, giving the whole workload to a single PU presents two main drawbacks that may cause the system performance to be less than the optimal. First of all, after sending the input data to the PU, the host remains idle most of the cases, leading to a wastage of computational power. On the other hand, the memory bandwidth of the PU (for example, in the GPUs) can act as a bottleneck, making the computational resources of these PUs to remain underutilized [17].

In order to tackle these issues, developers have tried to divide the total workload among many PUs using a load balancing schedule. Load balancing is a technique widely used in the computer science field, that aims to share the total workload across different computers, processes, or other resources, so that they all finish their respective tasks at the same time [6]. Performing a proper workload division among the different PUs can boost the computation performance of the whole system, avoiding possible bottlenecks and idle PUs. However, doing this work distribution is a more complicated problem than device mapping, as it not only has to take into account the behavior of the problem, but also the heterogeneity of the system.

Regarding different proposals of load balancing techniques, these can be divided into task-parallel and data-parallel approaches. The former relies on dividing the entire application into several kernels, and sending these kernels to the different PUs of the system [18, 19, 13]; whereas the latter approach splits the total work of a single kernel into many PUs, enabling cooperation between them [7, 9, 14]. The data-parallel approach has been the core of this work, as it is very suitable for applications to be run on a GPU.

2.3 Machine Learning for Device Mapping

Classical methods for solving the scheduling problem presented in Section 2.2 are deterministic models that often involve a profiling step, which adds an overhead to the whole computation process and consequently, ends up enlarging the execution time.

With the aim of solving this problem, many works have tried to use deep learning based methods, as these have proven to successfully learn complex heuristics. Additionally, using a data-based approach for selecting the schedule of a given kernel can automatically avoid the use of profiling, reducing the overhead of the computation. Nevertheless, using deep learning in this specific field presents two main challenges: finding an appropriate way of encoding the input data and lack of training data. The code can be used as a sequence of tokens as input for deep learning, but this may fail on correctly representing all the semantics of the code and the relationships of the variables and instructions. On the other hand, creating a dataset can be very challenging due to the reduced number of publicly available OpenCL kernels. Additionally, among all the available kernels, many of them are repeated or are very similar, meaning that the implementation for the same (or a closely related) problem can be found in different code sources. Apart from that, creating a dataset also involves having a well-defined methodology for taking measures (execution time, for example) to ensure that the results are valuable and reproducible.

Due to these two limitations, the load balancer implemented in this work takes as baseline works that perform the device mapping problem. This problem has a publicly available dataset that has been used by many works, thus using it as a starting point was easier than creating a new dataset. The following works have used different features, data encodings, and machine learning methods for the heterogeneous device mapping problem.

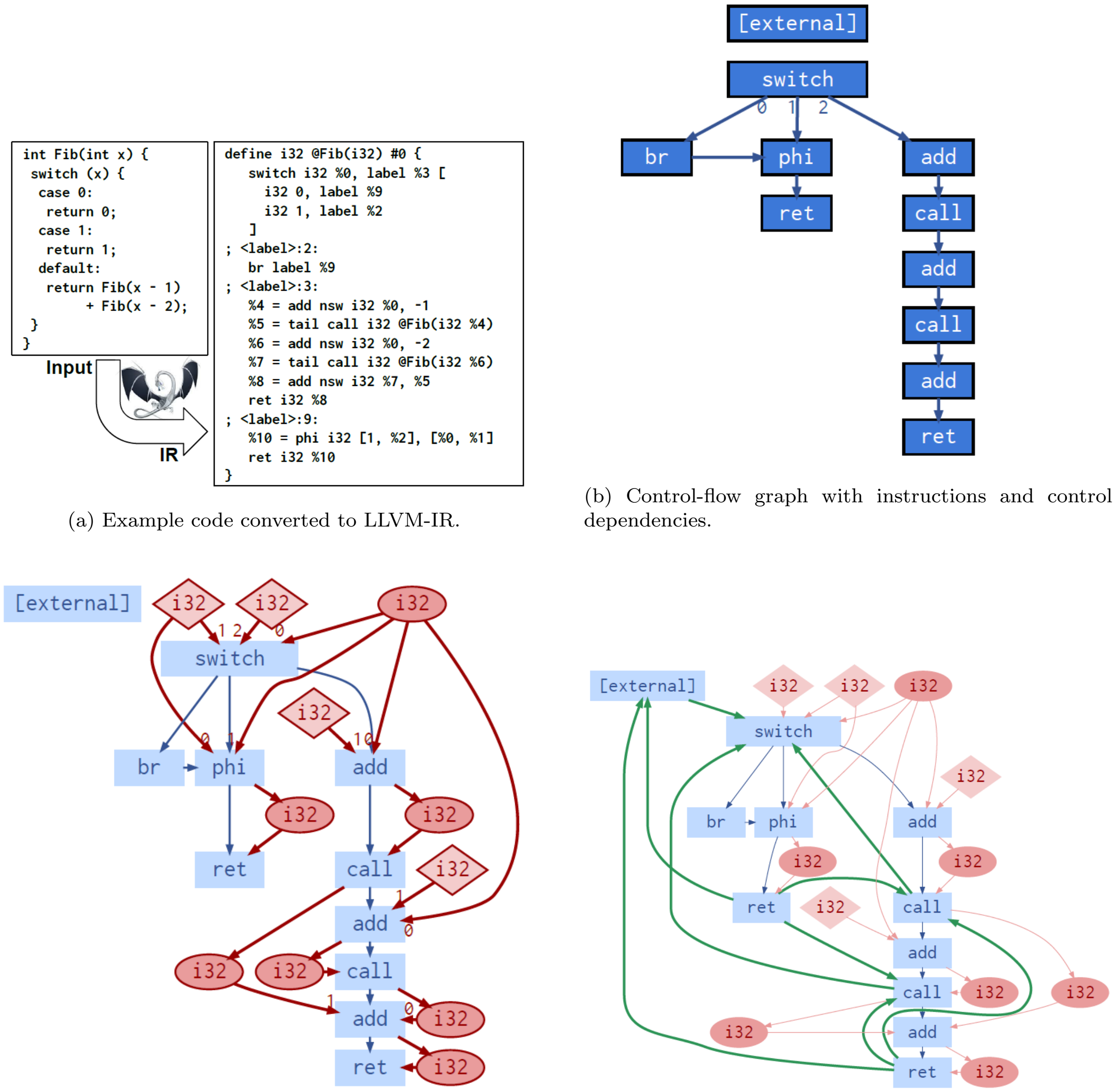
The first projects using data-based method for mapping a kernel to a specific processing unit were proposed by Grewe *et al.* and O'Boyle *et al.*, were they used a Decision Tree and a Support Vector Machine, respectively, for mapping a kernel to a CPU and GPU [20, 21]. Cummins *et al.* proposed DeepTune, a deep learning method composed of an embedding layer, 2 LSTM layers, and a Multilayer Perceptron [22]. Taking as input a sequence of pre-processed tokens and dynamic information of the runtime (size of the input data and work group size), it highly improved the accuracy obtained by the previous method. Additionally, they also created a dataset composed of 256 kernels which has become the *de facto* dataset for working on this area. Ben-Nun *et al.* created a program representation that sequentially processed the IR statements of a program to perform whole-program classification, obtaining accuracy values that are comparable to DeepTune [23]. Cummins *et al.* proposed a graph-based program representation: ProGraML [10]. This work replaced the LSTM layers of DeepTune by Graph Neural Networks (GNNs) and used as input graph representations of the input kernels, outperforming all the previously mentioned methods. This project is the baseline of this work and it is discussed in-depth in the following section. Parisi *et al.* made an extensive analysis of the dataset created by Cummins *et al.*, and proposed to use siamese networks for avoiding overfitting [24, 22]. Finally, Xiao *et al.* created Programmable Graph Learning framework, a pipeline that constructs a dynamic dataflow graph from a LLVM-IR, extracts the features of the graph by using random walks and multi-fractal analysis and maps the kernels to the appropriate device using GNNs [25].

2.4 ProGraML

This project is based on the program representation that the deep learning models receive as input. Using a program representation that contains both the semantics (the problem that the program aims to solve) and the different characteristics (types of variables that is uses, control flow, et cetera) of the program is crucial. In this work, we use ProGraML, an IR-based program representation that aims to describe programs as heterogeneous graphs, for representing the semantics of the input kernels.

2.4.1 Graph representation

ProGraML is an IR-based program representation that describes programs as directed multigraphs where instructions, variables, and constants are vertices, and relations between vertices are edges. These multigraphs have different types of edges for representing control, data, and call flow. In addition, the edges are augmented with position attributes for adding more information to the graph, for example, for encoding the order of operands to instructions or for differentiating between divergent branches in control-flow.



```
int Fib(int x) {
 switch (x) {
  case 0:
   return 0;
  case 1:
   return 1;
  default:
   return Fib(x - 1)
        + Fib(x - 2);
 }
}
```

```
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) Example code converted to LLVM-IR.

(b) Control-flow graph with instructions and control dependencies.

(c) Adding the data vertices and edges.

(d) Final ProGraML graph.

Figure 2.2: Conversion of a Fibonacci implementation to ProGraML. Image taken from [10].

A ProGraML graph $G = (V, E)$ is created by traversing a compiler IR. An empty graph $G = \emptyset$ is populated in three steps: control-flow, data-flow, and call-flow. In practice, these three steps are done in a single one $O(|V| + |E|)$. Figure 2.2 shows the process of creating a ProGraML a graph, as well the final result. It starts by compiling a C-like code to LLVM, and then it creates the graph performing the three mentioned steps.

- **(I) Control-Flow:**

  Sub figure 2.2b shows the first step after compiling the input code to LLVM: creating the control-flow. This step inserts a vertex per instruction and connects the control-flow edges. Moreover, control-flow edges have numeric positions that represent their order in instruction's successors. These positions appear in an ascending order.

- **(II) Data-Flow:**

  The data-flow step (see sub figure 2.2c) introduces constants and variables as graph vertices. After that, the process continues by adding data-flow edges to the graph. These edges represent the relation between constants and variables with the instructions that use them as operands, and also describe the variables that are created by certain instructions. In this way, users can easily identify the scope of each variable and constant. Finally, the last step is to augment the data-flow edges with a position attribute that represents the order of operands for instructions.

- **(III) Call-Flow:**

  As shown in Figure 2.2d, call edges represent the relation between the instructions that call to functions and the entry instructions of the called function. Each return vertex to the calling statements creates a call edge. An IR with F functions creates $|F|$ disconnected subgraphs, as the call edges do not span between functions (this does not happen with the data edges, where a global variable can appear in different functions). The process ends by creating additional vertex for representing external calls, if the program requires of external linkage.

2.4.2 Deep Learning Model description

The deep learning model proposed by Cummins *et al.* [10] is a model created specifically to work with the ProGraML graph representation. The original purpose of this model was to perform dataflow experiments, but the authors changed the dimensionality of the output prediction for the heterogeneous device mapping problem.

As illustrated in Figure 2.3, the model is a GNN based deep learning model. These models usually consist of three main parts: an embedding layer that encodes the data of the graph's nodes to dense vectors, a message passing neural network (MPNN) that updates iteratively T times a node's dense vector with the information of the node's neighbors, and a readout function that takes as input the updated node's vectors and creates a per-node or per-graph classification depending on the task. For instance, the dataflow experiments need per-node predictions, whereas the device mapping problem requires per-graph classifications. The main difference between these two types of problems is that, for a graph with n nodes and a dataset with l labels, the output of the readout head will be $n \times l$ for node classification, whereas for graph classification it will be $1 \times l$. Doing classification at graph level requires having a single dense vector per graph. Because of that, in these cases the model needs an additional step between the MPNN and the readout steps: the pooling operator. The pooling operator converts all the dense vectors of the nodes that belong to the same graph, and creates a single vector that represents the whole graph. Finally, the readout head can take the graph representations for performing the final classification.

Regarding the ProGraML model, it takes as input the augmented graph G explained in Subsection 2.4.1. The first step is to encode the information of every node in the graph into a dense vector. For this purpose, every node stores an abbreviation of the instruction or variable it represents. This abbreviation is then mapped to a unique integer index (Figure 2.3 refers to these indexes as *vocabulary ids*), which serves as an identifier (this text abbreviation to index mapping has been defined for LLVM-IR, and it also involves an *unknown element* for making the mapping more general). Finally, an embedding layer converts these identifiers into dense vectors. The vectors created by the embedding layer are trained alongside the other layers of the model.

Once the embedding layers convert the information of the nodes into dense vectors, the message passing neural network updates them in an iterative process. Each iteration is divided into three main steps: creation of the messages, message propagation, and aggregation and vertex state update.

Firstly, a linear layer takes as input the node states and creates the messages that will be sent (for clarification, the messages are dense vectors of the same size of the nodes states). By means of differentiating the type of message depending on the type of edge (control, data and call) that connects two nodes, the linear layer creates three different messages for every input node. The next step is to send the messages, always from the sources nodes to the target nodes. For every type of edge, the graphs

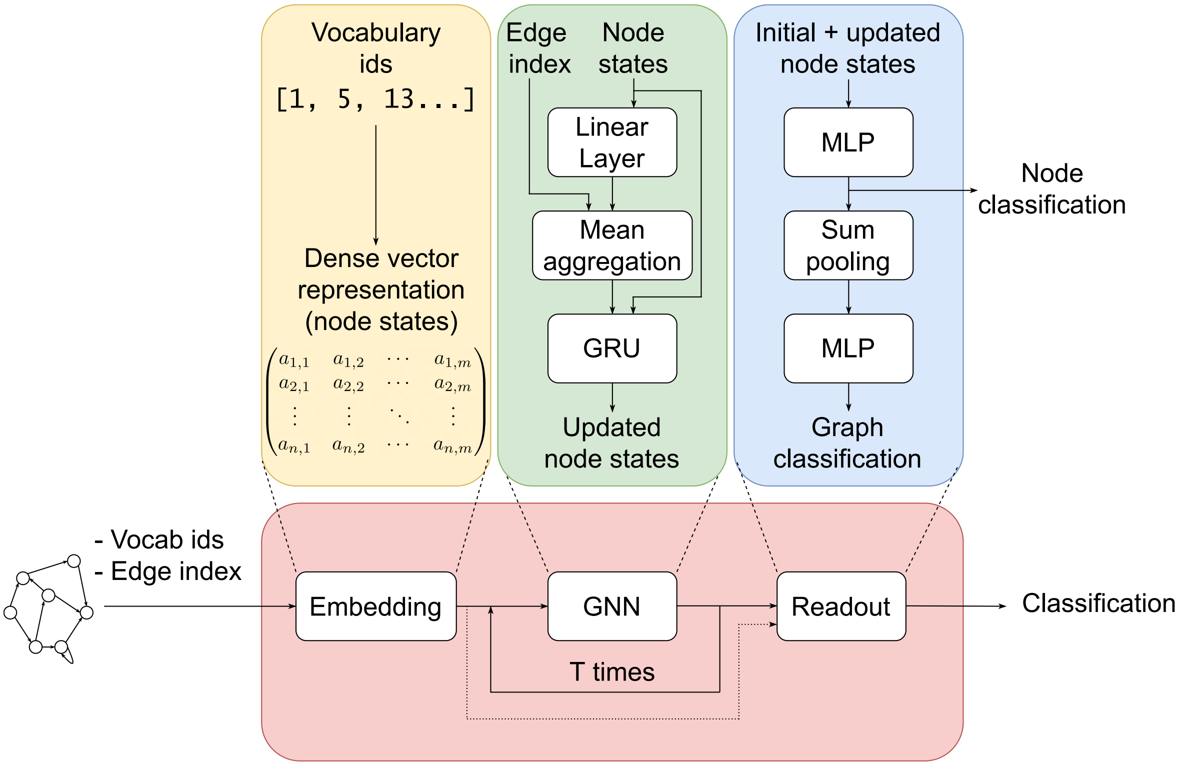

Vocabulary ids
[1, 5, 13...]
Dense vector representation (node states)
$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}$
Edge index
Node states
Linear Layer
Mean aggregation
GRU
Updated node states
Initial + updated node states
MLP
Node classification
Sum pooling
MLP
Graph classification
- Vocab ids
- Edge index
Embedding
GNN
T times
Readout
Classification

Figure 2.3: DataFlow model, including the input decoding, the message propagation, and the readout head.

store their corresponding adjacency matrix in sparse mode, meaning that every edge is represented with the id of the source node and the target node.

For choosing the messages that will be sent, for every type of edge, the model retrieves the messages of the given type using a look-up table that takes as input the source nodes of the edge type. After retrieving the messages, every target node in the graph will be assigned with the messages of their corresponding sources nodes, thus one target node can receive more than one message. All the messages received by a node are then mean aggregated, creating a single vector.

The last step to update the node's dense vectors, is to pass these dense vectors and the mean aggregated vectors through a Gated Recurrent Unit (GRU) layer to compute the node states of the next timestep [26]. The process of updating the states of a graph's nodes is repeated iteratively T times, depending on the task and the graph's structure.

Finally, in order to perform the classification task, the readout head takes as input the initial node states (the ones taken straight from the embedding layer) and the updated node states and passes them through a Multi Layer Perceptron (MLP), creating a per-node prediction. If the given task requires graph classification, then the layer includes a sum pooling operator. This sum pooling operator sums element wise all the node dense vectors that correspond to the same graph, creating a single dense graph vector. In the last step, an additional MLP takes as input the graphs dense vector and creates the per-graph classification.

Chapter 3

Methodology

This chapter shows an overview of the tools used for carrying out the experiments. These tools involve the machine and the software used for developing the project, as well as the chosen performance metrics and datasets. Finally, the chapter gives a brief explanation of the repository that stores the code of this project.

3.1 Experimental setup

This section specifies the experimental setup used throughout this work, so that future projects can easily reproduce the obtained results. Across all this work, the experimental platform always consists of an heterogeneous system: CPU and GPU. Table 3.1 shows the technical specifications of the system.

Hardware element	Version
CPU	Intel(R) Xeon(R) Bronze 3204 CPU @ 1.90GHz
GPU	NVIDIA GeForce RTX 4080
RAM	188GB DDR4
PCI-Express	PCI-Express 3.0 x16
STORAGE	2TB
OS	Ubuntu 20.04.6 LTS
Kernel	5.4.0-174-generic

Table 3.1: Hardware specifications of the used machine

The main programming language of the project is Python3 [27]. In addition, being a deep learning project, the implementation of this work relies on machine learning and deep learning libraries; Table 3.2 shows the libraries with their specific versions that we have used.

The project is mainly based on pytorch, which is a widely used deep learning library [28]. Apart from that, the models implemented in this project use Graph Neural Networks (GNNs) for solving the different tasks. In order to implement these

GNNs, we have used the pytorch-geometric library, because the pytorch library does not include this type of layers [29]. Pytorch-geometric is a deep learning library built upon pytorch that offers a wide variety of GNNs, graph operations and data structures ready to use for developers. Finally, we utilized the scikit-learn library for measuring the performance metrics and implementing machine learning models [30].

On the other hand, for testing the final models against kernels that were not part of the training set, we installed OpenCL[1] in our machine [5].

Finally, we have used Git for keeping track of the state of the project.

Software	Version
Python3	3.8.18
Pytorch	2.1.0
Pytorch-Geometric	2.4.0
Scikit-Learn	1.3.2
CUDA	12.0
OpenCL	2.2.18

Table 3.2: Summary of the used software.

3.2 Performance metrics and methodology

This project has involved comparing several deep learning models among them, as well as validating the final load balancing model, against kernels not seen during training. Because of that, it has been necessary to use classical machine learning accuracy metrics and measuring the execution time of the kernels.

Regarding the machine learning metrics, this project involves both classification (dataflow, heterogeneous device mapping and load balancing model) and regression (load balancing model) models, thus metrics for evaluating classification and regression problems were necessary. For the classification part, we used accuracy, F1 score, precision and recall, whereas for regression we chose mean squared error, root mean squared error and mean absolute error. For both cases, the metrics aim to describe the performance of the models in general terms. Appendix C summarizes the used metrics.

For measuring the execution time of the kernels, we followed the methodology proposed by Cummins *et al.*, where the total execution time of a kernel is the sum of the communication of sending and receiving the data from the host to the device and the kernel time itself [31]. Each kernel is run five times, and the average execution time is then used for evaluation.

[1] See Section 2.1 for OpenCL details.

3.3 Datasets

The aim of this section is to give an overview of the used datasets for training the deep learning models. This project uses two different datasets as baselines: the heterogeneous device mapping dataset and the DeepDataFlow dataset [22, 10]. Both of them are related with the ProGraML code representation seen in Section 2.4.

3.3.1 Heterogeneous Device Mapping dataset

The Heterogeneous Device Mapping dataset was first proposed by Cummins *et al.* and it consists of 256 OpenCL kernels sourced from seven benchmark suites on two combinations of CPU/GPU pairs (see Table 3.3 for more details of the benchmarks that were used for creating the dataset) [22].

Suite	Version	Benchmarks	Kernels	Samples
AMD SDK	3.0	12	16	16
NPB (SNU [32])	3.3	7	114	527
NVIDIA SDK	4.2	6	12	12
Parboil [33]	0.2	6	8	19
PolyBench [34]	1.0	14	27	27
Rodinia [35]	3.1	14	31	31
SHOC [36]	1.1.5	12	48	48
Total	-	71	256	680

Table 3.3: Composition of the Heterogeneous Device Mapping dataset.

Regarding the labelling process, the authors executed each input kernel in two different configurations of heterogeneous systems, creating in this way two different datasets: the AMD and the NVIDIA dataset. Table 3.4 gives details of the configurations of these two heterogeneous systems, being both of them different to our setup. Each kernel is labeled with the PU in which the execution time was the shortest.

Dataset Name	CPU	GPU
NVIDIA	Intel Core i7-3820	NVIDIA GTX 970
AMD	Intel Core i7-3820	AMD Tahiti 7970

Table 3.4: Heterogeneous Systems' configurations used for labeling the dataset.

Each dataset contains 680 labeled OpenCL kernels by varying dynamic input that can be provided to the code. Specifically, each example of the dataset contains the

OpenCL code, the CPU/GPU label, the execution time in CPU, the execution time in GPU and two auxiliary inputs: the size of the input parameters and the workgroup size (these two are the ones that have been varied in order to perform data augmentation).

3.3.2 DeepDataFlow dataset

The DeepDataFlow dataset was created by Cummins *et al.*, and it assembles 256M-line corpus of LLVM-IR files taken from a variety of sources such as BLAS, OpenCV or Tensorflow (see Table 3.5) [10]. Its main purpose is to train models for solving traditional dataflow tasks: control reachability, dominators, data dependencies, liveness and subexpressions.

Regarding the labelling process, all the ground truth labels have been computed using traditional analysis implementation [37]. For each of the tasks, and for every unlabeled graph of the dataset, n labeled graphs are created by selecting different $V_0 \in V$ source vertices. The number of graphs created n is proportional to the size of the graph:

$$n = \min\left(\frac{|V|}{10}, 10\right) \tag{3.1}$$

Every example graph in the dataset consists of an input graph, a source vertex (the original work refers to this vertex as *vertex selector*) that indicates the initial vertex that was used for computing the ground truth labels of the graph, and the binary labels of the graph's nodes. In addition, each graph also includes the number of steps that the iterative processes needed for computing the ground truth labels. These number of steps is used for producing subsets of the entire dataset.

The dataset follows a 3:1:1 ratio for training, validation and test instances. The examples coming from the same input graph were allocated to the same split, and the authors used the same random allocation of instances for the five tasks. Finally, being the five data flow analyses binary classification problems, the authors state that the dataset contains a strong class imbalance among the instances. An accuracy of the 86.92% can be obtained by always predicting the negative class, hence, F1-score, recall and precision have been used for evaluating the dataflow models.

3.4 The CHAI benchmark suite

Although evaluating the deep learning models by presenting classical machine learning metrics can provide a good intuition about a model's performance in a given task, testing the models against more realistic cases can provide additional information about

Source	Language	Domain	IR files	IR lines
BLAS 3.8.0	Fortran	Scientific computing	300	345,613
Github	C	Various	38,109	74,230,264
	OpenCL		5,224	9,772,858
	Swift		4,384	4,586,161
Linux 4.19	C	Operating systems	13,418	41,904,310
NPB [38]	C	Various	122	255,626
Cummins *et al.* [22]	OpenCL	Various	256	149,779
OpenCV 3.4.0	C++	Computer Vision	432	2,275,466
POJ-104 [39]	C++	Standard Algorithms	397,032	104,762,024
Tensorflow [40]	C++	Machine Learning	1,903	18,152,361
Total	-	-	461,182	256,434,462

Table 3.5: The DeepDataFlow LLVM-IR corpus. This data has been taken from the original work.

their limitations. These real examples must be standardized so that the results of future approaches are comparable with the ones of this work [41]. Having this into account, the evaluation of the last load balancing model was done using kernels taken from the CHAI benchmark [42].

The Collaborative Heterogeneous Applications for Integrated-architectures (CHAI) benchmark is designed for evaluating the architectures of a CPU-GPU collaboration. It consists of 14 benchmarks that can be executed using different collaboration patterns such as data partitioning or fine-grain task partitioning. In addition, each benchmark is implemented in seven different programming models like OpenCL, CUDA or C++ AMP.

Using CHAI for testing was due to two main reasons: on one side, these benchmarks are not part of the training dataset, therefore the obtained results describe in a fairer way the model's capacity. On the other hand, the data partitioning benchmarks are suitable for evaluating load balancing schedules, as they are already designed for giving different workloads to the PUs.

Chapter 4

Enhancing ProGraML Performance

This chapter describes the unexpected migration from the ProGraML's original implementation to the pytorch-geometric library. Initial measurements showed that ProGraML performance was poor in our setup, and this chapter begins analyzing why (see Section 3.1 for more details of the experimental setup). Then, the discussion continues detailing the migration process and the implemented performance enhancements. Finally, the last section validates the implementation by re-running the dataflow and device mapping experiments to ensure that the performance issue is solved and showcases the performance advantages of the new implementation.

4.1 Performance Analysis

The initial goal of this project was to build up our proposal from the original ProGraML implementation and try to make further contributions from that baseline [10]. Nevertheless, the code had a bug that made the GPU to stay nearly idle the whole training process[1]. In fact, we profiled the model and the results confirmed that the GPU was not working on a big part of the forward pass. Figure 4.1 shows the GPU activity of the forward pass, where the executed functions are represented with green, orange and pink frames, whereas the light gray time lapses represent that the GPU is idle. This problem caused the training times to be much longer than the ones reported in the former paper, preventing its use in any real case scenario [10]. Using the default hyper-parameters described in the paper, the resulting training time was almost 23 times slower than the one stated in the paper.

Regarding the bug, after having tried to debug the cause of such low GPU performance, we contacted the author of the work asking for advice. The author himself recognized the bug in the project dataloader, and he suggested us to migrate the code to pytorch-geometric. Based on the aforementioned facts, we decided to replicate the

[1] `https://github.com/ChrisCummins/ProGraML/issues/147`

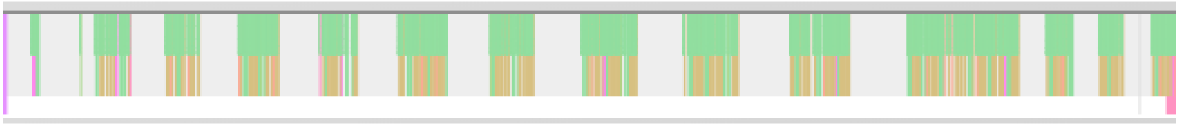


Figure 4.1: Activity of the GPU on the original code's forward pass. The spaces in light gray mean that the GPU is idle. Green, orange, red and purple colors represent functions exectued on the GPU. This entire forward pass lasts 51 seconds.

original datasets and models using pytorch-geometric.

4.2 ProGraML implementation with pytorch-geometric

As mentioned in Section 3.1, pytorch-geometric is a deep learning library built upon pytorch that allows working with Graph Neural Networks (GNNs), including ready to use layer implementations, graph operations, data structures and GPU support. In addition, the API is well documented, there are a variety of tutorials and an active community. Having all these facts into consideration and taking into account that the ProGraML format is based on graphs, we found pytorch-geometric suitable for migrating this project.

In this migration process, the three guiding principles were the following ones:

- The created copy of the model must have a very small classification accuracy difference to the original model.
- The created copy must me able to effectively use the GPU, and should be ready to train on any installed device.
- Migrating the code also involves adapting the datasets mentioned in Section 3.3 so that they can be used by this model or future ones.

The following subsections give some insights about the migration process of the deep learning models and datasets described in Sections 2.4 and 3.3, which consisted on three main steps: encoding the graphs of the datasets, migrating the dataflow and device mapping models and optimizing the training times of these models.

4.2.1 DeepDataFlow and Heterogeneous Device Mapping datasets.

As stated in subsections 3.3.2 and 3.3.1, both datasets are based on the ProGraML code representation. This representation has a characteristic that must be taken into account: the graphs have three different types of edges (control, data and call edges)

```
HeteroData(
  y=[223],
  nodes={
    x=[223],
    selector_ids=[223],
  },
  (nodes, control, nodes)={
    edge_index=[2, 104],
    edge_attr=[104],
  },
  (nodes, data, nodes)={
    edge_index=[2, 244],
    edge_attr=[244],
  },
  (nodes, call, nodes)={
    edge_index=[2, 46],
    edge_attr=[46],
  }
)
```

(a) Heterogeneous graph of the dataflow dataset. It stores one label per node in the graph, and every edge has its own position attributes. The root node used for labelling is stored in `selector_ids`.

```
HeteroData(
  y=[1],
  wgsize=[1],
  transfer_bytes=[1],
  nodes={
    x=[242]
  },
  (nodes, control, nodes)={
    edge_index=[2, 104],
  },
  (nodes, data, nodes)={
    edge_index=[2, 281],
  },
  (nodes, call, nodes)={
    edge_index=[2, 10],
  }
)
```

(b) Hetero graph of the device mapping dataset. It does not store the position attributes of the edges, while keeping the dynamic data as attributes (`wgsize` and `transfer_bytes`). In addition, it has a single value as label.

Figure 4.2: Differences between the dataflow and device mapping graphs. The data structures show the the attributes stored with their dimensions.

and therefore, using a data structure that considers that all the edges are the same is not enough to express all the information of a graph. Hence, we have used heterogeneous graphs for correctly encoding the graphs. Figure 4.2 show two examples of them. These heterogeneous graphs not only encode the vocabulary id of the graphs' nodes (attribute `x` in the Figure), but they also divide the edges by their corresponding type in sparse mode, where the first row represents the source nodes and the second row the target nodes of the edges (the attribute `edge_index` represents the adjacency matrix of each type of edge, in sparse mode). Regarding the mentioned datasets, they present some differences.

Regarding the dataflow dataset, Figure 4.2a shows an example of a graph with its respectives attributes. The graphs on this dataset need to represent the node that was used as a root node of the given dataflow experiment and the position attribute of the edges. Furthermore, the dataflow experiments are node classification problems. Having this into account, the attribute `selector_ids` stores a boolean array of size N, being N the number of nodes in the graph, that is used for representing the root node, where all the values are `False` except for the root node. This boolean array is necessary for extending the embeddings created with the vocabulary indexes of the nodes (the array is an additional input parameter of the embedding layer). In addition, `y` is a boolean array of size N is used for storing the labels of the nodes. Finally, `edge_attr` is an array of integers of size M_{type} (being M_{type} the number of edges of the given edge type) that stores the position attribute of each edge. These three arrays are stored alongside with the edges and the node vocabulary ids of the graph.

On the other hand, as seen in Figure 4.2b, the graphs of the heterogeneous device mapping dataset aim to predict the most suitable processing unit for running any given program. Taking into account that each input graph represent a program, the prediction to perform is done at graph level, and therefore, each graph will only have a single label `y` (CPU or GPU). Apart from that, the model also used dynamic information of the execution (`wgsize` and `transfer_bytes`), which should also be included inside of the data structure. In order to encode all this information properly, apart from the different edges and graph vocabulary ids, each graph on this dataset also includes one boolean variable that represent the label, and two float values that represent the dynamic data.

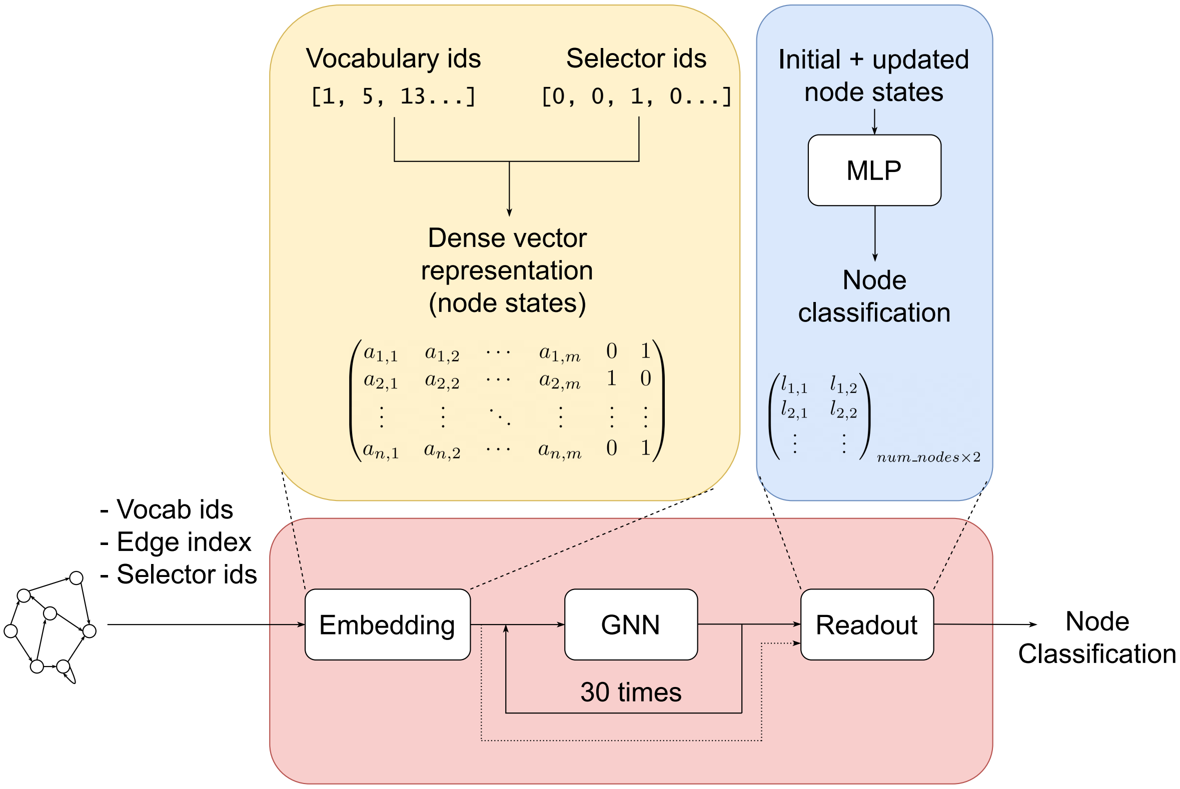

Vocabulary ids
[1, 5, 13...]
Selector ids
[0, 0, 1, 0...]
Dense vector representation (node states)
$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} & 0 & 1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} & 0 & 1 \end{pmatrix}$
Initial + updated node states
MLP
Node classification
$\begin{pmatrix} l_{1,1} & l_{1,2} \\ l_{2,1} & l_{2,2} \\ \vdots & \vdots \end{pmatrix}_{num\_nodes \times 2}$
- Vocab ids
- Edge index
- Selector ids
Embedding
GNN
30 times
Readout
Node Classification

Figure 4.3: ProGraML dataflow model.

4.2.2 ProGraML dataflow and heterogeneous device mapping models

We have migrated the ProGraML models trying to be as precise as possible to the original implementation. Although these two models are very similar, they also present some differences among them: the number of message passing iterations that they perform, the embeddings that they use and the output dimensionality of their readout heads.

Figure 4.3 shows the dataflow model. It uses the selector ids to extend the size of the node states, where the last two elements of every dense vector represent in binary form if a node is the root node of the experiment or not. Apart from that, the model performs 30 iterations of the graph neural network (GNN). This version of the GNN uses the position attribute of the edges for distinguishing non-commutative operations such as division, and the branch type in diverging control-flow. Finally, taking as input the initial node states (the ones taken straight from the embedding layer) and the updated node states, the readout head performs a node classification prediction for the dataflow tasks.

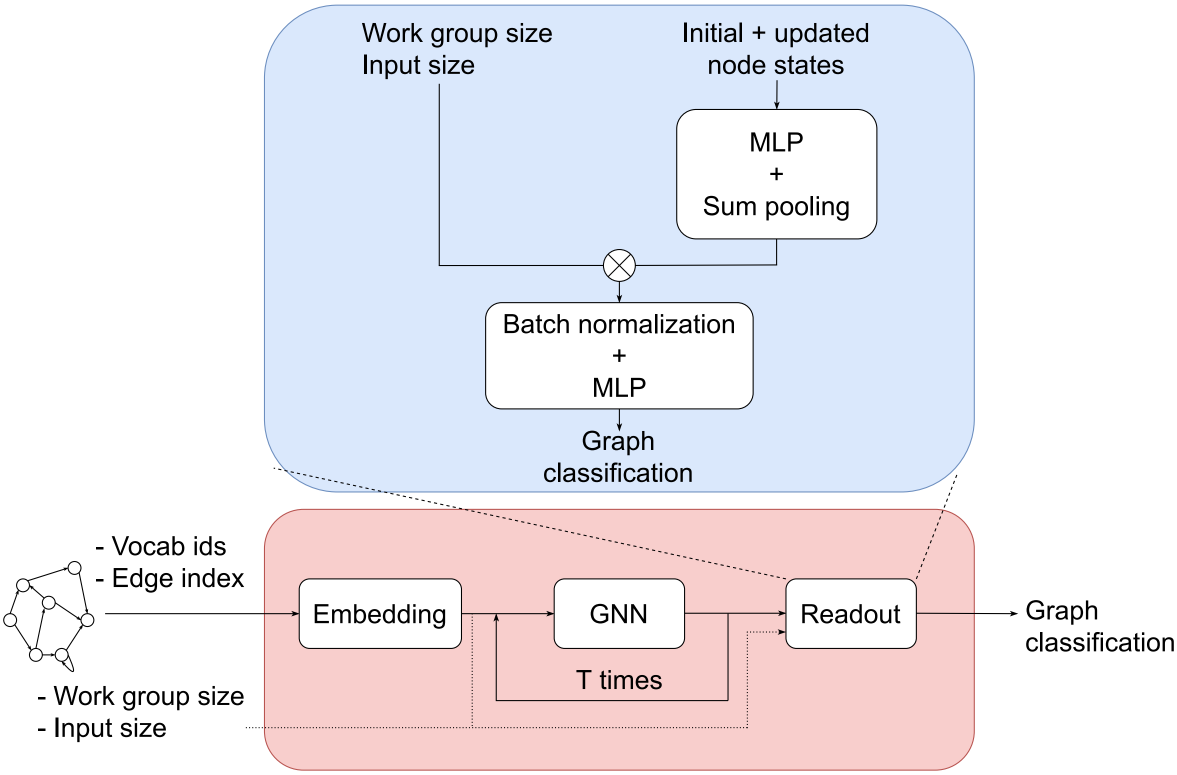


Figure 4.4: ProGraML device mapping model.

On the other hand, Figure 4.4 describes the heterogeneous device mapping model. This model only performs 6 graph neural network iterations, sharing the parameters of the layers between the even and odd steps (although it is not explicitly shown in the Figure, in practice the model uses 2 GNN layers, one for updating the odd time steps and the other for the even ones). In addition, the readout head creates a graph classification task using the provided dynamic information. After performing the pooling operation (create a single dense vector per graph), the dynamic information is concatenated with the respective graph dense vector. Finally, a batch normalization operator keeps the extended graph readout (graph dense vector + dynamic information) in scale and a final MLP creates the graph classification.

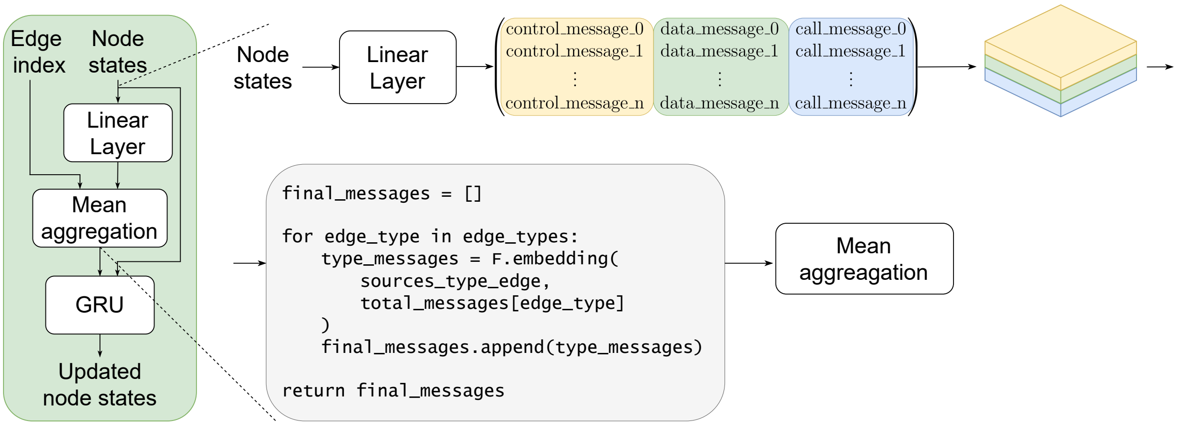
4.3 ProGraML Neural Network Enhancements

The migration process of the ProGraML's original code (which was written in pytorch and bazel) also involved some changes that were made for improving the long training times of the dataflow experiments. This section explains the two main changes made on the dataflow dataset and model.

4.3.1 Reshaping the messages of the Message Passing Neural Network

During the migration process, the custom message passing neural network presented a big performance issue. Figure 4.5 shows the original way of creating the messages: firstly, a linear layer takes as input the node states, and creates a 2 dimensional matrix of size *num_nodes* × (*hidden_size* × *edge_types*), where *num_nodes* is the number of nodes in the batch, *hidden_size* is the size of the dense vector that represent the states of the nodes and *edge_types* are the number of different types of edges. After that, the original implementation reshapes this matrix to a tensor of three dimensions of size *edge_types* × *hidden_size* × *num_nodes*. As mentioned in Section 2.4.2, the messages are always sent from the source node to the target node. For every type of edge (data, control and call), the model uses the source nodes of the given type and their corresponding messages for creating a look-up table. In this way, the model selects the correct messages to send to the target nodes. Although doing this reshape simplifies the implementation of the layer, it forces the layer to create as many look-up tables as types of edges of the input graphs, when it is possible to send the messages with a single look-up table.

The way of creating the messages has received a small modification in order to sending them creating a single look-up table. Figure 4.6 describes the new way of reshaping the messages: first of all, all the edges are stacked together in a single matrix of size 2 × (*num_edges*), where *num_edges* is the sum of the control, data and call edges. In addition, the implementation extracts the source nodes named



```
final_messages = []

for edge_type in edge_types:
    type_messages = F.embedding(
        sources_type_edge,
        total_messages[edge_type]
    )
    final_messages.append(type_messages)

return final_messages
```

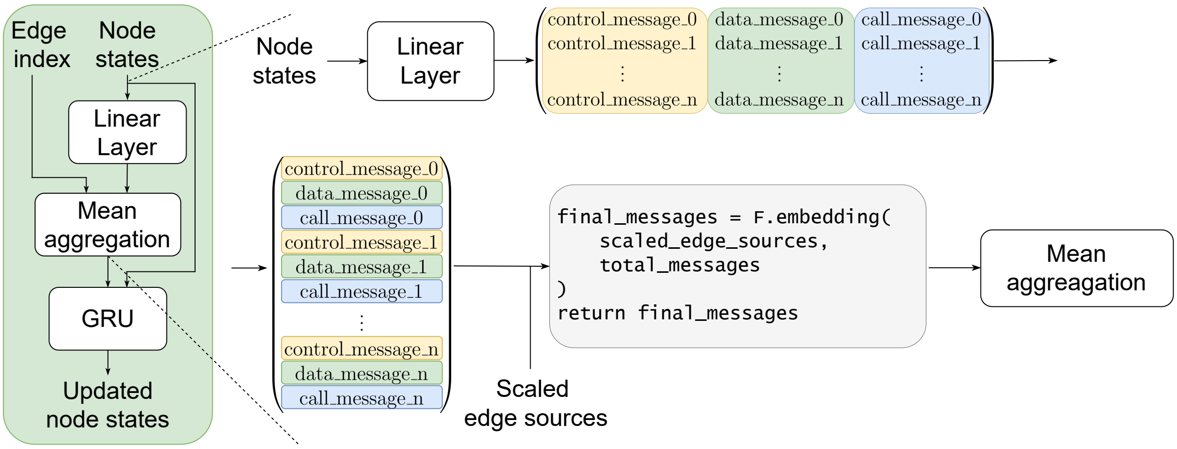
Figure 4.5: Original method for creating the messages of the ProGraML GNN.

edge_sources from the edges. When the messages are created, they are reshaped to a two dimensional matrix of size $(\mathit{num\_nodes} \times \mathit{edge\_type}) \times \mathit{hidden\_size}$, where all the messages corresponding to the same node are put one on top of the other. The idea of this new design is to access the messages using *edge_sources* in a single step. Nevertheless, if the vector *edge_sources* is kept on their own, the accessed messages will not be the correct ones, because the new reshaping creates an offset between a node's original and current positions. Hence, a re-scaling of the sources of the edges is needed in order to correctly access to the messages. The sources of the edges are pre-scaled (they do not change during execution) with the following equation:

$$\mathit{scaled\_edge\_sources} = (\mathit{edge\_sources} \cdot \mathit{edge\_types}) + \mathit{edge\_type}_i \tag{4.1}$$

where *edge_sources* are the sources of the edges, *edge_types* are the different number of edges and $\mathit{edge\_type}_i$ is the type of each edge.

By performing the mentioned reshape of the nodes' messages and the scaling of the edge sources, the process of sending messages can be done by creating a single look-up



```
final_messages = F.embedding(
    scaled_edge_sources,
    total_messages
)
return final_messages
```

Figure 4.6: Proposed method for creating the messages of the ProGraML GNN.

table, minimizing significantly the execution time of the forward pass of the model. In addition, as the change done only involves the way in which the messages are accessed, there is no impact on the classification accuracy of the model.

4.3.2 Training sampling method

As stated in Section 3.3.2, every graph of the dataflow dataset creates n different graphs by selecting different root nodes at every time. Taking into account that the dataset contains over 100,000 graphs, and that from every graph n instances are created, the number of training samples scale up very quickly.

The original implementation stores all the information of a graph in a single `.pb` file. At runtime, the application opens the file, and creates the n instances of the graph, by varying the root node and the labels of the nodes. This way of loading the graphs is very inefficient, because it forces the data loaders to create the graphs every time that a training is to be done. Moreover, this manner of processing the graphs is also redundant, because the same graph can be processed more than once in different trainings.

In order to solve this issue, when processing the graphs of a given dataflow experiment for the first time, the new implementation creates n different instances (the same graph but with different root node and node labels) for every input graph, and it stores every instance in separate files. After that, when the training process starts, it loads all the instances into memory, and samples them randomly at every epoch for training the model. This change helps reducing the execution time of the training by process, at the cost of incrementing the processing of data.

4.4 Evaluation of the created models

We evaluated the created implementation to ensure that the long training execution times are reduced and that the accuracy of the new models are similar to the original ones. For validating these two facts, we recorded the training times of the dataflow experiments and the accuracy metrics (we did not record the training time of device mapping problem because its dataset is smaller than the dataflow dataset, thus getting small execution times in dataflow would mean the same for device mapping) of both the dataflow and device mapping models.

4.4.1 Dataflow and Device Mapping results

Regarding the dataflow experiment, we followed the same training methodology as [10], taking the hyper parameters from the original implementation. The model used a learning rate of $2.5 \cdot 10^{-4}$ and performed $T = 30$ message passing timesteps. In addition, the model was trained with sets of 10k intervals for the first 50k training graphs, and with 100k intervals thereafter. Table 4.1 shows the results of the dataflow experiments; for every experiment, the table compares the results stated in the original paper with the ones obtained by our new implementation, making clear that in terms of classification both models have a very similar capacity.

Model	F1-Score		Precision		Recall	
	Original	Ours	Original	Ours	Original	Ours
Reachability	0.998	0.996	0.998	0.999	0.998	0.999
Dominance	1	0.999	1	0.999	1	1
Datadep	0.997	0.998	0.998	0.999	0.997	0.998
Liveness	0.937	0.956	0.962	0.969	0.916	0.932
Subexpressions	0.996	0.996	0.997	0.999	0.996	1

Table 4.1: Dataflow results after the migration.

As far as the heterogeneous device mapping model is concerned, the model was trained using both the NVIDIA and AMD datasets, following a 10-fold cross validation with 80/10/10 splits, where each fold was trained for 300 epochs and the model with the highest validation epoch was picked for testing. Moreover, a learning rate of 10^{-3} and $T = 6$ message passing iterations were used. Table 4.2 shows that the models of the original and the new implementation have achieved very similar results, which means that although the implementations are not identical, they are equally good on solving the given tasks.

Implementation	AMD Error [%]	NVIDIA Error [%]
Original	13.4	20.0
Ours	15.0	21.0

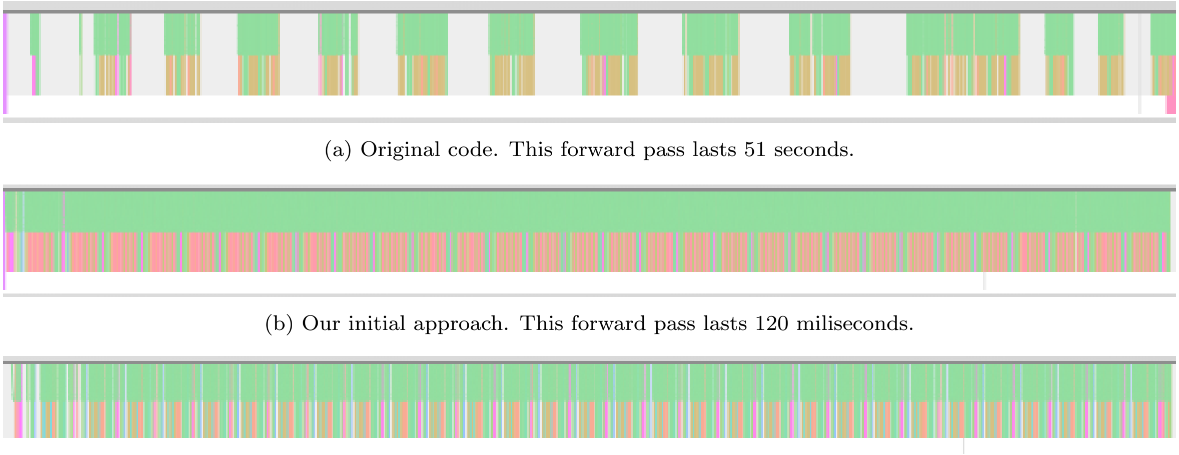
Table 4.2: Heterogeneous device mapping results after the migration.

4.4.2 Execution times

After ensuring that the new implementation has a very similar accuracy to the original code, we recorded the execution times of the different experiments to see the impact of the migration process.

The training times recorded with the original code were much larger than the ones described in the paper, and additionally the GPU usage was nearly around the 0%

during the whole process. Thanks to the migration and to the enhancements explained in Section 4.3, Figure 4.7 shows how these execution times are shorter while making a better utilization of the GPU.



(a) Original code. This forward pass lasts 51 seconds.

(b) Our initial approach. This forward pass lasts 120 miliseconds.

(c) Reshaping of the messages. This forward pass lasts 60 miliseconds.

Figure 4.7: GPU usage of the forward passes of the original implementation, our initial approach and the proposed reshaping of the messages. Green, orange, red and purple colors represent functions exectued on the GPU. The light gray color means no GPU usage.

As far as the performed improvements, although the pre-processing of the training graphs described in Section 4.3.2 is a costly operation, once it is done, the processed graphs can be used for performing experiments at any time without adding an extra overhead to the training. In addition, Table 4.3 shows the impact of the proposed way of reshaping the messages described in Section 4.3.1. It compares the execution times of the forward passes of the original implementation, our initial approach and the new reshaping of the messages (the table names them as original, initial and new, respectively). It also shows the speed-ups of the new reshaping with respect to the other two methods. Without having any impact on the model's accuracy, the proposed way of reshaping gets an speed-up of almost x2 with respect to the our initial approach, and an speed-up of x846 compared to the original implementation.

Regarding the dataflow training times, the problem was first seen when performing the preliminary tests of this project. Due to the large training times, we only measured the reachability test, as performing all the dataflow experiments with the original code would take more than one month. Diving into the numbers, the original implementation needed 7 days for finishing the full experiment, while our approach that included all the enhancements presented in Section 4.3 finished in 14 hours. In addition, the training times of the rest of the dataflow experiments have been reduced to less than one day. Notice that we run our approach with a batch size of 32 graphs, and that increasing the batch size would shorten even more the training times.

Implementation	Execution time (ms)
Original	51,579.983
Initial	118.898
New	60.832
Speed-up (New against original)	847.908
Speed-up (New against initial)	1.955

Table 4.3: Execution times of the forward passes of the original implementation, our initial approach and the new way of reshaping the messages (named as original, initial and new, respectively). The new approach gets an speed-up of x847 with respect to the original one, and almost a x2 speed-up compared to the initial approach without damaging the model's accuracy.

As an overall conclusion of the migration part, the created models have obtained a very similar classification accuracies to the original ones, meaning that although the implementations are not identical they are equally good on solving the given tasks. Moreover, we solved the problem of the low GPU usage, and the proposed enhancements achieve considerable speed-ups without affecting the model's accuracy. In addition, the training times of the dataflow experiments have been reduced to less than one day using a considerably small batch size. To finish with, the new implementation is based on a library that enables testing a wider number of Graph Neural Networks in an easier and faster way, which will help on making further improvements.

Chapter 5

Extending ProGraML for Load Balancing

While device mapping aims to find the most suitable device for an application within a system, load balancing aims to utilize all devices assigning a different amount of application's work to each of them. This chapter shows how the ProGraML's model designed for the device mapping problem can be extended to create a static load balancing scheduler. With the ProGraML graph of an OpenCL kernel and some runtime information (work group size and input arguments size), the new model predicts the amount of work that the CPU should execute. Please note the model works on heterogeneous systems composed by a CPU and a GPU.

We have formulated the load balancing task (CPU-GPU) as both a regression and classification problems, to see which is the most suitable approach to tackle this task. For the regression problem, the model has to predict the percentage of the total work that the CPU has to do, a value withing a range 0 to 100. On the other hand, instead of predicting a continuous value, in the classification problem the model has to predict a discrete scheduling, where the possible percentages of work given to the CPU are 0, 25, 50, 75, and 100%.

The load balancing model leverages the NVIDIA heterogeneous device mapping training dataset and the device mapping model. Using this dataset and model for load balancing has involved two main changes: relabelling the dataset, and adapting the output dimensionality of the readout head of the device mapping model. After creating the preliminary model, an ablation study has been carried out in order to improve the overall performance of the model. Finally, the model with the best accuracy has been tested against OpenCL kernels that were not seen during the training process.

5.1 Adapting the dataset

The load balancing dataset (both regression and classification) are based on the NVIDIA heterogeneous device mapping dataset. As the target value to predict has changed, the labels of the heterogeneous device mapping dataset also need to be changed, for both regression and classification tasks. In the original dataset, the execution time on each device determines the label for a given kernel (the execution time of the kernel on CPU and GPU are given): the selected device is the one with the lowest execution time. Taking the execution times of a kernel in the CPU and in the GPU, the following relation between both devices can be computed as:

$$f = \frac{T_{exec_{GPU}}}{T_{exec_{CPU}}} \tag{5.1}$$

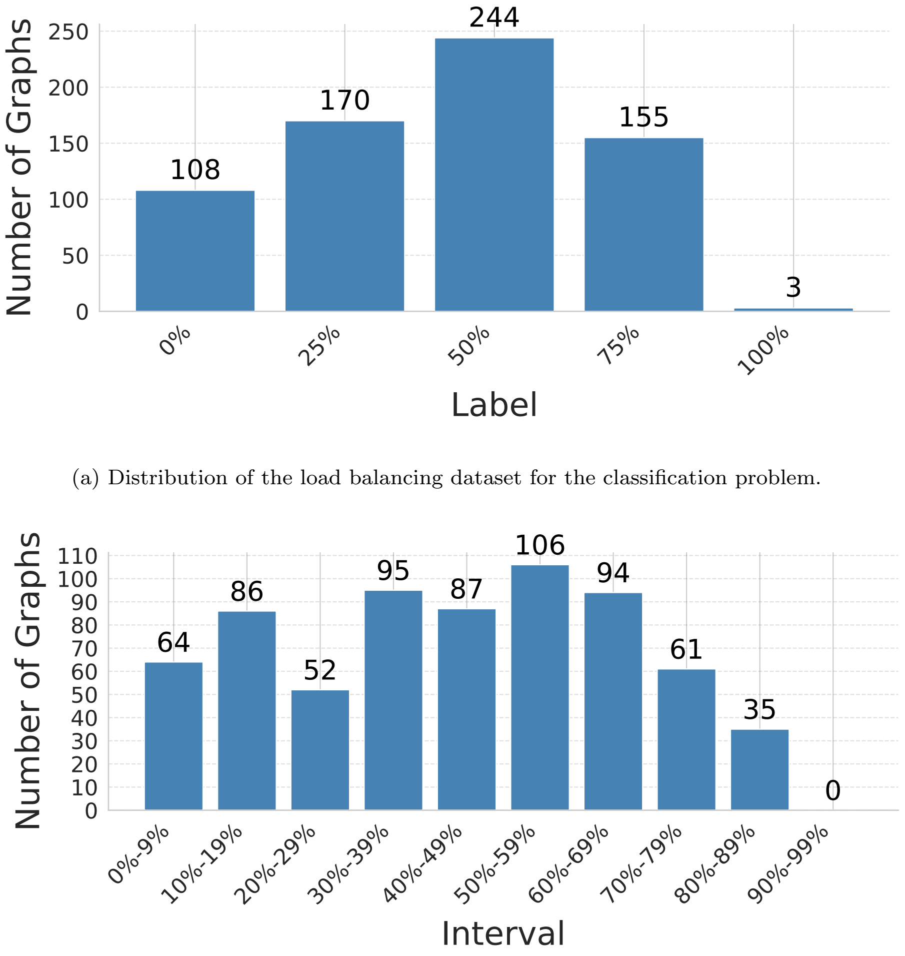
when creating the workload division, this should be proportional to the relative execution times f. For instance, if the execution time of a given task is much smaller in the GPU that in the CPU, then the GPU should receive more amount of work than the CPU. Then, the amount of work given to the CPU $load_{CPU}$ and to the GPU $load_{GPU}$ can be defined as:

$$load_{CPU} = f \cdot load_{GPU}. \tag{5.2}$$

This relation assigns a bigger proportion of work to the PU that has the smallest execution time for a given task. Also notice that the sum of $load_{CPU}$ and $load_{GPU}$ represent the entire workload, hence $load_{CPU} + load_{GPU} = 1$. Equation 5.2 can be extended as:

$$\begin{align}
load_{CPU} &= f \cdot load_{GPU} \tag{5.3}\\
load_{CPU} &= f \cdot (1 - load_{CPU}) \tag{5.4}\\
load_{CPU} &= f - f \cdot load_{CPU} \tag{5.5}\\
load_{CPU} + f \cdot load_{CPU} &= f \tag{5.6}\\
load_{CPU} \cdot (1 + f) &= f \tag{5.7}\\
load_{CPU} &= \frac{f}{1+f} \tag{5.8}\\
load_{GPU} &= 1 - load_{CPU}. \tag{5.9}
\end{align}$$

The regression label $label_{regr}$ results by multiplying $load_{CPU}$ by 100. Regarding the classification task, for each kernel in the dataset we compute the distance of $label_{regr}$ to every possible label (0, 25, 50, 75, 100). Then, the kernel receives the label $label_{class}$ that has the smallest distance.



(a) Distribution of the load balancing dataset for the classification problem.

(b) Distribution of the load balancing dataset for the regression problem.

Figure 5.1: Distribution of the load balancing datasets.

Figure 5.1 shows the distribution of the datasets after the relabelling process. It is interesting that for both cases, there are very few samples were the CPU receives the total workload, which also leads to a high class imbalance. This imbalance is more noticeable in the classification dataset, where the label 50% covers more than one third of the total data samples, and can be problematic when performing future experiments. On the other hand, the two distributions show that assigning the whole workload of a task to the CPU does not achieve a proper balancing. However, this is not true for the opposite case: the applications that are highly parallelizable will have very short execution times when running on the GPU, and consequently assigning most of the workload (or even the entire workload) to the GPU is the best load balancing option. Nevertheless, in general terms, assigning workloads bigger than a 30% and smaller to 70% to the CPU for achieving a good load balancing is the most suitable option for most of the cases.

5.2 Adapting the model

After relabelling the training dataset, we adapted the output of the device mapping model for predicting load balancing for the regression and classification tasks. Figure 5.2 shows the base-ProGraML load balancing model. It consits of four main elements, the embeddings layer, the graph neural network, the graph pooling operation and the readout head; and except for the readout output dimensionality, it is identical to the device mapping model[1].

Trying to have a better understanding of the model's behaviour and also with aims of improving its accuracy, after having made this change in the output, we carried out an ablation study that compares the accuracy of different configurations of models. The model configuration that obtained the best accuracy was then tested against kernels not seen during training.

5.2.1 Ablation study

As mentioned in the previous Section, the load balancing model consists of four main elements: the embeddings layer, the graph neural network, the graph pooling operation and the readout head. Taking this into account, the different configurations slightly modify these parts to see if the new combinations make a positive impact on the model's accuracy. By means of reducing the number of configurations to test, all the experiments keep the same embedding layer.

The first step has been to test the base-ProGraML model against classical machine learning methods. The rationale of this test is to ensure that using 'simpler' against deep learning-based methods does not provide an improve on the task's accuracy. Highly inspired by Grewe *et al.*, a decision tree has been chosen for the classification problem, whereas for the regression problem a linear regression has been trained [20, 43].

[1] Figure 4.4 shows the pooling operator as part of the readout head, while here it appears as a separate module. This is done to give more importance to the pooling operation.

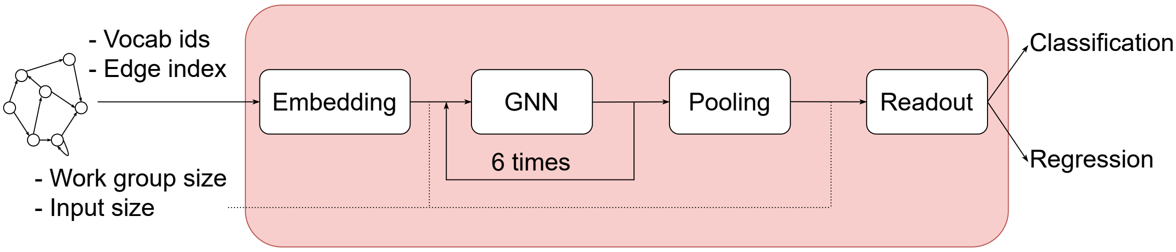


Figure 5.2: Overall schema of the load balancing model. The output of the readout head can be used for both regression and classification. The ablation study has covered different combinations of GNNs, pooling methods and combinations of the readout head, always keeping the same embedding layer.

In order to measure the impact of using Graph Neural Networks (GNN), we compared the ProGraML model with four different GNN configurations. These four models have also used a global mean pooling (the dense vectors of a graph are averaged element wise for creating a single dense vector) operation to create the graph representation. The compared models are the following: a multi-layer perceptron (Mean Pooling + MLP)[2], the ProGraML model using Graph Attention Network (Mean Pooling + GAT), the ProGraML model using Graph Convolutional Network (Mean Pooling + GCN) and the ProGraML model (Mean Pooling + ProGraML) [44, 45]. These tests try to expose the impact that the GNN creates on the final classification, that is why we compare the results of the load balancing model without GNN, the model with two GNNs that consider that all the edges are equal when sending the messages with the ProGraML model that sends different messages depending on the type of the edge.

Regarding the graph pooling operation three different options have been tried: the Top-K pooling operator, the differentiable pooling operator and a custom pooling operator called 'skips' [46, 47]. These configurations explore if using more sophisticated pooling methods helps the model on creating more accurate predictions.

The TopK and differentiable pooling methods are learnable methods that aim to select the nodes that provide more information at each time step. By extracting these nodes and linking them, smaller graphs are created. This iterative process continues until obtaining a single node graph.

The 'skips' method is a variation of the ProGraML model that is highly inspired in the skip connections seen in Convolutional Neural Networks, where after each message propagation step the node representations are mean pooled and stacked horizontally together. The readout head takes as input these stacked vectors for performing the final classification.

To finish with, the readout head of the three models that use a graph pooling operation (Diff pooling, TopK pooling and skips) has been extended with an additional linear layer.

Training methodology

In order to make a fair comparison between models, all of them have been trained and tested with the same methodology. All the models (including the decision tree and the linear regressor) have been trained following a 10-fold cross-validation, splitting the dataset into 80/10/10 for the training, validation and test datasets every fold.

[2] This is equivalent to a model with an embedding layer and a readout head.

All the deep learning models have been trained 300 epochs every fold, and the epoch with the greatest performance metric is chosen for doing the testing. Regarding additional hyper parameters, all the models have been trained with a hidden size of 64, a learning rate of 10^{-3} and a batch size of 256 graphs[3]. The classification models have been trained with a Cross Entropy loss function while the regression ones minimize a Mean Squared Error loss function.

Having both classification and regression models, the performance metrics used to evaluate them are different. For the classification part, the evaluated metrics are the classification accuracy, the F1 score, the precision and the recall, whereas the regression models are compared by looking at the mean squared error, the root mean squared error and the mean absolute error metrics.

Model	F1-Score	Acc.	Prec.	Rec.
Decision Tree	0.532	0.544	0.555	0.544
Base ProGraML	0.729	0.733	0.743	0.733
Mean Pooling + MLP	0.769	0.772	0.781	0.772
Mean Pooling + GCN	0.765	0.771	0.778	0.712
Mean Pooling + GAT	0.756	0.755	0.774	0.755
Mean Pooling + ProGraML	0.758	0.758	0.733	0.741
Diff Pooling + ProGraML	0.767	0.77	0.779	0.77
TopK Pooling + ProGraML	0.739	0.739	0.766	0.739
Skips + ProGraML	0.765	0.769	0.781	0.769
(Diff Pooling + Ext. Readout) + ProGraML	0.772	0.773	0.78	0.773
(TopK Pooling + Ext. Readout) + ProGraML	0.75	0.754	0.762	0.754
(Skips + Ext. Readout) + ProGraML	0.791	0.782	0.792	0.781

Table 5.1: Results of the ablation study for the classification problem.

Results and discussion

Looking at the results, it is clear that using deep learning methods against machine learning not only improves the performance of the model, but also avoids using expert selected features.

On the other hand, using different GNNs also have a significant impact on the model's performance. The GNN proposed in sends different messages for every type of edge, therefore the expressiveness of the model is bigger than the GCN and GAT based models [10]. In addition, using a simple MLP has a obtained a better or comparable performance to the GNN based models. Taking into account that the training dataset

[3] The batch size of the models that use the Diff Pooling operator have been lowered to 32 graphs, due to GPU memory restrictions.

Model	RMSE	MAE	MSE
Linear Regression	23.081	19.406	534.086
Base ProGraML	20.862	13.297	499.9
Mean Pooling + MLP	13.582	9.894	189.5
Mean Pooling + GCN	14.072	10.43	201.888
Mean Pooling + GAT	15.315	10.365	242.278
Mean Pooling + ProGraML	13.141	9.371	176.972
Diff Pooling + ProGraML	11.671	7.689	144.051
TopK Pooling + ProGraML	11.186	7.714	128.741
Skips + ProGraML	12.489	8.794	160.882
(Diff Pooling + Ext. Readout) + ProGraML	12.775	8.638	175.251
(TopK Pooling + Ext. Readout) + ProGraML	15.608	12.304	253.081
(Skips + Ext. Readout) + ProGraML	11.121	7.324	131.06

Table 5.2: Results of the ablation study for the regression problem.

is small, using a simpler model can help avoiding overfitting and therefore obtaining good scores. Nevertheless, the only way of improving the MLP model is by adding more linear layers to it, thus it presents less flexibility than the GNN models in terms of possible future improvements (as the model does not use GNNs, developers cannot try to improve the model's using different GNNs, pooling methods etc.). Apart from that, by comparing the base ProGraML and the mean pooling ProGraML scores, it is noticeable that the original graph pooling operator is acting as an information bottle neck, justifying the extra tests made with different pooling operators.

Another fact to highlight is that the performance differences among the models are easier to notice for the regression problem than for classification. This is probably due to the imbalanced distribution of the classification datase: there are only three samples with the 100% label and the 50% label covers more than one third of the entire dataset. This imbalance complicates the training process and also makes it more difficult to analyze the real differences between the models.

Finally, regarding the pooling based models, they all have obtained a better performance (specially in the regression problem). Nevertheless, using the Differentiable or the TopK pooling operators add extra parameters to the model, which make the training process more complex and making them more likely to over fit. In addition, due to the increase of parameters, the execution time of the forward pass of these models also increases, leading to larger training times.

Figure 5.3 shows a comparison between the models' accuracies against the execution times of their respective forward passes, where the best possible configuration is the lower right part. Among the models that use a more complex pooling method (top-k,

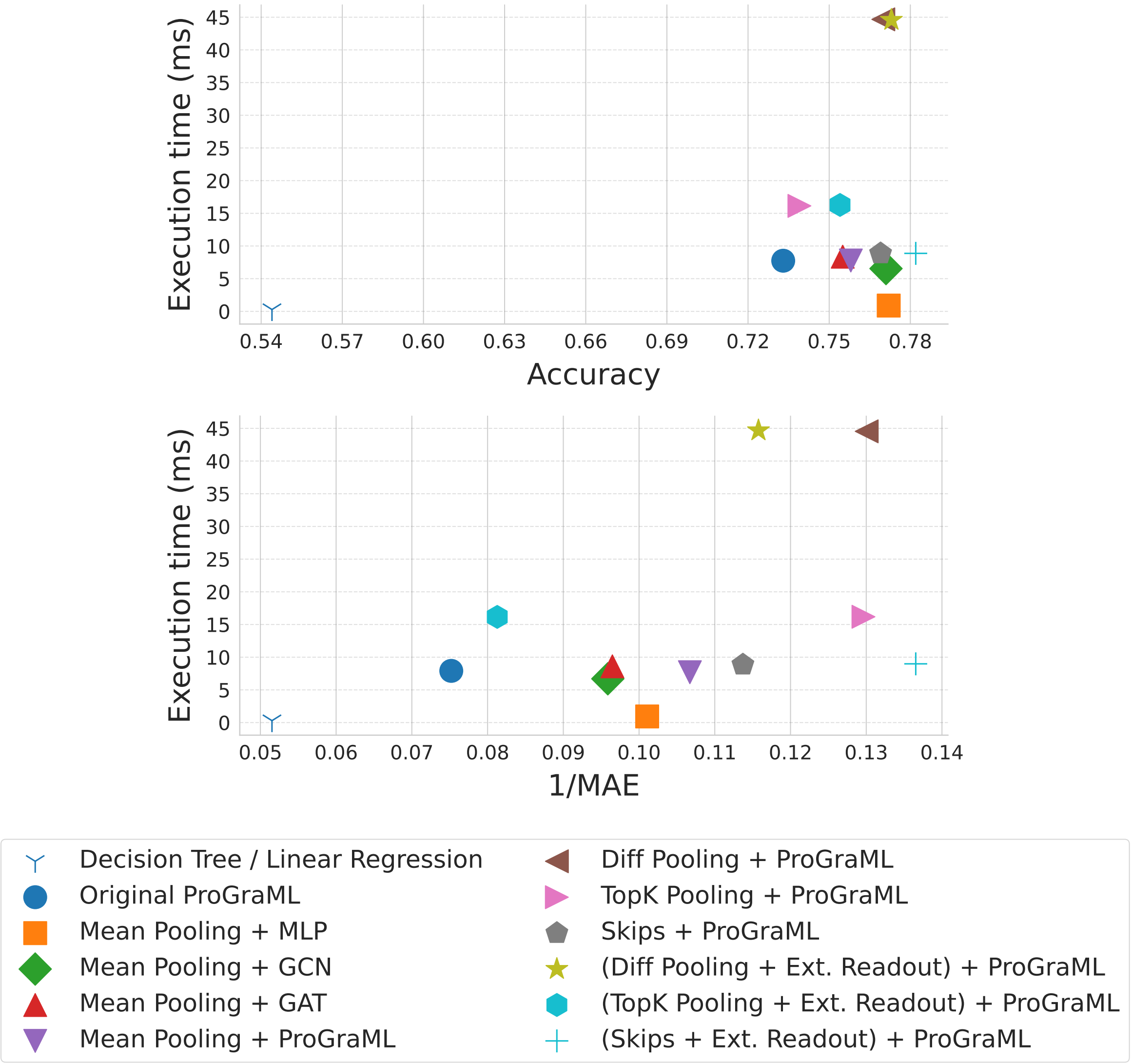


Figure 5.3: Comparison of the models' accuracies against the execution times of their respective forward passes. The top figure shows the classification models and the bottom figure the regression ones. For both cases, the bottom right is the best configuration, as it has the highest accuracy (or the smallest error) with the shortest execution times.

diff and skips), the skips model achieves the best accuracy with the lowest execution time. In addition, while adding a minimum extra overhead in comparison to its mean pooling version, the improvement made on the accuracy is noticeable. Having this into account, the skips model with the extended readout was chosen as the final model.

Figure 5.4 shows the model used for creating load balancing schedules with kernels not seen during training. The model uses skip connections and an extended readout for both the regression and classification problems; without adding too many extra parameters to the model, it obtains the best performance among all the tested variations.

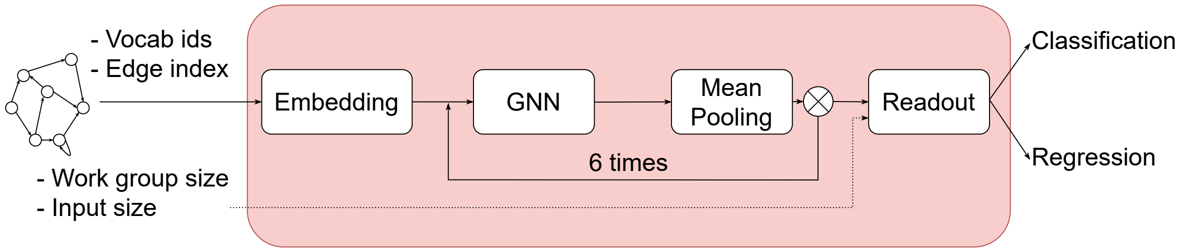


Figure 5.4: Load Balancing model: at each timestep a global mean pooling operation is performed, stacking together the graph's dense vectors. These dense vectors are given as input to the extended graph readout.

5.3 Testing against real kernels

After completing the ablation studies, the final step consist of testing the best model against kernels not seen during the training process. The testing has consisted of comparing the theoretical proportion that the CPU should receive, against the CPU workload proportion given by the model.

Regarding the used model, the ProGraML-skip with the extended readout model has been used for doing the testing, in both classification and regression ways. It has been trained using the whole dataset for 300 epochs, and using the same hyper parameters of the ablation study.

The chosen kernels for doing the testing have been taken from the CHAI benchmark suite for two main reasons: the benchmark suite is already thought for performing load balancing experiments and this suite has not been used for training the models, which adds an extra complexity to the test [42]. Among all the CHAI benchmarks, only the data-partitioning ones have been considered for making the tests.

In order to obtain the theoretical proportion, each input kernel has been executed 5 times in both the CPU and the GPU, and the mean of those execution times has been used for computing the theoretical workload (this has done following the Equation 5.3). After that, giving as input the work group size and the input size in bytes of the kernel, alongside the ProGraML graph to the model, the CPU workload has been predicted. Table 5.3 shows the comparison between the theoretical workload division and the predicted one.

The results of Table 5.3 show that the model is able to predict very accurate workload divisions, even though the technical setups for creating the dataset (this can be seen in Section 3.3.1) and the one used for testing differ. Moreover, the dynamic data given as input for the model can have a big impact on the prediction, thus users must select it carefully in order to obtain the best results.

The kernels with the worst results (SC and RSCD) contain a global synchronization part among the work items, making the execution of each thread to have a very dynamic

Problem name	Benchmark	Ratio CPU	Pred. Regr.	Pred. Classif.
Canny Edge Detection	CEDD	0.48	0.656	0.75
Image Histogram[a]	HSTI	0.566	0.492	0.75
Image Histogram[b]	HSTO	0.507	0.49	0.75
Padding	PAD	0.009	0.164	0
Bézier Surface	BS	0.682	0.54	0.5
Random Sample Consensus	RSCD	0.523	0.301	0
Stream Compaction	SC	0.011	0.43	0
In-Place Transposition	TRNS	0.45	0.44	0.75

[a] Input Partitioning

[b] Output Partitioning

Table 5.3: Comparison between the theoretical work division and the one predicted by the classification and regression load balancing models.

behaviour. Although this synchronization appears in most of the kernels, it is an important part of these two codes. Hence, a possible hypothesis is that the model is not able to make accurate predictions when the kernels have irregular or dynamic behaviors.

Regarding the training methodology, this problem should not be considered as a classification problem because the training dataset is highly imbalanced, adding an additional complexity to the whole training process. When the model does a wrong prediction, all the options are considered equally wrong. For instance, if the target label is 50%, predicting 25% is as wrong as predicting 0%. All this has also had an impact on the CHAI predictions: the classification model has predicted half of the kernels with an error higher to 20%. This is not the case for the regression model, as it has been able to 6 of the kernels with an error smaller to 20%, and among those 6 kernels, three of them were predicted with a difference smaller to a 10%. Consequently, considering the load balancing task as a regression problem facilitates the training process, and it achieves more accurate results.

As an overall conclusion of the load balancing model, the ablation study carried out shows that using different types of messages for heterogeneous graphs enriches the model's expressiveness, and that using a well-designed pooling operator can increase the model's accuracy. In addition, having such a small dataset, developers have to put special attention to avoid overfitting. On the other hand, the results against kernels not seen during training show that the proposed methodology is suitable for predicting load balancing schedules automatically, even when the experimental setups are different. Moreover, the regression model has achieved more accurate predictions than the classification model, probably due to the dataset distribution. Finally, extending this study with new kernels will give provide more insights about the model's weak points and possible improvements.

Chapter 6

Conclusions and Future Work

During this master's thesis, we have created a deep learning based load balancer, that taking as input an OpenCl kernel in ProGraML graph format, the size of the kernel in bytes and the work group size, it is able to predict the amount to assign to the CPU. For that purpose, we have first solved the performance issue of the ProGraML project, by migrating it to pytorch-geometric. After that, we have adapted the model and the training datasets for predicting load balancing. In addition, the accuracy of this model has been deeply examined, by performing an ablation study and by testing the best model against kernels not seen during training. Apart from all these contributions, the author of the original ProGraML work has merged some of the implemented code with the official repository; and has considered our work as a valid alternative for performing the deep learning experiments of the ProGraML work, accepting to put a link to our project on the original ProGraML repository. Moreover, we intend to publish the outcomes of this work as a research paper.

The new implementation of the ProGraML deep learning model achieves very similar performance to the original works in the dataflow and heterogeneous device mapping tasks, while solving the GPU issue of the original work. The new reshaping of the Graph Neural Network gets an speed-up in the execution time of the forward pass of almost x2 with respect to our initial approach, and a speed-up of x846 compared to the original implementation. In addition, using the new reshaping with the pre-processing of the data has reduced the training times of all the dataflow experiments to less than one day without affecting the models' accuracy. The new implementation is based on pytorch-geometric and is stored in a public repository, leaving the models and datasets used accessible to any user. Thanks to this, it will be easier to create new works and make further contributions on the area.

As a main conclusion, the device mapping model has been successfully adapted for predicting a load balancing scheduling. The ablation study has shown that using different messages for the different types of edges helps making more accurate predictions, and that using more sophisticated pooling methods also have a big impact on the model's accuracy. Although the number of tested configurations is relatively small, these two facts already provide a way to go for future projects, and help understanding better the used model. The final load balancing model has achieved an accuracy of 0.782 and a mean absolute error of 7.324 in the ablation study for the classification and regression problems. On the other hand, despite the fact that the setup used for labelling the dataset and our setup are different, when testing our model against real kernels not seen during training, the model has been able to distribute the workload of most of the cases in an accurate way. In fact, the regression model has been able to predict 6 of the 8 tested kernels with a difference of less than a 20% with respect to the theoretical work division, and among them those 6 kernels, three of them were predicted with an error less than 10%. Regarding the classification model, the model was not that accurate, predicting half of the kernels with an error higher to 20%. In addition, the classification model is harder to train due to the high class imbalance of its dataset. The preliminary results show the importance of setting correctly the dynamic data of the kernel, as it can totally change the prediction of the model. In addition, the model seems to struggle with all those kernels that have a dynamic behaviour. Nevertheless, in order to verify these results and to correctly identify the strengths and weaknesses of the model, it would be necessary to test it against more kernels.

Regarding some possible future works, the load balancer could be extended in three different ways. A possible initial approach would be to add more dynamic information to the readout head of the model, to test if it helps making more accurate predictions. Another interesting work would be to create a new graph pooling operator that is able to correctly encode the node's hidden vectors without loosing too much information. Finally, integrating the load balancer of this work with a production oriented tool, that automatically predicts the workload division and assigns the respective tasks to the devices, would help on having a better understanding of the model and making further contributions.

To finish with, this project has opened a new research line within the GAZ research group. We have put an special attention on following a well designed training methodology, while also focusing on the models' efficiency part. As it has been mentioned previously, creating efficient ways of managing deep learning projects is crucial for the incoming era, and we believe that this work can serve as a basis for future research lines.

Bibliography

[1] Some steps towards a safe and sustainable ai. *Nature Electronics*, 6(11):791–791, Nov 2023.

[2] Srini Bangalore, Arjita Bhan, Andrea Del Miglio, Pankaj Sachdeva, Vijay Sarma, Raman Sharma, and Bhargs Srivathsa. Investing in the rising data center economy. *McKinsey & Company*, 2023.

[3] Electricity 2024. Technical report, IEA, 2024.

[4] Indervir Singh Banipal and Sourav Mazumder. How to make ai sustainable, Feb 2024.

[5] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[6] Aaron Becker, Gengbin Zheng, and Laxmikant V. Kalé. *Load Balancing, Distributed Memory*, pages 1043–1051. Springer US, Boston, MA, 2011.

[7] Borja Pérez, E. Stafford, J.L. Bosque, and R. Beivide. Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 157:30–42, 2021.

[8] Yves Robert. *Task Graph Scheduling*, pages 2013–2025. Springer US, Boston, MA, 2011.

[9] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T. Lewis, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162, 2014.

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. Programl: Graph-based deep learning for program optimization and analysis, 2020.

[11] Maria Dávila, Raúl Nozal, Rubén Gran Tejero, María Villarroya, Darío Suárez Gracia, and Jose Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75, 03 2019.

[12] ARM. Heterogenous compute.

[13] A.B. Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *[1993] Proceedings Seventh International Parallel Processing Symposium*, pages 230–237, 1993.

[14] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, apr 1976.

[16] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[17] Mark S. Squillante. *Affinity Scheduling*, pages 11–16. Springer US, Boston, MA, 2011.

[18] M.H. Willebeek-LeMair and A.P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

[19] L.M. Ni and Kai Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Transactions on Software Engineering*, SE-11(5):491–496, 1985.

[20] Dominik Grewe, Zheng Wang, and Michael 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)*, pages 1–10, 2013.

[21] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[22] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, 2017.

[23] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics, 2018.

[24] Emanuele Parisi, Francesco Barchi, Andrea Bartolini, and Andrea Acquaviva. Making the most of scarce input data in deep learning-based source code classification for heterogeneous device mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1636–1648, 2022.

[25] Yao Xiao, Guixiang Ma, Nesreen K. Ahmed, Mihai Capota, Theodore Willke, Shahin Nazarian, and Paul Bogdan. End-to-end mapping in heterogeneous systems using graph representation learning, 2022.

[26] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[27] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[29] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[30] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[31] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99, 2017.

[32] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, Nov 2011.

[33] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.

[34] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[35] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[36] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey 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, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[37] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[38] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, sep 1991.

[39] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing, 2015.

[40] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[41] Peter Jamieson, Ahmed Sanaullah, and Martin Herbordt. Benchmarking heterogeneous hpc systems including reconfigurable fabrics: Community aspirations for ideal comparisons. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[42] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[43] Naomi Altman and Martin Krzywinski. Simple linear regression. *Nature Methods*, 12(11):999–1000, Nov 2015.

[44] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[45] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[46] Hongyang Gao and Shuiwang Ji. Graph u-nets, 2019.

[47] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019.

[48] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. Opencl task partitioning in the presence of gpu contention. In Călin Cașcaval and Pablo Montesinos, editors, *Languages and Compilers for Parallel Computing*, pages 87–101, Cham, 2014. Springer International Publishing.

[49] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[50] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[51] John L. Hennessy and David A. Patterson. *Computer architecture : a quantitative approach / John L. Hennessy, David A. Patterson.* Morgan Kaufmann series in computer architecture and design. Morgan Kaufmann, Waltham, MA, fifth edition. edition, 2012 - 2012.

Figure List

Table List

Appendix

Appendix A

Breakdown of task hours

Figure A.1 shows the Gantt diagram of the tasks described in Section 1.3. In addition, Table A.1 shows the number of hours spent in each task.

Task	Hours
Meetings	35
State-of-the-art	80
Performance issue	172
Migration to PyG	165
Load balancing model	123
Ablation study	85
Testing the model	20
Writing the document	115
Total	795

Table A.1: Number of hours spent in each task of the project.

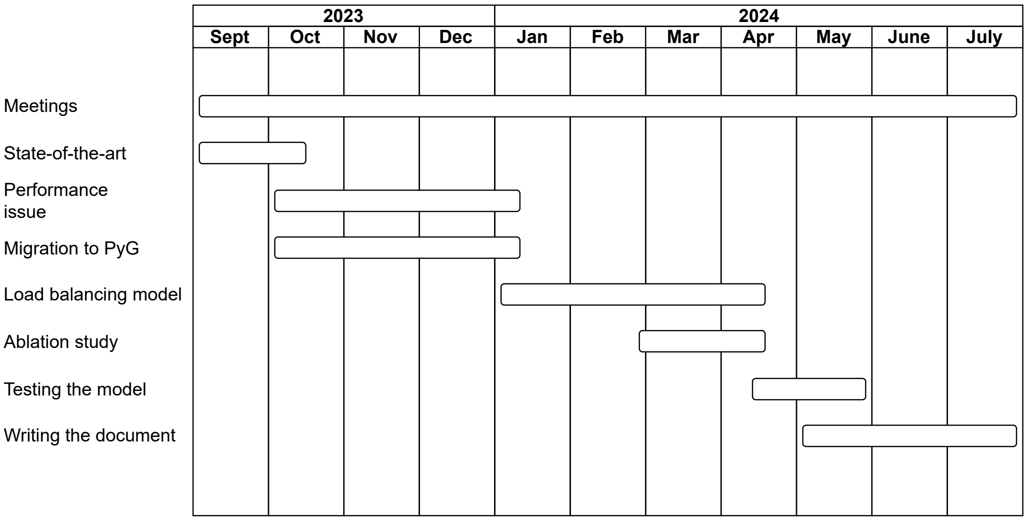


Figure A.1: Gantt diagram summarizing the temporal extent of each of the mentioned goals.

Appendix B

Accelerators

This chapter goes in depth into 4 types of processing units, giving details of their characteristics, their qualities and drawbacks.

B.1 CPU

Central Processing Units are the most important processing units of a computer, as they are designed for running general purpose programs. In addition, they usually take the role of master or host in the heterogeneous system, orchestrating the other PUs inside the system.

Although the design of the CPUs has suffered many changes since the 1950s, the main elements of this processing unit remain intact. First of all, an arithmetic-logic unit (ALU) for performing arithmetic and logic operations, registers for supplying operands to the ALU and for storing the results of the operations and the control unit for orchestrating the memory-fetching and the decoding and execution of instructions [51].

In order to provide higher performance, several CPUs are integrated in a single microprocessor chip, also called as multi-core processors. In this way, CPUs can provide parallel execution.

B.2 GPU

The Graphics Processing Unit (GPU) is a processing unit that was initially designed for accelerating computer graphics and image processing. Nevertheless, as the internal structure of these accelerators provide a high amount of data parallelism opportunity, these have been used as general purpose computing devices for specific problems.

The GPUs can be both integrated with the CPU in the same silicon die or on a dedicated graphics PCI-express card. In the first case, the CPU is fully integrated with the GPU, creating thinner and lighter systems, and also reducing power consumption and communication overhead. Nevertheless, this approach might not be enough when a higher computing performance is needed. Hence, discrete GPUs provide a better processing power at the cost of increasing the energy consumption, heat dissipation and communication overhead.

The GPUs are composed of thousands of processors, where each of these processors are smaller and more specialized than ones of a CPU. More specifically, GPUs have energy-efficient ALUs that have a long latency, but that are heavily pipelined in order to obtain a massive throughput. In addition, they use a fully banked cache subsystem for boosting the memory throughput. Nevertheless, the control unit of the GPUs is much simpler than that of the CPUs, and have neither the capability of performing branch prediction nor data forwarding.

B.3 FPGA

The field-programmable gate arrays (FPGA) are integrated circuits that are programmable and configurable by the user, so that ,one single FPGA can be optimized for many different tasks by programming an specific circuit.

An FPGA consists of an array of programmable logic blocks and interconnections that can be used for performing different logic functions. In addition, the logic blocks of most of the FPGAs also include memory elements (these memory elements can go from simple flip-flops to full memory blocks), allowing these processing units to perform specific functions that solve a given task. Thanks to this, FPGAs can solve specific tasks in a more efficient way than programs executed in general purpose processors.

As far as the main advantages of using FPGAs are concerned, they provide a high flexibility, allowing the users to reconfigure the processing unit to the specific task to be solve. Thanks to the complex combination of logic and memory blocks, this processing unit is suitable for solving both simple and complex tasks. Moreover, they also offer a high computing performance, with a high parallelism and throughput. On the other hand, programming FPGAs can be a complex task, and expertise using Hardware Description Languages is needed.

B.4 ASIC

An Application-Specific Integrated Circuit (ASIC) is an integrated circuit that has been customized for solving a specific computational problem, rather than for general tasks. Widely known ASIC examples are Google's Tensor Processing Units (TPU), an ASIC used for neural network machine learning, or the ASICs for mining digital currencies.

The ASICs are composed of many different circuits where each of them performs a smaller function. Combining these functions the general task that the ASIC was designed for is executed. Instead of creating logic blocks that can be programmed for different tasks as the FPGAs do, ASICs implement these blocks bare metal.

Regarding the advantages of using ASICs, these PUs are the most efficient computational solution (in both energy consumption and computational performance) for a given problem. Nevertheless, being so problem specific makes them very difficult to design and therefore, more expensive to develop. Additionally, the portability between different problems or tasks can also be challenging, even for variations of the same problem.

Appendix C

Summary of metrics

This chapter presents a summary of the machine learning metrics used for evaluating the proposed models. It involves both regression and classification metrics because of the design of the load balancing problem (see Chapter 5).

C.0.1 Regression metrics

The aim of this subsection is to explain the used regression metrics.

- **Mean squared error:**

  The Mean Squared Error (MSE) is defined as the mean of the squared residuals[1]:

  $$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{C.1}$$

  This metric puts a heavy penalty on the significantly large errors when squaring the difference, therefore, it is not very robust against outliers. Apart from that, MSE is not measured in the original units of the dependent variable, making it harder to interpret. The best possible MSE is having a value of 0, which means that the values to predict and the made predictions are the same.

- **Root mean squared error:**

  The Root Mean Squared Error (RMSE) is defined as the square root of the mean squared error:

  $$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{C.2}$$

[1] y_i = Ground truth value, $\hat{y}_i$ = Predicted value, N = Number of predicted elements.

Unlike MSE, RMSE is more robust to the outliers due to the square root, and additionally, the measure given by the RMSE is in the same unit as the dependent variable, simplifying the interpretation of it. However, althought the result will be in the same units, the magnitude is not comparable with values of the dependent variable. The best possible value for RMSE is 0.

- **Mean absolute error:**

  The Mean Absolute Error (MAE) is defined as the mean of the absolute differences:

  $$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{C.3}$$

  As the residuals are not squared, the unit of the MAE is the same as the target unit. Additionally, this metric is more robust to outliers than the MSE because all the errors are treated equally. The best possible MAE is 0.

C.0.2 Classification metrics

This subsection aims to gather all the used classification metrics.

- **Classification accuracy:**

  The accuracy can be defined as the ratio between the correctly made predictions and the total amount of predictions[2]:

  $$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{C.4}$$

  The best possible accuracy is 1 (all the predictions are true positive or true negative) and the lowest is 0 (all the predictions are false positive or false negative).

- **Precision:**

  The precision metric measures the proportion of positive predictions is actually correct. In other words, it measures how good a model is identifying true negative samples. The precision is computed as the ratio between the true positive samples and the sum of the true positive and true negative samples:

  $$precision = \frac{TP}{TP + TN} \tag{C.5}$$

[2] TP = True Positive; FP = False Positive; TN = True Negative; FN = False Negative

The precision metric reaches its best value at 1 (no true negative samples predicted), and its worst value at 0 (no true positive samples predicted).

- **Recall:**

  The recall metric measures the proportion of the number of true positive instances that are correctly classified. Stated differently, it evaluates the hability of a model for identifying false negative samples. The recall score is computed as the ration between the true positive samples and the sum of the true positive and false negative samples:

  $$recall = \frac{TP}{TP + FN} \tag{C.6}$$

  The highest possible value for the recall is 1 (no false negatives predicted), and the lowest is 0 (no true positives predicted).

- **F1-Score:**

  The F1-score is computed as the harmonic mean of precision and recall, and it provides the developer a combined idea of these two metrics:

  $$F1 = 2 \cdot \frac{Precision \times Recall}{Precision + Recall} \tag{C.7}$$

  The F1 score reaches its highest value at 1, and its lowest at 0. In addition, it gives a balanced metric of the precision and recall, making it very useful for classification tasks.