Locality analysis and its hardware implications for Graph Pattern Mining

Ana Arduengo García

Advisors: Miquel Moretó (UPC)
Arvind (MIT CSAIL)

Tutor: Miquel Moretó Planas

A thesis submitted in partial fulfillment of the requirements for the:

Bachelor's Degree in Informatics Engineering

Bachelor's Degree in Mathematics

May, 2023

Abstract

Graphs have numerous applications since they are simple yet effective to model real-life phenomenon. The strength of graph formalism is found in its generality and emphasis on relationships between points rather than the characteristics of individual points. Hence, there is a demand for proficient querying techniques on large data graphs.

Graph Pattern Mining (GPM) are the methods to find relationship over actual graphs. There are two major difficulties to computing GPM algorithms: (a) They have poor locality as a result of erratic accesses when traversing the nearby vertices; and (b) They require a lot of memory bandwidth. In other words, it is challenging to conceal the latency because the computation time following the retrieval of a piece of data from the memory hierarchy is brief. The memory bandwidth waste is another side effect of the poor locality.

Near-data processing (NDP) architecture is a promising solution to improve the performance of GPM applications. It is possible to integrate computation units in/or close to the memory module using NDP architecture, and we can reduce the amount of data that is moved back and forth between the CPU and memory module to achieve high performance and energy-efficient computation. To face these challenges, a system-level simulator, capable of simulating multi-threaded applications in a full-system environment with a complete operating system on a heterogeneous multi-core system, can be required.

In this work, we have addressed the acceleration of GPM applications from the perspective offered by the NDP architecture. We have developed a new simulation tool, based on the integration of two well-known simulators: ZSim (for the cores and the caches) and Ramulator (for the memory). The need to carry out this integration arises from the fact that the implementation available for the joint use of both simulators does not take advantage of the techniques that ZSim uses to reduce the loss of precision. We have implemented in simulation a state-of-the-art GPM accelerator based on the NDP architecture (NDMiner). The new simulation tool allows a detailed NDMiner profiling to identify its weak points. Therefore, it helps to design strategies that alleviate those bottlenecks and improve their performance. Consequently, after realizing experiments with the new simulator, we have elaborated a series of concrete proposals to solve some of the problems detected and to improve NDMiner.

Keywords: Graph Pattern Mining (GPM), Near-Data Processing (NDP), Graph Mining Acceleration, Locality, Hardware Simulation, ZSim, Ramulator

Mathematics Subject Classification (MSC): 68R10

Resumen

Los grafos tienen numerosas aplicaciones porque permiten modelizar fenómenos reales de forma sencilla y efectiva. El punto fuerte de la representación en grafos reside en su generalidad y en que el foco se dirige hacia la relación entre nodos más que hacia las características individuales de cada nodo. Por ello, existe una demanda creciente de técnicas especializadas para el reconocimiento de patrones representativos en grafos.

El reconocimiento de patrones en grafos (GPM) agrupa un conjunto de métodos para buscar relaciones entre sus nodos. La computación de los algoritmos GPM presenta dos grandes problemas: (a) Enfrentan una baja localidad, debido a los accesos erráticos al recorrer vértices cercanos:, y (b) Requieren un gran ancho de banda de memoria. En otras palabras, es difícil ocultar la latencia porque el tiempo de computación después de la recuperación de un grupo de datos de la memoria es corto. Otro inconveniente derivado de la baja localidad es el desperdicio de ancho de banda de memoria.

La arquitectura del procesamiento cerca de la memoria (NDP) es una prometedora solución para mejorar el rendimiento de las aplicaciones GPM. La arquitectura NDP permite integrar unidades de computación en/o cerca del módulo de memoria, reduciendo el movimiento de datos entre la memoria y la CPU y mejorando la eficiencia y el consumo de energía. Para avanzar en este desafío se precisa un simulador capaz de realizar aplicaciones paralelizadas en un sistema heterogéneo con múltiples cores.

En este trabajo, hemos abordado la aceleración de aplicaciones GPM desde la perspectiva ofrecida por la arquitectura NDP. Hemos desarrollado una nueva herramienta de simulación, basada en la integración de dos conocidos simuladores: ZSim (para los cores y las caches) y Ramulator (para la memoria). Hemos tenido que diseñar específicamente esta integración porque la implementación disponible para la utilización conjunta de ambos simuladores no aprovecha las técnicas que usa ZSim para reducir la pérdida de precisión. Luego hemos implementado en el simulador un acelerador GPM que utiliza la arquitectura NDP (NDMiner), entendemos que representa el estado-del-arte al respecto. La herramienta de simulación permite realizar un detallado "profiling" de NDMiner, muy útil para identificar sus puntos débiles. De esta forma, el simulador facilita el diseño de estrategias para mejorar el rendimiento del acelerador. Mediante una serie de experimentos en simulación, hemos elaborado una serie de propuestas concretas para solucionar los problemas detectados y mejorar NDMiner.

Palabras clave: Graph Pattern Mining (GPM), Near-Data Processing (NDP), Graph Mining Acceleration, Locality, Hardware Simulation, ZSim, Ramulator

Mathematics Subject Classification (MSC): 68R10

Resum

Els grafs tenen nombroses aplicacions perquè permeten modelitzar fenòmens reals de manera senzilla i efectiva. El punt fort de la representació en grafs resideix en la seva generalitat i en què el focus es dirigeix cap a la relació entre nodes més que no pas cap a les característiques individuals de cada node. Per això, hi ha una demanda creixent de tècniques especialitzades per al reconeixement de patrons representatius en grafs.

El reconeixement de patrons en grafs (GPM) agrupa un conjunt de mètodes per cercar relacions entre els nodes. La computació dels algorismes GPM presenta dos grans problemes: (a) Enfronten una baixa localitat, a causa dels accessos erràtics en recórrer vèrtexs propers:, i (b) Requereixen un gran ample de banda de memòria. En altres paraules, és difícil amagar la latència perquè el temps de computació després de la recuperació dun grup de dades de la memòria és curt. Un altre inconvenient derivat de la baixa localitat és el malbaratament d'amplada de banda de memòria.

L'arquitectura del processament a prop de la memòria (NDP) és una solució prometedora per millorar el rendiment de les aplicacions GPM. L'arquitectura NDP permet integrar unitats de computació a/o prop del mòdul de memòria, reduint el moviment de dades entre la memòria i la CPU i millorant l'eficiència i el consum d'energia. Per afrontar aquest desafiu cal un simulador capaç de realitzar aplicacions paral·lelitzades en un sistema heterogeni amb múltiples cores.

En aquest treball hem abordat l'acceleració d'aplicacions GPM des de la perspectiva oferta per l'arquitectura NDP. Hem desenvolupat una nova eina de simulació, basada en la integració de dos coneguts simuladors: ZSim (per als cores i les caches) i Ramulator (per a la memòria). Hem hagut de dissenyar específicament aquesta integració perquè la implementació disponible per a la utilització conjunta de tots dos simuladors no aprofita les tècniques que fa servir ZSim per reduir la pèrdua de precisió. Després hem implementat al simulador un accelerador GPM que utilitza l'arquitectura NDP (NDMiner), que representa l'estat de l'art. L'eina de simulació permet realitzar un detallat "profiling" de NDMiner, molt útil per identificar els seus punts febles. D'aquesta manera, el simulador facilita el disseny d'estratègies per millorar el rendiment de l'accelerador. Mitjançant una sèrie dexperiments en simulació, hem elaborat una sèrie de propostes concretes per solucionar els problemes detectats i millorar NDMiner.

Paraules clau: Graph Pattern Mining (GPM), Near-Data Processing (NDP), Graph Mining Acceleration, Locality, Hardware Simulation, ZSim, Ramulator

Mathematics Subject Classification (MSC): 68R10

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To understand a complex system, we must first understand how its components interact between them. To put it another way, a map of its wiring diagram is needed. A graph is a catalog of a system components, which are often referred to as nodes or vertices, as well as the direct interactions between them, which are referred to as links or edges. This graph representation provides a common language for studying systems that may be vastly different in appearance, nature, or scope [1]. Thus, graphs are mathematical structures that are used for describing relations between entities and are used almost everywhere.

Graph Mining is a collection of tools and techniques that are used to: (a) investigate the characteristics of real-world graphs; (b) predict how the properties and structure of a graph may affect a given application; and (c) create models capable of producing graphs that match the patterns that appear in real-world graphs.

Graph pattern mining (GPM) has emerged as a new class of data-intensive applications that has aroused interest in the architecture [2, 3, 4] and system [5, 6, 7] fields. GPM has a wide range of real-world applications, including pattern search over semantic data and motif extraction from gene networks. GPM differs fundamentally from general graph processing applications in some ways: (a) the computation uses more complicated iterations that could lead to load imbalance; and (b) there are numerous data accesses during the computation. As a result, using traditional hardware (such as a CPU or GPU) to speed up the computation is difficult.

Due to these challenges, graph mining algorithms have recently evolved and hardware has accelerated. The architectural paradigm known as Near-Data Processing (NDP), also referred to as Processing-in-Memory (PIM), has shown promise in overcoming the memory wall issue for data-intensive applications. The latency and energy overheads associated with data movements can be significantly reduced by moving the computations closer to the data locations in the main memory. Addi-

tionally, we can make better use of the memory device limited internal access paths and data bandwidth.

Increasing the data access bandwidth is the key to optimizing the performance of large-scale graph processing. However, there are five difficulties in achieving high bandwidth on conventional architectures [8, 9]:

- Intensive Data Access: On the one hand, graph applications typically result in a lot of requests for data access. Graph processing, on the other hand, has a high data-access-to-computation ratio, meaning that the majority of its operations are connected to data accesses.
- Irregular Computation: The power-law distribution causes a wide range in the computation workloads for various vertices. This will result in a serious problem with workload imbalance and communication overhead.
- Poor locality: For a straightforward operation on one vertex, access to all of its neighbors may be necessary. Such poor locality causes inefficient bandwidth utilization on multi-processing unit architecture and erratic access to global data.
- Redundant set intersection operations: GPM is defined as a search tree exploration process starting from each vertex in the graph, and then the following potential vertex is used to extend on the partially matched subgraph using set intersection and subtraction operations. The set intersection operation becomes the bottleneck due to the frequent computation of the candidate vertices and can consume the majority of the execution time. However, most of set intersection operations are unnecessary, i.e., the same set of vertices is being encountered, and the same set intersection operation is done.
- High Data Dependency: The graph vertices connections nature is what leads to the data dependency. It is challenging to investigate the parallelism in graph processing due to heavy dependencies. Data conflicts may appear.

The overall result is that memory has become a bottleneck and current frameworks are not suitable for this type of applications.

To face these challenges, a system-level simulator, capable of simulating multi-threaded applications in a full-system environment with a complete operating system on a heterogeneous multi-core system, can be required. Furthermore, the simulator should also be capable of a detailed system-level profiling to identify the bottlenecks, therefore, helping in designing strategies and architectures to alleviate these bottlenecks.

In this work, we develop a new simulation tool, based on ZSim+Ramulator, capable to replicate the operation of a state-of-the-art graph pattern mining accelerator

using near-data processing (NDMiner), analyzing their workload and identifying their bottlenecks.

In Section 1.1, we give a summary of the major issues and research issues that this work addresses. The main contributions of this work are then discussed in Section 1.2. Finally, we describe its structure in Section 1.3.

1.1 Motivation

Near-Data Processing (NDP) is an architectural paradigm to address the challenges for GPM applications. Typical 3D stacked memory based NDP systems contain massively parallel processing units, each of which can access its local memory as well as other remote memory regions in the system. In such an architecture, minimizing remote data access and achieving compute load balancing presents a critical trade-off, where existing solutions can only enhance one but sacrifice the other.

However, NDP systems provide an opportunity for emerging data-intensive applications. Currently, there are several research works on the integration of the computation unit with the logic layer of 3D-stacked DRAM. To develop these techniques, a system-level simulator, capable of simulating multi-threaded applications in a full-system environment with a complete operating system on a heterogeneous multi-core system, can be required.

The motivation for this work is to simulate the architecture of the-state-of-the-art NDP GPM accelerator. For this, we have to design a new simulation tool, which has used to analyze the workload of the NDMiner framework, detect their bottlenecks and propose strategies to solve the problems derived from the trade-off between data access optimization and load balancing. Consequently, we have elaborated a series of concrete proposals to solve some of the problems detected.

1.2 Contributions

On this thesis, we focus on a state-of-the-art architecture for GPM acceleration that moved the computation closer to the memory: NDMiner.

In order to be able to analyse its weak points and address possible solutions, we will go over the following steps:

- **Development of a new simulation tool based on ZSim+Ramulator.** We join two simulators: ZSim for the cores and the caches and Ramulator for the memory. The need to carry out this integration arises from the fact that

the available implementation does not leverage the techniques ZSim uses to reduce the loss of accuracy.

- **Implementation in simulation of NDMiner.** The NDMiner framework have been implemented in the simulator, and the results published by the authors have been reproduced, as verification of the correct installation and operation of the simulator.

- **NDMiner workload analysis.** Using the simulation tool, we have analyze the NDMiner workload, performing a detailed profiling identifying their main bottlenecks.

- **Solution proposals.** Using the analysis done in NDMiner we will propose solutions to alleviate the bottlenecks found.

1.3 Thesis Outline

Here, we provide a reader's guide and a summary of the material covered in the chapters. Most of the chapters stand alone to make them easier to read. The three major sections of the thesis are as follows:

Part I: Background

The objective of this part is to provide a context to the work. This part contains the justification of the research, a description of the main aspects addressed in it, the work itself, and a review of previous relevant studies.

Part II: Implementation and Results

This part describes the essential aspects of the work and therefore brings together the most relevant content. This part includes (a) the description of the simulation tool that we have developed, based on the integration of ZSim and Ramulator; (b) the implementation in simulation of NDMiner graph mining accelerator, which we have used as a reference for the state-of-the-art with NDP architecture; (c) the reproduction of the results published by the authors, as verification of the correct installation and operation of the simulator; (d) the NDMiner workoad analysis, identifying weak points; and (e) the proposal of concrete strategies that alleviate them and, therefore, improve the NDMiner performance.

Part III: Project Management, Budget, and Sustainability

This part describes the planning required by the project, the resources used for the work, and the management of difficulties encountered during its development. This part also covers the project budget and the three main elements of its sustainability: economic, social and environmental.

Part I

Background

Chapter 2

Graph Structures

2.1 What is a graph (data structure)?

Graph structures are a very natural way of representing natural links in our world. They are comprised of a set of vertices and the links between one another. An object such as a protein as part of a biological network or an individual in a social network is represented by a vertex in a graph. An interaction, like a friendship in a social network, or a biochemical relation between two proteins within a cell [10] corresponds to an edge in such a network. A common language to study systems that may be very different in scope, appearance, or nature is provided by this network representation. The network representation is the same for three distinctly different systems as seen in Figure 2.1.

This concept can conveniently be applied to real-world situations like human relationships or molecular interactions. Therefore there has been high interest in this type of structures in the last decades as a way to understand interactions between entities. Some of its most common applications can be found in social networks, cyber-security and e-commerce (Figure 2.2).

Mathematically, a graph G is defined as an ordered pair $(V(G), E(G))$ where $V(G)$ refers to the set of vertices and $E(G)$ to the set of edges. All elements in E are a pair of vertices of $V(G)$. Then, we say that u is connected to $v \in V(G)$ if $(u, v) \in E(G)$. We differentiate between directed and undirected graphs depending on if the pairs in $E(G)$ are ordered or not. In other words, depending on if (u, v) is equivalent to (v, u). If they are equivalent, then the graph is undirected. In this case the neighborhood of $v \in V(G)$, $N(v)$ is all u such that (u, v) or $(v, u) \in E(G)$. If they are not equivalent then the graph is directed and we separate the nodes linked to v in outward and inward neighbors depending on the order in the pair. In this thesis, we will focus on undirected graphs.

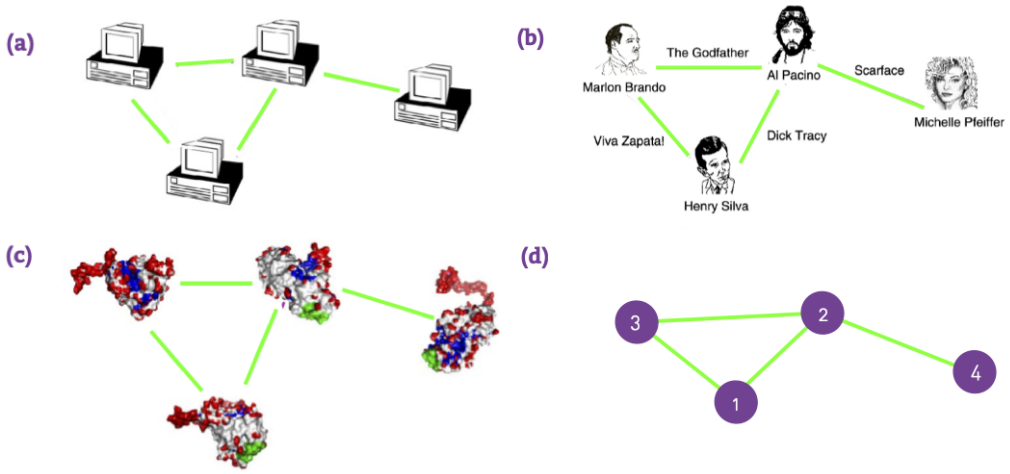

Figure 2.1: The same graph for different networks. (a) routers in the Internet; (b) actors that played in the same movie; and (c) proteins that can bind to each other in the cell. While the links and the nodes are different, the representation is the same, consisting of N = 4 nodes and L = 4 links, as shown in (d) [1].

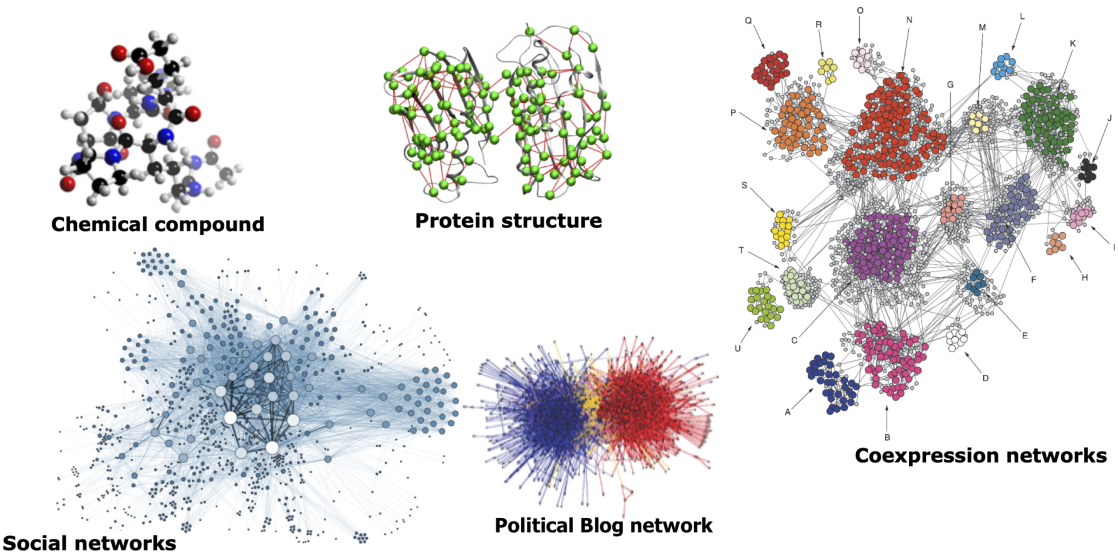

Figure 2.2: Graph usage in different fields.

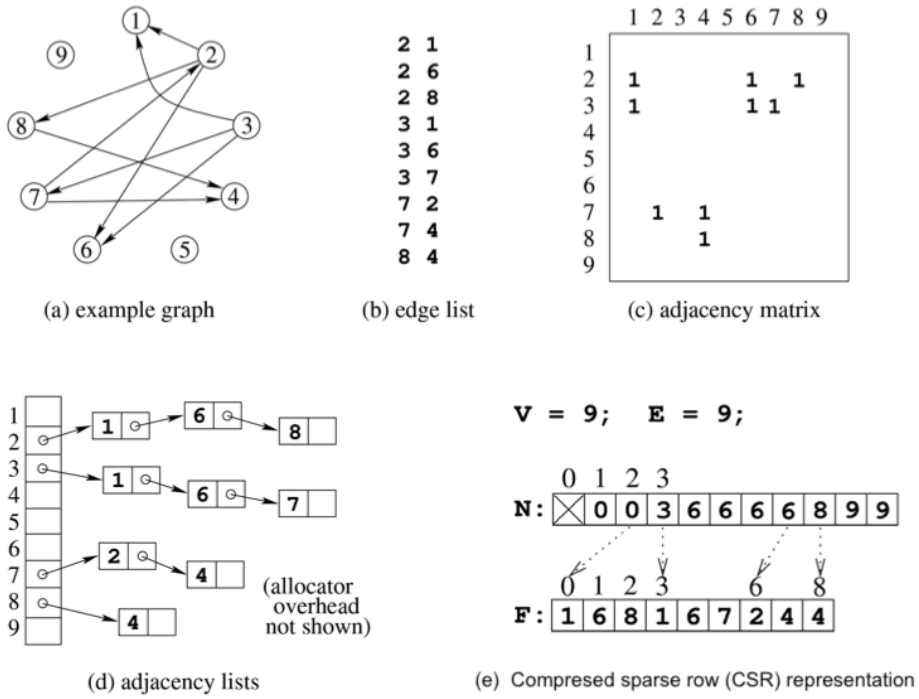

Figure 2.3: (a) An example graph; (c) An adjacency list for the example graph; (d) An adjacency matrix for the example graph; (e) CSR representation of example graph [11].

One of two common data structures is typically used to represent graphs: adjacency matrices (Figure 2.3(c)) and adjacency lists (Figure 2.3(d)). Both data structures are essentially arrays with vertices as their indices; In order to achieve this, each vertex must have a distinct integer identifier between 1 and V.

The adjacency list is the most typical data structure for storing graphs. An adjacency list is an array of lists, each of which contains the neighbors of a particular vertex (or, in the case of a directed graph, the out-neighbors).

For graph analysis applications in which the graph itself is not modified, Compressed Spares Row (CSR) representation (Figure 2.3(e)) is more frequently used. This representation consists of two arrays. One of them is the concatenated neighborhood lists of each vertex in the graph. The other is a pointer to the start of the neighborhood list itself. It is more compact, but it is not easy to modify any of the neighborhood lists. Therefore it is only suitable for static graphs and applications that do not apply transformations.

	$\|V\|$	$\|E\|$	**Max degree**
wiki	8 298	103 689	893
Citeseer	3 312	4 536	99
Github	37 700	289 003	6 809
Mico	100 000	1 080 156	1 359

Table 2.1: Datasets used in this work

2.2 Databases and real-world graphs

Real-world graphs also typically have the following characteristics [12]:

- Power law - A power law vertex degree distribution is a characteristic of many real-world graphs. The power law degree distribution has the effect that most vertices have low degrees, but a small number of vertices will have very high degrees. When processing a scale-free graph in parallel, this property, known as scale-free in literature, can result in a significant load imbalance.
- Small diameter or small world property - Despite being sparse, many graphs are linked together to form enormous connected components with tiny diameters. The longest shortest path between any two graph vertices is the diameter of the graph.
- Community Structure - Communities are the collective nodes that connect to form interconnected clusters. There are more connected edges than outgoing edges in a cluster. This characteristic causes some clusters to be highly interconnected while having few connections to other clusters.

The graph databases that we used and their key features are listed in the table in Table 2.1

Chapter 3

Graph Pattern Mining

The main goal of Graph Pattern Mining is, given a graph data set G and a set of graph patterns $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ find all embeddings of P in G. An example is shown in Figure 3.1. $\mathcal{P}$ is only one graph, a triangle and there are three different embeddings of it in G.

The study of patterns in actual graphs is relevant because it is important: [13]:

- Comprehending human (also protein, machine) behavior: By identifying patterns of connectivity, domain experts (biologists, sociologists) may be able to create hypotheses that explain when and how humans (proteins, machines), or their parts, interact with one another.
- Anomaly detection: neither anomaly nor "normal" can be perfectly defined but we can say that an anomaly is an event (node, subgraph), whose characteristics are very infrequent, and thus very different from most of the other population. We also search for local anomalies, that is, very particular edges, subgraphs, or nodes. They should differ from the "normal" patterns, so knowing the patterns that appear most commonly in graphs is necessary for the detection of such outliers.
- Graph compression: Data regularities that can be used to compress the data are represented by graph patterns. Keeping and compressing historical data is crucial for internet-scale graphs like Facebook, which at the time of writing this has "more than 950 million active users".

A naive approach to find the embeddings would be to enumerate all subgraphs of G with the same number of vertices as each $P \in \mathcal{P}$ and check if they are equivalent to G. Nevertheless, we work under the premise that $\mathcal{P}$ is known. In other words, we

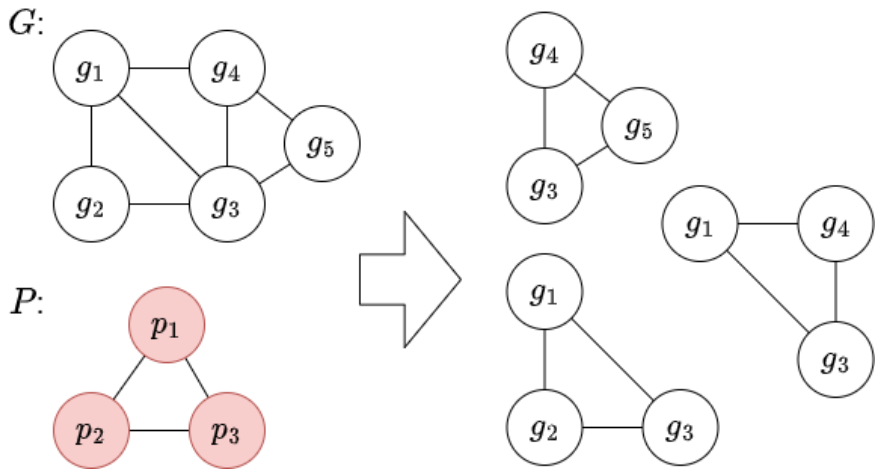

Figure 3.1: Example of the embeddings of a triangle found in the graph G

know how the topology of each $P \in \mathcal{P}$. We can take advantage of that information to narrow the search.

To illustrate how this can be done we walk through the example shown in Figure 3.1. Each step will be illustrated in Figure 3.2. We will map each vertex of P to possible candidates in G one by one. First, we start with node p_1 in P. We could map it to any other vertex in G, so we add all of them as possible candidates. Let's focus on the case where g_1 in G is mapped to p_1. Now, we decide we are matching p_2 in P. We know that p_1 and p_2 are connected, therefore, because we want to maintain the structure of the pattern, we also require the same thing from their correspondent mapping in G. In other words, the candidates to be mapped to p_2 will need to be connected to g_1. The same reasoning is followed for the mapping of p_3. As we can see, we have arrived to two possible mappings. Nonetheless, they are both the same. That is because the pattern P is symmetric. We could for example interchange any two nodes and we would still end up with the same pattern. To remove those symmetries we "block" those changes by imposing an order over the vertex in P. We not only consider the connectivity between the nodes in P but also the order we impose. In this case, we impose that if f is the mapping, then $f(p_1) > f(p_2) > f(p_3)$, where the relationship ">" is decided using the ids of the vertices.

Now, we will proceed to introduce the mathematical terminology to formally define GPM queries.

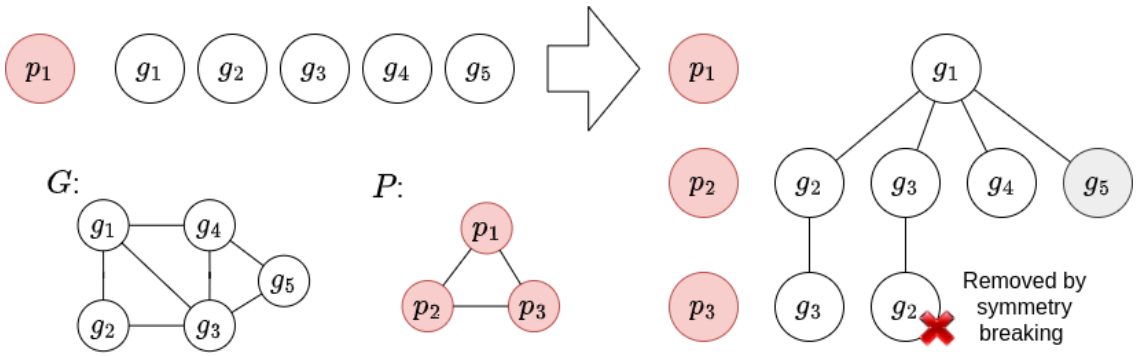

Figure 3.2: Matching the triangle to the Graph G. At each step, we extend the mapping by adding new vertices that comply with the topology of the pattern P we are searching for.

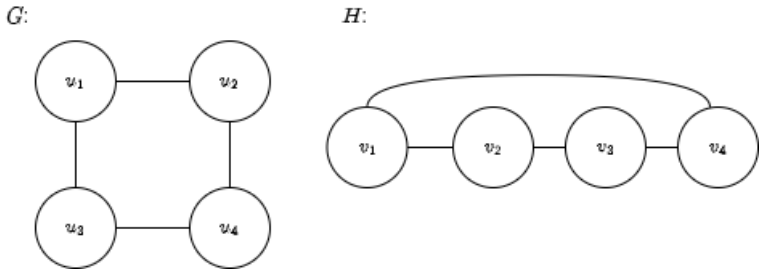

Figure 3.3: Example of an isomorphism between G and H. Every u_i is mapped to v_i.

3.1 Mathematical terminolgy

Definition 3.1.1 (Graph isomorphism)**.** Given G and H two graph structures. $f : G \to H$ is a **graph isomorphism** if it is a bijection between the vertex sets and if $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. In other words, it is a bijection between the vertex sets that preserves the topology of the graph. If such f exists then we say that G and H are **isomorphic** and we denote it with $G \cong H$.

If we ignore differences between individual nodes, isomorphism captures the idea that two networks have the same edge structure, that is, the same topology. This concept is illustrated in Figure 3.3. The graph structure is the same, and they are isomorphic, despite their apparent differences.

Definition 3.1.2 (Graph automorphism)**.** A **graph automorphism** is an isomorphism from the graph onto itself. It is a form of symmetry of the graph.

Definition 3.1.3 (Partial order)**.** The **Partial order** is a set of order rules we impose over the identifiers of the vertex to eliminate the possibility of an automorphism (Figure 3.4). That is, to avoid non-canonical matches we break the symmetries of a

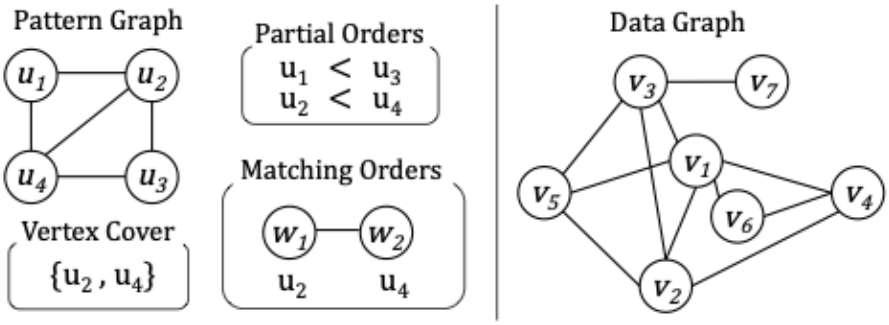

Figure 3.4: Data graph, Pattern graph, and partial and matching orders definition [14]

.

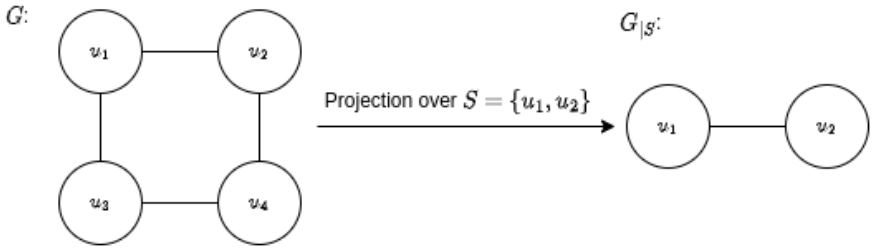

Figure 3.5: Projection of G over $S = u_1, u_2$.

pattern p by enforcing a partial ordering on matched vertices [15]. For this, we must enumerate all automorphisms of the pattern p to find symmetries, and iteratively sorting pairs of symmetric vertices until the only automorphism that matches the ordering is the one mapping each vertex to itself [14]. The partial ordering for the Figure 3.4 example pattern is $u1 < u3$ and $u2 < u4$. From now on, we will refer to the partial order as $\mathcal{PO}$

Definition 3.1.4 (Connected graph)**.** Given G a graph structure. We say it is a **connected graph** if and only if $\forall u \in V(G), \exists v \in V(G)$ such that $(u, v) \in E(G)$.

Definition 3.1.5 (Projection of a graph)**.** Let $S \subseteq V(G)$. The **projection** of $V(G)$ over S is the graph H such that $V(H) = S$ and $(u, v) \in E(H)$ if and only if $u, v \in S$ and $(u, v) \in E(G)$ (Figure 3.5).

Definition 3.1.6 (Matching order)**.** It is the order in which we will be mapping the vertex of P to G (Figure 3.4. Let N_P be the set of natural numbers smaller or equal than $|V(P)|$. The **matching order** of the subgraph P is a bijection $\pi : V(P) \to N_P$ such that $\forall i \in N_P$, the projection of P over $\pi^{-1}([1..i])$ is a connected subgraph of P. From now on we will also refer to $\pi(u)$ as the **matching order** of u and we will denote the projection of P over $\pi^{-1}([1..i])$ as M_i^π.

The connectivity requirement is not usually included in the matching order definition, but it is employed in the most common algorithms because it allows to cut

down in the number of iterations executed in the search.

Notation 1. We will denote the projection of P over $\pi^{-1}([1..i])$ as M_i^π. This will be the subgraph of P that has already been mapped at step i.

Notation 2. For clarity, unless indicated otherwise, we will use u_i to refer to $\pi(i)$.

Definition 3.1.7 (i-partial match)**.** Let $0 < i \leq |V(P)|$ and π a matching order of P. An **i-partial match** of P is an isomorphism $f : M_i^\pi \to G$. We will denote the space of these isomorphisms as R_i^π. We will refer to $f(M_i^\pi) = A_i^f$ or **i-antecesors** of f. This is the subgraph of G that has been mapped to M_i^π at step i.

Definition 3.1.8 (Backwards neighbors)**.** Let $u \in V(P)$ such that $\pi(u) = i$. The **backwards neighbors** of u is the set $N_+^\pi(u) = \{w \in N(u) | \pi(w) < i\}$, i.e, they are the neighbors of u that have smaller matching order.

Definition 3.1.9 (Candidate set)**.** Let π be a matching order of P. Let $i < |V(P)|$. Given f an i-partial match, the (i+1)th **candidate set** $C(f|i+1) = \{v \in G | \forall u \in N_+^\pi(u_{i+1}), (f(u), v) \in E(G)\} = \bigcap_{v \in A_i^f | f(u)=v,\, u \in N_+^\pi(u_{i+1})} N(v)$.

Definition 3.1.10 (Result set)**.** At step $i+1$ the result set is $Z_{i+1} = C(f|i+1) \setminus A_i^f$. It is those candidates vertices that have not been yet mapped to a $u_j, j < i+1$. If $f = \{(u_1, v_1), ..., (u_i, v_i)\}$, it is the set of vertices $v \in V(G)$ such that $f' = f \cup (u_{i+1}, v) \in R_{i+1}^\pi$.

3.2 Execution model

Using the terminology presented, the search model for one P is shown in Algorithm 1. At each i-step, we take all i-partial matchings and try to expand them with the mapping of 1 more vertex. In line 6, we impose the connections pattern P have. In line 7 we remove redundancies, in other words, we avoid a vertex appearing more than once. Finally, we apply the partial order rules to avoid automorphisms in the final mapping. This gives us the set with the vertices that are candidates to be mapped to the $i+1$ vertex in P. This algorithm finds all embeddings because at each step we guarantee that all vertex that maintain the initial topology of the pattern are added as candidates. At the same time, we also guarantee that vertex that do not maintain it are added.

As we had seen in the example in Figure 3.2, at each step we were extending a tree. From now on, we will refer to it as **Search Tree**. In Algorithm 1 we are expanding said tree in a BFS manner. Still, a DFS execution in which we complete each whole isomorphism one by one is more beneficial for hardware as it better exploits temporal reuse in the data. In line 6, we do the intersection of neighborhood lists of nodes previously matched. If we go branch by branch, there is a higher chance the neighborhood lists of the ancestors will still be present in the cache.

Algorithm 1: GPM search in BFS

Data: $G, P, \pi, \mathcal{PO}$
Result: R_n^π *where* $n = |V(P)|$

$R_1 \leftarrow \{f \mid \quad f = \{(u_1, v)\} \quad \forall\, v \in V(G)\}$;
$i \leftarrow 1$;
while $R_i \neq \emptyset$ *and* $i \leq n$ **do**
  $R_{i+1} \leftarrow \emptyset$;
  foreach $f \in R_i \in$ **do**
    $C(f|i+1) \leftarrow \bigcap_{v \in A_i^f | f(u)=v,\, u \in N_+^\pi(u_{i+1})} N(v)$;
    $Z_{i+1} \leftarrow C(f|i+1) \setminus A_i^f$;
    $\widetilde{Z_{i+1}} \leftarrow ApplyPO(\mathcal{PO}, A_i^f, Z_{i+1})$;
    $R_{i+1} \leftarrow R_{i+1} \cup \{f \cup (u_{i+1}, v),\, \forall v \in \widetilde{Z_{i+1}}\}$
  $i \leftarrow i + 1$;

3.3 Programming example

In this section, we present how the algorithm already presented looks when it goes over the Search Tree in a DFS manner for the triangle (Algorithm 2). The final result in this case is the set of vertices of G in the order they have been mapped to the triangle.

Algorithm 2: Triangle search in DFS

Data: G
Result: F

$F \leftarrow \emptyset$;
foreach $v0 \in G$ **do**
  foreach $v1 \in N(v0)$ **do**
    if $v1 > v0$ **then**
      break ;
    foreach $v2 \in N(v_0) \cap N(v1)$ **do**
      if $v2 > v1$ **then**
        break ;
      $F \leftarrow F \cup \{(v0, v1, v2)\}$

Chapter 4

Related Work

4.1 GPM acceleration problems

General purpose processors are not designed to deal with this type of workloads. There have been numerous proposals to design hardware accelerators that better address their main necessities. In this chapter, we will go over the opportunities to accelerate GPM with hardware and describe some proposals from the literature.

4.1.1 Parallelism

In Chapter 3 we saw that, in a pattern aware search, we build several Search Trees that we extend at each step to find the embeddings of the pattern in the graph data set. These Search Trees are independent from one another, and they unfold an opportunity for parallelism. Each of them can be fully extended without any need for synchronization. In order to exploit it, processors can take advantage of already existing parallel environments such as hybrid clusters, multicores, and GPUs [16]. There are some challenges a high parallel system needs to overcome [17, 12]. First, the workload is difficult to distribute due to the power-law distribution in graphs. The threads/cores that take the Search Trees with highly connected root nodes will have a much higher workload. In order to address this issue work stealing paradigms have been proposed [18]. Then, it becomes a problem of data transfer. If a core wants to steal the work a different core started, it will also need to take the current progress (Search Tree) as it is a crucial part for finishing the computation.

4.1.2 Specialized computation units

Normal processors are equipped with an arithmetical logic unit (ALU). The capabilities they offer do not align with the necessities of set operations. Figure 4.1 is an analysis of the weight of set operations in GPM workloads for GPUs. For every application, it is over 60%. Amdahl Law is a very common concept in computer science. It states how much speedup we can gain if we only improve a part of the workload. A clear and very intuitive conclusion is that to obtain the best improvements we should focus on the part of the computation that takes up most of the execution. If we apply Amdahl Law to figure 4.1 we can conclude that we can improve performance a lot if we optimize set operations. Several hardware accelerators have implemented special set operation units [19, 20].

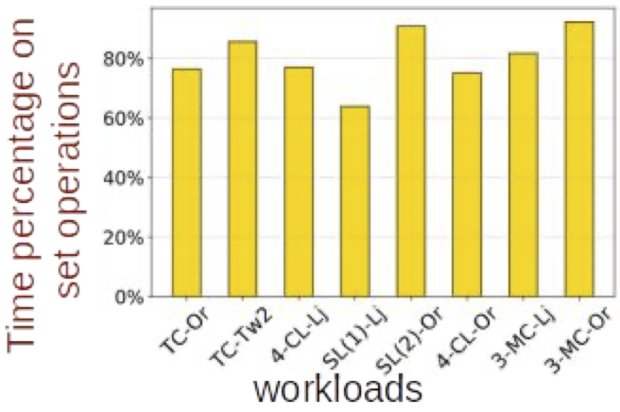

Figure 4.1: Set operations are the majority of the workload [19].

4.1.3 Optimized memory hierarchy

Graphs suffer from very irregular memory access patterns due to their connection focus nature. For this reason, typical solutions to optimize memory accesses do not work here. Prefechers are based on identifying a pattern in the access addresses. Caches, optimize the memory accesses based on the observation that data has spacial and temporal locality. Because of how randomized accesses are in these workloads, spacial locality does not apply. The only thing left is temporal locality, in other words, reusing the same data. What ends up happening is that, in graph applications, accesses typically end up relying on main memory as we can see in Figure 4.2.

Some proposals focus on ways to pre-process so it is stored in memory in a way that better aligns with current memory access optimizations. One of the simplest pre-processing strategies is ordering vertex by degree. If the graph is saved in CSR

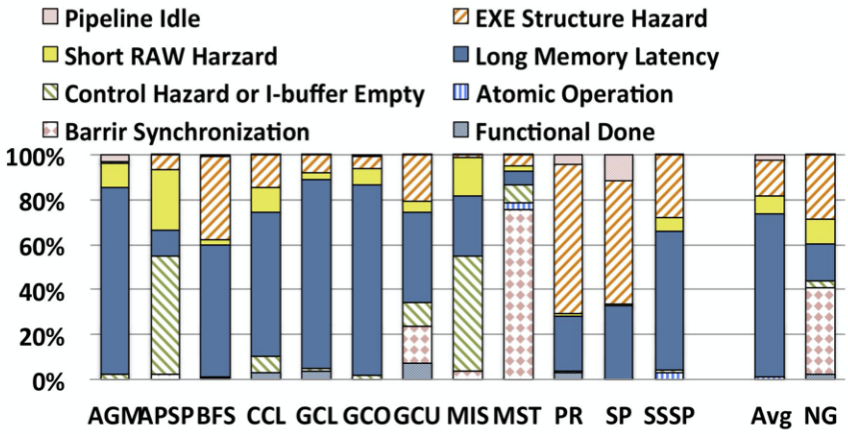

Figure 4.2: On average, long memory latency is the biggest bottleneck that causes over 70% of all pipeline stalls in graph applications [21].

format that will mean that the neighborhood lists of all of the highest connected nodes will be together. There is a very high chance that we will constantly move between them, as these nodes are also likely connected with one another. By sorting the vertices by degrees we aim at putting together the main cluster (collection of highly interconnected nodes). Other proposals try and define more fine-grained clusters to improve the same concept.

Other proposals change the whole processor architecture to a more memory focused approach. They put computation closer to the memory. If the computation takes place inside the memory, it is called **Processing-in-Memory (PIM)** and **Near-Data-Processing (NDP)** if it is close to it.

Near-Data Processing (NDP)

Processors usually follow the Von-Neumann arquitecture model (Figure 4.3). The computation is fully relegated to the cores and the only task of the memory is storing the data. Near-Data-Processing and Processing-In-Memory aims at breaking that separation by putting computation closer or inside it, respectively (Figure 4.4). Some new memories like HBM (High Bandwith Memory) or HCM (Hybrid Memory Cube) have already been designed with a logic layer that can support small computation components.

One of the key properties these systems try to exploit is the the difference in bandwidth. The amount of data that can move from the memory to the NDP logic per time unit is much bigger than to the cores. The main execution is still taken care of by the cores. Therefore, the main goal is pre-process the data in NDP to reduce it and then later send it to the cores to continue their execution.

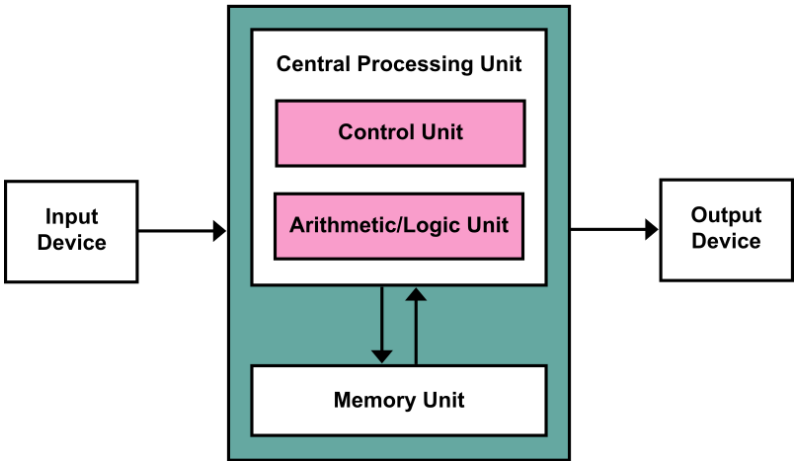

Figure 4.3: A von Neumann architecture scheme.

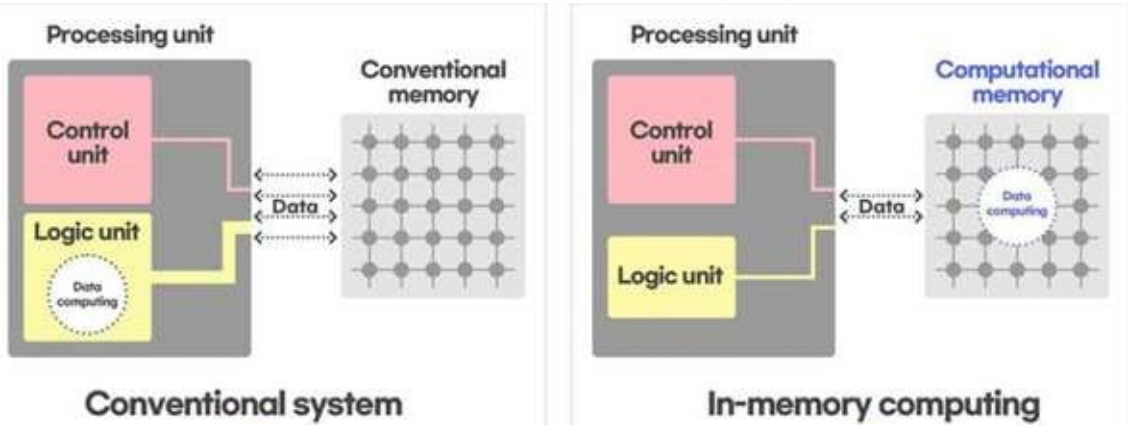

Figure 4.4: Concept of NDP computing architecture.

NDP also comes with some limitations, the most important for us being:

- **Simple logic:** The logic we are able to put so near to the memory has to be simple, as there is not a lot of area. Also, we need to take into account that the memory goes at a different frequency compared to the cores, making the computation slower.
- **Coherence with the caches cannot be broken:** Data can also be copied in the caches along the processor. Caches have protocols to ensure that different copies along the caches are consistent with one another. MESI (Modified-Exclusive-Shared-Invalid) is one of the most spread. It keeps track of how the current state of a cache line and makes the necessary changes when a core issues a request. The same coherence protocol cannot be extended to main memory given that it would be unfeasable to keep track of the state of all the data we have. Therefore, if the NDP logic is allowed to modify the data in main memory, there needs to be a system in place to notify the caches, or the data itself is marked as non-cacheable.

Why NDP for GPM?

We now list the properties of GPM that make them suitable for Near-Data (NDP) architectures:

- **Read only data and a static structure**: The range of GPM problems we are considering work with static graphs, in other words, they do not change their structure as the execution progresses. At the same time, GPM only aims at searching for embeddings, it does not modify the data structure at any point. Therefore, if the data is only going to be read, all copies will be consistent with one another.
- **Early reduction of data:** In GPM, we have two opportunities to reduce the read amount of data and only move to the core what is necessary to continue with the execution.
  - **Symmetry breaking:** Part of the neighborhood lists are discarded while applying the symmetry imposed in the pattern to avoid automorphisms.
  - **Set difference and intersection:** In GPM, for every operation we want to do (set intersection or difference), we have to load both inputs into the cores. However, we only use the result to continue the execution of the program, and it takes a much less space. For these two operations, the result is the subset of both or one of the inputs.

$$X \cap Y = Z, \quad Z \subseteq X, Y \tag{4.1}$$
$$X \setminus Y = Z, \quad Z \subseteq X \tag{4.2}$$

If they are computed in the memory, we can see a high reduction in the data movement between the core and main memory.

- **Simple arithmetic operations:** Symmetry breaking can be implemented with comparator logic. When the data read surpasses a threshold, it is stopped. The intersection of sets can be simply done with adder and comparator logic. The same goes for the difference.

For these reasons, it might be beneficial offloading some of the set operations and symmetry breaking to NDP engines to try and obtain a better performance.

4.2 Previous works

The development of Graph Pattern Mining accelerators always goes in hand with the latest software developments. The algorithm presented in this thesis, commonly referred to as "think-like-an-embedding", was already being used early in frameworks like Arabesque traversing the search tree in BFS [6]. Then, several other proposals started adding more optimizations. G-Miner [22] introduced task-based asynchronous execution, Fractal [18] suggested using DFS to reduce the memory footprint, and aDFS [23] used an almost-DFS exploration that combined the advantages of both DFS and BFS. Using this programming paradigm, Gramer [4] was the first hardware accelerator for graph pattern mining, and it achieved a significant improvement compared to software approaches.

AutoMine [5] was the first to represent graph pattern mining tasks as set intersect and substract operations, and used a custom compiler to create optimized plans for each pattern. Such pattern-aware paradigm was able to match the performance of highly optimized algorithms, with additional optimizations like symmetric breaking and search order scheduling in: GraphPi [24], Pangolin [25], GraphZero [26], and Sandslash [27]. These optimizations exploit locality, reduce memory consumption, and mitigate the overheads of dynamic memory allocation and synchronization [28].

This improvement on the algorithm leads to a new generation of accelerators, including FlexMiner [19] (a standalone accelerator), IntersectX [3] (an ISA extension for general-purpose processors), and SISA [2] (a design for processing in memory). This new methods expose a high level of parallelism. Instances of more recent methods are: Fingers [29], which computes the set operations in parallel with vectorized

units and uses a pseudo-DFS order to expose enough independent tasks to effectively utilize resources; and ABNDP [30], which uses a distributed DRAM cache, where the computation load can be distributed and balanced, without significantly increasing remote accesses [28].

In this section we will focus on two of the last proposed architectures to analyze how to tackle the different improvement opportunities.

4.2.1 FlexMiner

FlexMiner is a hardware accelerator developed by researchers at MIT [19]. It has several DFS walkers that traverse the Search Trees with specialized units for set operation units. Each Processing Element works independently from another, as a Search Tree is always completed by only one Processing Element. By doing so it is able to exploit the opportunities for parallelism as well as optimizing set operations. In Figure 4.5 we can see how a single PE is built.

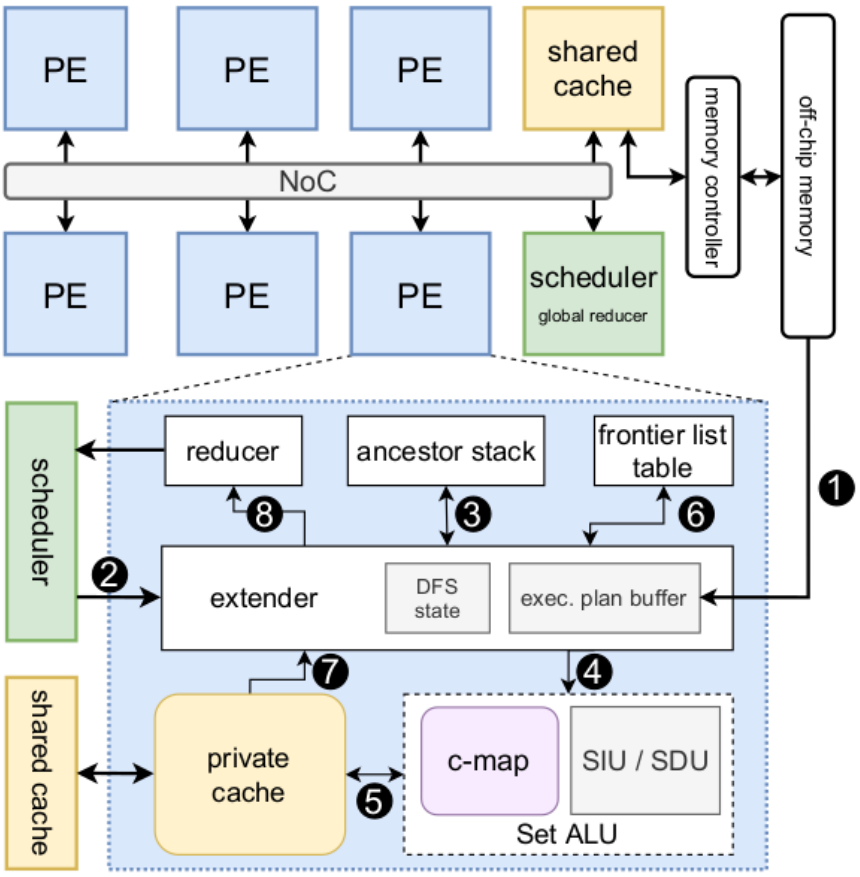

Figure 4.5: FlexMiner architecture [19]

The step required to complete each pattern is loaded by an off-chip memory into the extender. Then, the workload is distributed by the scheduler. It assigns a vertex v of the graph to each PE. Then, the PE starts creating and extending the Search Tree with v as root. The ancestor stack saves the vertices of G that have been mapped to the pattern so far in the branch we are traversing (partial match). The frontier list table saves the information necessary to access the candidate sets that we progressively compute at each extension step. At each extension, we only take one vertex of the candidate set to add to the stack because we are following a DFS model. The rest of the candidate set will be used when we traverse back.

At each extension step, the set operation unit loads the neighborhood lists necessary to compute the candidate set. The generated result set is loaded into the cache and its information is saved into the frontier list table. Then, the extender takes one vertex and loads it into the ancestor stack. The same steps are repeated until the mapping from graph to pattern is completed. Then, we traverse back in a DFS manner to obtain the next embedding. The same process is repeated over and over until the Search Tree is complete.

In the traverse of the search tree, there are a lot of redundant operations due to the associative property of set intersections. In Figure 4.6 there is an example of a pattern in which we will be able to reuse the intersection done to compute the candidate vertices for p_3 in the computation of the candidate vertices for p_4. This type of reuses is easier to exploit if we are traversing the Search Tree in a DFS manner. We need to keep fewer results in memory because we are tackling the Search Tree branch by branch. FlexMiner takes advantage of what DFS walk is doing and adds another element to automate these reuses: the c-map. We will not go into the design. Still, it is worth mentioning it is another optimization on the DFS walk.

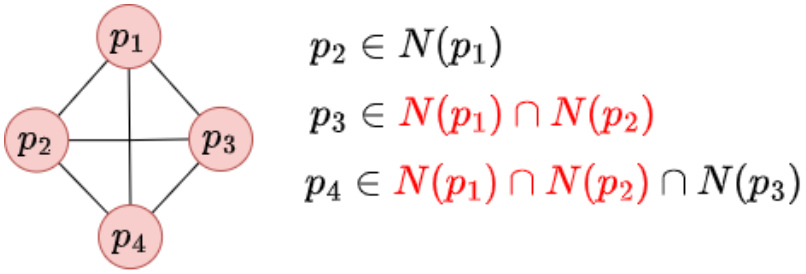

Figure 4.6: In the same branch of the search tree, a lot of operations can be reused if we apply the associative property of set intersection.

By optimizing the two first points (parallelism and specialized computation units) and the DFS walk (and the computation reuses that come with it) in hardware, FlexMiner significantly improved the performance obtained by state-of-the-art software optimization (GraphZero).

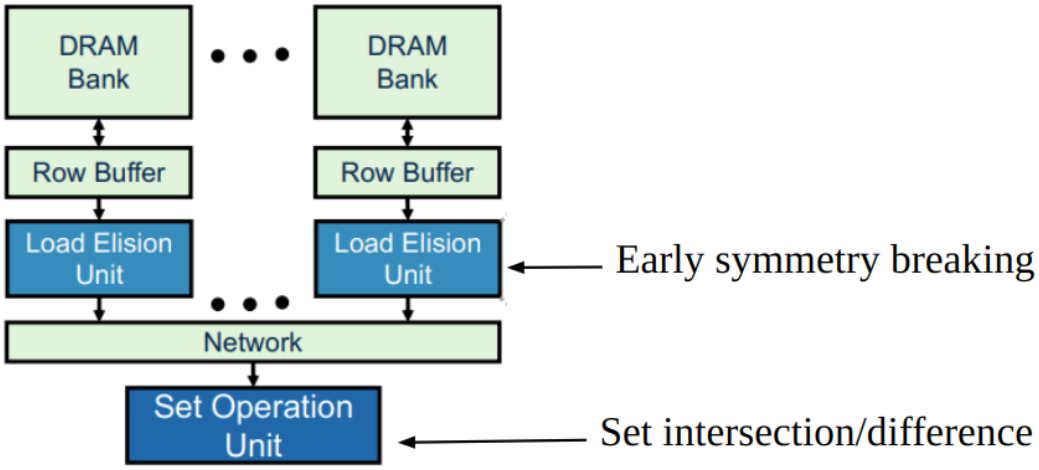

Figure 4.7: Main hardware components of NDMiner [20].

4.2.2 NDMiner

FlexMiner improved GPM performance by mainly focusing on the computation of the workload. NDMiner [31] is a different proposal developed at the University of Michigan that focuses on the memory side. They move the set operation units closer to the memory (outside the network) and apply the symmetry breaking at bank level in the Load Ellision unit. An overview of the components can be seen in Figure 4.7.

To understand why each of these components is in those places we need to look into how sets are being saved in the memory. Each neighborhood list is saved only in one bank. Then, we interleave the neighborhood lists between banks. The neighborhood list of vertex 0 will be in the first bank, the list of vertex 1 in the next bank, and so on.

The Load Ellision Unit works under the premise that neighborhood lists are ordered. This is a common assumption in GPM because it makes it easier to perform the set operations. To apply the symmetry breaking, we put an upper bound to the ids of the vertices of the set. Once that upper bound is surpassed, we cut off the reading of the set. Because the set is all in the same bank this can be done at that level.

The set operation unit needs to be able to access all neighborhood lists. Because they are distributed among all channels, ranks, and banks it needs to be able to access all the memory. Therefore, it needs to be put after the network. This will mean that the main memory bandwidth set operation units see is the same as the cores. We observe that the actual position of these components will highly depend on how the graph is mapped to the memory. If neighborhood lists were distributed among different banks and at each bank we could perform a part of each operation,

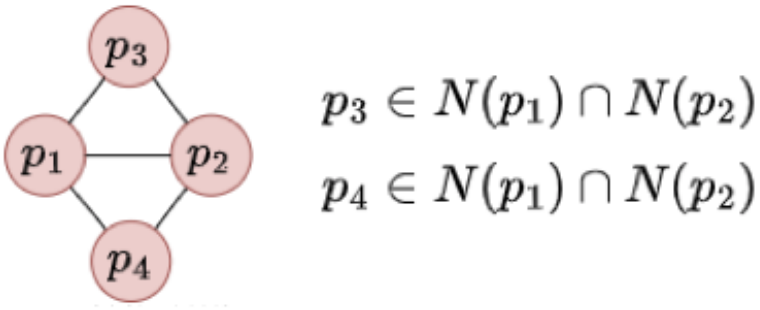

Figure 4.8: Redundant operations for the diamond pattern.

then we could move the set operation unit to bank level. The NDP engine issues requests for the sets it needs to the main memory. To allow ND cores to finish their computation as early as possible, it avoids having concurrent operations that access the same banks. It does so by reordering the requests that arrive at the ND engine.

These hardware solutions are also combined with software optimizations. If two candidate sets are computed with the same operation only one request is sent to the memory and it is attached to both candidate set. This happens for instance, for the diamond pattern as we can see in Figure 4.8. Both p_3 and p_4 have the same connectivity, and therefore their candidates will be computed in the same way in a branch of the Search Tree. For both, only one request for a set operation is sent to the NDP engine.

How much each of these modifications increases NDMiner performance is shown in Figure 4.9.

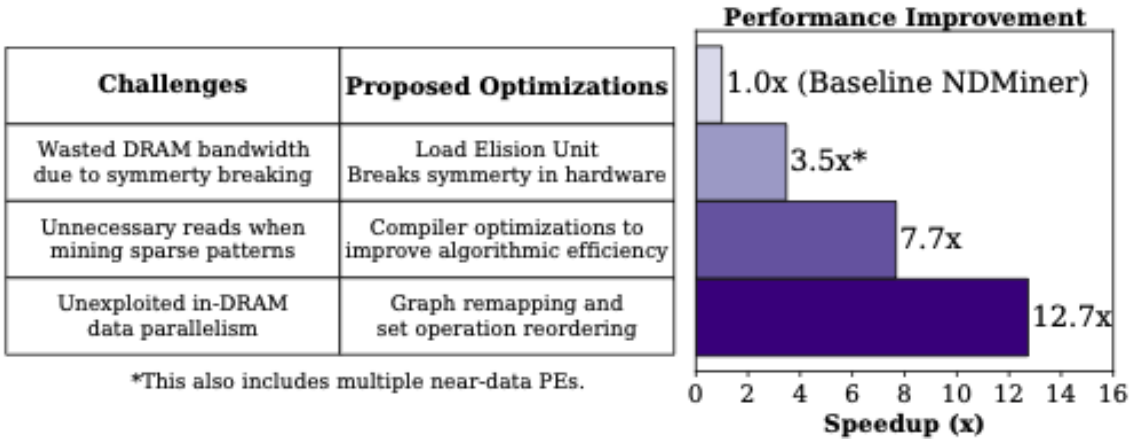

Figure 4.9: The difficulties of accelerating GPM workloads led to NDMiner optimizations and corresponding performance gains. Optimizations build up as the bars descend [31].

There are several questions we raise with this design:

- How are cores leveraged? All of the computation is relegated to the NDP engine in this design. Therefore, if we introduce more cores to leverage the parallelism, do we bottleneck by the NDP engine?
- Do we exploit any of the reuse? Every time we do an operation the request is sent to the memory. Still, there where no mechanisms put in place to reduce redundant operations.
- Are we able to get results faster? The bandwidth the NDP core is experiencing from main memory is the same as the core because it is outside the network. Therefore, it is not taking advantage of the higher bandwidth of the internal memory.

The main goal of this thesis will be implementing the NDMiner design in a complete simulation framework to address these questions.

As a summary of these two accelerators we present Table 4.1 to represent the different points each of them exploits.

Table 4.1: FlexMiner - NDMiner comparisson

	Parallelism	Specialized set operation units	Reduce redundant operations	Specialized memory
FlexMiner	✓	✓	✓	×
NDMiner	✓ [1]	✓	×	✓

[1] Potentially several threads could work in parallel and send requests to the NDP engine

Part II

Implementation and Results

Chapter 5

Implementation

The main goal of this work is to implement the architecture proposed in NDMiner in simulation. NDMiner focuses only on what happens in main memory while leaving the rest of the processor out of the picture. In order to be able to have detailed control over the memory we use a Ramulator [32], a popular main memory simulator that implements different memory models. Then, for the processors and caches we use ZSim[33], a simulator developed in Standford and MIT.

The amount of memories ZSim offers is pretty limited and their configuration is not standarized. Therefore there is value in joining it with Ramulator, as it allows us to try more memory configurations and have a finer control. There have been previous proposals to join them both [34, 35]. Still, the execution was not integrated, as ZSim was used only to generate cache traces that were later fed to Ramulator. If we analysie how ZSim actual works, we loose the accuracy of the thread contention simulation. We make a static and very rough estimation of when main memory will serve our request and continue. Ideally, there should be an event created for the main memory access that would be part of the chain of events.

First, we will look at the internal mechanisms ZSim and Ramulator use for its simulation so we can later integrate them together.

5.1 ZSim

The main contribution ZSim has made is allowing the simulation of multicore systems in a scalable way. In order to fulfill that goal, they divide the simulation in two phases they later parallelise.

- **Bound phase**. The contention between threads is not taken into account.

For each instrumented instruction we record the path it needs to do in order to finish. Essentially, it is a path discovery phase.

- **Weave phase**. We adjust the timing taking into account thread contention. The elements in the processor are divided into domains whose simulation executes in parallel. The only moment threads will have to synchronize is when there is a path that crosses domains.

We periodically change between those two phases only simulating a small number of cycles each time. One of the main observations that allow ZSim to follow this execution model is that path-altering interaction between threads is rare in a small number of cycles. Thus, there is only a very small loss of accuracy.

The instances when a path crosses domain are marked during the bound phase. Therefore, the workload during the weave phase is reduced as it will only need to take care of thread synchronization in those points. Previous frameworks used parallel discrete event simulator (PDES). Events generated during the simulation are distributed among the different host cores. If they are pessimistic then threads will synchronize every time a conflict might happen. If they are optimistic then they will continue with the execution and rollback later if needed. As the amount of cores increases so do the interaction between events, difficulting scaling this system.

When incorporating our own objects in ZSim, we need to make sure we are creating a valid chain of events during the bound phase. There are different core models. `SimpleCore` and `NullCore` do not register the chain of events they generate when executing instructions such as loads or stores. That is because they do not model contention: `SimpleCore` has a fixed latency (it does not consider thread contention) and `NullCore` does not perform simulation of the instructions. Therefore, if we want our chain of events to be registered and simulated we need to use `TimingCore` or `OOOCore`.

When a core issues a request to the memory hierarchy the objects it needs to access will add events to the chain if necessary. This is done with the `TimingRecord` object which is later pushed to the `EventRecorder` of the core. The chain will later be taken by the core and linked to any previuos event if necessary.

To simulate dependencies between events we use "parental relationships". They are not one to one, as an event can have multiple children. This would happen, for instance, if an access to a cache later provokes a write back and another access, both of which might generate new events. Therefore, the writeback and the new access will have the same "parent" event. An event cannot start simulation during the weave phase until all its parents have finished. Only then, the event will be queued on its domain queue to wait for simulation. Inserting the domain crosses is not our responsibility, as it will be done by the `CoreRecorder` when the full chain has returned to the core.

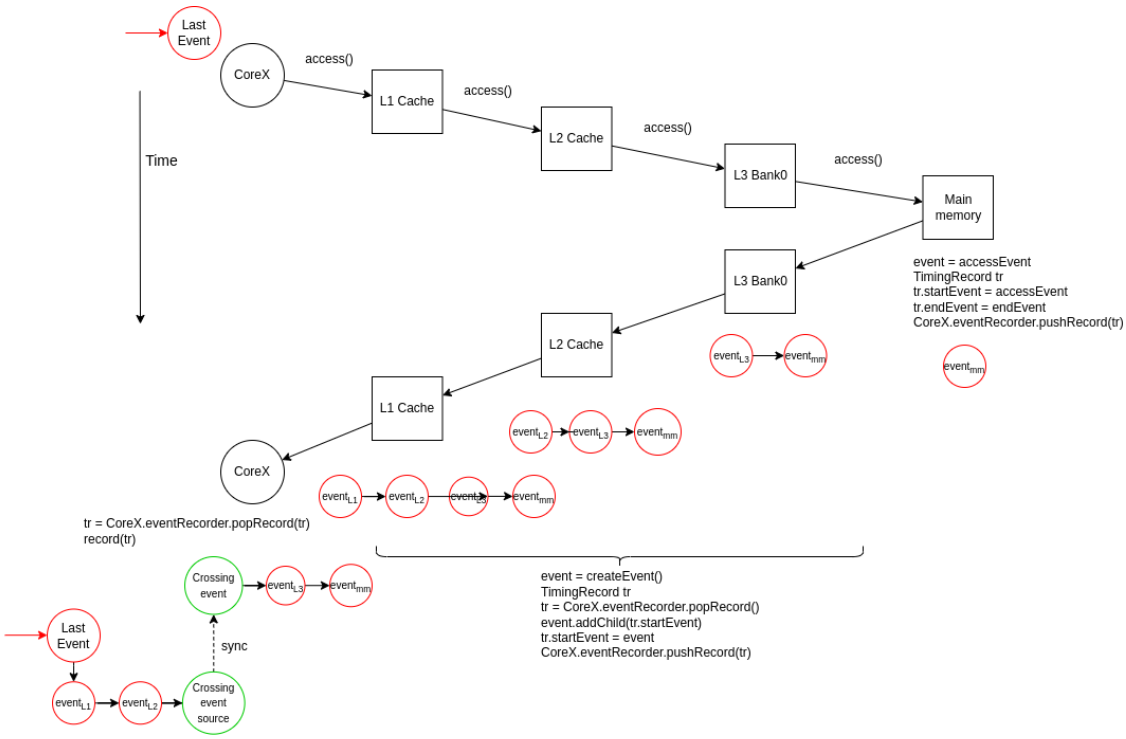

Figure 5.1: Example of how the chain of events is created in a memory access.

In Figure 5.1 is an example of how the chain of events is created and returned to the core for a memory access. Every miss in a cache triggers an access to a lower level memory. When we arrive to the level where data is found we create a `TimingRecord` and push it to the `EventRecorder`. Then we recursively go back and we extend the chain of events as necessary. Finally, the core records the access by linking the chain to any previous event and introducing any `CrossingEvent` if the path chained domain at any point.

Finally all of this events will be simulated during the weave phase to adjust the timing skewes caused by thread contention and main memory access latency.

To do the simulation in parallel, ZSim creates several processes and threads. Therefore, objects used for the simulation have to be saved in shared space, in the global heap. To do so, they can inherit from `GlobalAlloc`, an object already provided by ZSim.

Summarizing, when extending ZSim as programmers our responsability is:

- Creating a coherent chain of events that reflects the dependencies between actions.
- Saving objects in shared memory.

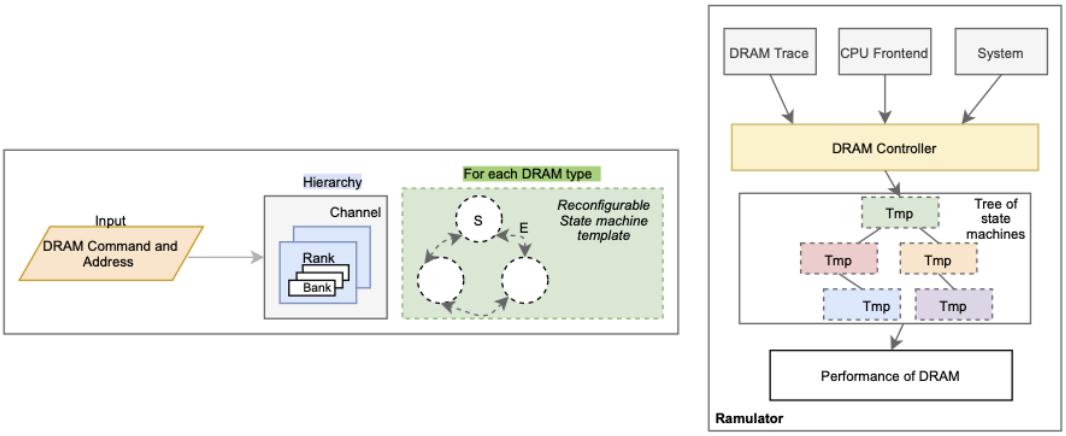

Figure 5.2: High-level overview of Ramulator [36].

5.2 Ramulator

Ramulator is a well known main memory simulator. It implements a variaty of DRAM models and standars such us LPDDR3/4, DDR3/4, GDDR5, HBM, WIO1/2, SALP, TL-DRAM, ALDRAM, RowClone and SARP. A high level design is as shown in the Figure 5.2 which employs a re-configurable tree of state machines for modelling the hierarchy of a DRAM type.

The underline simulation is much simpler than in ZSim, as it is cycle driven. It ticks every cycle and checks the necessary changes that have to made to update the memory.

The way Ramulator is modeled, each of the levels of the main memory hierarchy (for instance, channel, rank, bank group, bank) is represented under the same object: `DRAM`. To be able to issue requests along the hierarchy we only need 3 main lookup functions for all levels.

- `decode()`: Returns the prerequisite command for the request
- `check()`: Returns if the command can be issued
- `update()`: Changes the state of the level according to the command that has just been issued.

Each of this functions is defined as a vector of lambda functions in the spec. If a certain level does not have enough information then the request keeps going down.

All memory controllers in Ramulator are centralized in a `Memory object` that takes care of performing the mapping from address to memory and forwarding the request to the correct controller.

5.3 Integrating ZSim and Ramulator

The approach taken introduces new objects in ZSim that will become the bridge between the two simulators. Essentially, we will have one bridge for every channel or controller as they work independently from one another. Every time there is a request for the memory we create a new Event and push a `TimingRecord`. This event will follow the same flow as shown in the previous example in Figure 5.1. Ramulator is cycle driven, that meaning that it ticks every cycle and tries to schedule something. In order to emulate that behaviour we can also create a new Event that is triggered every memory cycle. When it simulates it ticks the controllers in Ramulator and gets reintroduced to be simulated in the next memory cycle. Because of the difference in frequencies between main memory and the system, we might not be able to tick exactly with the memory cycles. Therefore we tick as close as possible to them.

When the request is sent to Ramulator we attach a callback that will take care of finishing the event so the events attached as children can keep being simulated. Every time a request finishes in Ramulator (sending the response back to the core for read requests and issuing the memory writes for write requests) we will need to execute said callback.

A simple solution to integrate Ramulator while keeping the custom mappings from address to memory it offers, is centralizing all requests in one place, as Ramulator does. By doing so, the last level caches would only have to forward their requests to this centralized point and they would later redistribute. Nevertheless, we would be adding more synchronization points, as the arrival and redistribution of requests will incur in a lot of domain crossing. Thus, we eliminate the centralized memory for all controllers and we keep them independent. For that, we put in a separate object the mapping from address to memory and we link the last level caches to it, so they can know which channel/controller they have to forward their requests to.

In total, three new objects were created to bridge both simulators.

- `RamulatorMemoryManager:` Objects in Ramulator are heavily templated by the specification of the memory. This object just takes care of removing those templates in order to keep the ZSim code as undisturbed as possible.
- `RamulatorMapper:` Implements all of the functionality necessary to use custom mappings from address to memory.
- `RamulatorMemory:` It takes care of the events related to serving requests in main memory (access event, tick event).

We are introducing in ZSim a cycle driven object, which slows down the execution. Still we are able to combine the multicore scalability of ZSim with the flexibility

and portability of Ramulator memory models. This model will allow us to keep trying different configurations and distributions of the NDP logic in the future with different memory models.

5.4 Implementation of NDMiner

In NDMiner they store each neighborhood list in one bank and they interleave the sets by vertex id. Therefore, the NDP engines need to be able to access every bank and hence every channel. That puts the NDP engines on the ZSim side, as we do not have any elements in Ramulator that currently centralizes all memory controllers.

A NDP request is currently formed by the input neighborhoods, a filter and the type of operation. The input sets are marked with the vertex who owns the neighborhood list.

Now, we will go over how the main hardware proposals of NDMiner were implemented:

5.4.1 Reordering the NDP operations

NDMiner proposed implementing a reordering of set operations to leverage bank, rank and channel parallelism. These reorderings were done in windows of 1024 ND requests. To maximize the parallelism the main goal is trying to avoid having concurrent accesses that take sets from the same bank.

Our implementation takes a more dynamic approach. When a NDP request arrives to the PIM engine we queue it. Then, when we are trying to schedule a new NDP request we search for the one that interferes less with the NDP operations that are currently in-flight. In case of a match we use the criteria First Come First Served. In the initial configuration we set the queue size to 32 NDP requests.

Now, in the NDP engine, we would need to have a table that keeps track of how many ND cores are accessing each bank. Later, we have set up Ramulator so it prioritizes those accesses that incur a row hit. Still, there is an upper bound to how many times we can prioritize those accesses that access the same bank. By doing so, we avoid request starvation in the controller.

5.4.2 Load Ellision Unit

The actual data of the application does not exist in the simulation. Therefore, the best solution we found was already encoding in the request how many cache lines

we would have to read before getting to the threshold imposed by the symmetry breaking. We recognize that actually implementing the Load Ellision Unit incurs an extra delay not modeled in this simulation.

5.4.3 Chain of events of a NDRequest

Going back, our responsibility when implementing a ZSim object is creating a coherent chain of events that can later be returned to the core. For that, we need to find which stages a request needs to go through to be fullfilled and what are the dependencies between them. The steps a NDP request does before ending are the following:

- **Enqueue:** Put it in the NDP request queue so we can later decide the most convenient request to schedule.
- **Dequeue and schedule:** The NDP request is chosen for execution. We assign a NDP core.
- **Issue reads:** Start issuing read requests to main memory to read both inputs.
- **Issue writes:** The NDP engine needs to write back the result to main memory (the address had to be previously allocated by the user). For the writes there is no need to wait for a response, as the WAW and RAW dependencies are already taken care of by main memory. Therefore, the end of the chain will be marked before, as no other event will depend on the data being already in main memory.

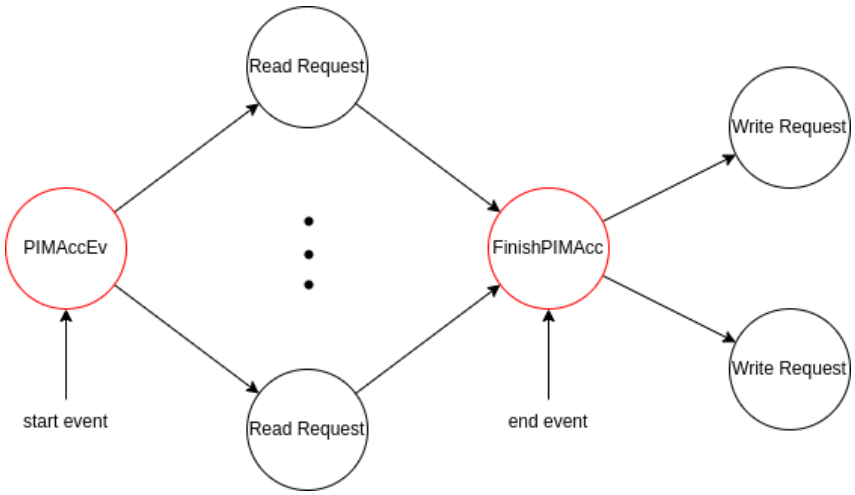

Figure 5.3: Chain of event crated during a NDP access.

This steps get reflected in the chain of events shown in Figure 5.3. NDPAccEv will finish when the request is dequeued. Only then, when we have left the queue and have a NDP core assigned, we will be able to start issuing the reads. FinishNDPAccEv marks the end of the NDP operations. Apart from signaling to the core, that it can continue with its execution, it also frees the NDP engine the request was using and signals that the results can be written back to main memory.

The collection of objects that were implemented to integrate the NDP engine are the following:

- `NDP`: Carries the execution of all the NDP access.
- `CustomAllocator`: Takes care of mapping the sets to main memory as we want, interleaved by vertex id.
- `GraphOperator`: Loads the Graph into ZSim. It is an adaptation of code provided by Berkely as a benchmark for graph applications. We created a library from it and stored the objects shared among processes in the global heap of ZSim.

Chapter 6

NDMiner workload analysis

6.1 Experimental setup

In this chapter we will execute different workloads to answer the questions raised about NDMiner. We will take the same memory configuration as NDMiner did and compare it with a baseline software execution.

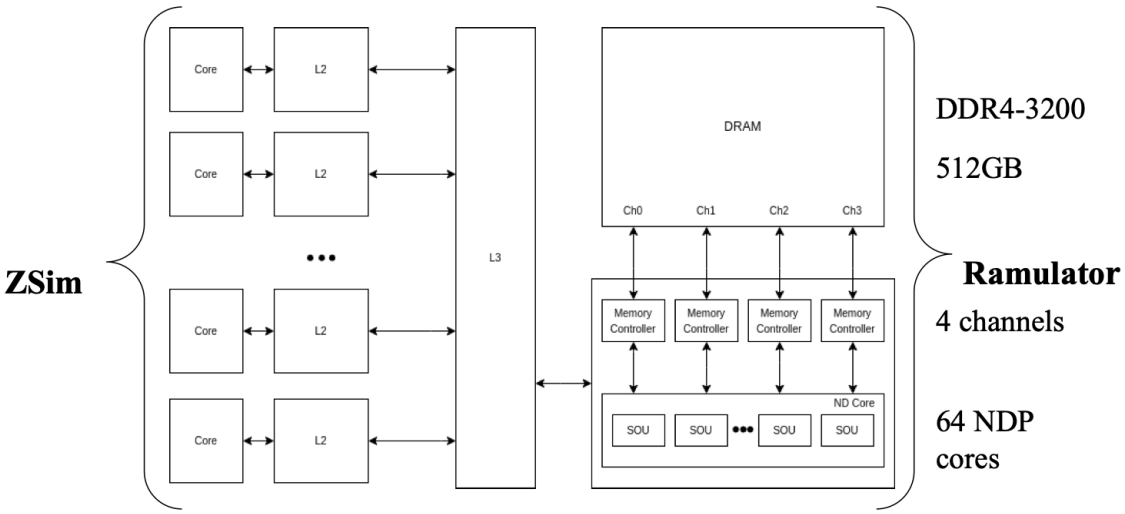

Figure 6.1: NDMiner experimental setup.

For the DRAM memory, the setup is shown in Figure 6.1.

Each L3 bank is 4MB and there is one per core. By doing so, when we grow to 64 cores, we use the same L3 size as the baseline in NDMiner: 256MB.

6.2 Results

NDMiner only used Ramulator and was trace based (only memory accesses, specifically intersections, are taken into account). When using ZSim all core instructions and the interaction between threads will be simulated. Therefore, we should expect to get lower speedups compared to the ones presented in NDMiner.

For every experiment we will divide the different Search Tree among different threads. To reduce context switching between threads, we will create the same number of threads as cores there are in the model and each core will get one thread. For NDMiner, the user has to allocate the space for the result of the set operation. To minimize the effects of dynamic allocation we have created a pool of space for each individual thread. Instead of constantly allocating and deallocating space the thread reuses the same space over and over.

As the baseline for execution without NDMiner cores, we use a simple triangle counting algorithm shown in Algorithm 3. Even though some further optimizations can be applied, we have left the algorithm as it is to show the benefit of doing filtered intersections in the NDP engine instead of in the core.

Algorithm 3: Triangle search

```
   Data: G
   Result: count
1  count ← 0;
2  foreach v0 ∈ V(G) ∈ do
3      foreach v1 ∈ N(v0) s.t v1 < v0 do
4          foreach v2 ∈ N(v0) ∩ N(v1)  s.t  v2 < v1 do
5              ++count
```

Figures 6.2 and 6.3 show the results obtained in the NDMiner simulation for the triangle counting for `wiki` and `GitHub` graph problem. The simulation has been carried out with a different number of cores, and the cycle numbers are compared with/without the implementation of near-data processing, obtaining the speed-up achieved with NDP.

In the figures, we can observe that the improvements NDMiner offer very fastly stall. For `wiki` it stops significantly improving after 8 cores. There is also a big increase in the execution time for 64 cores. The decrease in performance is a strange result and bad work distribution and thread contention are contributing to the results. Still, if we look into how congested the memory is and compare it for 32 and 64 cores we get the histogram in Figure 6.4. The initial part mostly corresponds to the initialization of the program (we have not yet started issuing ND processing

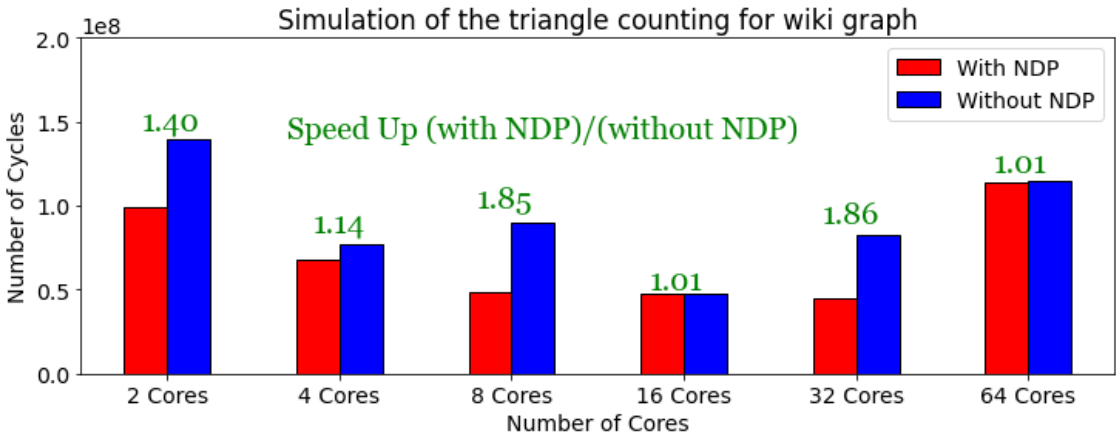

Figure 6.2: Results: Cycle count for triangle counting in the `wiki` graph. At the top of every bar we show the speedup when we compare the execution time of NDMiner with our baseline triangle counting for every number of cores.

requests).

During the main execution of the triangle search, we get the queue in the memory controller is fuller for 64 requests. This make sense as the only thing cores are doing is issuing ND processing requests and waiting for the results. There is not significant computation going on in between. Therefore, all 64 cores will constantly be putting pressure on the ND engine.

Next, lets look at the `GitHub` dataset. It is much bigger, so the work distributes better among the cores and the results we get make more sense. The observation we make is still the same: the application stalls very early, with just 4 cores. In this case, we have measured congestion by counting how many requests get delayed because the controllers are full. The results are shown in Figure 6.5.

Clearly, memory congestion is a key factor in the stalling of the performance improvements. In NDMiner, execution cannot continue in the cores until the set intersection has been completely fullfilled. Moreover, we got a very interesting result: for `wiki`, the ND engine, issued **2.23 times** more requests to the memory than the cores without ND, and for `Github`, **1.34**. The difference between datasets also explains why the speedup NDMiner offers is so different for each of them. Even though we apply early symmetry breaking, NDMiner does not reduce data accesses, but increases them. That can be explained if we look closer to the workload we are executing. Going back to Algorithm 3, in the second loop we go over v0's neighborhood list. This same neighborhood list will later be used for the next inner loop. We have **temporal locality** and it can be exploited by the caches. Moreover this locality is inherent to the pattern we choose for our GPM problem. Because we are working with pattern aware workloads, that is an information we know beforehand.

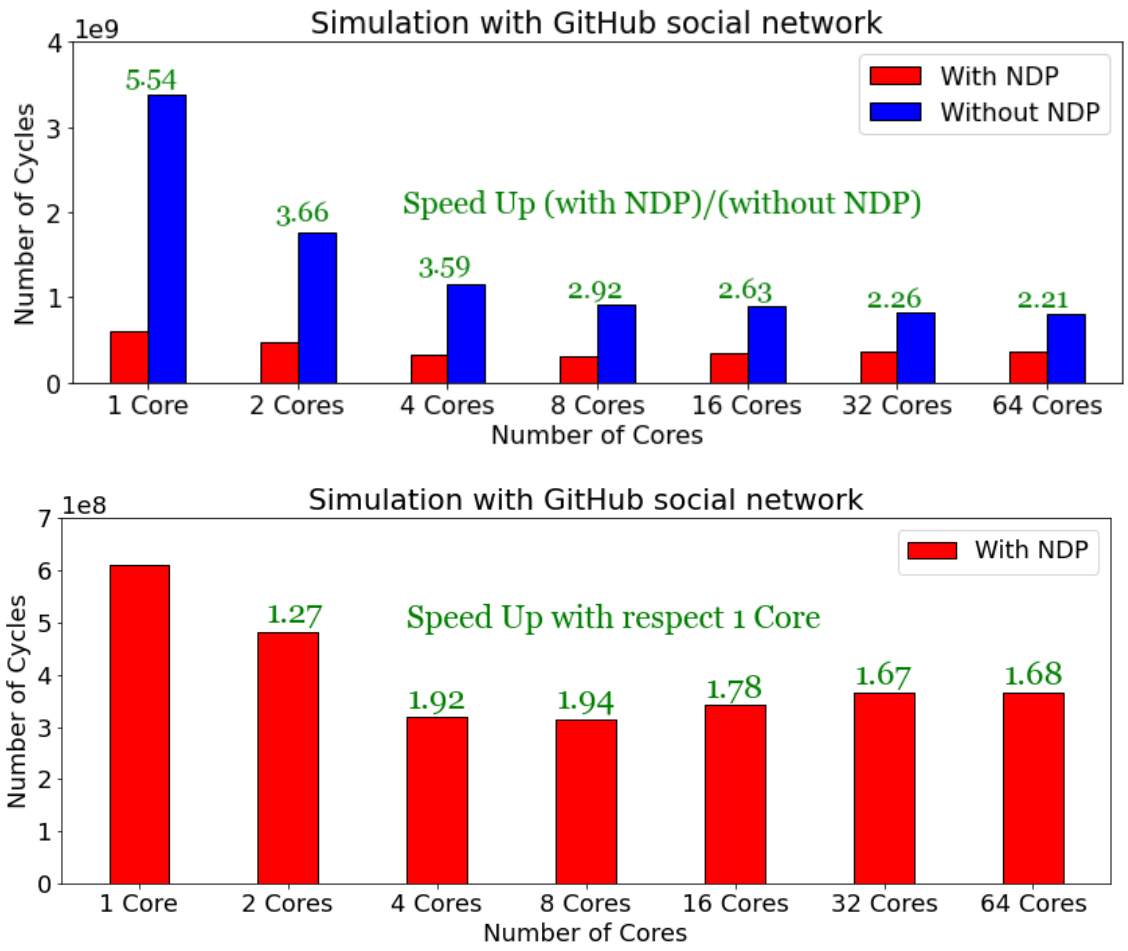

Figure 6.3: Results: Cycle count for triangle counting in the `Github` graph. At the top of every bar we show the speedup when we compare the execution time of NDMiner with our baseline for every number of cores. At the bottom, we show the results only for NDMiner. Over every bar we show the speedup with the 1 core execution as the baseline.

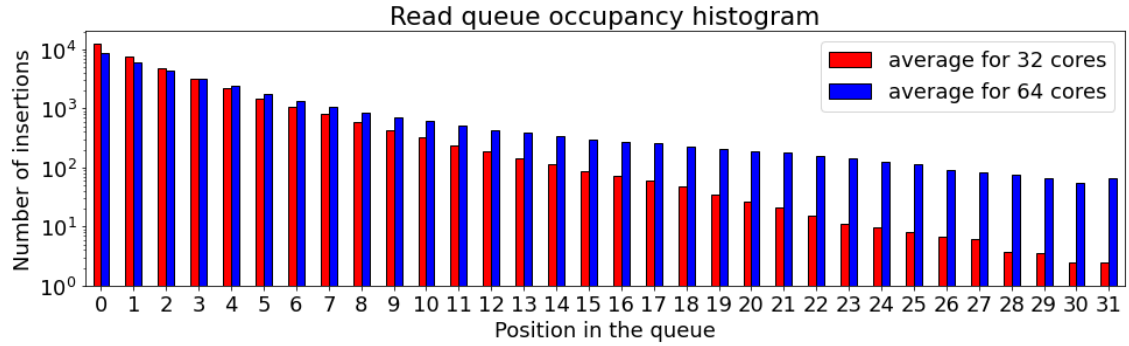

Figure 6.4: Read queue occupancy histogram of triangle counting problem for `wiki` graph

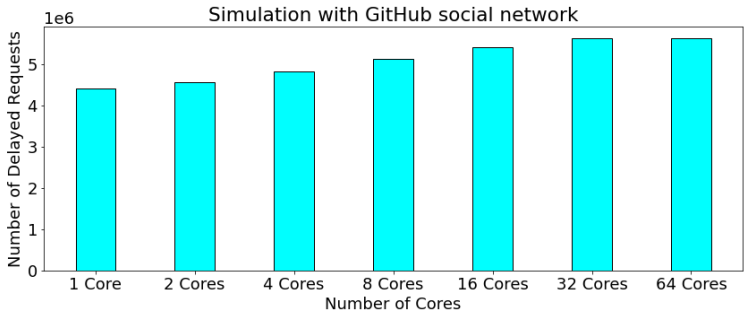

Figure 6.5: Number of the deleayed requests for `GitHub` graph

In summary, we extract the following key observations:

- **Detailed experiments unveils limited benefit of NDMiner:** When simulating the whole system we are not able to replicate the speedups mentioned in the NDMiner paper. This is due to the following reasons:

  1. We not only simulate memory accesses, but the whole program.
  2. Cores stall while they wait for the previous NDP request to be satisfied. Therefore, we also simulate the wasted cycles in between requests.
  3. Cores introduce contention.

- **We are not reducing main memory traffic:** Even though NDMiner implements an specialized memory hierarchy for Graph Pattern Mining, it is not actively reducing memory accesses. Going back, in previous works it had been shown that DFS provided better results for Graph Pattern Mining because it reduced the memory footprint. That was because a lot of results in the same branch of the Search Tree could be reused, as it happens in the triangle search. If all operations are relegated to main memory we cannot leverage thees operation reuses. Main memory is still a bottleneck. The main way we were reducing traffic in the network was by using the Load Ellision Units but they are not as efficient as actually reusing previous results saved in the caches.

- **NDMiner can achieve limited parallelism:** Adding more threads do not improve performance. More threads only translates into more requests for the NDP engine that saturates the main memory network very early on. Therefore we are only putting more pressure over a system that is already saturated.

- **Caches can be beneficial:** The difference in total read requests sent to the memory suggest that caches can be leveraged. Still, we observe a much higher benefit from `wiki`. That shows that the benefit we extract depends on the graph size and how much the sets pollute the caches. As graph sizes increase

the benefits are not as clear, as we also increase cache pollution. Efficiently predicting and exploiting reuse is not an easy task for graph applications. In the next chapter we will go over some techniques to predict which sets will have higher locality so we can efficiently use the limited space available in the cache system.

Chapter 7

Proposed solutions

In the previous chapter we have seen that offloading all workload to memory does not scale well. The performance improvements quickly stall when memory gets saturated. Furthermore we find there are three main benefits obtained by doing some of the computation in the cores:

- **Increased scalability:** By putting some of the workload on the cores we intend to balance out the added pressure it puts on the ND engine. Instead of just sending out requests, it also contributes to the completion of them.
- **Overlapping of the accesses with computation:** In NDMiner intersections are blocking operations. The core sends a request for an intersection and it can not continue until it has finished and the ND engine signals it. If instead we do an intersection in the core, by using a the memory hierarchy and a prefetcher, we can overlap memory accesses with computation.
- **Leveraging the caches:** In the previous chapter we showed that there is locality in GPM applications. For the triangle it is inherent to the pattern. Nevertheless, during this chapter we will try and identify locality more generally.

7.1 Predicting reuse

It seems intuitive that the higher the degree of the node, the higher the chances of it appearing again. Previous works [4] have implemented locality aware accelerators following this intuition. Nodes are given an score and if it is high then it is saved in a high priority memory. We define k as the range of the proximity of the neighbors taken into account. For instance, if $k = 0$ we only take the actual node. If $k = 1$,

the node and its neighbors. If $k = 2$, the node, its neighbors and the neighbors of the neighbors. And so on. They define the priority of a node with:

$$NON_k(v) = \prod_{dist=0}^{k} \sum_{v' \in N(dist,v)} degree(v') \cdot c_k \tag{7.1}$$

where NON stands for *node occurrence number* and c_k decides how much weight the neighbors at each set distance have.

7.1.1 Choosing the right k

The nodes to take into account can increase exponentially with k. There might be a trade-off between accuracy and complexity that has to be taken into account. To test how different $k's$ perform we tried a range of them and compared them using *ROC (receiver operating characteristic curve) curves* to see how well they identify the 10% most used intersections. Here, we are performing a binary classification where we are trying to predict if our set is within this 10% group or not. ROC curves can help us ascertain the quality of our predictor. This curves plot the false positive rates against the true positive rates as the threshold that divides the positives from the negatives changes. In this case, the threshold is compared with the *node occurrence value* (*NON*) to decide a group for the classification of a set. The threshold starts so high that no *NON* surpass it. Therefore, no set is put into the high locality group. That translates into us not correctly identifying any of the high locality sets (0 True Positive Rate) and not us incorrectly identifying a low locality set as high locality (0 False Positive Rate). If we look into Figure 7.2 as an example we see that the curve starts in the $(0, 0)$ coordinate. As the threshold decreases more sets are identified as high locality and thus, both the True and False Positive Rate increase. If the classification method works well, the TPR increases faster than the FPR. Finally, the threshold is so low that all sets fall under the high locality group. Therefore we have a TPR and a FPR of 1 [$(1, 1)$ coordinate in Figure 7.2].

We obtain different curves for different for different ks.

The Area Under the Curve (AUC) is a measure of the quality of our classification: The greater, the better. We can see in Figure (7.2) that simpler models, with low k obtain better results than models with high k. This is a little counter-intuitive because it seems that with high k we work with more information, but given our results it appears that more information also adds more noise.

A true positive in this workload is identifying one of the top 10% intersections as "important". A false positive would be doing so with a less reused intersection. When searching for the pattern in Figure 7.1, we obtained the results show in Figure 7.2 and 7.3 for both *citeseer* and *mico*.

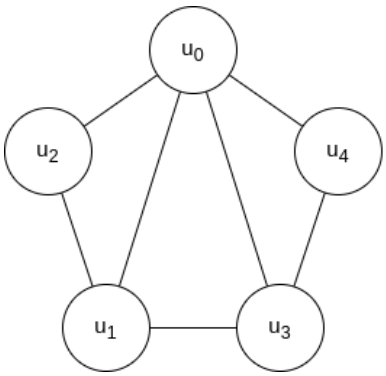

Figure 7.1: Pentagon pattern

Surprisingly, there is no significant difference between $k = 0$ and other approaches. If anything, that is the one that works best. In Table 7.1 we show the AUC (Area Under the Curve) for both datasets. We can see that the value is slightly higher sometimes for $k = 1$, but not enough to justify the increase in complexity. Thus, from now on we will continue with $k = 0$.

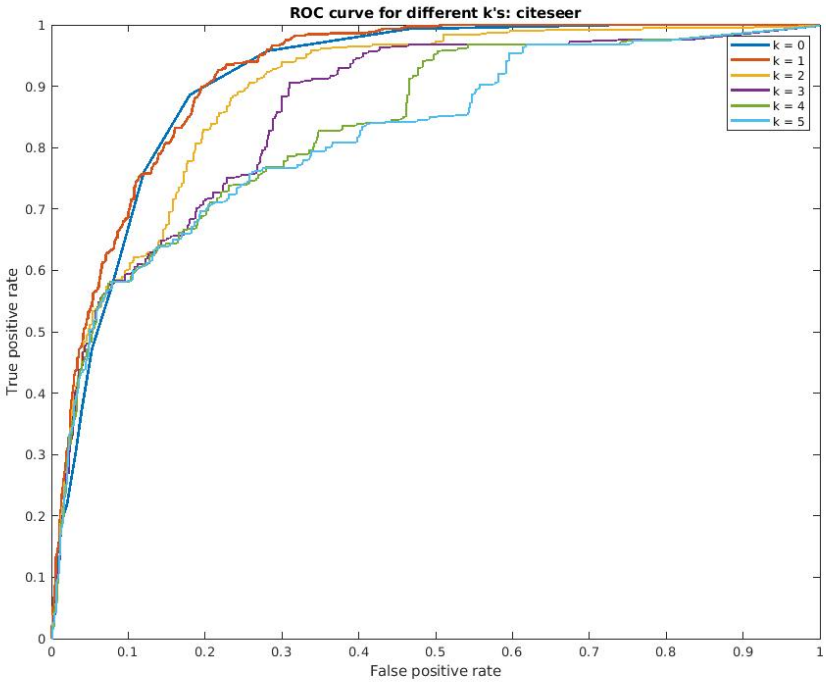

Figure 7.2: *ROC curve* for the citeseer dataset

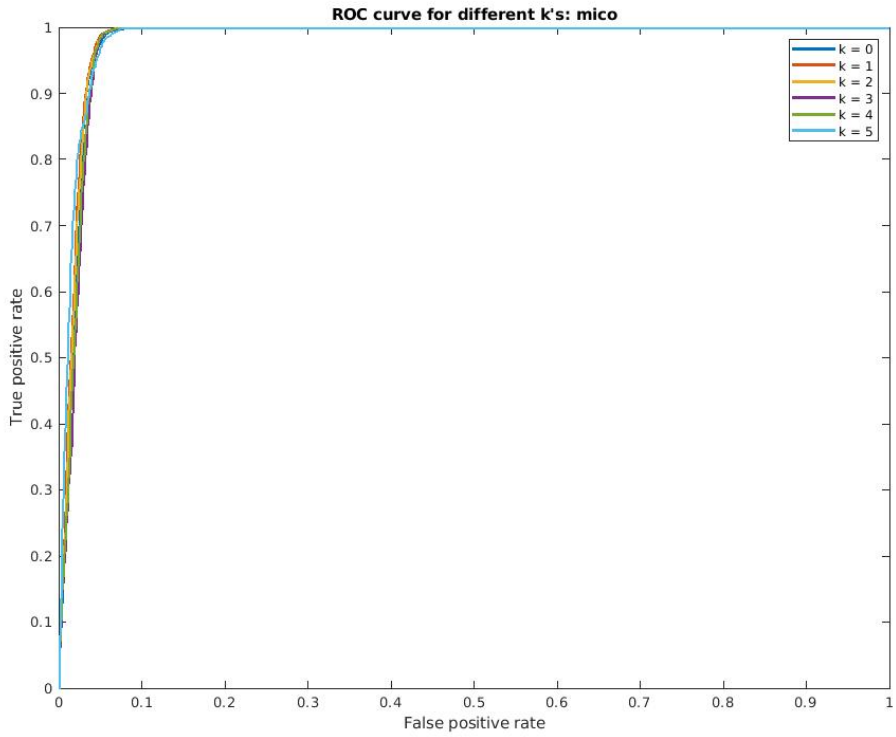

Figure 7.3: *ROC curve* for the mico dataset

Dataset k	0	1	2	3	4	5
Citeseer	0.9110	0.9198	0.8914	0.8618	0.8338	0.8241
Mico	0.9820	0.9839	0.9832	0.9799	0.9811	0.9856

Table 7.1: Area under the curve for the different datasets and k's

7.1.2 Using locality to balance the workload

We want to address the scalability issue for NDMiner. Going back, we saw that the bottleneck is still in main memory. Thus, we need to be careful when deciding which data we move to the cache. The only way we will be able to reduce the number of requests is if we only bring data with locality so it can later be reused.

From the previous results we extract the trade-offs for loading data into the cache shown in Table 7.2. We then obtain the different types of operations by doing the cross-product of the different inputs and we consider the trade-offs of performing them in the ND engine as shown in Table 7.3.

Taking this results into consideration, we propose that the cores dynamically steal work from the ND engine taking into account the potential benefit of doing so.

Table 7.2: Trade-off of bringing data to the cache

Degree	Locality	Space
High	✓	×
Low	×	✓

Table 7.3: Trade-off of doing an operation in the ND-engine

	Reduces cache pollution	Low impact on Main Memory
high × high	✓	×
high × low	∼	∼
low × low	×	✓

First, **Low** × **Low** operations would always stay in the ND engines as it is the data that has lower locality.

Then, if the ND engine starts getting congested, the cores start stealing **High** × **High** operations as they are the ones that present highest locality and the ones that will potentially relieve the pressure on the memory. If there is more congestion and the caches are not highly polluted, then we can also start stealing **High** × **Low** operations.

7.1.3 Overlapping computation and memory accesses to hide latency

We make another observation about the algorithm we employ for GPM. If we look into how an extension works in GPM in Figure 7.4, we typically compute the next candidate set to later iterate over it. To start the second inner loop there is no need for the intersection operation to be complete. If we just have some elements we can continue with the execution while we wait for the others.

```
for (Vertex v0 : G(V))
{
    VertexSet C1 = compute_candidates()
    for (Vertex v1 : C1)
    {
        ...
    }
}
```

No need to wait for full results

Figure 7.4: There is no need to wait for the full result in order to continue with the execution.

By using this observation, we propose that if cache pollution gets high and we find that **High** $\times$ **High** operations are constantly evicting data from one another in the caches we divide the computation between main memory and core. To be able to parallelize both parts, we need them to be separated by range (both parts have to be orthogonal), as shown in the equation below. Neighborhood lists are typically order from lowest to biggest identifiers. From now on we will refer to the section with the lowest ids as the head and the one with the highest as the tail.

$$X \cap Y = (X_1 \cup X_2) \cap (Y_1 \cup Y_2) = \tag{7.2}$$

$$= (X_1 \cap Y_1) \cup \cancel{(X_1 \cap Y_2)} \cup \cancel{(X_2 \cap Y_1)} \cup (X_2 \cap Y_2) \tag{7.3}$$

We now make another observation: high degree nodes are not only more frequent as inputs, but also in other vertex neighborhood lists. Therefore, there is a higher likelihood the vertex will be in the results of the set operations.

We propose pre-processing the graph and relabel the vertex identifiers in such a way that the ones with higher degree get the lowest identifiers. By doing so, high degree nodes will fall in the head of the neighborhood list. By applying the previous observation we conclude that by just doing the set operation that involves the head, we will be getting a good portion of the final results.

Finally, we can use those results obtained with the heads to continue the execution in the core while we wait for the ND engines to do the other part. In this way, we avoid bringing more data to the cache, but we also hide the latency accessing main memory incurs.

Part III

Project Management, Budget, and Sustainability

Chapter 8

Project Management

This chapter describes the planning of the project (defined from an initial planning, whose modifications and adjustments will be discussed), the resources used to carry out the work, and the management of obstacles and difficulties encountered during its development.

The work has been developed during a stay at the MIT Computer Science and Artificial Intelligence Laboratory (MIT-CSAIL), as a visiting student in Arvind's Lab. The dedication to the project has been exclusive, with the application of about 8 hours of work a day to it, totaling about 1000 hours of personal work.

8.1 Task definition

All of the tasks necessary for the project's advancement will be discussed in this section. Each of them will be described, and additional details like duration and potential dependencies will be discussed. The table in Figure 8.1 shows an overview of the tasks.

8.1.1 Project management tasks

For the project to develop properly, a group of tasks related to project management are crucial. This group of duties, among other things, aids in establishing the project's context and scope, planning the various activities that are carried out, and other matters pertaining to sustainability and the economy. [37].

An initial review of the work must be included in this set of tasks, since the stay as a visiting student had originally been planned for the participation in designing a

ID	Name	Time (h)	Dependencies
T1	**Project management**	**90**	
T1.1	Context and scope	24	
T1.2	Project planning	20	T1.1
T1.3	Budget and sustainability	8	T1.2
T1.4	Final document	16	T1.1, T1.2, T1.3
T1.5	Meetings	22	
T2	**Research**	**160**	
T2.1	Research on general topics of Graph Mining	80	
T2.2	Research on GPM acceleration methods	80	T2.1
T3	**Implementation**	**520**	
T3.1	(ZSim+Ramulator) simulator development	400	T2.1, T2.2
T3.2	Implementation of NDMiner in simulation	120	T3.1
T4	**Experimentation**	**120**	
T4.1	Program the experiments	40	T3.2
T4.2	Test de correctness of the implementation	40	T4.1
T4.3	NDMiner workload analysis	40	T4.2
T5	**Final**	**110**	
T5.1	Gather information	15	T4.3
T5.2	Write down the Bachelor's Thesis	80	T5.1
T5.3	Defense preparation	15	T5.2
Total		**1000**	

Figure 8.1: Summary of project tasks.

special parallel processor inspired by PRISC architecture, published by Arvind and Nikhil. However, once at CSAIL, the possibility arose of joining a working group on Graph mining acceleration, led by researcher Xuhao Chen. In order to "maximize immersion" as a visiting student at CSAIL, we decided to face the collaboration challenge that was posed to us, although this would undoubtedly require extending the initial research stage that would be required for the development of the project.

The activities in this group are the following [37]:

- Context and scope (T1.1): Set the project's context and define its scope. Discuss the methodology and defend the choices made. It includes the presentation at the CSAIL and the initial rethinking of the object of the project, and it takes around 24 work hours. It has no dependencies.
- Project planning (T1.2): Organize the various tasks that make up the project's development and schedule them to be completed by the deadline while taking risky scenarios and backup plans to avoid them into consideration. This task lasted 20 work hours. It depends on T1.1.
- Budget and sustainability (T1.3): Determine a potential project budget and evaluate the project viability. This task takes approximately 8 work hours. It depends on T1.2.

- Final document (T1.4): Create the final document by merging the updated documentation from the earlier tasks. This task takes approximately 16 work hours. It depends on three tasks: T1.1, T1.2, and T1.3.

- Meetings (T1.5): Weekly meetings to review the project's status and determine the next move. This task takes approximately 22 hours. It has no dependencies.

The total dedication to this set of tasks has been about 90 hours of work, which represents approximately 9% of the total personal dedication to the project.

8.1.2 Research

Apart from project management, the project can be divided into three main phases, which would be: (a) research; (b) implementation; and (c) experimentation [37].

The research component (T2) has a heavier theoretical load. It takes a lot of time to work on this component because the two other stages rely heavily on it. The research includes search of information about graph pattern mining (GPM), ranging from a more general approach (graph theory, graph mining, etc.) to more specific aspects, such as GPM algorithms, acceleration techniques, simulation packages, etc.

Once sufficient understanding of the topic has been attained, the focus of the research shifts to studying different open-source frameworks for GPM acceleration, both from the system approach and the dedicated hardware, to select the frameworks to be compared.

These tasks can be broken down into:

- Research on general topics of graph theory, graph mining methods, graph sampling algorithms, and graph pattern mining algorithms (T2.1): This task takes about 80 work hours. It has no dependencies.

- Research on graph pattern mining (GPM) acceleration methods (T2.2): Study different GPM acceleration frameworks and try to understand the basis of the applied techniques (algorithm optimization and hardware architecture). This task takes approximately 80 work hours. It depends on T2.1.

The total dedication to these tasks has been about 160 hours of work, which represents approximately 16% of the total personal dedication to the project.

8.1.3 Implementation

The implementation stage (T3) consists in the development of the new simulation tool and the implementation of the NDMiner acceleration framework in this simulation platform. Then the testing stage can start. Thus, the implementation consists of:

- Development of simulator (ZSim+Ramulator) (T3.1): We join two simulators: ZSim for the cores and the caches, and Ramulator to the DRAM memory. This task takes approximately 400 work hours. It depends on T2.1 and T2.2.
- Install the chosen acceleration framework NDMiner (T3.2): It includes access to the documentation and source code, proceeding to its installation and implementation in the simulator framework. This task takes approximately 120 work hours. It depends on the task T3.2.

The total dedication to these tasks has been about 520 hours of work, which represents approximately 52% of the total personal dedication to the project.

8.1.4 Experimentation

The experimentation phase is crucial because the findings will determine how it turns out. The next tasks must be accomplished for carried out this part:

- Program the experiments (T4.1): Choose the datasets for the experiments, and design experiments to analyze the NDMiner workload. This tasks takes approximately 40 hours. It depends on T3.2.
- Test that the implementation is correct (T4.2): The simulator installation must be thoroughly tested. For this, the results published by the authors have been reproduced as verification of the correct installation and operation. This task takes approximately 40 work hours. It depends on T4.1.
- Analyze the NDMiner workload (T4.3): The simulator installation must be thoroughly tested. For this, the results published by the authors have been reproduced as verification of the correct installation and operation. This task takes approximately 40 work hours. It depends on T4.2.

The total dedication to these tasks has been about 120 hours of work, which represents approximately 12% of the total personal dedication to the project.

8.1.5 Memory writing and oral defense

It is time to add this section to the project's memory and finish the writing of it once the experimentation phase is complete and the results have been obtained. To sum up, after the project is complete, the oral defense is the next step, so answers to potential questions from the senior tribunal members should be prepared in advance. The following tasks represent these final phases:

- Gather information (T5.1): Gather the results that the experiments have produced. This task takes approximately 15 work hours. It depends on T4.3.
- Write the Bachelor's Thesis (T5.2): This task takes approximately 80 work hours. It depends on T5.1.
- Defense preparation (T5.3): Prepare a response to any inquiries the tribunal members may have. This task takes approximately 15 hours. It depends on T5.2.

The total dedication to these tasks has been about 110 hours of work, which represents approximately 11% of the total personal dedication to the project.

8.2 Resources

In this project, four groups of resources have been used throughout the different work phases: human, information resources, hardware and software.

8.2.1 Human resources

Three groups of people have been involved in carrying out the work, performing specific functions:

- Researcher: is the responsible for the project. The researcher must take care of: (a) the planning; (b) the implementation of the software, algorithms and acceleration frameworks; (c) perform the experiments, and obtain and analyze the results; and (d) write the final report of the project.
- Supervisor: They are the people of the research group in which the researcher has been integrated, who provide their background and technical guidelines for carrying out: (a) previous theoretical research; and (b) planning and conducting the experiments.

- Tutor or Project Director: In order for the project to succeed, the person filling this role must direct and support the research.

8.2.2 Software resources

The software used for this project include:

- GitHub: To access the open-source codes of the GPM accelerator frameworks used in the project.
- Overleaf : is an online application that allows the writing of LaTeX documents.
- Google Meet: for the meetings with the tutor to discuss the advances and new discoveries.

8.2.3 Hardware resources

A computer is the main tool to develop this work. The personal computer used is an LG Gram Laptop 15Z90N-V-AA78B Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz, 16GB of RAM. Also a CPU with an NVIDIA GeForce MX130 GPU, and a GPU server (CSAIL equipment) have been used to carry out the experiments.

8.2.4 Information resources

In order to obtain sufficient knowledge, scientific papers, books and other documentation have been essential for the development of the project, so they should be considered as information resources. The essential part of said documentation is referenced in the bibliography included in the project report.

8.3 Risk management: overcoming obstacles

Despite the initial planning of the project, it is normal for problems or obstacles to arise during its implementation that may imply the need to modify the initial plan.

The most important obstacles and potential risks in a project such as the one we have carried out, we consider to be the following:

- Bad deadline forecasting: The time required can be underestimated. This is quite common, at least in some of the tasks, so it is convenient to rethink the plan, taking into account the dependencies and the state of the project, which allows adjusting the dedication of the rest of the tasks so as not to compromise the final execution period of the project.

- Problems with the software: For the implementation of the accelerators it is necessary to obtain the open-source code from the GitHub repositories, and it is very likely that installation or execution problems will appear during the validation of the results. In these situations, it is convenient to go to the requests provided in this type of repositories to obtain help, or resort to the advice of the supervisors and the tutor to obtain some type of guide in this regard, in addition to working more hours in order to lessen the effect of these obstacles.

- Inexperience in the research field: It is very common that the area of research practically new for the researcher, the number of hours necessary to achieve a good level of understanding of the concepts can be considerable. This can delay the project, so it may be necessary to increase the dedication, at least during the initial phase of the theoretical investigation, seeking help from supervisors who know the research area best, and, in any case, with planning that don't underestimate the dedication required in the initial documentation phase.

Chapter 9

Budget and Sustainability

All the various facets of the project's budget estimation will be covered in this section. This comprises different elements such as labor costs per task, amortization, generic calculated costs, etc.

The issues corresponding to the sustainability of the project will be considered, taking into account the main three elements (economic, social and environmental) of the term [37].

9.1 Budget

9.1.1 Labor costs per task

Each task is developed by a specific group of individuals. Each of these workers has a set hourly rate. The following is a list of all the various roles that were involved in the project's tasks:

- Project manager: in charge of the project's proper development through planning and supervision. The supervisor, the tutor and the author will all take on this role.
- Researcher: In charge of reviewing technical documentation, Prepare the experiments, examine the findings, and draw conclusions. The author will take on this role.
- Programmer: Responsible for installing the code of simulators and GPM accelerator, and for verifying their proper functioning by reproducing the published results. The author will take on this role.

- Technical writer: In charge of recording the project various stages, including the findings and outcomes. The author will take on this role.

The cost per hour of each role will be multiplied by the amount of time spent on the activity to determine the staff cost per activity. The results of each role will then be added to determine the task's overall cost. Based on the salary scales obtained on platforms such as glassdoor and payscale, we have estimated the following salary costs per hour worked (in which the corresponding social charges are included) for each of the professional roles that we have identified in carrying out the project:

- Project manager = 36€/h
- Researcher = 30€/h
- Programmer = 20€/h
- Technical writer = 20€/h

The total labor costs associated with the completion of the project are: 22,600 €

9.1.2 Generic costs

Amortization The project has lasted 6 months, during which the computation and auxiliary equipment described in the previous chapter has been made available.

To calculate the amortization costs of computation equipment, we assign a useful life of 4 years (48 months) for this equipment, and we consider that the total value of said equipment is 6,000 €.

The amortization cost is obtained as:

$$\text{Amortization} = 6000 \times \left(\frac{6}{48}\right) = 750€$$

Electricity and internet We have estimated monthly costs for electricity consumption and internet connection of 125 € per month, so the total cost for these concepts to carry out the project is 750 €.

Other generic costs We have assigned additional generic costs for different concepts (use of laboratory rooms, heating, etc.) for the execution of the project of 1,000 €.

Total generic costs Adding the different sources of cost, the total generic costs imputed to the realization of the project is 2,500 €.

9.1.3 Extra costs

Contingencies We must onsider adding a second budget to the overall budget to cover any potential unforeseen issues that might occur. In this scenario, 15% of the overall cost will be taken into account.

Thus, if the estimated cost (labor and generic costs) of carrying out the project is:

$$22,600 + 2,500€ = 25,100€,$$

then the contingency costs would be 3,750 €.

Travel expenses Taking into account that the project has been carried out in the MIT CSAIL, we have considered that additional costs must be assigned to the execution of the project, associated with the travel of the visiting student from Barcelona to Boston. The cost imputed for this concept is 1,150 €.

9.1.4 Total cost

Finally, all previously calculated partial costs must be summed to determine the expected overall cost of the project. Then, the total cost of the project is 30,000 €.

9.2 Sustainability

The creation of a sustainability report for an engineering project is now standard procedure in all significant ICT organizations. [38]. The three sections of the project sustainability analysis are as follows: [39]:

- The project put into production (PPP), which takes into account design and planning, development and also the project implementation.
- The useful life or exploitation stage of the project, which starts once it is implemented and ends with its dismantling.
- The potential hazards that could arise during the project development and operation.

Each of these parts is analyzed from three points of view: social, economic and environmental, the three parts of sustainability. Thus, the sustainability report is structured in three sections [38]:

- environmental impact study.
- economic impact study.
- social impact study.

Following the Socratic method [39], the sustainability report is made in the form of questions that are posed and the corresponding answers. In any case, we consider that in a project like the one we have addressed, the sustainability measures that we can apply can be included in what could be called "technical sustainability", understanding that this refers to the specific choices we make that cause the system to provide the desired outcomes. This covers choices made for hardware (memory, architecture,networks), software (code, complexity, algorithms), as well as issues like system latency, testing needs, or scale-up/scale-out requirements. You might vaguely think of these as steps that would be part of what you might think of as "traditional" software engineering.

9.2.1 Environmental dimension

- PPP: Have you calculated the project environmental impact? - Since this project is almost entirely based on software, it is a little challenging to estimate how it will affect the environment. The amount of electricity wasted during some stages, particularly the experimentation phase, is a consideration for the environment.
- PPP: Did you make any plans to lessen its effects, such as by reusing materials? - In this project, the only reused resources are the open-source codes (developed by other researchers) from GitHub repositories.
- Useful Life: How is the issue you want to address currently being resolved (state of the art)? - The project carried out aims to to improve performance of GPM acceleration methods. Advances in GPM acceleration can be expected to contribute to faster and more energy efficient applications (better hardware and software), which will contribute positively towards sustainability.

9.2.2 Economic dimension

- PPP: Think about the project completion costs you have projected - To estimate the cost of the work, only the human resources and the hardware, as

well as other costs as contingency, since the software resources are free and the technical documentation has been obtained from the servers available at MIT CSAIL. Perhaps some additional cost should have been charged for these services. However, in general we consider that the costs of the project have been correctly estimated, although there is some possible deviation or error in any of the concepts.

- Useful Life: How does the problem you want to solve (state of the art) relate to currently resolved economic issues (costs, etc.)? - This research topic is still fairly recent. The Near-Data Processing and load balanced are techniques that improve the efficiency of GPM acceleration methods.

- Useful Life: How will your solution improve financial problems (costs, etc.) in comparison to other options already in place? - The project aims to improve the performance of GPM acceleration methods, and therefore to improve their behavior from the energy and economic point of view.

9.2.3 Social dimension

- PPP: What do you hope to accomplish through this project in terms of personal development? - Through this project, I will become more familiar with the world of research, how it operates, and what this kind of project entails. It is a challenge I can overcome with the abilities I have developed throughout my time in college. I now have a better understanding of what it takes to schedule, manage sustainability concerns, and manage a project because I have worked on real projects and had some responsibility.

- Useful Life: How is the issue currently being solved (state of the art)? - Currently, the way to solve GPM problems is to do an experimentation phase with the input graph considering a GPM acceleration framework. The proposed solution for load balanced and Near-Data Processing contributes to improve the efficiency of these methods.

- Useful Life: Compared to other options currently available, how will your solution enhance the quality of life (social dimension)? - This project can help to improve the performance of GPM acceleration frameworks.

- Useful Life: Is the project actually needed? - This project is part of the research on a real need that is imposed by the computation of GPM problems. Since GPM applications in very large graphs have spread, and it is of great importance in multiple sectors, it is crucial to make GPM accelerators more effective.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

In this thesis, we have studied Graph Pattern Mining and how it can be accelerated through hardware. During the development, we have arrived to the following conclusions:

- Set operations and memory accesses bottleneck GPM. This bottlenecks have been addressed by different hardware accelerators: FlexMiner and NDMiner, respectively. Still, there has not been an scalable proposal that addresses both at the same time.
- Simulations of Near Data Computing also need to take into account the whole processor. When we considered, not only the memory, but also the processors and the caches, we got different results than NDMiner.
- There is locality in GPM that can be exploited to reduce the pressure on the main memory.

10.2 Future work

The final part of this thesis was not implemented in simulation. Given the amount of time invested in joining ZSim and Ramulator and later implementing NDMiner in this new framework, we were not able to test these ideas. Therefore, further investigation is required to determine if they could actually improve the balancing of the workload and the scalability of the solution proposed by NDMiner.

First, there should be some software testing to have an intuition of the impact this changes can have. Then, a more specific architecture needs to be defined and lastly it has to be implemented and tested.

During the whole development of NDMiner we have assumed the whole graph fitted in one DIMM chip, and thus, there was no communication needed between different ND engines in different DIMMS or between main memory and disk. Graph nowadays have billion of nodes, so this assumption cannot be applied in a big part of the workloads. This unrealistic assumption is common among hardware proposals that accelerate graph applications. If we are trying to improve scalability this is an issue that also has to be addressed.

Bibliography

[1] Albert-László Barabási and Márton Pósfai. *Network Science*. Cambridge University Press, 2016.

[2] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 282–297, 2021.

[3] Gengyu Rao, Jingji Chen, and Xuehai Qian. IntersectX: An Accelerator for Graph Mining. *arXiv*, 2012.10848, 2020.

[4] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 895–907, 2020.

[5] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *27th ACM Symposium on Operating Systems Principles (SOSP)*, page 509–523, 2019.

[6] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *25th Symposium on Operating Systems Principles (SOSP)*, page 425–440, 2015.

[7] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 763–782, 2018.

[8] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions*

on Computer-Aided Design of Integrated Circuits and Systems, 38(4):640–653, 2019.

[9] Chuangyi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xinyu Chen, Xiaofei Liao, and Hai Jin. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *Journal of Computer Science and Technology*, pages 339–371, 2019.

[10] Kayhan Erciyes. *Guide to Graph Algorithms. Sequential, Parallel and Distributed*. Springer International Publishing AG, 2018.

[11] Terence Kelly. Programming workbench: Compressed sparse row format for representing graphs. *;login: Usenix Magazine*, 45(4), 2020.

[12] Brahim Betkaoui. *Reconfigurable computing for large-scale graph traversal algorithms*. Dissertation, Department of Computing, Imperial College London, 2014.

[13] Deepayan Chakrabarti and Christos Faloutsos. *Graph Mining. Laws Tools, and Case Studies*. Morgan and Claypool Publishers, 2012.

[14] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[15] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *International Conference Research in Computational Molecular Biology (RECOMB)*, pages 92–106. 2007.

[16] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. A Survey on Subgraph Counting: Concepts, Algorithms, and Applications to Network Motifs and Graphlets. *ACM Computing Surveys*, 54:1–36, mar 2021.

[17] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 03 2007.

[18] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *International Conference on Management of Data (SIGMOD)*, page 1357–1374, 2019.

[19] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 581–594, 2021.

[20] Nishil Talati. *Optimizing Emerging Graph Applications Using Hardware-Software Co-Design*. Dissertation, Department of Computer Science and Engineering, University of Michigan, 2022.

[21] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on GPUs: Where are the bottlenecks? In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149, oct 2014.

[22] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Thirteenth EuroSys Conference*, 2018.

[23] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *USENIX Annual Technical Conference (ATC)*, pages 209–224, 2021.

[24] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[25] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. 13(8):1190–1205, may 2020.

[26] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Operating Systems Review*, 55(1):21–37, jun 2021.

[27] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A two-level framework for efficient graph pattern mining. In *ACM International Conference on Supercomputing (ICS)*, page 378–391, 2021.

[28] Mingyu Gao. Algorithm evolution and hardware acceleration of Graph Mining. *ACM SIGARCH*, (https://www.sigarch.org/algorithm-evolution-and-hardware-acceleration-of-graph-mining/), 2021.

[29] Qihang Chen, Boyu Tian, and Mingyu Gao. Fingers: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 43–55, 2022.

[30] Boyu Tian, Qihang Chen, and Mingyu Gao. ABNDP: Co-optimizing data access and load balance in near-data processing. In *28th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), volume 3, page 3–17, 2023.

[31] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. NDMiner: Accelerating Graph Pattern Mining using Near-Data Processing. In *49th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, page 146–159, 2022.

[32] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15, 03 2015.

[33] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *40th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, page 475–486, 2013.

[34] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks. *arXiv*, 2105.03725, 2023.

[35] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NAPEL: Near-memory computing application performance prediction via ensemble learning. In *56th Annual Design Automation Conference (DAC)*, pages 1–6, 06 2019.

[36] Madhurya Kumaraswamy. Near-Memory Computing: Application Profiling and Offloading. Master's Thesis, Department of Mathematics and Computer Science, Parallel Architecture Research Eindhoven (PARsE), Eindhoven University of Technology (TU/e), 2021.

[37] Carlos Gascón Domínguez. Comparative Evaluation of Software Acceleration Techniques for Graph Neural Networks Computing. Bachelor's Thesis, Facultat d'Informàtica de Barcelona (FIB), Universitat Politècnica de Catalunya, 2021.

[38] Juan Climent, Jose Cabré, Fermín Sánchez-Carracedo, Carme Martín, Eva Vidal, and David Lopez. El informe de sostenibilidad del trabajo de fin de grado del área de las ingenierías. *Revista de Docencia Universitaria (REDU)*, 16:75, 12 2018.

[39] Fermín Sánchez-Carracedo, Jose Cabré, Jordi Garcia Almiñana, Eva Vidal, David Lopez, Marc Forment, and Carme Martín. Guía del estudiante para elaborar el informe de sostenibilidad del TFG. In *XXII Jornadas sobre la Enseñanza Universitaria de la Informática (JENUI)*, 07 2016.