NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DAVID MACHADO DAS NEVES

BSc in Computer Science and Engineering

GPU ACCELERATED DYNAMIC GRAPH PROCESSING

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

April, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

GPU ACCELERATED DYNAMIC GRAPH PROCESSING

DAVID MACHADO DAS NEVES

BSc in Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA FCT, NOVA University Lisbon

Examination Committee

Chair: Carla Ferreira
Full Professor, NOVA FCT, NOVA University Lisbon

Rapporteur: Pedro Ribeiro
Assistant Professor, FCUP

Adviser: Hervé Paulino
Associate Professor, NOVA FCT, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2024

GPU Accelerated Dynamic Graph Processing

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [26].

To the loving memory of my grandfather Jacob Van Eck, for having seeded my interest in science from a young age, and for always inspiring and motivating me in life and in all my academic endeavors.

Acknowledgements

I would like to express my gratitude to everyone who has accompanied and supported me throughout these last academic years.

To start, I would like to thank my thesis advisor Hervé Paulino for all the support and mentorship he has given me during the development of this thesis, but also, for all the interesting research opportunities he has provided me in the last three years.

To the NOVA School of Science and Technology, and in particular the Department of Computer Science, thank you for providing me, and all my colleagues, with an education of excellence, but also for fostering such a great community.

I would also like to thank all of my friends who have made these last five and a half years so enjoyable and memorable, including, but not limited to: Rodrigo Mesquita, Ricardo Valverde, Gonçalo Condeço, Ricardo Monteiro, Rita Costa, Guilherme Gil, Francisco Simões, Joana Paiva, Tomás Santos, André Costa, João Palma, André Matos, Ruben Belo, Guilherme Martins, and João Pio.

Finally, I would like to deeply thank my parents, my brother, my dear Joana, and the rest of my family for all their continuous love and support.

Abstract

Graphs are ubiquitous in today's large-scale computing systems. With the rapid advance of technology and applications, the size of these graphs, and the complexity of the computations performed on them is constantly increasing. This has led to a need for the development and research of parallel graph processing, in order to overcome the computational limits of single-processor-based solutions. At the forefront of this research, is Graphics Processing Unit (GPU)-accelerated graph processing, as the GPU's massive parallelization capabilities allow for extremely efficient execution of algorithms on large graphs.

In the last years, various GPU-accelerated graph processing frameworks have emerged and started being utilized in the industry. However, practically all of these frameworks only support static graphs, meaning that whenever the graph is updated, it must be entirely retransmitted to the GPU for further processing. In a landscape where many large graphs are constantly evolving, like for example social and financial networks, this becomes an issue. Some dynamic GPU-accelerated graph processing frameworks have already emerged, however, they are few and their usability is still limited, making this an open and relevant area of research.

In this thesis, we present Marrow-Graph, a fast GPU-accelerated dynamic graph processing library, built using the Marrow framework. The library supports efficient graph updates, provides a simple yet expressive programming model, and includes multiple commonly used graph processing algorithms. To achieve this, the graph is stored both on the host and device using Marrow collections, allowing one to execute algorithms efficiently on the device while performing lazy graph updates on the host. Additionally, we demonstrate how our solution offers better usability and conciseness compared to competitors, and exhibits promising performance. Marrow-Graph outperforms FaimGraph and Hornet in some algorithms, achieves speedups up to x260 compared to state-of-the-art Central Processing Unit (CPU)-based solutions, and provides faster real-time edge insertions using small to medium update batches.

Keywords: Dynamic Graph, Graph Analytics, GPU, Marrow, CUDA

Resumo

Actualmente, grafos podem ser encontrados numa grande parte dos sistemas informáticos de grande escala. Com o rápido avanço da tecnologia e das aplicações, o tamanho destes grafos e a complexidade das computações efectuados sobre estes, estão a aumentar constantemente. Isto tem levado à necessidade de desenvolver e investigar o processamento paralelo de grafos, de modo a ultrapassar os limites computacionais das soluções baseadas em processadores únicos. Na frente desta investigação está o processamento de grafos acelerado por GPU, uma vez que as capacidades de paralelização massiva da GPU permitem execuções extremamente eficiente de algoritmos em grafos de grandes dimensões.

Nos últimos anos, várias *frameworks* de processamento de grafos acelerados por GPU têm surgido e começado a ser utilizadas na indústria, no entanto, a maioria suportam apenas grafos estáticos, o que significa que sempre que o grafo é atualizado, este é retransmitido por completo para a GPU para processamento posterior. No cenário atual, em que muitos grafos de grandes dimensões estão em constante evolução, como por exemplo as redes sociais e financeiras, isto torna-se um problema. Já surgiram algumas *frameworks* de processamento de grafos dinâmicos acelerados por GPU, no entanto são poucas e a sua usabilidade ainda é limitada, tornando esta uma área de investigação aberta e relevante.

Nesta dissertação apresentamos o Marrow-Graph, uma biblioteca de processamento de grafos dinâmicos acelerada por GPU, desenvolvida com a *framework* Marrow. A biblioteca suporta actualizações eficientes sobre grafos, providencia um modelo de programação simples mas expressivo, e inclui multiplos algoritmos comuns de processamento de grafos. Para tal, o grafo é armazenado tanto no *host* como no *device* utilizando as coleções do Marrow, o que permite executar algoritmos de forma eficiente no *device*, e efetuar atualizações *lazy* sobre o grafo no *host*. Além disso, demonstramos como a nossa solução oferece melhor usabilidade e concisão em comparação com os concorrentes, e apresenta um desempenho promissor. O Marrow-Graph supera o FaimGraph e o Hornet em alguns algoritmos, atinge *speedups* até x260 em comparação com as soluções do estado da arte baseadas em CPU, e permite inserções em tempo real de *edges* mais rápidas usando *update*

batches pequenos a médios.

Palavras-chave: Dynamic Graph, Graph Analytics, GPU, Marrow, CUDA

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

AST Abstract Syntax Tree *(pp. xi, 14, 15)*

BAL Blocked Adjacency List *(pp. xiv, 4, 41, 49, 50, 52, 53)*
BFS Breadth-First Search *(pp. 25, 30, 56, 62, 67, 68, 73, 75, 79)*

CPU Central Processing Unit *(pp. v, vi, 2, 3, 5, 23, 24, 29, 30, 75, 76, 79, 80)*
CSR Compact Sparse Rows *(pp. xi, 2, 3, 17–19, 23–27, 77, 78)*
CUDA Compute Unified Device Architecture *(pp. xiv, 3–6, 8, 10–12, 14, 31, 32)*

DRAM Dynamic Random Access Memory *(p. 6)*

GAS Gather Apply Scatter *(pp. 23, 25)*
GPC Graphics Processing Cluster *(p. 6)*
GPGPU General Purpose computing on Graphics Processing Units *(pp. 5, 7, 8)*
GPU Graphics Processing Unit *(pp. v, vi, xii, 1–7, 11, 16, 19, 21–23, 25–33, 42, 63, 66, 69, 71, 73, 75–81)*

PCSR Packed Compressed Sparse Row *(p. 18)*
PMA Packed Memory Array *(p. 18)*
PR Page Rank *(pp. 25, 56, 62, 65, 67, 68, 75, 80)*

SM Stream Multiprocessor *(pp. xi, 6, 7, 9, 10)*
SOA Structure of Arrays *(p. 26)*
SpMV Sparse Matrix–Vector Multiplication *(pp. 56, 63, 67, 68, 73, 76, 79, 80)*
SSSP Single Source Shortest Path *(pp. 25, 30, 56, 62, 65–68, 70, 71, 73, 75)*
STINGER Spatio-Temporal Interaction Networks and Graphs Extensible Representation *(p. 27)*

TC Triangle Counting *(pp. 56, 58, 62, 67, 68, 75, 79)*

TPC Texture/Thread Processing Cluster *(p. 6)*

VRAM Video Random Access Memory *(pp. 66, 69, 77–79)*

WCWS Warp Cooperative Work Sharing *(p. 28)*

1 Introduction

1.1 Motivation

Graphs are one of the most common and useful data structures in math, science, and engineering. Many problems can fundamentally be described as relationships between nodes with a set of associated properties, i.e. a graph. Once a problem is represented as a graph, it enables one to perform analysis and computations over it using algorithms and functions that have already been studied and optimized for decades[1] [43]. For these reasons, and given the ease of storing and processing large graphs using computers, graphs are being used in today's large-scale computing systems for a large variety of purposes [38, 42, 43]. Some of these include: 1. Keeping track of relationships between users in a social media platform. 2. Optimizing packet routing in a computer network represented as a graph. 3. Enabling internet searches using graphs to build indexes of web pages. 4. Implementing recommendation systems by representing users, items, and their interactions as a graph. 5. Studying and understanding complex biological systems with graphs that can represent, for example, protein interactions.

Given the rapid evolution of technology, and more specifically the rise of *big data*, the size of graphs and the complexity and scope of the computations being performed on graphs is always increasing [20]. This has resulted in a large interest in the research of parallel graph processing to overcome the limitations of single processor-based machines' computational resources [8]. The two main approaches that have emerged are distributed graph processing, involving graph partitioning and processing over multiple machines, and accelerated graph processing, which utilizes accelerators like the GPU to process graphs faster. While the former can be more capable when dealing with the largest graphs present today, the latter has the potential to allow for more efficient and faster processing for all but the largest graphs. This is so, given the lack of communication overhead associated with distributed memory systems, and the ability to massively parallelize algorithms. Research in this field is becoming ever more relevant and crucial to ensure

[1] Centuries when considering the work done in graph theory since 1735 outside the realm of computer science.

that systems dependent on large graphs can continue to scale sustainably.

1.2 Problem

A lot of graphs being processed today, are not only large but are also constantly evolving. Two illustrative examples are social networks and financial networks. [19]. Nodes and edges in such graphs are constantly being created and updated. This has led to the need and development of dynamic graph processing systems. Most of these systems are aimed at either single-processor shared-memory machines, or distributed environments, given that efficient dynamic data structures are already commonly used in such systems.

As we have discussed before, accelerated graph processing, using GPUs for example, has the potential to yield the best-performing graph processing algorithms. For this reason, a number of GPU graph processing frameworks have been developed in the last two decades [42, 6, 14]. These frameworks have proven to be very efficient and essential for many applications. The only caveat is that most GPU based graph processing frameworks today are designed for static graphs. This means that whenever a graph is updated, it must be completely re-transmitted to the GPU for further processing, making such frameworks not well suited for frequently mutating graphs.

A few GPU-based dynamic-graph processing frameworks have already emerged, such as Hornet [6] and FaimGraph [46]. However, the number of frameworks that support dynamic graphs is still quite limited, and the existing solutions do not seem to be in active development. This is mainly due to the limitations associated with GPUs. GPUs are designed to operate efficiently over regular data structures that store data in a compact and contiguous layout. Graphs can be stored using compact and contiguous data structures such as CSR, however, such data structures typically do not allow for efficient updates, often requiring reconstructing the whole graph. Another issue is managing communication with the GPU efficiently, given that communication between the host and device requires expensive operations. These limitations have proven to be challenging (but not impossible) to overcome when developing dynamic-graph processing frameworks accelerated by GPUs.

Given that GPUs are unmatched in terms of efficient parallel computing capabilities, researching and developing frameworks that efficiently support dynamic graphs on GPUs is relevant, and possibly crucial, for the future of graph analytics and graph processing.

1.3 Solution Proposal

In this thesis, we present Marrow-Graph, a fast dynamic graph processing library designed for CPU-GPU systems. Marrow-graph supports operations both on the CPU and GPU. A graph is stored both on the host and device and consistency between them is automatically ensured. Updates to the graph can be performed efficiently on the host (and lazily

synchronized [2]), while heavy algorithms can be executed and accelerated on the device. Contrary to static graph processing frameworks, updates to the graph do not require re-transmitting the whole graph to the device, but rather only require updating the segments that have changed. Additionally, the library includes multiple widely used graph processing algorithms but also allows the user to develop their own algorithms based on a set of simple primitives: `advance`, `compute`, `filter` and `segmented intersection` (inspired by Gunrock [42]).

In order to conceive this library, we make use of the Marrow [30] framework, an algorithmic skeleton-based parallel programming framework, focused on GPU acceleration. This framework allows one to easily create data structures, that are stored both on the host and device, and to perform operations either on the CPU or GPU, without worrying about data consistency since it is guaranteed by Marrow via lazy synchronization. Marrow offers a set of powerful high-level primitives to create and manipulate these data structures and compiles them down automatically to low-level operations for a specified back-end like CUDA. We take advantage of these high-level primitives and containers as a basis for our implementation.

To store the graph, Marrow-Graph utilizes a blocked adjacency list data structure, shared between the host and device, inspired by FaimGraph. Such a data structure allows for efficient updates, contrary to static graph data structures such as CSR, while also achieving good performance in parallel graph analytics. Sharing the graph between the host and device also offers additional benefits beyond the aforementioned efficient and lazy updates. It enables further investigation into the potential advantages of switching between CPU and GPU execution based on the specific workloads performed on a graph. Additionally, it enables future out-of-GPU-memory graph representations for graphs that do not fit into GPU memory.

We present a novel solution that uniquely combines a simple but powerful programming model inspired by Gunrock and an efficient dynamic graph data structure inspired by FaimGraph, which additionally, is shared and lazily synchronized between the host and device. We also assess the library's performance by running a set of graph constructions, updates, and algorithms on real large-scale graphs, and compare its execution times against some of the state-of-the-art GPU accelerated graph processing frameworks like Gunrock [42], Hornet [6] and FaimGraph [46].

1.4 Contributions

The main contributions of this work are:

- Design of a programming model and interface that allows for graph manipulation, and intuitive development of graph processing algorithms.

[2] Lazy synchronization is a strategy that delays the synchronization of an update until its value is needed.

- Development of Marrow-Graph, a GPU-accelerated dynamic graph processing library that implements the previous abstraction using a BAL data structure.
- Evaluation of our solution against several state-of-the-art GPU-accelerated graph processing frameworks.

1.5 Structure

This document is composed of six chapters:

- The first chapter introduces the motivation for this work, and provides an overview of the developed solution.
- The second chapter starts by providing the background required for the comprehension and development of Marrow-Graph. This includes an overview of the architecture and execution model of the GPU, an overview of CUDA, an introduction to Marrow's base concepts and architecture, an exploration of the most common data structures to store graphs, an overview of the existing programming models to process graphs, and finally, a survey of the existing graph processing solutions.
- In the third chapter we formally introduce Marrow-Graph's programming model and interface and detail its implementation using the BAL data structure. Additionally, we provide a breakdown of all the operators, graph manipulation functions, and some of the algorithms supported by Marrow-Graph.
- In the fourth chapter we evaluate Marrow-Graph based on its correctness, expressiveness, and conciseness, and its performance (experimentally). For the latter two, we compare Marrow-Graph against the state-of-the-art GPU-based graph processing frameworks.
- The fifth chapter offers a comprehensive discussion on the established goals and whether they were met. Additionally, we outline future work for enhancing Marrow-Graph's capabilities and performance.

2

Background and Related Work

In this section, we present the background required for the understanding and the conception of our solution. We also discuss the current state-of-the-art graph processing frameworks we considered relevant and that can be used for comparison against our solution.

We start by presenting some basic notions about the architecture and execution model of the Graphics Processing Unit (GPU), as well as the most famous and best performing GPU programming model named Compute Unified Device Architecture (CUDA) [12]. We then present the Marrow framework and give a brief detailing of its inner workings. Finally, we study how graphs can be stored and processed on the GPU. To this end, we present the various data structures that can be used to represent graphs, the existing programming models to process graphs, and the existing state-of-the-art high-performance graph processing frameworks.

2.1 GPU

The most common usage of the GPU is to render images intended for output to a display device. However, these chips can also be used for any generic computations . Using a GPU for other types of computations besides image rendering is known as General Purpose computing on Graphics Processing Units (GPGPU). The main difference between a GPU and Central Processing Unit (CPU) lies in their core count and complexity. GPUs contain thousands of "low"frequency (around 1 to 3 GHz) and low complexity compute cores, while CPUs typically have between 1 to 32 cores with a high frequency (around 2 to 6 GHz) and high complexity. This type of architecture makes GPUs the ideal platform for highly parallelizable applications [12]. CPUs are designed to achieve a good performance in a wide variety of tasks. To this end, the CPU's design is aimed at minimizing latency, having a powerful arithmetic logic unit, big caches, and utilizing techniques like branch prediction and data forwarding. The GPU's design, on the other hand, is aimed at maximizing efficiency, leveraging large numbers of efficient compute units, arithmetic logic units, and pipelining [9].

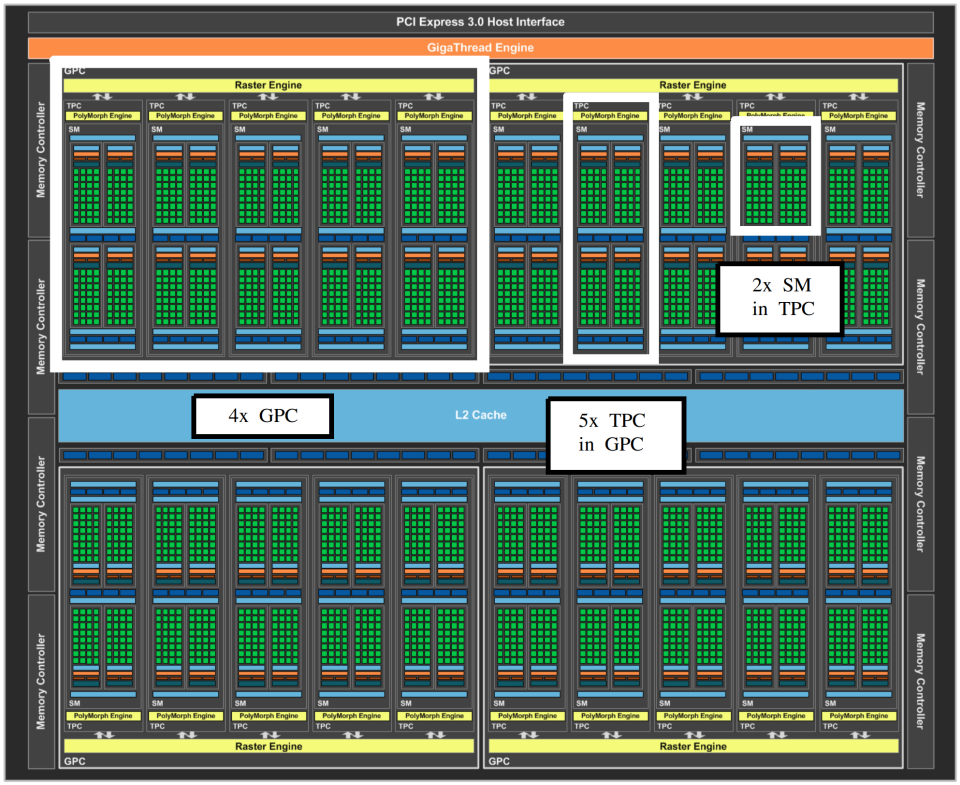


Figure 2.1: GP104 Pascal Architecture with 40 Stream Multiprocessor (SM) units

2.1.1 Architecture

Given that we will be working with the CUDA interface, which only supports Nvidia GPUs, we will focus on the architecture of Nvidia graphics cards, specifically the Pascal architecture[1]. It should be noted, however, that a lot of these concepts are also applicable to architectures from other GPU manufacturers.

As seen in Figure 2.1, the GPU is divided into multiple Graphics Processing Cluster (GPC)s, which contain multiple Texture/Thread Processing Cluster (TPC)s, which typically contain two SMs plus memory controllers. Each SM is composed of compute cores, special function units, double precision units, and other specialized units. The SM's main purpose is hosting the execution of warps of 32 threads. These will be addressed in more detail in section 2.1.2 [10, 11].

The memory in a GPU can be divided in two categories, device memory and on-chip memory. The device memory is a global high-bandwidth Dynamic Random Access Memory (DRAM) memory. This memory has a high capacity but also high latency, in the order of hundreds of clock cycles. On-chip memory is composed of shared memory registers, constant memory, L1 and L2 caches, and texture caches. Threads running in the same SM have access to a shared memory, meaning that memory accesses to shared memory within a SM are faster than accesses to global memory but slower than accesses

[1]More modern Nvidia architectures are more complex containing, for example, dedicated Tensor Cores for machine learning purposes which are not directly relevant for this work.

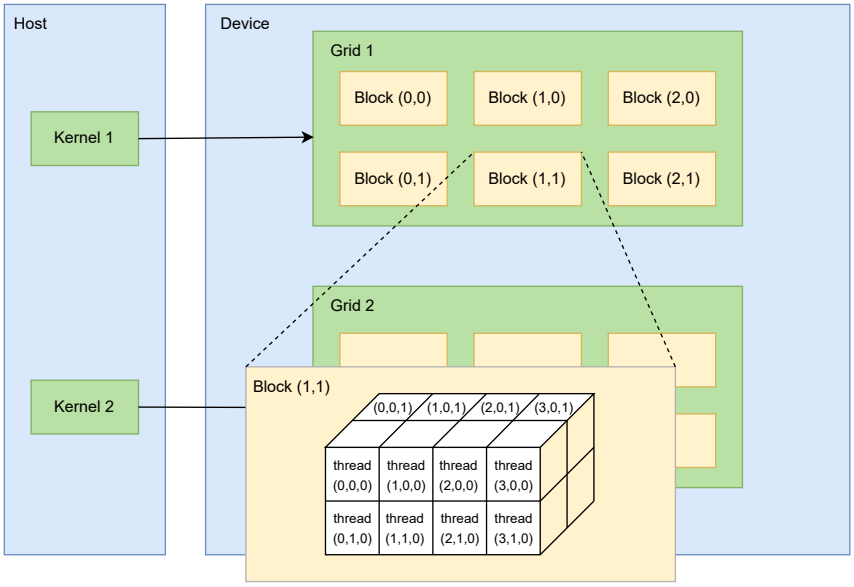


Figure 2.2: Grid with 3D blocks (adapted from diagram in [23])

to local memory. Shared memory within a SM has a very limited capacity, in the order of dozens of kilo-bytes [38].

2.1.2 Execution Model

When doing GPGPU, we usually refer to our main system as the host, and our accelerator as the device. The host configures and sends kernels to the device, and also transfers and allocates data between the two. The device executes the kernels specified by the host.

A kernel is essentially a function that is executed using a grid of blocks of threads on the device. Each block is executed by a SM and each thread is executed by a SM's core. All threads execute the same instructions and have access to their thread ID (ID within a block), block ID, and the dimensions of the grid. These IDs are used to select the correct data to process and make control decisions. in Figure 2.2 is an example of such a grid. These grids are defined by the programmer and can have between 1 and 3 dimensions (both the block layout and thread layout within the blocks), which impact how the IDs of both the blocks and threads are distributed [38]. Block sizes have limits that differ from architecture to architecture (Ex: 1024 x 1024 x 64).

Once a kernel is invoked, the GPU assigns each block to a SM (hardware is free to assign blocks to any SM), where each SM can handle multiple blocks. Blocks are in turn executed using warps (units of 32 threads). These warps execute in lock-step and with no order guarantees, meaning that all threads within a warp execute the same instruction in each clock cycle, but there are no guarantees in which order the warps will be executed. To improve performance, if a given warp has to execute an expensive operation, like reading from global memory, a SM can start executing operations from another warp while the previous awaits for a result [9]. This technique is called latency hiding.

```
#define N 2048
#define THREADS_PER_BLOCK 512
__global__
void add(int* res, int* b, int* c, int n) {
    int index = threadIdx.x + blockIdx.x*blockDim.x;
    if(index < n)
        res[index] = b[index] + c[index];
}

int main() {
    int *res, *b, *c;
    int *d_res, *d_b, *d_c; // Device data
    int size = N * sizeof(int);

    // Allocate and init host data
    b = (int*)malloc(size);
    c = (int*)malloc(size);
    res = (int*)malloc(size);
    init_data(b, c, N);

    // Allocate device data
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);
    cudaMalloc((void**)&d_res, size);

    // Copy data from host to device
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_c, c, size, cudaMemcpyHostToDevice);

    // Specify grid and invoke kernel
    long nb = (N+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;
    add<<<nb, THREADS_PER_BLOCK>>>(d_res, d_b, d_c, N);

    // Copy result data from device to host
    cudaMemcpy(res, d_res, size, cudaMemcpyDeviceToHost);

    // Deallocate memory
    cudaFree(d_b); cudaFree(d_c); cudaFree(d_res);
    free(b); free(c); free(res);
}
```

Listing 1: Vector addition in CUDA.

2.2 CUDA

The most popular and best performing GPGPU programming model is Nvidia's CUDA. Within a C++ program, CUDA allows us to transfer data between the host and device, specify a kernel using an annotated function, and invoke it specifying the layout of the associated grid. A typical CUDA program is composed by the following steps:

- Allocate and initialize data on the host.
- Allocate memory on the device.
- Copy data from the host to device.
- Specify the grid layout and invoke a kernel defined using an annotated function that has access to a `threadIdx`, `blockIdx`, and `blockDim`.
- Copy result data from device to host.

- Deallocate memory.

Listing 1 provides an example of such a program that adds two vectors on the device. Matrices `b`, `c` and `res` are allocated and initialized on the host. Then matrices `d_b`, `d_c` and `d_res` are allocated on the device, and `b` and `c` are copied to `d_b` and `d_c` respectively. The kernel `add` is invoked using a grid with `nb` blocks and `THREADS_PER_BLOCK` threads in each block. Note that we round up the number of blocks `nb` in line 31. The grid dimensions are specified within the ⋘⋙ delimiters. In the `add` kernel, the `threadIdx` and `blockIdx` fields are used to compute the index of the elements each thread should add. Finally `m_res` is copied to `res` and the memory is freed.

2.2.1 Good Practices and Optimizations

Grid and Block Geometry. To reach good performance, one should use large numbers of threads, preferably multiples of 32 (number of threads per warp), and multiple blocks, preferably multiples of the number of SMs. This ensures that all cores of all SMs are being used as much as possible. In order to hide synchronization latency, one can use as a rule of thumb: Number of thread blocks > 2 * number of SMs.

Control Flow Divergence. Warps are executed in lock-step, meaning every thread executes the same instruction simultaneously. Within a kernel, if a conditional statement is executed, which is dependent on the thread or block IDs, some threads can end up in different "paths". This is known as control flow divergence. As seen in Figure 2.3, once there is a branch, all threads that took path A execute first, while the others are idle. When all the threads of path A have executed all instructions of that particular branch, all the threads that took path B execute, while the others are idle. At the end of the branch, all threads continue executing in parallel. Having threads idle is of course bad for performance. The higher the degree of divergence the worse the performance. This means that conditional statements which result in this kind of branching, should be avoided as much as possible.

Thread Cooperation. Threads within a block can cooperate via shared memory, atomic operations, and barrier synchronization. Each thread has its own local data, but threads within a block can use shared data. Data to be shared between threads can be declared using the `__shared__` keyword. There also exist primitives to perform block-wide barriers and atomic operations on data, however, these operations come at a performance cost, meaning that they should be used sparingly.

Overlapping Execution and Communication. The latency associated with transferring data between the host and device is very high. This means that the device cores can become idle for lots of clock cycles while communication with the host is being performed. In order to avoid this, it is possible to conduct communication asynchronously and bi-directionally

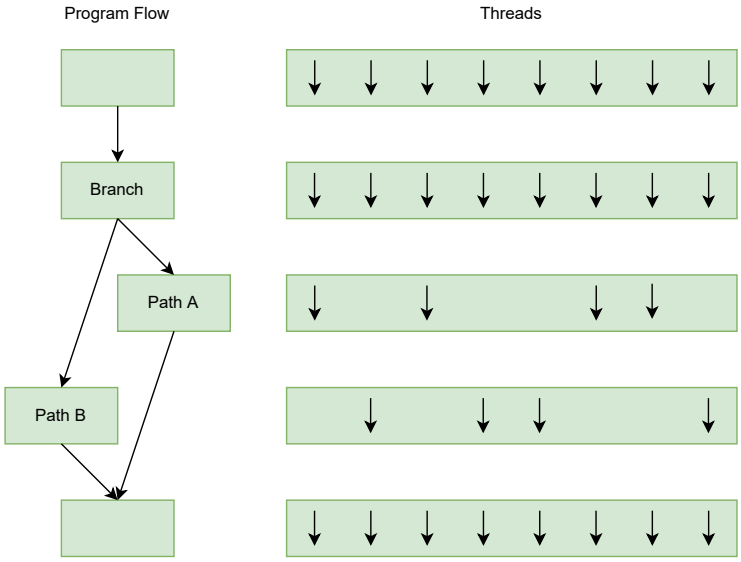


Figure 2.3: Control flow divergence.

while the device is performing computation. For example, considering a scenario where a lot of data has to be processed, it might be possible to partition the data into N partitions, send the first partition to the device, and then while the second partition is being sent, the first partition can start being processed. CUDA streams and events can be used for this purpose. Other usages of these mechanisms include asynchronous requests to the device to allow for concurrent execution between the host and device.

Global Memory Accesses. Given that accesses to global memory are slow, it is important to make these accesses as efficient as possible. When fetching data, the bus can handle 128 bytes. The main caches are also divided into 128-byte cache lines. Accesses without caching are done via 32-byte segments (given that warps have 32 threads). To reduce the number of transactions required to fetch data, it is important to keep the memory accesses coalesced and aligned with the 128-byte cache lines and with the 32-byte segments. This way all the threads can fetch their respective data simultaneously.

Shared Memory Accesses. Within a SM, threads can access a shared memory. This shared memory is divided into 32 banks which are 8 bytes (2 words) wide. If two (or more) different threads within a warp access the same bank, but do not access the same word, a bank conflict occurs . Whenever there is a bank conflict, the accesses are performed sequentially. To avoid this, one should arrange the data accesses in such a way that avoids these bank conflicts as much as possible. This can require using padding (sacrificing some shared memory space in favor of less latency) or remapping of the data. It should be noted that for devices of compute capability 2.0 and higher, the ability to multicast shared memory accesses has been added to mitigate this issue [41]. This means that multiple accesses to the same location by any number of threads within a warp are served simultaneously.

```
1  using T = float;
2  marrow::vector<T> x(100), y(100);
3  x.fill([] (int i){return rand();});
4  y.fill([] (int i){return rand();});
5
6  marrow::scalar<T> min = marrow::reduce<min<T>>(x);
7
8  auto sum = y + x;
9  marrow::vector<T> result_sum = min * sum;
10
11 auto dif = y - x;
12 marrow::vector<T> result_dif = min * dif;
```

Listing 2: Marrow basic program.

2.3 Marrow

Marrow [30, 40, 2] is a high-level C++ parallelization library that implements an algorithmic skeletons programming model. Skeletons comprise common algorithms such as map, scan, and reduce, which can be nested to express complex computations. Marrow parallelizes and executes these computations, and supports multiple backends to do so. It supports CUDA and OpenCL backends for GPU offloading, and an OpenMP backend for multi-threaded execution. Data in a marrow program can be represented using smart containers including vectors, arrays, scalars, matrices, and tensors. These containers offer a unified address space, meaning that the programmer does not need to worry about managing host and device memory (when dealing with GPU-based backends). A data container can be seemingly accessed both on the host and the device. The programmer simply instantiates a set of containers, then applies skeletons and transformations over these containers, and marrow handles all the memory management and skeleton parallelization and execution.

2.3.1 Using Marrow

Listing 2 shows a simple C++ program written using marrow. In lines 2-4, we create two vectors of equal size. In line 6, the min value from vector `x` is calculated using the `reduce` skeleton. In lines 8-9 the vectors are summed together (the + operator is equivalent to using the skeleton `map<plus>`) and then the result is multiplied by the previously computed scalar `min`. Lines 11-12 have a similar logic. As we can see, the programmer only specifies the containers and operations over these containers that he wishes to perform, and marrow converts these operations into kernels, launches them, and manages their execution and synchronization on the GPU (assuming a backend such as CUDA is used). If a CUDA backend is used, all skeletons are executed on the device. In this program, the vector `x` is initially allocated and filled on the host[2] in line 3, then in line 6, the vector `x` is transferred to the device and the `min` scalar is computed by a kernel on the device as well.

[2]Alternatively, we can configure marrow to perform this fill lazily on the device for better performance.

Commonly used functions like map, reduce, scan and filter are provided by marrow, and can be applied to the various containers. Marrow also provides common operators such as min, max, plus, minus, multiplies, etc to use alongside the skeletons (for example `reduce<max>` or `map<plus>`). Alternatively, the programmer can define his own functors and pass them to the skeletons. Operator overloading is also provided for common map operations such as arithmetic functions between the different container types. In listing 2 we can see that to perform arithmetic operations between vectors, instead of using a map skeleton with the corresponding arithmetic operator, the programmer can simply use the overloaded functions plus, minus, multiplication, etc.

Marrow also supports custom functions to process and generate containers. These are denominated `marrow::function` and operate as a more generic map operator. Marrow functions can take as arguments any number of marrow containers of any size, any number of primitive data types, and return a single marrow container. These functions can also receive coordinates as a parameter, which is useful when dealing with backends such as CUDA. Moreover, the parameters of a marrow function can optionally be marked as `in`, `out`, or `inout`, to specify if a parameter is intended as an input, an output container, or both. The geometry of the function, i.e. the grid geometry in the case of a backend such as CUDA, is implicitly calculated by marrow, but can also be set by the programmer.

Listing 3 illustrates a marrow function `mandelbrot_fun` that can take as input a `vector<int>` and an integer `n`, and outputs a `vector<int>` representing the pixel values of a discrete mandelbrot set. In lines 1-2 we define an auxiliary function `divergence` annotated with `marrow_function`. This annotation is necessary for dealing with backends such as CUDA that require decorators such as `__device__`. In lines 4-14 we define a functor `mandelbrot_fun` that extends `marrow::function` passing as template parameters the functor itself, the return type, and the types of the arguments. In the body of the function we describe the logic we wish to perform on each element `pos` of the input container. In lines 17-22 we define a C++ function that starts by generating a vector `positions` of size $n * n$ of counting integers [3], then instantiates a `mandelbrot_fun` object, and finally applies `positions` and `n` to the marrow function and stores the result in the marrow vector `x`. Note that this specific example could also be written using a map skeleton with a similar functor instead.

Listing 4 illustrates a more complex marrow function `foo_fun`. In line 21 we pass a container of integers, a container of floats, and a single integer `n` to `foo_f`. However, the functor receives as parameters an integer, a collection of floats (float pointer), and another integer (besides the function coordinates `c`). This implicitly states that the function will operate in parallel over the elements of `indexes` while delegating accesses to the container `values` to the programmer. In this case, we access the elements of `values` using the function coordinates. By providing the correct data types in the function's template parameters, Marrow is able to handle this distinction. Finally, the function will return a

[3] Not actually a vector but rather an expression that evaluates to such a vector.

```
marrow_function
int divergence(int depth, marrow::complex<float> c0) { ... }

struct mandelbrot_fun : public marrow::function<mandelbrot_fun, int, int, int> {

    marrow_function
    int operator()(int pos, int n) {

        int x = pos / n;
        int y = pos % n;
        marrow::complex<float> c0(XMIN + x * SIZEX / n, YMIN + y * SIZEY / n);
        return divergence(DEPTH, c0);
    }
};

marrow::vector<int> mandelbrot(int n) {
    mandelbrot_fun mandelbrot_f;
    auto positions = iota(n*n); // expression that evalutes to a marrow::vector<int>
    marrow::vector<int> x = mandelbrot_f(positions, n);
    return x;
}
```

Listing 3: Marrow mandelbrot function.

```
marrow_function
float bar(float x, float y) { ... }

struct foo_fun : public marrow::function_with_coordinates<foo_fun, float, int, float*, int> {

    marrow_function
    float operator()(coordinates_t c, int index, float* values, int n) {

        int coordinate_index = c[0] % n;
        return bar(values[index], values[coordinate_index]);
    }
};


marrow::vector<float> foo(int n) {

    foo_fun foo_f;
    marrow::vector<int> values = ...;
    marrow::vector<float> indexes = ...;
    // set values of indexes and values
    marrow::vector<float> result = foo_f(indexes, values, n);
    return x;
}
```

Listing 4: Marrow foo function.

container of the same size as `indexes`, but of type float.

You may notice that in listings 2 and 3 we sometimes use the `auto` keyword when defining the variables that receive the result of a given operation over containers. The reason is that whenever we apply a skeleton or function over a set of containers, instead of triggering the corresponding computation, we are simply defining a template expression that describes the desired computation. Only when such an expression is assigned to a container is the expression actually evaluated and therefore computed. This means that in line 8 of listing 2, no computation is launched, and the type of `sum` isn't a container, but rather a marrow expression. Only in line 9, where the result of the expression `min * sum`

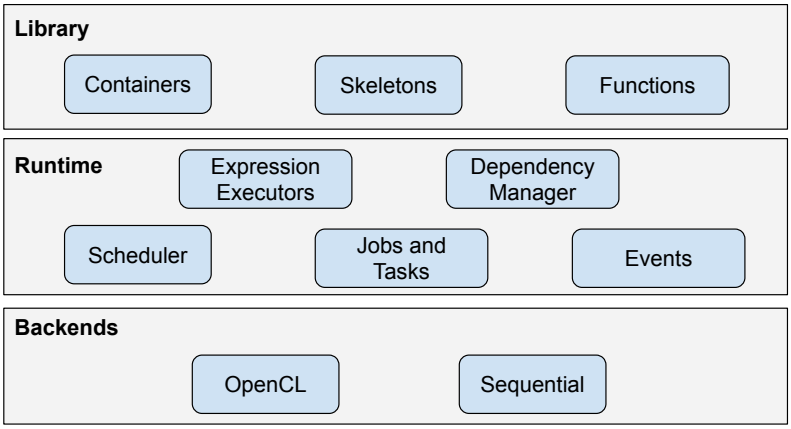


Figure 2.4: Marrow architecture.

is assigned to the vector `result_sum`, is the expression actually evaluated. This allows marrow to compute the vector `result_sum` in a single kernel (more on this later). As a rule of thumb, whenever the result of a computation is explicitly required or is used in multiple succeeding computations, we assign it to a corresponding container, otherwise, the `auto` keyword can be used to simply obtain the desired marrow expression.

2.3.2 Marrow Architecture

Marrow is comprised of 3 layers (as seen in Figure 2.4): API, runtime, and backends. The user only interfaces with the upper layer. This layer includes all the supported containers, functions, operators, and skeletons. During compilation, kernels are generated from the computations the user-defined, using the marrow API, for a specified backend such as CUDA. Its then the job of the runtime layer to launch and manage these kernels, as well as ensuring any necessary data synchronization. To this end, the runtime is composed of a scheduler, a dependency manager and a task manager. These components utilizes generic modules such as marrow events and marrow allocators, which in turn delegate backend-specific calls to the backend layer.

As previously mentioned, to avoid generating a separate kernel for each operation/function, marrow expressions are subjected to lazy evaluation. This ensures that a marrow expression's result is only computed when it is required or specifically requested. To this end, marrow makes use of Abstract Syntax Tree (AST)s which can include multiple operations/functions. Potentially a single kernel is generated for an entire AST. Returning to Listing 2, in lines 8 and 11, the `auto` keyword is used to define the type of the variables `sum` and `dif`, meaning that these variables store marrow expressions. In line 9 we have another variable assignment, but this time, the type of the variable is a marrow container. This informs marrow that the expression on the right-hand side of the expression should be evaluated, and its result stored in the variable `result_sum`. The AST that is computed in this line is represented in Figure 2.5 (as well as the ASTs that are computed in lines 6 and 12 respectively). We can see that the AST includes both the sum between `x` and `y`,

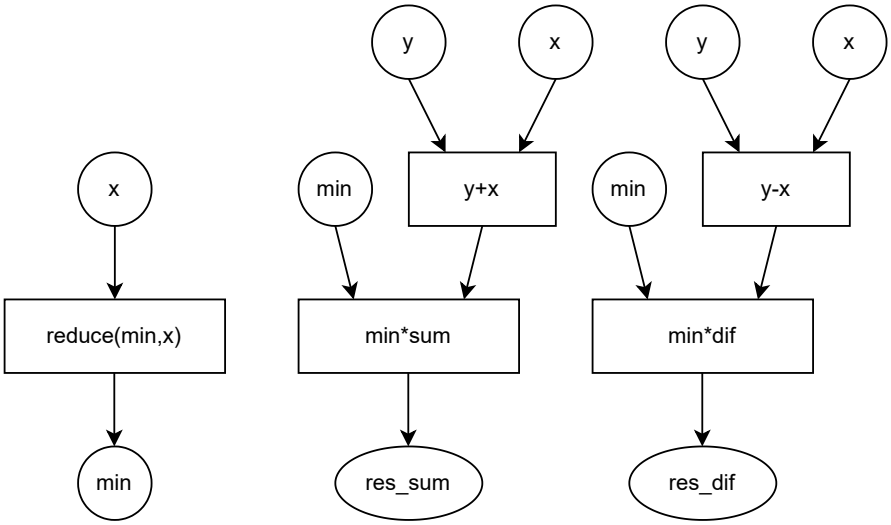


Figure 2.5: Marrow program ASTs.

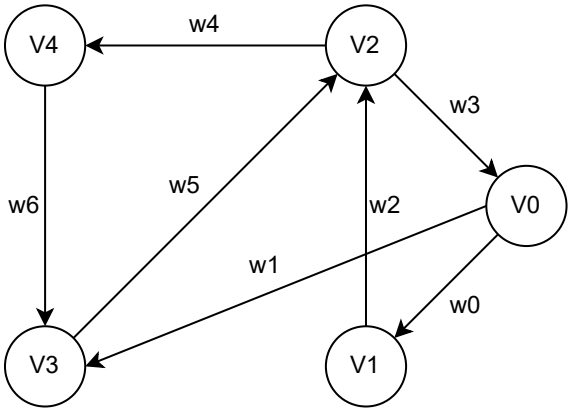


Figure 2.6: Directed graph.

and the multiplication of `min` with the previous result. This optimization can drastically decrease the communication and memory transferred between the host and device. Note that sometimes it is not desired to use the `auto` keyword if a variable is later used multiple times, like variable `min` in line 6 for example, since this will result in the value being computed multiple times in different kernels.

Another optimization implemented by marrow is the usage of non-blocking kernel execution. Whenever a computation is launched, like in lines 6, 9, and 12 from listing 2, the program follows to the next instruction without blocking until the computation is over. Of course, if the result of a computation is required in another line of the program, the program waits for the result to be computed before proceeding. All this logic is handled by the runtime dependency manager and scheduler.

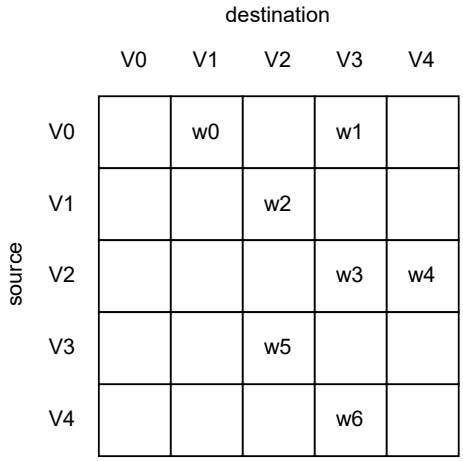


Figure 2.7: Adjacency matrix representation of graph 2.6.

2.4 Graph Data Structures

We will now present the different data structures that can be used to represent graphs and discuss which are better suited for our purpose. This is not an exhaustive list of all the specific data structures used by the various graph processing frameworks, but rather a high-level view of the most common structures to represent graphs. Since our goal is to develop a GPU-based dynamic graph library, these three characteristics should be taken into account:

1. **Random access speed:** Fast random accesses allow for fast algorithms to be performed over the graph and help us leverage parallelism.

2. **Compactness:** A compact representation of the graph allows us to take as much advantage of the limited GPU memory as possible, and to reduce the amount of data transfers between the host and device.

3. **Update-friendliness:** A dynamic graph data structure must allow for efficient updates to the graph.

The most common data structures used to represent graphs are the following:

Adjacency Matrix. Storing a graph in an adjacency matrix is straightforward. Each cell $C_{i,j}$ (row i, column j), stores the weight of the edge that goes from vertex V_i to vertex V_j, if the edge exists. This representation allows for simple memory allocation strategies and fast random accesses. However, for sparse graphs[4], this leads to a lot of wasted memory and sparse data, as can be seen in the adjacency matrix in Figure 2.7 (opposed to contiguous data, which usually allows for better performance).

Adjacency List. An adjacency list represents a graph as an array of lists. For each vertex V_i, we define a list of adjacencies with all the vertices V_j, for which there exists an edge

[4]A graph in which most pairs of vertices are not connected by edges.

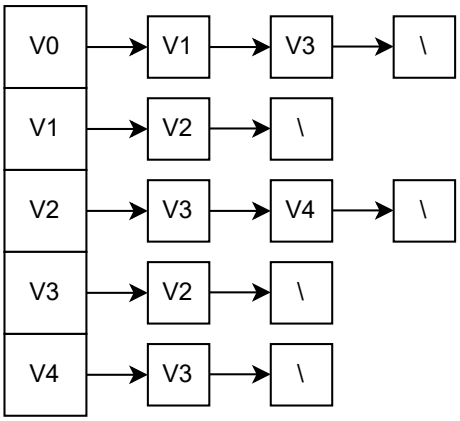


Figure 2.8: Adjacency list representation of graph 2.6.

from V_i to V_j. This way, no memory is wasted for non-existing edges, contrary to an adjacency matrix. The main issue with this approach is related with the way these lists are stored. The different lists will most likely not be stored contiguously. Moreover, some list implementations, like linked lists, do not store the the elements of a list contiguously. This kind of memory layout usually does not perform well in high-performance and parallel environments. Edge searches are also slower compared to an adjacency matrix given that the neighbors of the source index must be traversed to search for a specific edge. An example of an adjacency list can be seen in Figure 2.8.

In order to improve data locality, some graph representations use blocked adjacency lists [46, 13]. In blocked adjacency lists, neighboring vertices of a given vertex v are stored in fixed-sized blocks. For adjacency lists composed of multiple blocks, each block points to the next one. A small block size results in the same restrictions of a linked list whilst a large edge block has similar drawbacks of an adjacency matrix. This strategy allows one to trade some memory overhead for better locality.

Compact Sparse Rows. Compact Sparse Rows (CSR) is considered to be the most compact data structure to store graphs. This data structure consists of 3 arrays (or vectors) as seen in Figure 2.9. The two "parallel" arrays, `col_indexes` and `values`, store the information about the edges and have a length equal to the number of edges in the graph. The array `row_offsets`, has a length equal to the number of vertices in the graph. Each element O_i of the array `row_offsets`, stores the offset where the information, about the outgoing edges from vertex V_i, starts. `col_indexes` stores the indexes (or IDs) of the destination vertices of the edges, and the array `values` stores the corresponding weights. Given that all the outgoing edges from a given vertex V_i are stored contiguously, the degree of V_i can be calculated using the offsets array as follows: `row_offsets[i+1] - row_offsets[i]`. Even though CSR features good compactness and locality, updating a graph represented using CSR, requires reconstructing the whole data structure.

In order to address the last issue, some bodies of work [7, 15] have proposed modifications to CSR, in order to support relatively efficient manipulations to the graph. CSR++ [15]

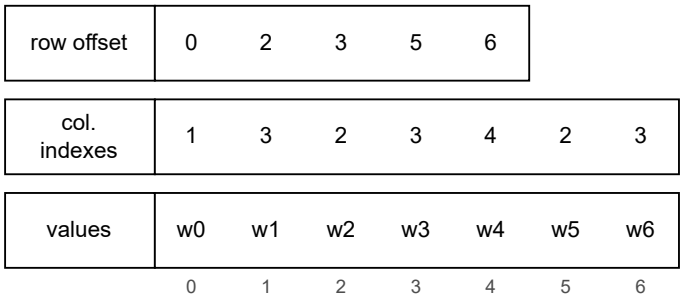


Figure 2.9: CSR representation of graph 2.6.

uses a segmentation technique, keeping vertices and neighbor edges compacted. It stores vertices in segments, where each segment is a fixed-size array. Edges are stored using per-vertex arrays. Vertex property values are stored in parallel arrays within each segment. For edge properties, a similar segmentation approach is used. Each vertex structure stores a pointer to an array of edge property values. The property values can be located using offsets and calculations based on the type, index, and degree of the vertex. Packed Compressed Sparse Row (PCSR) [7] bases its implementation on a dynamic array-based data structure called Packed Memory Array (PMA). PCSR is similar to CSR, but leaves space between elements, allowing for significantly faster insertions and deletions while slowing traversal times by a constant factor. As far as we know, none of these CSR adaptations have been implemented in GPU environments.

Vector Graph [5]. Vector graph stores a graph using a segmented vector. Each segment corresponds to a vertex, and the number of elements in a given segment corresponds to the degree of the associated vertex. The values kept in the elements of the segmented vector are pointers to the other end of the edge. More vectors can be added to store information about weights, vertex IDs, etc. Figure 2.10 shows an example of a vector graph representation. Vector `vertex` stores the ID of the vertex of each segment. Vector `segment-descriptor` stores the degree of the vertices of each segment. Vectors `cross-pointer` and `weights` store the pointers to the other end of the edges and the weights associated with each edge. For undirected graphs, this representation results in duplicated information. Like CSR, updating a graph requires reconstructing the whole data structure.

Slotted Pages [21]. Slotted pages is a data structure that represents a graph as a set of pages. Each page is stored in a contiguous area of memory. A page is divided into two logical areas. An area of slots, which typically starts at the end of the page and grows towards the beginning of the page, and an area of records, which typically starts at the start of the page and grows towards the end of the page. The slots store the adjacency lists of the vertices stored in the page. The records store information that allows one to get the location of an adjacency list for a given vertex ID. Some implementations of slotted pages also include metadata at the start of the slotted page, like the number of vertices stored on the page. All slotted pages have the same fixed size. Vertices and edges can be

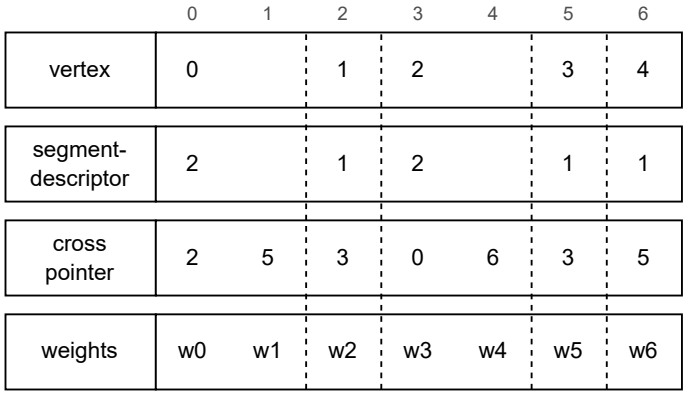


Figure 2.10: Vector graph representation of graph 2.6.

added to a slotted page until it is full, at which point, a new slotted page is created. If an adjacency list of a vertex does not fit in a single slotted page, extra Large Adjacency Pages are created. This data structure is an appealing data structure for dynamic graphs, and has been used by dynamic graph processing frameworks like TurboGraph [21], since it allows for regular accesses, is semi-compact, and allows for efficient updates.

Conclusions

From Table 2.1 we derive a comparison between the previously presented data structures. CSR and vector graph offer the best compactness and memory locality but are subjected to extremely slow updates. However, CSR variants that trade some compactness and locality for faster updates can be a viable option for representing a dynamic graph on a GPU. Adjacency Matrices, even though offer the fastest edge searches, have the worst spacial complexity and are generally too sparse considering the limitations of GPU memory (this doesn't apply when considering dense graphs, however, a lot of large-scale real-life graphs are sparse). Adjacency lists offer good characteristics overall, but as discussed before, their memory layout isn't well suited for GPUs. Blocked adjacency lists and slotted pages offer a good balance between compactness, memory locality, and update efficiency. In both cases, compactness and locality are proportional to the size of the blocks or pages. Blocked adjacency lists offer some faster updates and search operations compared to slotted pages. This is due to the overheads associated with searching for a page that contains a given vertex, and the in-page reallocations. Slotted pages can however offer better memory locality for sparse graphs, given that a single page can store multiple adjacency lists.

2.5 Graph Programming Models

In order to process graphs, one can consider different units of parallel execution and different programming models. In this section, we introduce some of the most common units and models, and in the following section, we discuss how these are implemented by some of the existing graph processing frameworks.

	Adjacency Matrix	Adjacency List	Blocked Adjacency List	CSR	Slotted Pages
Dynamic	yes	yes	yes	no	yes
Directed	yes	yes	yes	yes	yes
Mem. Locality	poor	poor	$\propto s_b$	optimal	$\propto s_p$
Spacial Cost	$O(n^2)$	$O(n+m)$	$O(n+(1+f_b)*m)$	$O(n+m)$	$O(n+(1+f_p)*m)$
Search Edge (v_1, v_2)	$O(1)$	$O(deg(v_1))$	$O(deg(v_1))$	$O(deg(v_1))$	$O(log(n_p)+deg(v_1))$
Add Vertex	$O(n^2)$	$O(1)$	$O(1)$	$O(n+m)$	$O(1)$
Del Vertex v	$O(n^2)$	$O(deg(v))$	$O(deg(v))$	$O(n+m)$	$O(log(n_p)+s_p)$
Add Edge	$O(1)$	$O(1)$	$O(1)$	$O(n+m)$	$O(log(n_p)+s_p)$
Del Edge (v_1, v_2)	$O(1)$	$O(deg(v_1))$	$O(deg(v_1))$	$O(n+m)$	$O(log(n_p)+deg(v_1)+s_p)$

Table 2.1: Graph data structure comparison with an approximation of each one's spacial complexity and temporal complexity for a set of operations. Note that for vertex deletion complexity, we are ignoring possible necessary destination vertex deletions (n = |vertices|, m = |Edges|, s_b = block size, s_p = page size, n_p = number of pages, $f_b \propto S_b$ = blocked adjacency list fragmentation proportion, $f_p \propto S_p$ = slotted pages fragmentation proportion).

2.5.1 Units

Vertex-Centric. In vertex-centric programming models, the processing algorithms are expressed from the perspective of a vertex. This means that the user defines a vertex-local function , which can receive and send information from/to its neighboring vertices. The vertex therefore becomes the unit for parallel execution. This concept was formally introduced by Pregel [29] and later used by other frameworks like GraphLab [27] and PowerGraph [18].

Edge-Centric. In edge-centric programming models, the processing algorithms are expressed from the perspective of an edge. This paradigm was adopted by systems like X-Stream [34] and Chaos [35]. This unit can potentially ease load balancing given that it does not deal with variable vertex degrees like the vertex-centric unit.

Sub-Graph-Centric. Both vertex-centric and edge-centric models are fine-grained, given that the unit of parallel execution is the vertex and the edge respectively. This can lead to large communication overheads in distributed systems. Sub-graph-centric models therefore aim to reduce these overheads by using a sub-graph structure as the unit of parallel execution (coarse-grained). Two approaches are used for this end, partition-centric and neighbourhood-centric. The first partitions the graph into multiple sub-graphs which can be processed in parallel. Each vertex can send information to any other vertex within the same partition, opposed to a vertex-centric approach, where a vertex can only send information to its neighboring vertices. This paradigm has been adopted by Giraph++ for example. Neighbourhood-centric models on the other hand, allow for multiple sub-graphs to exist within a physical partition. This paradigm facilitates the implementation of graph algorithms that operate on ego networks; networks built around a central vertex of interest. In such algorithms, the user might want to define a set of custom sub-graphs which are built around vertices and their multi-hop neighbors. Shared-state updates are

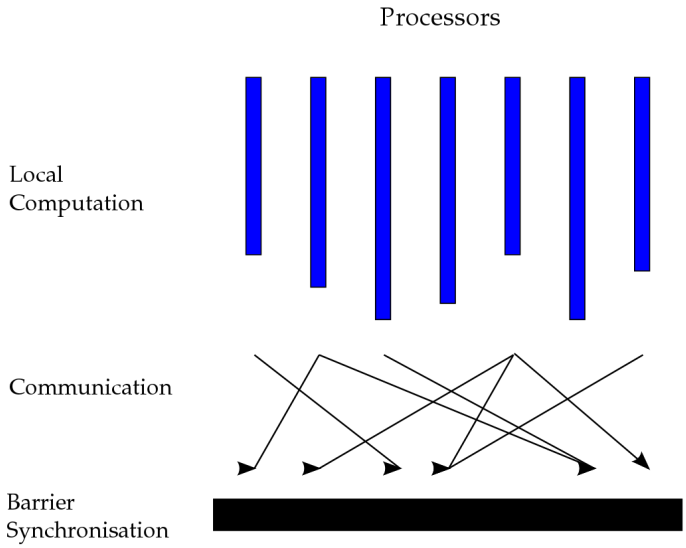


Figure 2.11: Bulk synchronous parallel superstep (taken from [44]).

used to exchange information between sub-graphs in the same partition, and replicas and messages are used to exchange information between sub-graphs in different partitions [22].

2.5.2 Models

Bulk Synchronous. Bulk synchronous executes an algorithm in supersteps. A superstep consists of a sequence of three steps (as shown in Figure 2.11):

1. **Concurrent Computation:** Each processor performs a local computation.
2. **Communication:** Processors exchange data.
3. **Barrier Synchronization:** Synchronizes all processors.

In graph processing, during each superstep, a user-defined function is applied to all the vertices asynchronously. The results are then sent to each vertice's neighbors. When implementing this model using GPUs, a superstep is performed by one or more kernels. Threads within a kernel execute concurrently. Communication between threads is performed via local-memory and communication between supersteps via global-memory. Synchronization between supersteps is achieved by a barrier that waits for the kernel (or multiple kernels) to finish before continuing. This model has been adopted by frameworks like Pregel [29], Gunrock [42] and Medusa [48], and is the best suited for GPU acceleration, given that a superstep can simply be modeled with a kernel launch.

Scatter-Gather. This model consists of two phases, each dictated by a user-defined function. In the scatter phase, each vertex sends messages to its neighbors. In the gather

phase, each vertex receives messages from its neighbors and performs a computation, possibly updating its state.

Gather-Apply-Scatter. This model was introduced by PowerGraph [18] to solve issues present with other models regarding power-law graphs, given that big differences between the degrees of vertices could lead to computational load imbalances (vertices with high degrees become stragglers). To solve this, the vertex-local function/program is decomposed into multiple phases, which can be more evenly distributed across a cluster. It consists of three phases, each dictated by a user-defined function and executed in parallel. In the gather phase, each vertex receives information from its neighboring vertices. In the apply phase, each vertex performs a computation and possibly updates its state. In the scatter phase, each vertex sends information regarding its new state to its outgoing edges. Some algorithms do not require the gather phase.

2.6 Graph Processing

In this section we will present some of the existing state-of-the-art high-performance graph processing frameworks, discussing their key features, data models, and programming models. All the frameworks that were analysed can be found in table 2.2, however, in this section, we will only address the ones we found to be the most relevant for our purpose. Given our goal, we will give a more detailed analysis of the GPU based solutions.

2.6.1 Distributed

As we can see in Figure 2.2, most distributed graph frameworks, i.e. intended for clusters, are vertex-centric, with some exceptions, and represent their graphs using data structures which can easily be partitioned, like edge lists and adjacency lists.

GraphIn. GraphIn [36] adopts a hybrid data structure using a compressed matrix format to store a static version of the graph and edge lists to store updates. The edge list allows for fast updates, while the compressed format allows for fast parallel computation. GraphIn merges these two data structures whenever it is required.

Chaos. Chaos [35] has its foundations on X-Stream [34], using sequential storage accesses and an adaptation of X-Stream's streaming partitions which allow for parallel execution. Its design ensures that the storage devices are busy as much as possible in order to optimize for the bandwidth bottleneck. Both Chaos and XStream are novel in using an edge-centric (instead of vertex-centric) implementation of the scatter-gather model, and for streaming unordered edge lists instead of utilizing random accesses which are slower when using storage devices [8].

PowerGraph. PowerGraph [18] is an extension of GraphLab, and offers a different approach to representing distributed graphs by exploiting the structure of power-law graphs. It presents e graph-parallel abstraction based on Gather Apply Scatter (GAS), which eliminates the degree dependence found in vertex-centric programs. It does so by leveraging the GAS decomposition, factoring the vertex programs over edges. This way, PowerGraph maintains the "think-like-a-vertex"paradigm but is able to distribute the computation of a single vertex over a cluster.

GraphX. Most of the frameworks so far are specialized graph processing systems. GraphX [17] on the other hand, is built on top of Spark, a general-purpose distributed dataflow framework. It presents a common graph API, which is implemented using a few basic dataflow operators, like join, map, group-by, etc. To achieve a good performance, graph-specific optimizations are implemented using distributed join optimizations and other distributed techniques. Internally the graphs are stored using a pair of vertex and edge collections built on the Spark RDD.

Pregel. Google's Pregel library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' outgoing edges. Its main contribution is the introduction of a simple vertex-centric API that automatically scales efficiently.

2.6.2 CPU-Based

Shared memory multi-core CPU machines have the advantage of not having to deal with communication overheads and data partitioning, however, they also have limited potential for parallelization. Given that modern CPUs can perform complex operations very fast, frameworks on these machines can take advantage of more complex data structures, like hash-tables [32], snapshot-based data structures [28] and even NUMA-Aware data layouts [47], which can be harder to implement efficiently in distributed or GPU-based environments. We will now discuss three cases of study, chosen for their novel data structures and/or simplicity.

LLAMA. LLAMA [28] is a graph storage and analysis system that supports mutability and out-of-memory execution. It performs comparably to immutable main-memory systems for graphs that fit entirely in memory and outperforms out-of-memory systems. It is based on the CSR data structure but is augmented to allow for mutability and persistence with the usage of multi-versioned array snapshots. The graph is stored as a series of snapshots. When the graph is loaded, it is represented as a single snapshot, then with each update batch, a new snapshot is added. This allows for easy and fast updates and the ability to perform temporal graph analysis. Internally, LLAMA stores an edge table per snapshot, each storing consecutive adjacency list fragments [5], and a

[5] A vertex's adjacency list can be spread across multiple snapshots. The section of an adjacency list contained in a single snapshot is called an adjacency list fragment [28].

single vertex table, shared by all snapshots, that maps the vertex IDs to the indexes in the edge table. Whenever it is necessary, snapshots can be merged. LLAMA outperforms other state-of-the-art graph processing frameworks like X-Stream [34] and Graphlab [27] in several benchmarks.

Ringo. Ringo [32] is aimed at multi-core single-machine big-memory systems. The authors claim that graph processing often requires random data access patterns and deals with poor data locality, and therefore a single big-memory machine can be the most appropriate hardware for dealing with all but the largest graphs. They also analyzed that most of today's [6] real-life large graphs, can easily fit in a big-memory machine (with around 1 terabyte of memory).

In terms of architecture, Ringo combines a simple Python front-end, which interfaces with a scalable parallel C++ back-end, responsible for rapid data handling and manipulation. Its key features are: 1. A system for interactive and exploratory analysis of large graphs with hundreds of millions or even billions of edges. 2. Integration between graph and relational table processing, as well as conversion primitives between these two representations. 3. A simple programming model and operations to construct various types of graphs. 4. Competitive performance against distributed systems on all but the largest graphs.

In terms of graph analytics, 200 different graph functions are supported through its core graph analytics package SNAP. SNAP is a highly efficient C++ graph analysis library which was extended for Ringo with several new components, including parallel graph algorithms using OpenMP.

In order to efficiently support dynamic graphs, Ringo represents a graph as a hash table of nodes. Each node maintains a sorted adjacency vector of neighbors. It is claimed that this allows for similar space requirements to those of CSR while allowing for fast updates and a small impact on the graph algorithms' performance.

Ligra. Ligra [39] is a lightweight CPU-based graph processing framework aimed at shared memory systems. It provides a simple yet efficient solution to develop graph traversal algorithms using a similar operator abstraction to Gunrock [42]. Ligra's implementation employs CilkPlus or OpenMP to handle multi-threading. One of its key strengths lies in a robust load-balancing strategy built on CilkPlus [25], a fine-grained task-parallel library for CPUs. In terms of graph representation, a CSR-like data structure is used to store the graph in memory.

Ligra's authors claim that all but the largest real-life graphs can fit in modern shared memory machines. They leverage this fact by demonstrating that Ligra is highly effective for processing such graphs, yielding significantly better performance when compared to other distributed-memory graph processing systems. Particularly, it excels when dealing with algorithms that require processing only a subset of vertices in each iteration.

[6] As of 2015, when the paper was published.

Ligra features two fundamental routines: one for mapping over edges and another for mapping over vertices. The first can be applied to any subset of vertices, making the framework useful for many graph traversal algorithms that operate on subsets of the vertices. Moreover, the routines automatically adapt to the density of vertex sets. Several algorithms that take advantage of this characteristic were implemented including Breadth-First Search (BFS), graph radii estimation, graph connectivity, betweenness centrality, Page Rank (PR), and Single Source Shortest Path (SSSP). These algorithms are described in Ligra in a simple and concise manner, while performing well even when compared to highly optimized code, allegedly outperforming previously reported results on machines with higher core counts.

2.6.3 GPU-Based

A couple of GPU accelerated graph processing frameworks have already emerged. Most of these frameworks focus on static graphs, often using compact data structures like CSR to store them (as seen in Figure 2.2) to ensure good data locality. This is desired since GPUs' architecture is optimized to operate over contiguous data. There are however some exceptions like Hornet [6] and FaimGraph [46] which trade some compactness for update efficiency.

Gunrock. Gunrock [42] is a high-performance framework to process graphs on the GPU, offering a high-level programming model that allows developers to easily design new graph processing applications. It achieves this by implementing a data-centric model based on the concept of a frontier and operators that can be applied on a frontier. A frontier is composed of a subset of edges or vertices of a graph. All operators performed on a frontier are bulk-synchronous and subjected to load-balancing and work-efficiency optimizations.

Both Pregel and PowerGraph use programming models focused on defining sequences of computation steps. Gunrock, on the other hand, focuses on defining manipulations over a frontier. One benefit of such a model is the ability to easily switch between an edge-centric and a vertex-centric paradigm. Models like GAS and message-passing are focused on operations over vertices, making them harder to use when describing edge-centric computations.

Gunrock offers one compute operator and three traversal operators: advance, filter, and segmented intersection. These operators can be visualized in Figure 2.12 and are defined as follows:

- **Advance:** An advance operator generates a new frontier from the input frontier by visiting its neighbors. Each input item maps to multiple output items from the input item's neighbor list.

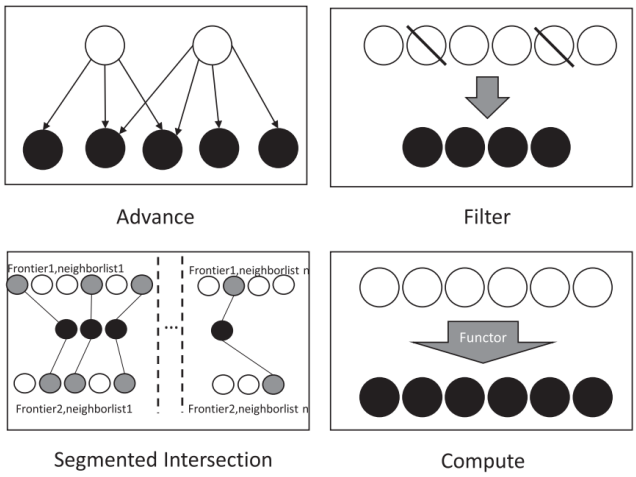


Figure 2.12: Gunrock's operators. Input frontier in white/grey and output frontier in black (taken from [42]).

- **Filter:** A filter operator generates a new frontier from the input frontier by choosing a subset of the input frontier based on programmer-specified criteria. Each input item maps to zero or one output item.

- **Segmented Intersection:** A segmented intersection operator takes two input frontiers F and F' with the same length and generates both the number of total intersections and the intersected node IDs as the new frontier. The intersections are performed between the neighbors of all the pairs of vertices $(v_i, v'_i), v_i \in F \wedge v'_i \in F' \wedge i \in \{0, |F|\}$. The resulting frontier is the set of common neighbors between the pairs of vertices (v_i, v'_i).

- **Compute:** A compute operator defines an operation to be applied on all elements of its input frontier. A programmer-specified compute operator can be used together with all three traversal operators.

A new graph primitive can be composed as a sequence of these four operators, which are then executed sequentially by Gunrock. It is often desired that primitives run to convergence. Gunrock generally expresses a convergence as an empty frontier. However, a developer can use different criteria, which can be specified using a computation operator.

Gunrock stores per-vertex and per-edge data as a Structure of Arrays (SOA), allowing coalesced memory accesses. Both vertex-centric and edge-centric operations are supported. To achieve this. the graph is stored using a CSR sparse matrix for vertex-centric operations by default, but users can also choose an edge-list-only representation for edge-centric operations.

Gunrock shows better performance than other GPU-based graph processing frameworks like MapGraph [16] and CuSha [24], and competitive performance against nvGRAPH.

cuSTINGER. cuSTINGER [19] is designed for streaming graphs that evolve over time. Contrary to static graph processing frameworks, which require the entire graph to be re-transmitted to the GPU whenever an update is issued, cuSTINGER only transmits the updates themselves, which drastically decreases the amount of memory being transferred. The GPU then applies the update internally. cuSTINGER handles all the memory allocation, allowing the programmer to focus on designing the graph processing algorithms and not worrying about the implementation of the dynamic data structure. Even though cuSTINGER is mainly aimed at dynamic graphs, it also supports static graph processing. Multiple memory allocators, which influence the amount of memory being allocated for each vertex, are provided. When dealing with static graphs, a compact memory allocator can be chosen. For dynamic graphs, another allocator can be chosen that trades memory usage for better update performance.

cuSTINGER is a GPU extension of the Spatio-Temporal Interaction Networks and Graphs Extensible Representation (STINGER) data structure. STINGER's layout is based on the blocked adjacency lists data structure discussed in section 2.4. In cuSTINGER, the lists of edge blocks (per vertex) are replaced with a single adjacency array (per vertex). This allows for better edge locality within an adjacency list, for better memory utilization, and only requires one memory allocation per adjacency list (allows for a more efficient memory manager). When the arrays are allocated, some extra space is left for future inserts. cuSINGER takes advantage of an advanced memory manager that minimizes the number of calls made to the system memory allocation and deallocation functions, and groups adjacency lists into large chunks for better locality.

The following graph update operators are supported: edge insertion, edge deletion, vertex insertion, and vertex deletion. Given that the GPU is a highly parallel system, Oded Green and David A. Bader recommend that updates be grouped into batches.

Since 2017, cuSTINGER has been superseded by Hornet.

Hornet. Hornet [6] is a graph framework/data structure designed for dynamic sparse graphs and matrices. It is scalable with the input size and doesn't require memory reallocation when updates are issued. Compared to aimGraph [45] and cuSTINGER [19], it provides better memory usage, faster initialization, and faster updates. It only requires about 5% to 25% additional memory when compared to CSR, and outperforms Gunrock [42] in a suite of algorithmic benchmarks.

Hornet is composed of two layers, a user interface and an internal representation abstracted from the user. From the user's perspective, each vertex has two associated fields: the number of neighbors, and an adjacency list that references those neighbors.

To ensure fast updates, Hornet does not use the standard memory allocation function. Instead, a data manager implements this function with the usage of three data structures: 1) Block arrays that store adjacency lists, 2) A vectorized bit tree that can efficiently reclaim empty blocks, and 3) A B+Tree to manage the block-arrays. A block-array is simply an array composed of equally sized chunks. The vectorized bit tree is used to find empty

blocks within block-arrays at high rates. It does so by using a simple and fast lookup strategy. cuSTINGER on the other hand, requires computationally intensive searches for this purpose. The B+Tree is responsible for finding block-arrays that have free blocks. Each node of a B+Tree is a tuple <data, key>. The data field points to the block-array and the key stores the number of free blocks in that block-array. Given that searches in such a tree take $log(t)$ time, and that the number of block-arrays of a given size is relatively small, it is able to execute these searches very fast.

The following graph updates are supported: insertion and deletion of vertices, insertion and deletion of edges, and update of values of vertices and edges. Vertex insertions and deletions are expressed using sequences of edge insertions and deletions. Updates are grouped in batches to maximize throughput.

Dynamic Gunrock. Muhammad A. Awad et al. [4] proposed a dynamic graph data structure that utilizes a hash table per vertex, each storing a vertex's adjacency list. The authors claim that with this approach it is possible to gain significant speedups both in insertion (3.4 - 14.8x faster) and deletion (7.8x times faster) rates when compared to other state-of-the-art solutions like Hornet [6] and FaimGraph [46]. This is achieved given hash tables' fast query rates and their ability to ensure uniqueness while performing updates. Regarding the "front-end", the framework is integrated into Gunrock [42].

The proposed data structure is composed of a vertex dictionary and a set of adjacency lists (stored using hash tables). The vertex dictionary is represented using a fixed-size array, where vertices are indexed by vertex ID. For the adjacencies, one hash table is used per vertex. Each hash table is composed of a set of buckets. To calculate the number of necessary buckets, a user defined load factor combined with the graph's connectivity information is used in order to reduce memory allocation overhead. Additionally, a dynamic memory allocator is used whenever new slabs need to be allocated as a hash table grows.

The data structure is built on top of Slab Hash [3], a dynamic hash-table for the GPU. In order to ensure good performance, all operations are implemented using a Warp Cooperative Work Sharing (WCWS) strategy. In WCWS, each thread has an independent task assigned to it, but all threads within a warp cooperate with each other to collectively perform one independent task at a time. This type of strategy allows for GPU optimized memory access patterns.

Most graph updates only require appending or removing elements from the hash tables, allocating new slabs when needed. Vertex deletion requires a more sophisticated algorithm, given that it requires deleting all adjacencies that include the vertex at hand. To mitigate load imbalance, a queue is utilized to manage the deletion of vertices.

FaimGraph. FaimGraph [46] is a fast fully dynamic graph framework developed by the same authors of aimGraph [45], improving on the previous work, and assessing its limitations. It achieves high update rates while keeping a low memory footprint by

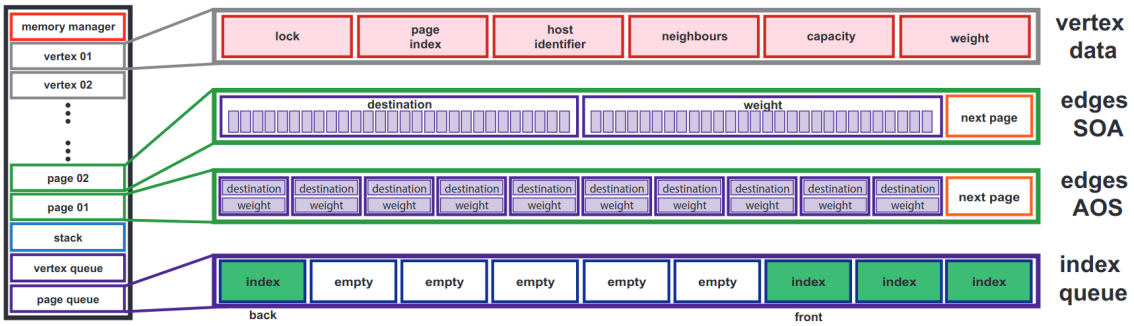


Figure 2.13: Faimgraph memory layout (taken from [46]).

taking advantage of memory management directly on the GPU. With this approach, it is possible to perform initialization in parallel (up to 300 times faster than previous work) and perform updates directly on the GPU. Queuing techniques are also used to reclaim memory, reducing the memory footprint, and allowing for occasional faster inserts. Unlike other GPU dynamic graph frameworks that only support edge insertions and deletions, like aimGraph [45], FaimGraph allows for fully dynamic graphs (vertex insertion and deletion).

During initialization, FaimGraph allocates a single large memory block on the GPU, to avoid CPU round-trips, which is than managed by the framework's memory manager. This memory block is mostly composed of the queuing structures, vertex data, and edge data. It also keeps track of memory sections, graph properties, free pages, and unused indices. Figure 2.13 shows a visualization of the device memory layout.

As stated before, the main mechanism to reclaim memory uses a set of index queues. Whenever a vertex is deleted, its index is enqueued to the respective index queue. The same applies, whenever an adjacency page is deleted. During resource allocation, threads first try to dequeue a deleted index. If successful, the memory relative to the index is reclaimed, otherwise, the vertex or page region is increased.

Vertex data is stored as a dynamically growing array of structures. The number of attributes for each vertex isn't limited, given that the memory manager can adapt to any vertex size. Storing vertices contiguously in this manner allows for coalesced memory accesses and simple indexing. Additionally, the vertex identifier used on the CPU is mapped to the device identifier. Edges are stored as a linked list of fixed-sized pages of edges. A larger page size allows for more efficient traversal, while small pages allow for less memory fragmentation and allocation overhead. A page size of 64 bytes was chosen by the authors.

Both vertex and edge insertions and deletions are performed as expected. Appending and removing values from the data structures, increasing the vertex or page regions when needed, and reclaiming memory using the index queues as discussed before. Shuffle operations are also utilized to ensure sorting order when performing updates.

According to the authors, FaimGraph outperforms multiple state-of-the-art frameworks, including Hornet [6], aimGraph [45], cuSTINGER [19] and GPMA [37], in a suit of benchmarks that test initialization times, graph update rates and algorithmic performance.

Previous Marrow-Graph. Previous work [31] has already studied the feasibility of developing a dynamic graph data structure and framework using Marrow (with an OpenCL backend). In this section, we provide a brief overview of the implementation and features of the previously developed Marrow-Graph.

The chosen data structure to store graphs was slotted pages [21] (discussed in greater detail in section 2.4) represented using marrow containers. The graph is stored using a set of fixed-sized slotted pages, each represented as a marrow array. Each slotted page stores the attributes of a set of vertices and their respective adjacency lists. Besides the slotted pages, two auxiliary data structures are used. One to keep track of large adjacency lists, represented as a marrow vector, and another, which is only present on the GPU, to store the device addresses of each slotted page.

Graph manipulations are performed on the CPU. The data structure is updated on the host and later synchronized to the device automatically by Marrow in a lazy manner. Edge insertions are performed by inserting the new edge in the respective slotted page. If the slotted page is full, a new one is allocated and marked as a large adjacency list. Vertex insertions are implemented by appending a new vertex to the last slotted page. If the last slotted page is full, a new one is allocated. Deletions follow a similar logic (if a page becomes empty, it is deleted), however, vertex deletions bring the added complexity of needing to delete all edges that include the deleted vertex. It should be noted that these manipulations can often result in reallocations, but only inside a given slotted page. This means that if the size of the pages is kept "small", the reallocations do not involve a large number of elements.

For graph processing, the solution uses the same set of primitives defined by Gunrock [42]: compute, advance, filter, and segmented intersection. These operations are further discussed in section 2.4. All of these operations were implemented using preexisting marrow functions combined with custom OpenCL functions, defined over marrow containers. With these basic operations, it is possible to implement practically all major graph processing algorithms. Because of time constraints, however, only BFS and SSSP were implemented. Another limitation found, is the inability to combine the compute operator with the other three, as proposed by Gunrock.

Regarding the performance of the manipulation operations, the construction of the graph can not be properly accelerated due to the nature of the slotted pages data structure and its representation using Marrow. Deletions are accelerated through the use of a range function. Insertions are relatively costly but do not require any previous knowledge about the number of vertices and edges.

When compared to Gunrock, Marrow-Graph shows competitive performance in the BFS benchmarks when considering the time of the graph construction plus the time of

Framework	Intended Hardware	Internal Data Structure	Dynamic	Unit	Programming Model
Gunrock	GPU	CSR or Other		VEC	BSP
D. Gunrock	GPU	Vector of HTs	✓	VEC	BSP
Hornet	GPU	Adjacency Lists & Trees	✓	VC	BSP
CuSTINGER	GPU	STINGER	✓	VC	BSP
FaimGraph	GPU	Blocked Adjacency Lists	✓	VC	BSP
CuSha	GPU	G-Shards and CWs		VC	BSP
MapGraph	GPU	CSR		VC	BSP
Medusa	GPU	Custom		VEC	BSP
LLAMA	MC CPU	CSR-Like	✓	VC	-
GraphLab	MC CPU	CSR-Like	✓	VC	-
Ligra	MC CPU	CSR-Like		VC	-
Ringo	MC CPU	HT of Vectors	✓	VC	-
Polymer	MC CPU	CSR-Like		VC	-
GraphIn	Cluster	CSR & Edge Lists	✓	VC	I-GAS
Chaos	Cluster	Edge-List		EC	Scatter-Gather
PowerGraph	Cluster	CSR-Like	✓	VC	GAS
GraphX	Cluster	Spark RDD	✓	VC	BSP
Pregel	Cluster	Adjacency Lists		VC	GAS

Table 2.2: Graph processing frameworks (MC: Multi-Core, VC: Vertex-Centric, EC: Edge-Centric, VEC: Vertex & Edge-Centric, HT: Hash-Table).

Framework	Public Repository	Compiles & Runs	Directed	Weighted	VVEA	Language	FDP
Gunrock	✓	✓	✓	✓		CUDA	
D. Gunrock	✓		✓	✓		CUDA	✓
Hornet	✓	✓	✓	✓		CUDA	
FaimGraph	✓	✓	✓	✓	✓	CUDA	✓

Table 2.3: GPU-Accelerated graph processing competitors (VVEA: Variable Vertex and Edge Attributes, FDP: Full Dynamic Processing).

the algorithm's execution time, but falls behind when discarding graph construction time. Compared to Hornet [6], it shows significantly slower execution times in all benchmarks. The author explains that Marrow-Graph is slower in most benchmarks due to the significant overhead associated with the advance operator to locate vertices in the correct pages. Another factor is related to the usage of OpenCL, while other frameworks, like Hornet and Gunrock, are implemented using CUDA, which is optimized to offer better performance with Nvidia graphics cards.

2.6.4 Conclusions

The potential competitors (the ones we will evaluate against) to our solution are found in Table 2.3. Currently we will only consider GPU-based solutions, but distributed solutions can also be relevant for future work that explores multi-GPU systems. Gunrock is the state-of-the-art static GPU-accelerated graph processing framework, so even though it is not a direct competitor, considering it only supports static graphs, it can be used as a baseline for algorithmic performance evaluation. Regarding its dynamic counterpart [4],

we were not able to successfully compile its publicly available implementation (found on Github as a branch in Gunrock's repository). For our direct competitors, this leaves us with Hornet and FaimGraph. cuSTINGER is not considered since it has been superseded by Hornet. All three competitors Gunrock, Hornet, and FaimGraph, are directed, weighted and are implemented using CUDA. FaimGraph, contrary to the other frameworks, allows for user-defined vertex and edge attributes. This can be useful for more specialized graphs. Another noteworthy observation is that even though Hornet supports GPU-accelerated processing over dynamic graphs, only a small number of algorithms have actually been implemented to run on the fully dynamic data structure.

3

Marrow-Graph

In this chapter, we present the design and implementation of Marrow-Graph. We start by defining a programming model (section 3.1), then present the corresponding designed interface (section 3.2), and finally describe the implementation of the library, including its data structure (section 3.3), and the various supported operators (section 3.4) and algorithms (section 3.5).

3.1 Programming Model

We will start by defining Marrow-Graph's programming model. Our goal while designing this model was to provide an interface that was as simple and expressive as possible. As discussed in Section 2.5, bulk-synchronous is the most natural model for GPU applications. More specifically, we believe that Gunrock's [42] programming model fits our purposes well. It allows one to highly optimize the small set of operations, and facilitate the development of algorithms utilizing these simple but fast operations. Another obvious benefit lies in the fact that Gunrock is an open-source project which already offers a wide array of algorithms implemented using this programming model.

Similarly to Gunrock [42], Marrow-Graph provides two traversal operators: *advance* and *filter*. Both these operators map an input frontier to an output frontier, where a frontier consists of the IDs of a subset of the graph's vertices. As previously mentioned in Section 2.6.3, the compute operator can be fused together with the other traversal operators. So instead of having a dedicated compute operator, each transversal operator takes a compute function as input as well as any necessary compute arguments. The absence of a dedicated segmented intersection operator will be addressed later.

The filter operator has the following specifications:

$$\texttt{filter}(\text{Frontier } F, \texttt{FilterFun } f, \texttt{FilterArgs...}\ a) : \texttt{Frontier}$$

where the first parameter F is the input frontier composed by a set of vertex IDs. The second parameter f, is a function that acts both as the compute operator and the filter selector. This function is applied to every vertex v referenced by F, and must return either

0 to exclude v from the filtered frontier or return 1 to include v in the filtered frontier. Any additional parameters of f, besides vector v, must be passed to the filter operator via the variadic arguments a. This allows the programmer to pass and process any auxiliary problem data. The output of the operator is frontier $F' \subset F$, containing only the vertex IDs corresponding to the vertices for which f returned 1.

We now move on to the advance operator which has the following specification:

$$\texttt{advance(Frontier}\, F\texttt{, ComputeFun}\, c\texttt{, ComputeArgs...}\, a\texttt{) : Frontier}$$

where, similarly to the filter operator, the first parameter is the input frontier composed by a set of vertex IDs. The second parameter c expresses the compute function. This function is applied to every edge e traversed during the advance operator, i.e. all the outgoing edges of the vertices referenced by the input frontier. Given that vertex and edge information is stored separately in Marrow-Graph, c must receive as parameters a source vertex v_s, a destination vertex v_d, and the edge e connecting v_s to v_d. Any additional parameters of c, besides v_s, v_d, and e, must be passed to the advance operator via the variadic arguments a. The output of the advance operator is a frontier F' containing the IDs of all the unique neighbors of the vertices referenced by F.

In Section 2.6.3 we described the segmented intersection as an operator that can be applied over two frontiers. The latest versions of Gunrock however do not include a dedicated segmented intersection operator. Rather, a segmented intersection function, which can be invoked in the middle of a compute function, is provided. This means that the segmented intersection function can be applied over two vertices instead of two frontiers, i.e. it intersects the neighbors of the two vertices. Given that the compute function that is passed to the advance operator operates over edges composed of two vertices, the idea is that a segmented intersection can be executed during an advance. This is equivalent to performing a segmented intersection between a frontier F and the frontier F' obtained from performing an advance over F. According to Gunrock's authors, having such a segmented intersection function instead of a dedicated operator, is both more useful, and more efficient, given that we can perform an advance and a segmented intersection simultaneously. We decided to follow the same philosophy and incorporate a segmented intersection function which can be invoked in the middle of an advance's compute function c. The segmented intersection function has the following specification:

$$\texttt{segint(Graph}\, g\texttt{,Vertex}\, v_a\texttt{, Vertex}\, v_b\texttt{, IntersectFun}\, t\texttt{, IntersectArgs...}\, a\texttt{) : int}$$

where the first parameter includes the graph data required to perform the intersection (more on this later). The second and third parameters v_a and v_b correspond to the vertices to intersect. The fourth parameter t is a function that allows the user to express some logic to be performed over each intersection (we will later see that some algorithms require this). Function t must receive as parameters the two vertices being intersected v_a and v_b, and the intersected vertex v_i. Any additional arguments can be passed using the variadic

```
template<typename vertex_attributes>
struct vertex {
    idx_t idx;
    std::size_t degree;
    idx_t adjacency_list;
    idx_t adjacency_list_end;
    vertex_attributes attributes;
};

template<typename edge_attributes>
struct edge {
    idx_t dst;
    edge_attributes attributes;
};

template <typename vertex_attributes, typename edge_attributes>
class graph {

    using Vertex = vertex<vertex_attributes>;
    using Edge = edge<edge_attributes>;

    idx_t add_vertex(vertex_attributes attributes) ;
    bool remove_vertex(idx_t idx) ;
    bool add_edge(idx_t src, idx_t dst, edge_attributes attributes) ;
    bool add_edge_batch(edge_insertion_batch<edge_attributes>& batch) ;
    bool remove_edge(idx_t src, idx_t dst) ;
    bool remove_edge_batch(edge_deletion_batch<edge_attributes>& batch) ;
    bool edit_edge(idx_t src, idx_t dst, idx_t new_dst) ;
    vector<idx_t> get_connection(idx_t idx) ;
    std::size_t get_degree(idx_t idx) ;
    std::size_t get_number_of_vertex() ;
    std::size_t get_number_of_edges() ;
    void sort() ;

    template <bool return_frontier = true, bool balanced = true>
    template <typename compute_fun, typename... compute_args>
    vector<idx_t> advance(vector<idx_t>& frontier, compute_fun& cfun, compute_args&... cargs);

    template <typename filter_fun, typename... filter_args>
    vector<idx_t> filter(vector<idx_t> &frontier, filter_fun& ffun, filter_args&... fargs);
};
```

Listing 5: Graph interface.

arguments a. The output of the segmented intersection function is an integer representing the total number of intersections between v_a and v_b.

3.2 Interface

We now present Marrow-Graph's interface that implements the previously described programming model and also provides a set of graph manipulation operations (the complete C++ abstract class can be found in Listing 25 of Appendix A). Graphs represented in Marrow-Graph are directed and optionally weighted (using the optional attributes). As we can see in Listing 5, three structs are provided. The `vertex` struct includes a vertex's ID, its degree, pointers to the start and end of its adjacency list, and a templated field for storing custom vertex attributes. The `edge` struct includes the ID of the destination vertex (the source vertex is implied) and a templated field for storing custom edge attributes. Finally, the `graph` class offers a set of methods for manipulating and queering the graph,

```
1  template<typename edge_attributes>
2  struct edge_insertion_batch {
3      std::size_t size;
4      std::map<idx_t, std::vector<edge<edge_attributes>>> edges;
5
6      void add(idx_t src, idx_t dst, edge_attributes attributes);
7  };
```

Listing 6: Edge insertion batch.

as well as the advance and filter operators.

Regarding graph manipulation functions, we offer a method to create a new vertex `add_vertex` which returns the ID of the generated vertex. Then using the generated IDs, the user can invoke the methods `remove_vertex`, `add_edge`, `remove_edge`, and `edit_edge`. Additionally, an `add_edge_batch` method and a `remove_edge_batch` method are provided to perform optimized insertions and deletions of batched edges. As seen in Listing 6, an edge batch is composed by a `std::map`, whose key is the ID of a source vertex, and value is a vector of outgoing edges. This allows Marrow-Graph to perform edge insertions and deletions optimally by iterating over consecutive edges that share the same source vertex.

3.2.1 Operators

As discussed previously in Section 3.1, the advance and filter operators operate over frontiers and parameterized functions. To express these concepts in Marrow-Graph's interface, we use `marrow::vectors` of IDs to represent the former, and resort to templated functors and variadic templated arguments for the latter. Note that even though in Listing 5 the advance and filter operators have a return type `vector<idx_t>` to make it clear that these functions return a new frontier, in the actual implementation of Marrow-Graph, these functions have `auto` as their return type. This allows Marrow to return an expression that evaluates to the desired container, rather than returning the actual container. This is of course much more efficient, given that containers are heavier objects than Marrow expressions.

Advance. The `advance` method (Lines 35 to 37 of Listing 5) receives as input a templated compute function and its corresponding arguments (expressed with a variadic template). The `compute_fun` type must be a functor whose function call operator `()` adheres to the following signature:

```
void operator()(Vertex& sv, Vertex& dv, Edge& e, Args&... a) { ... }
```

As specified by our model, the advance's compute function describes an operation that is performed over a single edge traversed during advance. Marrow-graph ensures that this function is actually applied over all the outgoing edges of the input frontier. The parameters `sv`, `dv` and `e` represent a single outgoing edge: `sv` contains the data and attributes of the source vertex, `dv` contains the data and attributes of the destination

```
1  struct vertex_attribute {};
2  struct edge_attribute { int weight; };
3
4  struct compute_total_weight_fun {
5      marrow_function
6      void operator()(Vertex &src, Vertex &dst, Edge &edge, int* weights) {
7          marrow::atomic::add(&weights[dst.idx], edge.weight);
8      }
9  };
10
11 graph<vertex_attribute, edge_attribute> g = ...;
12 vector<int> weights(g.get_number_of_vertex());
13 fill(weights, 0);
14
15 idx_t vertex_id = ...;
16 vector<idx_t> frontier = { vertex_id };
17 vector<idx_t> advanced_frontier = g.advance(frontier, compute_total_weight_fun(), result(weights));
18 int total_weight = reduce<plus>(weights);
```

Listing 7: Advance example.

vertex, and `e` contains the attributes of the edge connecting these vertices. Besides these three arguments, the compute function can also receive and operate over any number of auxiliary arguments. These arguments must either be basic data types, Marrow containers or `result` wrapped Marrow containers. If the compute function changes the state of a Marrow container, it can be passed to the advance operator using the `result` wrapper, informing Marrow-Graph that the container is a result that is computed on the device. We will discuss this in more detail in section 3.3.1. Note however, that although the auxiliary arguments we pass to the `advance` function can be Marrow containers, the type of the arguments of the `cfun` function must be their kernel type counterparts, similarly to how we define and use Marrow functions. For example, we might pass a `vector<int>` as an auxiliary argument to advance, but receive a `int*` in the `cfun`. Given that the advance operator is executed on the device, the device code must operate over basic data type pointers rather than complex Marrow containers.

While developing Marrow-Graph, we found out that some algorithms use the advance operator without requiring the resulting advanced frontier. These algorithms only require the compute operator to be applied over all outgoing edges. For this reason, the `compute` function has an additional `return_frontier` template (set to true by default), which can be set to false, in order to run an optimized version of the operator which returns an empty frontier. Besides the `return_frontier` template, we can see that there is also a `balanced` boolean template. This template only has an effect whenever `return_frontier` is set to false, allowing the optimized advance operator to run either in a balanced or unbalanced manner. We will further discuss this in Section 3.4.2.

Listing 7 shows an example of how the advance operator might be used to compute the sum of the weights of all the outgoing edges of a given vertex. For the sake of simplicity, we will assume this graph has not been subjected to vertex deletions which could affect the required size of the `weights` container. In Lines 11-13 we start by instantiating a graph `g` and a vector `weights` with all values initialized to 0. In Lines 15-16 we define a

```
1  struct vertex_attribute { char active = 1; };
2  struct edge_attribute {};
3
4  struct remove_inactive_fun {
5      marrow_function
6      void operator()(Vertex &v) {
7          return v.active ? 1 : 0;
8      }
9  };
10
11 graph<vertex_attribute, edge_attribute> g = ...;
12 vector<idx_t> frontier = { ... };
13 vector<idx_t> filtered_frontier = g.filter(frontier, remove_inactive_fun());
```

Listing 8: Filter example.

frontier with a single vertex (the vertex whose neighbor weights we will be summing). In Line 17 we call the `advance` operator over the frontier, passing it an instance of the `compute_total_weight_fun` functor and the `weights` vector. Looking at Lines 4-9, we can see that this functor follows the previous signature rules, and atomically sums an edge's weight to the `weights` array at the index corresponding to the destination vertex. This means that after the advance operator finishes, `weights` will contain all the weights of the outgoing edges in the corresponding positions of the destination vertices. All the other positions of the container will remain equal to 0. Finally, In Line 18 we sum all the values of `weights` with a Marrow reduce, and obtain the total sum of weights. You may have noticed that instead of passing the `weights` vector directly in Line 17, we actually pass `result(weights)`. As previously mentioned, this informs Marrow-Graph that the `weights` argument is a result that is computed during the `compute` function.

Filter. The `ffun` parameter and corresponding `fargs` parameter from the filter method (Lines 39 to 40 of Listing 5) follows similar restrictions to the previously discussed compute function and arguments. The main difference is that filter does not operate over edges, but rather directly over the vertices of the input frontier. Therefore, the `filter_fun` type must be a functor whose function call operator `()` receives a single vertex as input and any number of auxiliary arguments:

```
int operator()(Vertex& v, Args&... a) { ... }
```

Additionally, the return type of the operator must be an integer, instead of void, to allow one to express the filter selection logic described by our model.

Listing 8 shows how the filter operator can be used to filter-out inactive vertices from a frontier. In this example, we define a graph whose `vertex_attributes` includes a `active` field. In Line 13 we apply the filter operator over an input frontier and pass it an instance of the `remove_inactive_fun` functor. This functor returns 1 if the input vertex is active and 0 otherwise. This means that the filter operator will return a frontier that only includes the vertices that have the `active` field set to 1.

```
1  template<typename vertex_attributes, typename edge_attributes,
2      typename on_intersection_fun, typename... on_intersection_arguments>
3  marrow_function
4  int segmented_intersection(graph_bal_t<vertex_attributes, edge_attributes> &graph,
5      vertex<vertex_attributes>& vertex_a,
6      vertex<vertex_attributes>& vertex_b,
7      on_intersection_fun& on_intersection,
8      on_intersection_arguments&... on_intersection_args);
```

Listing 9: Segmented intersection function.

Segmented Intersection. In order to provide the segmented intersection specified by our model, we defined a marrow function (that can be invoked in the middle of a compute function that is executed on the device) whose signature can be seen in Listing 9. The function returns an integer with the number of intersections and receives as input: a graph struct, the two intersecting vertices, and a `on_intersection_fun` alongside its `on_intersection_arguments`. The graph struct `graph_bal_t` found in the code snippet, is specific to the blocked adjacency graph implementation. In the future, this struct can easily be templated to provide a more generic function that supports multiple implementations. Regardless, although this parameter is required to compute the intersection, it can be mostly ignored by the user. As stated before, even though the function returns the total number of intersections, some algorithms require performing some logic for each intersection. This is done via the `on_intersection_fun`. The `on_intersection_fun` type must be a functor whose function call operator `()` adheres to the following signature:

```
void operator()(Vertex& va, Vertex& vb, Vertex& vi, Args&...) { ... }
```

Like our model specifies, this function operates over a single intersection, and its then Marrow-Graph's job to ensure this method is called for every intersection. Parameters `va` and `vb` represent the two vertices being intersected, while `vi` represents the intersected vertex (a neighbor that `va` and `vb` have in common).

Listing 10 shows how the segmented intersections function can be used to compute the total number of intersection between the vertices of a frontier F and its advanced frontier F'. In Lines 16-19 we define a graph, a vector to store intersection counts, and a frontier. In Line 20 we apply the advance operator over the frontier passing it an instance of a `compute_intersection_count_fun` functor and all the necessary arguments, including an instance of the `on_intersection_fun`. The `vertex_intersection_count` is wrapped as a `result` since it is a result manipulated by the compute function. In this example, since we do not want to perform any logic for each intersection, the body of the `on_intersection_fun` functor can stay empty. Note that our compute function defined In Lines 7-14 has a slightly different signature than the one we established before. The difference is the first parameter `graph`. Given that it is mandatory to pass this struct to the `segmented_intersection` function, a Marrow-Graph compute function can optionally contain the graph struct as its first parameter. In Line 11 we call the `segmented_intersection` function passing it the graph struct, the source and destination

```
1  struct on_intersection_fun {
2      marrow_function
3      void operator()(Vertex& vertex_a, Vertex& vertex_b, Vertex& intersection_vertex) {
4      }
5  };
6
7  struct compute_intersection_count_fun {
8      marrow_function
9      void operator()(Graph& graph, Vertex &src, Vertex &dst, Edge &edge,
10         int* vertex_intersection_count, on_intersection_fun& on_intersect) {
11         int nintersections = segmented_intersection(graph, src, dst, on_intersect);
12         marrow::atomic::add(&vertex_intersection_count[dst.idx], nintersections);
13     }
14 };
15
16 graph<vertex_attribute, edge_attribute> g = ...;
17 vector<int> vertex_intersection_count(g.get_number_of_vertex());
18 fill(vertex_intersection_count, 0);
19 vector<idx_t> frontier = { ... };
20 vector<idx_t> advanced_frontier = g.dvance(
21     frontier, compute_intersection_count_fun(), result(vertex_intersection_count), on_intersection_fun());
22 int total_intersection_count = reduce<plus>(vertex_intersection_count);
```

Listing 10: Segmented intersection example.

vertices (which are provided to any advance's compute functor), and the `on_intersect` functor. The result is stored in a local variable `nintersections`. In Line 12 we atomically add the intersection count to the `vertex_intersection_count` container in the corresponding index of destination vertex. Finally, in Line 22, we compute the total intersection count with a Marrow reduction over the `vertex_intersection_count` vector.

Similarly to Gunrock, in order to achieve an efficient and fast implementation of the `segmented_intersection` function, it is assumed that the adjacencies of the intersecting vertices v_a and v_b are sorted. The motives for this are discussed in Section 3.4.3. If a user wants to ensure that these adjacencies are sorted before performing a compute with a segmented intersection, the `sort` function (Line 33 of Listing 5) can be invoked. Additionally, for the blocked adjacency graph implementation, when initializing a graph, the user can specify that the graph should track unsorted adjacency lists. This adds some slight overhead to the graph manipulation functions, but improves the `sort` function performance drastically, as it only sorts the adjacency lists that might have become unsorted.

3.3 Data Structure

We will now discuss the data structure chosen for the internal implementation of Marrow-Graph. We decided to base our data structure on FaimGraph [46], as it has proven to be a versatile yet highly efficient data structure for storing dynamic graphs. FaimGraph is comprised by a vertices vector, a set of blocked adjacency lists, and two deleted index queues as seen in Figure 2.13. In Marrow-Graph, we use the same sub-data structures, with the addition of two index vectors storing the links between blocks. Figure 3.1 shows a representation of the data structure indicating each corresponding Marrow collection.

```
template<typename vertex_attributes, typename edge_attributes, std::size_t block_size>
class graph_bal : public graph<vertex_attributes, edge_attributes> {
    ...
    std::size_t nvertex = 0;
    std::size_t nedges = 0;
    vector<vertex<vertex_attributes>> vertices;
    vector<array<edge<edge_attributes>, block_size>> blocks;
    vector<idx_t> block_links;
    vector<idx_t> block_links_reversed;
    std::queue<idx_t> deleted_vertices;
    std::queue<idx_t> deleted_blocks;
    bool track_unsorted_adjacencies;
    std::set<idx_t> unsorted_adjacency_lists;
}
```

Listing 11: graph_bal fields.

The vertices are stored using a `marrow::vector` of `vertex`s. We chose a Marrow vector, given that vertices must be accessible both on the host, to perform updates and query the graph, and on the device, to execute the operators. Additionally, the vector data structure allows for insertions and deletions of vertices.

The adjacencies are stored using a `marrow::vector` of fixed size `marrow::array`s of `edge`s. Each edge stores the destination vertex and optional attributes. Given that the adjacency lists are supposed to be stored using fixed-size blocks, the most fitting container is a variable-sized Marrow vector of fixed-sized arrays. This way, we can easily insert and remove new blocks as needed, and still achieve good memory locality given that consecutive blocks (or arrays in this case) will be stored contiguously in memory. Similarly to the vertices, we use a Marrow container instead of a `std` container, since the adjacency lists must be accessible both on the host and on the device.

Block links are stored using `marrow::vector`s of indexes. Once again, a Marrow vector allows us to insert and delete new block links whenever a block is added or removed and makes the container accessible both on the host and device. The advance operator requires the block links to be accessible on the device in order to iterate over the adjacency lists comprised of multiple blocks.

Finally, the deletion queues are stored with basic `std::queue`s. We do not use Marrow containers as the deletion queues are only accessed on the host (all graph updates are performed on the host).

Listing 11 shows the actual class and fields used to define the Blocked Adjacency List (BAL) implementation of Marrow-Graph. Besides the aforementioned data structures, two counters `nvertex` and `nedges` are also included to keep track of the total number of vertices and edges in the graph, and a `unsorted_adjacency_lists` set is used to track unsorted adjacency lists. If a graph is flagged to track unsorted adjacencies, whenever edges are inserted or deleted, Marrow-Graph verifies if the corresponding adjacency list becomes unsorted, and if it does, the adjacency list's source vertex ID is added to the `unsorted_adjacency_lists` set. We will see in Section 3.3.3 how this allows for more efficient graph sorting.

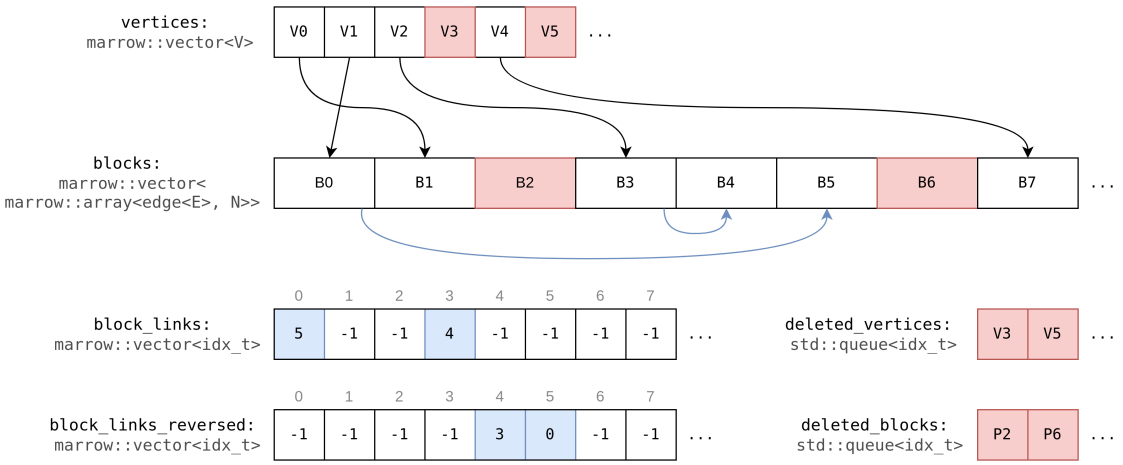


Figure 3.1: Marrow-Graph's data structure.

In terms of indexing, each element of the `vertices` vector stores an index to the first and last adjacency blocks. The `block_links` and `block_links_reversed` vectors can then be used to traverse the blocked adjacency lists in either direction. Looking at Figure 3.1 we can see that, for example, vertex `V1` has an adjacency list that starts at block `B0` and ends at block `B5`. Looking at the `block_links` container, we can see that indeed the link at position 0 stores the index 5. Then the link at position 5 stores the index -1. A value of -1 indicates that there does not exist another link, i.e. the adjacency list ends.

3.3.1 Host-Device Synchronization

Initially, the graph is solely stored on the host. Once an operator (advance or filter) is invoked, all the necessary containers are allocated and synchronized to the device. Given that the operators are implemented using Marrow functions and Marrow skeletons, this allocation and synchronization is ensured naturally by Marrow. All graph manipulations (adding, removing, and editing edges and vertices) are performed on the host. Once again, Marrow tracks any changes and synchronizes the dirty containers automatically whenever an operator is executed.

Figure 3.2 shows the memory layout once a graph has been loaded and synchronized with the GPU. As we can see, most containers are simply stored as contiguous vectors on the host, and have a replica stored on the device. One more nuanced container is the `vector<array<N,T>>`, which we use to store blocks (or adjacencies). Assuming a fixed array size of N and a variable vector size of S, a `vector<array<N,T>>` is stored on the host using two data structures. A contiguous block of data of length $N * S$, and a vector of Marrow arrays of length S. These arrays only contain metadata and point to the contiguous memory block. This allows user-friendly manipulations of the data structure on the host while allowing for efficient memory transfers and memory accesses to/on the device. If the host would only store a vector of arrays containing the data, transferring the whole vector to the device would require N separate memory transfers, rather than a single large

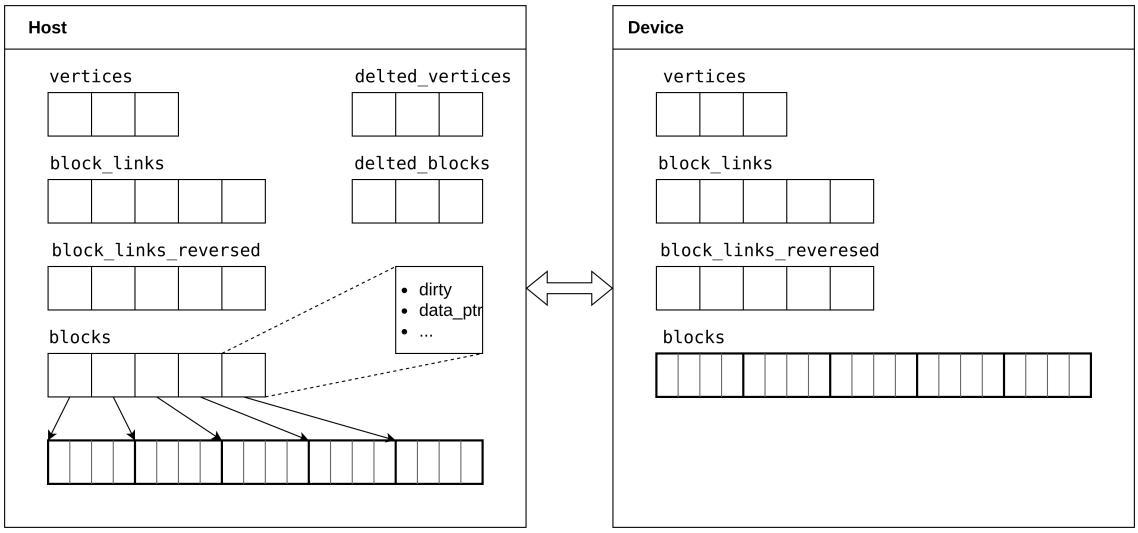


Figure 3.2: Host/device memory layout.

memory transfer of $N * S *$ sizeof(T) bytes. The vector of arrays that we store on the host, also allows Marrow to track singular dirty arrays. When Marrow synchronizes the vector to the device, if it's been already allocated, only the dirty arrays are synchronized, instead of the entire vector.

While developing algorithms using Marrow-Graph, the user might utilize Marrow containers to store useful problem data and process this data during a Marrow-Graph operator. For example, the `weights` vector used in Listing 7. As we discussed before in Section 2.3, when we are defining a Marrow function, we can specify if its parameters are input, output or input & output parameters. When we specify that a given parameter container is meant for output, this informs Marrow that the function will change the data of that container. The result is that the container's replica (the version of the container stored on the device) is marked as dirty. This is important given that when we later access the container on the host, it is first properly synchronized. While developing Marrow-Graph, we were of course careful to mark any parameters of the internal Marrow functions as output parameters if they were manipulated by the function. Even so, given that Marrow-Graph allows the user to pass to the operators, templated functors, and arguments, it is impossible to track which containers might be manipulated by said functors. Moreover, these functors can not be expressed using regular Marrow functions, given that they are invoked by device kernels (during an advance or filter), rather than by host functions. This means we can not specify at the functor level what parameters should be output parameters. For example, in Listing 7, the `compute_total_weight_fun` functor, passed to the advance operator, manipulates the data of the `weights` container, but we can not mark the `weights` parameter as an output parameter, as this functor cannot be defined using a `marrow::function` (since it is invoked by a device kernel). To overcome this, whenever the programmer invokes an advance or filter operator with a functor that manipulates Marrow containers, these containers can be passed to the operators as `result(container)`

Algorithm 1: Add Vertex

Input: vertex_attributes attributes
Output: idx_t

```
1  idx ← graph.nvertex;
2  if ¬ is_empty (graph.deleted_vertices) then
3  |  idx ← pop (graph.deleted_vertices);
4  end
5  v ← vertex<vertex_attributes>();
6  v.idx ← idx;
7  v.degree ← 0;
8  v.adjacency_list ← −1;
9  v.adjacency_list_end ← −1;
10 v.attributes ← attributes;
11 push_back (graph.vertices, v);
12 graph.nvertex ← graph.nvertex +1;
13 return idx;
```

(like we see in Line 17 of Listing 7). The `result` class is a container wrapper which informs Marrow-Graph that the container will be manipulated by the compute or filter operator. In practice, this means Marrow-Graph will mark the container's replica as dirty after the operator has been executed. If the container is later accessed on the host, it will first be properly synchronized. Note that if a filter or compute operator manipulates a Marrow container that is only useful on the device, i.e. it is never later accessed on the host, the usage of the `result` wrapper is optional.

3.3.2 Updates

We will now describe the implementation of the various Marrow-Graph's update functions. All of these functions manipulate the graph data structure on the host, and all synchronization with the device is handled by Marrow.

Add Vertex. To create a new vertex (Algorithm 1), we start by checking if the `deleted_vertices` queue has any elements (Line 2). If it does, we pop a value to the `idx` variable (Line 3), otherwise, we leave `idx` equal to the next available index (Line 1). A new vertex object is then created with the corresponding ID and attributes (Lines 5-10) and is pushed to the `vertices` vector (Line 11). The total number of vertices is updated, and the generated ID is returned (Lines 12-13).

Remove Vertex. To remove a vertex given its ID (Algorithm 2), we push the ID to the `deleted_vertices` queue (Line 1), delete all the blocks of its adjacency list (Lines 2-9), and decrement the number of vertices (Line 10). In order to remove all the blocks from the adjacency list, we get the first block index from the vertex's `adjacency_list` field (Line 2), and then use the `block_links` to iterate through the rest of the blocks (Line 7) until an index of -1 is reached (Line 3). Removing a single block involves adding its index to the

Algorithm 2: Remove Vertex

Input: idx_t idx

```
1  push (graph.deleted_vertices, idx);
2  block_idx ← vertices [idx ].adjacency_list;
3  while block_idx ≠ -1 do
4  |   push (graph.deleted_blocks, block_idx);
5  |   block_links_reversed [block_idx ] ← -1;
6  |   prev_block_idx ← block_idx;
7  |   block_idx ← graph.block_links[block_idx ];
8  |   graph.block_links[prev_block_idx ] ← -1;
9  end
10 graph.nvertex ← graph.nvertex −1;
```

Algorithm 3: Add Edge

Input: idx_t src, idx_t dst, edge_attributes attributes

```
1  new_edge ← edge<edge_attributes>();
2  new_edge.dst ← dst;
3  new_edge.attributes ← attributes;
4  src_vertex ← graph.vertices[src ];
5  block_offset ← src_vertex.degree % block_size;
6  block_idx ← src_vertex.adjacency_list_end;
7  if block_offset == 0 then
8  |   block_idx ← get_new_block (src_vertex, block_idx);
9  end
10 graph.blocks[block_idx ][block_offset ] ← new_edge;
11 src_vertex.degree ← src_vertex +1;
12 graph.nedges ← graph.nedges +1;
```

`deleted_blocks` queue (Line 4), and setting the corresponding block links to −1 (Lines 5 and 8). Note that this algorithm does not take into account incoming edges. This is, any vertices that might have outgoing edges to the removed vertex, will continue to do so. We decided not to remove incoming edges given that this is an expensive operation. One workaround is implementing soft deletes by marking removed vertices as deleted using the vertex attributes.

Add Edge. To insert a new edge given its source and destination vertices, as well as its edge attributes (Algorithm 3), we start by creating a `new_edge` object with the corresponding destination vertex and attributes (Lines 1-3). We then get the block index and block offset where the edge should be inserted (Lines 4-9). The block index is obtained from the `adjacency_list_end` field from the vertex (Line 6). The offset is computed using the vertex's degree and block size. If the block offset is 0 (Line 7), this means a new block must be created, as the last block is already full. We do so using the function `get_new_block` (Line 8). Finally, we insert the edge in the correct block and offset (Line 10), increment the vertex's degree, and increment `nedges` (Lines 11-12).

The `get_new_block` (Algorithm 4) starts by checking if there is an available block in the

Algorithm 4: Get New Block

Input: Vertex src_vertex, idx_t prev_block_idx
Output: idx_t

```
1  new_block_idx ← graph.blocks.size;
2  if ¬ is_empty (graph.deleted_blocks) then
3  |   new_block_idx ← pop (graph.deleted_blocks);
4  |   graph.block_links[new_block_idx ] ← -1);
5  |   graph.block_links_reversed[new_block_idx ] ← prev_block_idx;
6  end
7  else
8  |   push_back (graph.blocks, _);
9  |   push_back (graph.block_links, -1);
10 |   push_back (graph.block_links_reversed, prev_block_idx);
11 end
12 if src_vertex.degree == 0 then
13 |   src_vertex.adjacency_list ← new_block_idx;
14 end
15 else
16 |   graph.block_links[prev_block_idx ] ← new_block_idx;
17 end
18 src_vertex.adjacency_list_end ← new_block_idx;
19 return new_block_idx;
```

Algorithm 5: Edit Edge

Input: idx_t src, idx_t dst, idx_t new_dst
Output: bool

```
1  block_idx, offset, found ← find_edge (src, dst);
2  if ¬ found then
3  |   return false;
4  end
5  graph.blocks[block_idx ][offset ].dst ← new_dst;
6  return true;
```

`deleted_blocks` queue (Line 2). If so, it pops the deleted block index into `new_block_idx` (Line 3), and updates the corresponding block links (Lines 4-5). Otherwise, a new empty block (a Marrow array of edges) is pushed to the `blocks` vector (Line 8), and new links are added to the block links containers (Lines 9-10). In Line 12 we check if the degree of the vertex is 0, meaning this is the first block of the adjacency list. If so, the `adjacency_list` field of the vertex is set to the new block index, otherwise, the block link of the previous block in the adjacency is updated. Finally, the `adjacency_list_end` is updated to the new block index, and the index is returned (Lines 18-19).

Edit Edge. To edit an edge given its source and destination vertices, and the new destination vertex (Algorithm 5), we start by finding the edge block and block offset using the `find_edge` function (Line 1). If the function did not find the edge (Line 2), we return `false` to indicate that the edit failed. Otherwise, we edit the edge at block `block_idx`

Algorithm 6: Find Edge

Input: idx_t src, idx_t dst
Output: idx_t, idx_t, bool

```
1  block_idx ← graph.vertices[src ].adjacency_list;
2  offset ← 0;
3  found ← false;
4  while block_idx ≠ −1 ∧ ¬ found do
5  |  while offset < graph.block_size ∧¬ found do
6  |  |  e ← blocks [block_idx ][offset ];
7  |  |  if e.dst == dst then
8  |  |  |  found ← true;
9  |  |  end
10 |  |  else
11 |  |  |  offset ← offset +1;
12 |  |  end
13 |  end
14 |  if ¬ found then
15 |  |  block_idx ← block_links [block_idx ];
16 |  end
17 end
18 return block_idx, offset, found;
```

at position `offset`, so that the destination vertex now references `new_dst` (Line 5), and return `true`.

The `find_edge` function (Algorithm 6), iterates through the adjacency list of the source vertex until an edge to `dst` is found, or the adjacency list ends. Similarly to the vertex removal function (Algorithm 2), the iteration starts at the block referenced by the source vertex's `adjacency_list` field (Line 1), and iterates through the blocks using the `block_links` container (Line 15), until an index of −1 is reached or the edge has been found (Line 4). Additionally, the offsets of all the blocks are iterated from 0 to `block_size`, or until the edge has been found (Line 5). Finally the block index, offset and `found` boolean are returned (Line 18).

Remove Edge. To remove an edge given its source and destination vertices (Algorithm 7), we start by obtaining the source vertex and checking if its degree is 0 (Line 2). If so we return `false` to indicate the deletion failed. Otherwise, we find the edge to remove using the `find_edge` function (Line 5). If the edge was not found, we once again return `false` (Line 7). Otherwise, we obtain the last edge of the source vertex's adjacency list (Lines 9 and 10). If the destination vertex is different from the last edge (Line 11), this means we are removing an edge in the middle of the adjacency list. In this case, instead of moving a bunch of edges to the left to fill the gap of the edge, we are bout to remove, we simply move the last edge to the slot of the edge we want to remove (Line 13). Removing the last edge then simply means decrementing the source vertex's degree (Line 33) and decrementing `nedges` (Line 32). But if the last edge is at a block offset of 0 (Line 15), this means that besides decrementing the degree, we also have to remove the last block of

Algorithm 7: Remove Edge

Input: idx_t src, idx_t dst
Output: bool

```
1  src_vertex ← graph.vertices[src ];
2  if src_vertex.degree == 0 then
3  |  return false;
4  end
5  block_idx, offset, found ← find_edge (src, dst);
6  if ¬found then
7  |  return false;
8  end
9  last_edge_offset ← modulo (src_vertex.degree - 1, graph.block_size);
10 last_edge ← graph.blocks[src_vertex.adjacency_list_end][last_edge_offset ];
11 if dst ≠ last_edge.dst then
12 |  // Copy last edge to the slot of edge we want to delete;
13 |  graph.blocks[block_idx ][offset ] ← last_edge;
14 end
15 if last_edge_offset == 0 then
16 |  // Delete last block;
17 |  push (graph.deleted_blocks, src_vertex.adjacency_list_end);
18 |  if src_vertex.adjacency_list_end == src_vertex.adjacency_list then
19 |  |  // Adjacency list only has a single block that needs to be removed;
20 |  |  block_links [src_vertex.adjacency_list] ← -1;
21 |  |  block_links_reversed [src_vertex.adjacency_list_end] ← -1;
22 |  |  src_vertex.adjacency_list ← -1;
23 |  |  src_vertex.adjacency_list_end ← -1;
24 |  end
25 |  else
26 |  |  prev_adjacency_list_end ← src_vertex.adjacency_list_end;
27 |  |  src_vertex.adjacency_list_end ← block_links_reversed
   |  |    [src_vertex.adjacency_list_end];
28 |  |  block_links_reversed [prev_adjacency_list_end ] ← -1;
29 |  |  block_links [src_vertex.adjacency_list_end] ← -1;
30 |  end
31 end
32 nedges ← nedges −1;
33 src_vertex.degree ← src_vertex.degree −1;
34 return true;
```

the adjacency list (given that the last edge is the only edge left in the block). In this case, we start by pushing the index of the last block to the `deleted_blocks` queue (Line 17). We then check if the block we are removing is the only block in the adjacency list. This can be done by verifying if the start and end indexes of the source vertex's adjacency list are the same (Line 18). If so we update the corresponding block links (Lines 20-21), and set the start and end indexes of the adjacency list of the source vertex (`adjacency_list` and `adjacency_list_end`) to −1 (Lines 22-23). Essentially deleting the whole adjacency list. Otherwise, we simply update the `adjacency_list_end` field (Line 27) and update the corresponding block links (Lines 28-29).

Algorithm 8: Filter

Input: vector<idx_t> frontier, filter_compute_fun filter_compute, filter_compute_args... fc_args
Output: vector<idx_t>

```
1 flags ← graph_bal_flags (frontier, graph.vertices, filter_compute, unwrap (fc_args)...);
2 filtered_frontier ← marrow::filter (frontier, flags);
3 dirty_results (fc_args)...;
4 return filtered_frontier;
```

3.3.3 Sorting

Like we discussed in Section 3.2.1, sorting the graph's adjacency lists can be useful to apply segmented intersections. We now discuss the `sort` function implementation. The `sort` function starts by verifying if the graph has been flagged to track unsorted adjacencies. If so, it iterates over the `unsorted_adjacency_lists` set and individually sorts each one. Otherwise, it iterates over every single adjacency list of the graph and sorts each one. The implementation of the `graph_bal_sort_adjacency_list` function can be found in Listing 23 of Appendix A. We will not give a detailed description of this code given that it is just an implementation of the standard quick sort algorithm, but applied to a blocked linked list. We chose this algorithm because of its good average case time complexity of $O(n \log(n))$.

3.4 Operators

In this section, we describe the implementation of the two transversal operators, and their corresponding compute and segmented intersection functions, for the BAL implementation of Marrow-Graph.

3.4.1 Filter

The filter operator (Algorithm 8) receives as input a frontier represented by a `marrow::vector` of indexes, a filter/compute functor `filter_compute_fun`, and a variadic set of arguments `fc_args`. It starts by invoking the `graph_bal_flags` function, which receives all of the filter's parameters as well as the `vertices` vector and returns a `marrow::vector` of flags with the same size as the input frontier. Each flag has either a value of 0 or 1 and is the result of applying the `filter_copmute` functor to the frontier. The `fc_args` variadic arguments are passed through the `unwarp` function to unwrap any possible `result` arguments (mentioned in section 3.3.1). In Line 2, we apply a `marrow::filter` to the frontier, passing it the `flags` vector, which returns a `marrow::vector` containing the subset of vertex IDs for which `flags` is 1. In Line 3, we mark any result arguments as dirty, and finally in Line 4, we return the filtered frontier.

The C++ implementation of the `graph_bal_filter` function can be found in Listing 12. The `filter_compute` parameter is represented using a templated functor, and the `fc_args`

```
1  namespace func {
2      template <typename vertex_attributes,
3          typename filter_compute_fun, typename... filter_compute_args>
4      struct graph_bal_flags : function</*...*/>
5      {
6          /* ... */
7          marrow_function
8          int operator()(idx_t frontier_idx, vertex<vertex_attributes>* vertices,
9              filter_compute_fun filter_compute, filter_compute_args... comp_args) {
10
11             vertex<vertex_attributes>& vert = vertices[frontier_idx];
12             return filter_compute(vert, comp_args...);
13         }
14     };
15 }
16
17 template<typename vertex_attributes, typename filter_compute_fun, typename... filter_compute_args>
18 static inline auto graph_bal_flags(
19     vector<idx_t>& frontier,
20     vector<vertex<vertex_attributes>>& vertices,
21     filter_compute_fun& filter_compute,
22     filter_compute_args&... fc_args)
23 {
24
25     static func::graph_bal_flags<vertex_attributes,
26         filter_compute_fun, kernel_type_t<filter_compute_args>...> singleton;
27     return singleton (frontier, vertices, filter_compute, fc_args...);
28 }
```

Listing 12: Graph BAL flags.

with variadic templated parameters. The function simply creates a singleton instance of the `graph_bal_flags` Marrow function (Line 25 and 26) and applies all the input parameters to the singleton (Lines 27). The `graph_bal_flags` Marrow function is defined to operate over every frontier index. In Line 11, the vertex for the given frontier index is obtained, and then the `filter_compute` functor is invoked with the vertex and the variadic arguments. Given that the `filter_compute` functor will return either 0 or 1, by returning the result of the functor invocation, this `marrow::function` will generate a container of flags with the same size as the input frontier. In practice, this function will result in a kernel that is executed by a number of threads equal to the number of elements in the input frontier. Each thread executes the `filter_compute` functor, passing it the correct vertex object, as well as the auxiliary variadic arguments, just like we described in our model 3.1.

3.4.2 Advance

The advance operator (Algorithm 9) receives as input a frontier represented by a Marrow `vector` of indexes, a compute functor `compute`, and a variadic set of arguments `comp_args`. It starts by invoking the `graph_bal_get_degrees` function, which given the frontier and the `vertices` container, returns a Marrow `vector` containing the degrees of all the vertices referenced by the frontier. The `comp_args` variadic arguments are passed through the `unwarp` function to unwrap any possible `result` arguments (mentioned in section 3.3.1). We will not give a detailed description of the `graph_bal_get_degrees` function as it simply calls a Marrow function, that for each frontier index, returns the degree of the

Algorithm 9: Advance

Input: vector<idx_t> frontier, compute_fun compute, compute_args... comp_args
Output: vector<idx_t>

```
1 degrees ← graph_bal_get_degrees (frontier, graph.vertices);
2 degrees_scan ← marrow::scan<plus> (degrees);
3 advanced_frontier_size ← degrees_scan [size (degrees_scan)];
4 frontier_size ← size (frontier);
5 advanced_frontier, unique_flags ← graph_bal_advance (frontier, degrees_scan,
    graph.vertices, graph.blocks, graph.block_links, frontier_size, advanced_frontier_size,
    compute, unwrap (comp_args)...);
6 unique_frontier ← marrow::filter (advanced_frontier, unique_flags);
7 dirty_results (comp_args)...;
8 return unique_frontier;
```

corresponding vertex, generating the degrees vector. In Line 2, we perform a Marrow scan to cumulatively sum all the degrees. This is useful since the `degrees_scan` can be used to index where the neighbors of each vertex of the input frontier end/start in the advanced frontier (see Figure 3.3 for clarification). Additionally, the last position of the degrees scan is equal to the size of the advanced frontier (Line 3). In Line 5, we invoke the `graph_bal_advance` function. The function receives as input, the frontier, the degrees scan, the advanced frontier size, the compute function and arguments, and an assortment of necessary graph data. The function returns the advanced frontier but also returns a vector of flags with the same size as the advanced frontier. This vector indicates which IDs in the advanced frontier are duplicates or not. A value of 0 in the `unique_flags` vector indicates that the corresponding position in the advanced frontier is a duplicate and should not be included in the final result. In Line 6, we apply a Marrow filter over the advanced frontier passing it the `unique_flags` vector, which returns a frontier containing only the unique elements of the original `advanced_frontier`. In Line 7 we mark any result arguments as dirty, and finally in Line 8, the `unique_frontier` is returned. Even though removing duplicates has significant overheads, the performance benefits that are achieved in algorithms that require performing many iterations invoking the advanced operator, greatly outweigh these overheads. Additionally, not removing duplicates leads to issues when trying to implement some algorithms.

The C++ implementation of the `graph_bal_advance` function can be found in Listing 13. For the sake of conciseness, we omitted the parameters of the function, as well as the arguments passed to the corresponding Marrow function (Line 14). However, these can be implied by Algorithm 9 and the Marrow function found in Listing 14. In Lines 1 and 2, we define two static Marrow vectors: `_unique_flags` is used by the advance operator later, and `_duplicates` is used internally to compute the `_unique_flags` vector. The reason these are global, rather than local, variables, is to avoid spending time reallocating these vectors when the advance operator is called many times sequentially. The `_duplicates` vector's size is equal to the number of vertices in the graph (Line 8). This means it is

```
1  static vector<int> _duplicates;
2  static vector<int> _unique_flags;
3
4  template<typename vertex_attributes, typename edge_attributes, std::size_t block_size,
5      typename compute_fun, typename... compute_args>
6  static inline auto graph_bal_advance(/*...*/)
7  {
8      _duplicates.resize(vertices.size());
9      _unique_flags.resize(advanced_frontier_size);
10     fill(_duplicates, 0);
11
12     static func::graph_bal_advance<vertex_attributes, edge_attributes,
13         compute_fun, kernel_type_t<compute_args>...> singleton;
14     auto /*vector<idx_t>*/ advanced_frontier = singleton (/*...*/);
15     return advanced_frontier;
16 }
```

Listing 13: Graph BAL advance.

an expensive vector to allocate, and likely does not need to be reallocated often between advance invocations (unless the number of vertices of the graph is constantly changing). This is less relevant for the `_unique_flags` whose size is dependent on the frontier size, which often changes between advance invocations. In Lines 8-10, we resize the `_duplicates` and `_unique_flags` vectors (in case they already have the correct size this has minimal overhead), and fill `_duplicates` with zeros. In Lines 12-14 we create a singleton instance of the `graph_bal_advance` Marrow function and invoke it with all the necessary parameters. Finally, in Line 15 we return the advanced frontier. The unique flags do not have to be returned given that they are a global variable and can be accessed by the caller function.

The implementation of the `graph_bal_advance` Marrow function can be found in Listing 14. Once again, for the sake of conciseness, we omitted some template parameters and other noncrucial details. We left the last template parameters (Line 4) visible, to demonstrate that parameters `duplicates` and `unique_flags` are marked as output parameters. They are marked as output parameters because we update these containers during the function. Additionally, `duplicates` is also an input parameter (therefore marked as `inout`) given that it is initially updated to be filled with zeros. Contrary to the previously defined Marrow functions, `graph_bal_advance` implements a `function_with_coordinates` and is not applied to all the elements of a specific container. Rather, this function's result size is specified manually (omitted from this listing) to be equal to the `advanced_forntier_size`. This means that this function will result in a kernel that is executed by a number of threads equal to the advanced frontier size, and each thread (which has access to its coordinates) computes and returns a single vertex ID of the advanced frontier.

In Lines 13-16, we start by performing a binary search to find the index of the thread ID `tid` in `degrees_scan` and use this index to obtain the corresponding source vertex. Given that each thread is assigned to an element of the advanced frontier, which contains the neighbors of the vertices of the input frontier, we must first compute its source vertex. For clarification, Figure 3.3 shows an example of a frontier, its advanced frontier, the degrees

```
1  template<typename vertex_attributes, typename edge_attributes,
2      typename compute_fun, typename... compute_args>
3  struct graph_bal_advance :
4    public function_with_coordinates</*...*/, inout<int*>, out<int*>, compute_fun, compute_args...> {
5      /* ... */
6      marrow_function
7      idx_t operator()(coordinate_t* coordinate, idx_t* frontier, std::size_t* degrees_scan,
8        vertex<vertex_attributes>* vertices, edge<edge_attributes>* edges,
9        std::size_t block_size, idx_t* block_links, std::size_t frontier_size,
10       std::size_t advanced_frontier_size, int* duplicates, int* unique_flags,
11       compute_fun compute, compute_args... comp_args) const {
12
13         const std::size_t tid = coordinate[0];
14         int frontier_vertex_n = binary_search(tid, frontier_size, degrees_scan);
15         idx_t vertex_idx = frontier[frontier_vertex_n];
16         Vertex& vertex = vertices[vertex_idx];
17
18         std::size_t neighbour_n = tid - (frontier_vertex_n == 0 ?
19             0 : degrees_scan[frontier_vertex_n - 1]);
20         idx_t block_idx = vertex.adjacency_list;
21         std::size_t block_n = neighbour_n / block_size;
22         std::size_t block_offset = neighbour_n % block_size;
23         for (int i = 0; i < block_n; i++)
24             block_idx = block_links[block_idx];
25         Edge _edge = edges[block_idx * block_size + block_offset];
26
27         int prev = marrow::atomic::add(&duplicates[_edge.dst], 1);
28         if(prev > 0)
29            unique_flags[tid] = 0;
30         else
31             unique_flags[tid] = 1;
32
33         if constexpr (std::is_invocable<decltype(compute),
34           Graph&, Vertex&, Vertex&, Edge&, compute_args&...>::value) {
35             Graph graph = { block_size, vertices, edges, block_links };
36             compute(graph, vertex, vertices[_edge.dst], _edge, comp_args...);
37         }
38         else {
39             compute(vertex, vertices[_edge.dst], _edge, comp_args...);
40         }
41         return _edge.dst;
42     }
43 };
```

Listing 14: Graph BAL advance marrow function.

of the frontier, and the scan of the degrees. Assuming `tid` is equal to 2, in order to compute the value of the advanced frontier at index 2, we must first obtain the corresponding source vertex. Looking at Figure 3.3, we can see that this is vertex `V2`. The way we obtain `V2`, is by searching for the `tid` in the degrees scan. Even though the value 2 is not present in the degrees scan vector, we know it falls between value 1 at index 0, and value 4 at index 1. Given that the values of the degrees scan point to the end of each set of adjacencies in the advanced frontier, the search returns index 1. Index 1 contains the source vertex `V2`. Since we know that the degrees scan is a sorted list, we use a binary search for maximum efficiency. Its code can be found in Listing 21 of Appendix A.

With the source vertex, we can now obtain the outgoing edge assigned to a thread. We start by computing the neighbor offset (Line 18) by subtracting the degree, of the previous source vertex, from `tid`. For example, returning to Figure 3.3, assuming `tid` is equal to 2,

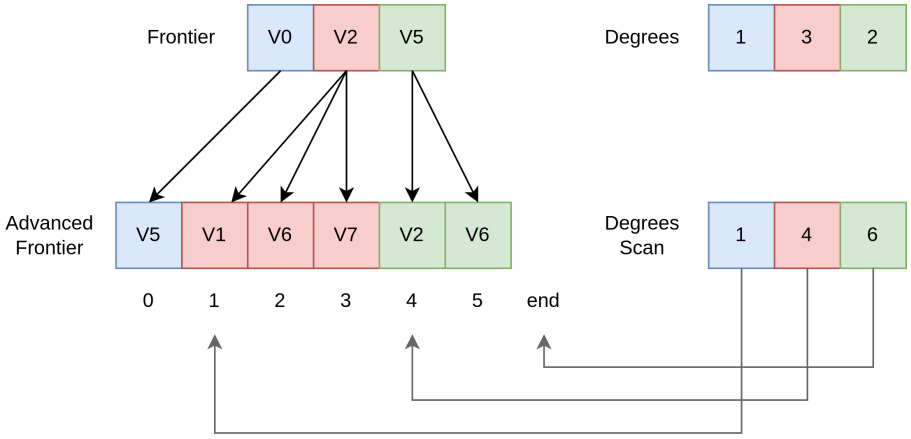


Figure 3.3: Advanced frontier diagram.

the offset of the neighbor `V6` of the source vertex `V2` can be computed by subtracting the degree of the source vertex `V0` from `tid`. We can confirm that $\texttt{tid} - \texttt{deg(V0)} = 2 - 1 = 1$ is indeed the offset of `V6` in the adjacency list of `V2`. With the neighbor offset `neighbour_n`, we can compute the block index and inner-block-offset where the edge is stored (Lines 20-22). In Lines 23-25 we iterate through the block list until we reach the desired block, and obtain the edge. In the end of the function, the destination of the edge is returned (Line 41).

Lines 28-31 update the `unique_flags` container. Each time an outgoing edge is obtained, we atomically increment the `duplicates` value for that edge's destination vertex. If the previous value of `duplicates` is greater than 0, i.e. there already exists an entry in the advanced frontier for that destination vertex, we set `unique_flags` at `tid` to 0.

In Lines 33-40, we invoke the compute operator, passing it the source vertex, destination vertex, edge, and variadic compute arguments, just like we specified in our model (Section 3.1). Additionally, we check whether or not a `graph_bal_t` struct should also be passed to the compute functor (Line 33-34). As we saw in Listing 10, we sometimes need to pass this struct to the compute functor, in order to be able to use the `segmented_intersection` function. We therefore verify what parameters the functor is expecting, and pass them accordingly.

Frontier-less Advance. We previously mentioned in Section 3.2 that it is possible to invoke an optimized variant of the advance operator that does not return the advanced frontier. Let us call this variant the frontierless advance. It was also mentioned that both a balanced and unbalanced version of the frontier-less advance can be chosen. Starting with the balanced frontier-less advance, this variant operates identically to the default advance operator but does not require dealing with duplicate removal. This means that a set of instructions and device-executed operators can be omitted. In the advance Algorithm 9, we omit the last filter (Line 6) which removes duplicates. In the advance function (Listing 13), we omit Lines 8-10 which setup the `unique_flags` and `duplicates` containers. And finally in the advance Marrow function (Listing 14), we omit Lines 27-31

which deal with updating the `unique_flags` and `duplicates` containers.

Even though the balanced frontier-less advance operator allows for a significant optimization compared to the default advance operator, for sparse graphs (which should have small adjacency lists), it is possible to optimize even further. We do so in the unbalanced frontierless advance. Note that even in the balanced frontier-less advance, we perform at least four host-device communications. The first when we invoke the `graph_bal_get_degrees` function (Algorithm 9), the second when we perform a scan over those degrees, the third when we access the last element of the scan result (which is stored on the device), and the fourth when we actually execute the `graph_bal_advance` function. The unbalanced frontier-less advance reduces these host-device communications to a single one. The actual invocation of the advance function. Given that we do not need to generate an advanced frontier, we do not necessarily need to calculate the `advanced_frontier_size` (Line 3 of Algorithm 9). We, therefore, do not compute the degrees vector (Line 1), do not perform the scan (Line 2), and do not access the scan result on the host (Line 3). Instead, we only invoke a `graph_bal_unbalanced_frontierless_advance` function. This function instantiates and invokes a Marrow function that, contrary to the default Marrow advance Marrow function, is executed by a number of threads equal to the size of the input frontier. This means that, instead of binary searching for the source vertex, each thread loops through its neighbors, and invokes the `compute` functor for each neighbor. The complete implementation of this Marrow function can be found in Listing 24 of Appendix A. For dense graphs that might have very long adjacency lists of widely varying sizes, this can become quite inefficient, since there will be a large load imbalance between threads. However, for sparse graphs, with potentially small adjacency lists, the reduced host-device communication can overcome the load imbalance overhead. We were able to achieve up to x8 algorithmic speedups using the unbalanced frontier-less advance, when compared to the default advance operator, and up to x1.7 algorithmic speedups using the balanced frontier-less advance.

3.4.3 Segmented Intersection

As we already discussed Section 3.2, the `segmented_intersection` is a device function that intersects the (sorted) neighbors of two input vertices (Listing 9). The function iterates through the adjacency lists of both the vertices, and whenever it encounters a shared neighbor, it increments an intersection counter and invokes the `on_intersection` functor. At the end of the function, the counter is returned. All the necessary data to traverse the adjacency lists is accessible through the `graph` struct. Given that it is assumed that the adjacency lists of both vertices `vertex_a` and `vertex_b` are sorted, the function makes use of two indexers, one for each vertex's adjacency lists. It then increments both indexers if there is an intersection, otherwise, it only increments the indexer which is indexing a neighbor with a smaller vertex ID. This ensures that all possible intersections are caught, while only having to iterate through each adjacency exactly once. The full implementation

of the `segmented_intersection` device function can be found in Listing 22 of Appendix A.

3.5 Algorithms

Besides the basic operators, Marrow-Graph includes a suite of implemented algorithms. The algorithms that have been implemented so far are:

- BFS: Traversal algorithm that starts from a given source vertex and explores all its neighbors at the present depth level before moving on to vertices at the next depth level. This algorithm is useful to find short paths between nodes (not necessarily the shortest). Some applications include network routing and web crawling.

- SSSP: Algorithm that finds the shortest paths from a given source vertex to all other vertices in the graph. Marrow-Graph follows Dijkstra's implementaiton. Some applications include network routing and path-finding in robotics and video-games.

- Triangle Counting (TC): Algorithm that determines the number of triangles (three nodes that are mutually connected) in a graph. This metric is useful for understanding the clustering coefficient of the graph, which provides insights into the network's topology and density.

- PR: Algorithm that is used to rank the nodes in a graph based on their importance. It assigns a score to each node, indicating its relative significance within the graph, with a node's score being tied to the number of nodes referencing it. This algorithm was originally used by Google to rank webpages.

- Sparse Matrix–Vector Multiplication (SpMV): Algorithm that multiplies a sparse matrix (represented by the graph) with a vector. SpMV is used in various applications including scientific computations and machine learning.

Most of the logic of these algorithms was based of Gunrock's implementations given that a similar programming model is used. Of course, the specific syntax to express these algorithms is quite different and is explored in Section 4.2. We will not give an exhaustive description of each algorithm's implementation, but rather, will only focus on a subset that we found worth discussing.

3.5.1 SSSP

We start by presenting the SSSP algorithm, as it follows the most common algorithmic pattern, of advancing and filtering a frontier until it is empty while updating problem data during each advance phase. The `sssp` function (Listing 15) starts by initializing a set of Marrow vectors. A `distances` vector to store the result of the algorithm, and two `visited` and `revisit` vectors to track already visited vertices. The `dist_fill` functor used In Line 5, fills all the elements of the container with the `FLT_MAX` value, except for the `source` index, which is set to 0. In Lines 11-12 we instantiate the compute and filter

```
1  vector<float> sssp(graph_bal<vertex_attributes, edge_attributes> &graph, idx_t source) {
2
3      auto nvertex = graph.get_number_of_vertex();
4      vector<float> distances(nvertex);
5      distances.fill_on_device(dist_fill(source));
6      vector<int> visited(nvertex);
7      visited.fill_on_device(0);
8      vector<int> revisit(nvertex);
9      visited.fill_on_device(0);
10
11     shortest_path<vertex_attributes, edge_attributes> sp;
12     remove_completed_paths<vertex_attributes> rm;
13     vector<idx_t> frontier;
14     frontier.push_back(source);
15
16     while (frontier.size() != 0) {
17         vector<int> advanced_frontier = graph.advance(frontier, sp, result(distances), visited, revisit);
18         vector<int> filtered_frontier = graph.filter(advanced_frontier, rm, visited, revisit);
19         frontier = filtered_frontier;
20     }
21     return distances;
22 }
```

Listing 15: SSSP function.

functors, and In Lines 13-14 we create a frontier with a single vertex, the input source vertex. The actual algorithm (Lines 16-20), runs until the frontier is empty (Line 16). In each iteration, we advance the frontier with the `sp` functor, and filter the advanced frontier with the `rm` functor. The `distances` container is wrapped as a `result` given that it is computed during the compute function, and might later be accessed by the host. Finally, once the frontier is empty, the `distances` vector is returned.

The compute functor `shortest_path` can be found in Listing 16. The compute operator is executed over every edge traversed during the advance of a frontier. This means that during every `shortest_path` compute function, we update the distances vector, taking into account the weights of the edges being traversed. We start by calculating the new total distance to the neighbor (the destination vertex of the traversed edge) In Line 6. We then set the distance to that neighbor as the min value between the distance that is already computed, and the new distance we calculated (Line 7). We update the `visited` vector to signal that this source vertex has been visited (Line 10), and, in case the newly calculated distance has replaced the old distance to the neighbor vertex, we mark the neighbor vertex to be revisited (Line 11-12). Given that there is a new shortest path to `dst`, we must reconsider this vertex in the next iteration.

The filter functor `remove_completed_paths` can be found in Listing 17. It starts by checking if the degree of the source vertex is 0, in which case, it can be excluded from the filtered frontier (Line 5-6). This is not necessary for the correctness of the algorithm, but makes it slightly more efficient, given that neighbor-less vertices will not have any impact in the advance phase. We then check if the vertex has been marked for a revisit (Line 8), in which case, we set both the visited and revisit flags to 0. Finally, we include the vertex from the filtered frontier if the visited flag is set to 0, otherwise, the vertex is

```
template <typename vertex_attributes, typename edge_attributes>
struct shortest_path {
    marrow_function void operator()(vertex<vertex_attributes>& src, vertex<vertex_attributes>& dst,
      edge<edge_attributes>& edge, float *distances, int *visited, int *revisit) {

        float distance_to_neighbor = distances[src.idx] + edge.attributes.weight;
        marrow::atomic::min(&distances[dst.idx], distance_to_neighbor);
        float recover_distance = distances[dst.idx];

        marrow::atomic::exch(&visited[src.idx], 1);
        if (recover_distance == distance_to_neighbor)
            marrow::atomic::exch(&revisit[dst.idx], 1);
    }
} ;
```

Listing 16: SSSP compute function.

```
template <typename vertex_attributes>
struct remove_completed_paths {
    marrow_function int operator()(vertex<vertex_attributes>& v, int *visited, int *revisit) {

        if(v.degree == 0)
            return 0;

        if(revisit[v.idx]) {
            marrow::atomic::exch(&revisit[v.idx], 0);
            marrow::atomic::exch(&visited[v.idx], 0);
        }
        return visited[v.idx] ? 0 : 1;
    }
};
```

Listing 17: SSSP filter function.

excluded. This ensures that we do not revisit vertices that have already been visited, and whose shortest distance has not been updated in the last advance phase. Additionally, this ensures that the frontier converges to empty, which is necessary for the algorithm loop to end.

3.5.2 Triangle Count

We now present the TC algorithm given it is the main application and motivator for the segmented intersection function. The `tc` function (Listing 18) starts by initializing a vector to store the triangle counts of all the vertices (the result), with all values set to 0. We then initialize a frontier that contains the IDs of all the vertices of the graph. Assuming there are no deleted vertices, we can do so using the `iota` function that generates a vector of N counting integers $[0, 1, 2, \ldots, N]$. In Lines 8-9 we instantiate the compute functor `trig_count` and the on intersection functor `trig_count_seg`. In Line 11 we perform a single advance over the frontier containing all the vertices in the graph, which computes the triangle count for each one. Since we do not require the resulting advanced frontier, we call the optimized unbalanced frontier-less advance operator, by setting the corresponding boolean templates to false. Finally, In Line 12-13, we sum all the triangle counts with a Marrow reduce, and return the result. Note that the result is divided by 3. Because

```
1  int tc(graph_bal<vertex_attributes, edge_attributes> &graph) {
2
3      auto nvertex = graph.get_number_of_vertex();
4      vector<int> vertex_triangle_count(nvertex);
5      vertex_triangle_count.fill_on_device(0);
6      vector<idx_t> frontier = iota(nvertex);
7
8      triangle_count_fun<vertex_attributes, edge_attributes> trig_count;
9      on_intersection_fun<vertex_attributes> trig_count_seg;
10
11     graph.advance<false,false>(frontier, trig_count, trig_count_seg, vertex_triangle_count);
12     int tcount = reduce<sum<int>>(vertex_triangle_count) / 3;
13     return tcount;
14 }
```

Listing 18: TC Function.

```
1  template <typename vertex_attributes, typename edge_attributes>
2  struct triangle_count_fun {
3      marrow_function void operator()(/*...*/) {
4          if(dst.idx > src.idx) {
5              segmented_intersection(graph, src, dst, on_intersec, vertex_triangle_count);
6          }
7      }
8  };
```

Listing 19: TC compute function.

the `triangle_count` stores, for each vertex v, the number of triangles in the graph that include v, we must divide the total sum of triangle counts by the number of vertices in a triangle, i.e. 3.

The compute functor `triangle_count_fun` can be found in Listing 19. We omitted the functor's parameters for conciseness. As we can see, the only thing the functor does is call the `segmented_intersection` function. It passes all the necessary parameters, but most importantly, passes the `on_intersec` functor, which contains all the important logic of the algorithm, and the `vertex_triangle_count` container that is manipulated in `on_intersec`. The only logic that is present in the compute functor, is the conditional statement (Line 4), which prevents the algorithm from counting repeated triangles. Given that this algorithm is implemented for undirected graphs, the same triangle can be counted twice in each direction. We prevent this by only considering triangles in the direction of ascending indices.

The on intersection functor `on_intersection_fun` can be found in Listing 20. The function starts by checking whether the intersection `intersection_vertex` does not coincide with one of the intersecting vertices `vertex_a` or `vertex_b`. If not, the triangle count of the intersection vertex is incremented atomically.

To clarify why this works, we present an example using Figure 3.4. Looking at this undirected graph, it is obvious that it contains 3 triangles. When we perform an advance over the frontier containing all the vertices in the graph, we are essentially traversing all the edges of the graph. We can see that when we traverse, for example, the edge

```
1  template <typename vertex_attributes>
2  struct on_intersection_fun {
3      marrow_function int operator()(
4        vertex<vertex_attributes>& vertex_a,
5        vertex<vertex_attributes>& vertex_b,
6        vertex<vertex_attributes>& intersection_vertex,
7        int* vertex_triangle_count) {
8
9          if (vertex_a.idx != intersection_vertex.idx && vertex_b.idx != intersection_vertex.idx) {
10             marrow::atomic::add( &(vertex_triangle_count[intersection_vertex.idx]), 1);
11         }
12     }
13 };
```

Listing 20: TC on intersection function.

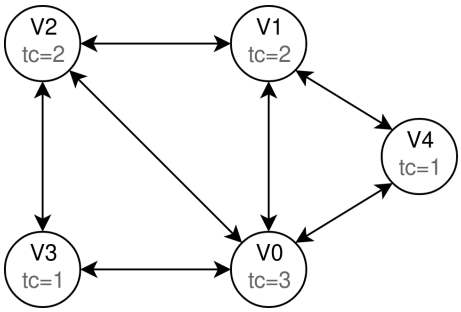


Figure 3.4: Triangle count example.

$\{V_0, V_4\}$, a segmented intersection is performed between these two vertices, resulting in a single intersection (a single neighbor in common) V_1. With this, the triangle count of V_1 is incremented to 1. This count is incremented again in the traversal of edge $\{V_0, V_2\}$, since V_1 is also a common neighbor between these vertices. This results in a total triangle count of 2, like we can see in the figure. Turning our attention to vertex V_4, we can see it has a total triangle count of just 1. The traversal of edge $\{V_0, V_1\}$ results in a `tc` increment for V_4 (since V_4 is a common neighbor of V_0 and V_1), but the traversal of the edge $\{V_1, V_0\}$ does not. It does not because V_0 has a lesser ID than V_1, and In Line 4 of Listing 19, we can see that we only perform segmented intersections between edges whose destination vertex ID is greater than the source vertex's ID. In practice, each vertex ends up with a triangle count equal to the number of triangles that include that specific vertex. V_4 is part of a single triangle, V_1 is part of two triangles, V_0 is part of three triangles, etc. If we sum all the triangle counts and divide them by 3, we get $(3 + 2 + 2 + 1 + 1)/3 = 3$, the number of triangles in the graph.

3.6 Marrow Adaptation

Due to some technical challenges with Marrow during the development of this thesis, an adapted and simplified runtime was developed and used to implement Marrow-Graph. Because of time constraints, we were not able to run our solution on the original and complete version of Marrow. This is something we would like to tackle in future work.

Nevertheless, the solution is working correctly and showing promising results. In this section, we present the differences and relevant technical details regarding the adaptation, which we will call *lmarrow*.

lmarrow's syntax is a direct subset of Marrow's, supporting the main smart containers: `array`, `vector`, `vector<array>` and `scalar`, the main skeletons: `map`, `filter`, `reduce` and `scan`, and the same generic `function` primitive. Just like Marrow, *lmarrow* allocates and synchronizes the smart containers automatically in a lazy manner, and executes the skeletons and functions on the device. The main differences in the adaptation are:

1. The lack of Marrow-expressions, meaning that skeletons always return the resulting containers and not the corresponding expression. Therefore, skeleton composition is not supported.

2. The lack of a scheduler that manages the execution of kernels asynchronously and deals with data dependencies. *lmarrow* invokes most device operations sequentially on the default stream.

3. *lmarrow* only supports CUDA, and does not include multiple backends.

For the specific needs of Marrow-Graph, these drawbacks fortunately are not too concerning in terms of performance. Regarding the first, Marrow-Graph does not directly make use of skeleton composition. Regarding the second, given the bulk-synchronous nature of Marrow-Graph's programming model, the potential for asynchronous kernel management is limited (although not nonexistent). Regarding the third, we planned on only using the CUDA backend during this thesis.

Given the aim of this thesis, we will not give a detailed description of *lmarrow*'s implementation. Additionally, since we plan to eventually run our solution on the original complete version of Marrow, we left all the code ready to be run on the non-adapted library. For example, we used the `auto` keyword whenever a Marrow expression can be captured instead of a container. This means that while using *lmarrow*, these variable types will be containers, but once Marrow is integrated, these variables will be typed with the lighter-weight Marrow expressions. For the rest, from a user perspective, every thing stays the same, and almost all information provided in Section 2.3 is still relevant.

4

Evaluation

In this section, we evaluate Marrow-Graph against a set of state-of-the-art graph processing frameworks. The main goals of this evaluation are 1. Verifying the correctness of Marrow-Graph's operations and algorithms. 2. Assessing Marrow-Graph's expressiveness and simplicity. 3. Evaluating Marrow-Graph's performance.

4.1 Correctness

To verify the correctness of Marrow-Graph, we try to answer the following questions:

- Do all graph mutation operations result in the expected graph state?
- Do all the traversal operators output the expected result?
- Do all of Marrow-Graph's algorithms output the expected result?

To this end, multiple unit tests (using Google Test) were developed during Marrow-Graph's development. Additionally, we compare Marrow-Graph's algorithmic outputs with Gunrock's. The developed unit tests include: 1. Testing the advance and filter operators for both small/trivial examples, and using large graphs and complex compute and segmented intersection operators. 2. Testing all the graph mutation functions. 3. Testing the loading of large batches of edge and vertex insertions and deletions and ensuring a correct final state.

To ensure the correctness of the developed algorithms, we executed all the algorithms on Marrow-Graph and Gunrock, using real-life-large graphs, namely RoadNet-CA (Table 4.3) and hollywood-2009 (Table 4.3), and compared their respective outputs. For Breadth-First Search (BFS), Single Source Shortest Path (SSSP), and Triangle Counting (TC), we were able to achieve the exact same output results as Gunrock. Page Rank (PR) deals with many floating point operations, for this reason, very similar output results were obtained, but with some small deviations. To ensure that the deviations were indeed just related to floating-point rounding issues, we compared all the outputted elements and ensured that their absolute differences were bellow `FLT_EPSILON`. `FLT_EPSILON` is

a commonly used floating-point value to access floating-point similarity. Its defined by the standard library as the "difference between 1 and the least value greater than 1 that is representable" [1]. For Sparse Matrix–Vector Multiplication (SpMV) we were able to achieve very similar results, but again with slight deviations. We were not able to achieve an output result with absolute differences smaller than `FLT_EPSILON` for SpMV. The exact cause for this has not been completely determined, but might also be related to floating point rounding errors.

We can conclude that Marrow-Graph's manipulation functions and operators are indeed following the expected behavior, and that all the algorithms are outputting mostly identical results to the current state-of-the-art Graphics Processing Unit (GPU)-accelerated graph processing framework.

4.2 Expressiveness and Simplicity

To access the expressiveness and simplicity of Marrow-Graph, we try to answer the following questions:

- Can Marrow-Graph express most graph-analytics algorithms?
- Can algorithms and programs be expressed in Marrow-Graph in a simple and concise manner?

4.2.1 Evaluation Methodology

For this evaluation, we compare the implementations of a set of common graph algorithms against Gunrock, FaimGraph, and Hornet. As metrics to answer the previous questions, we will consider:

Implementable Algorithms: Number of common algorithms that have been implemented, or can theoretically be implemented.

Conciseness: How much code and boilerplate is required to express an algorithm.

Readability: How readable or obfuscated the implementations of algorithms are.

For the first, we will compare the total set of implemented algorithms by each framework. For the second, we will compare the total number of useful lines of code that comprise each framework's implementation. When we refer to useful lines of code, this means discarding any empty lines, comments, and code unrelated to the algorithm (for example debug logs and validation functions). For readability, we will directly compare some of the implementation details and syntax of each framework.

Algorithm	Gunrock	Hornet	FaimGraph	Marrow-Graph
Betweeness Centrality	✓	✓	✓	imp.
Breadth-First Search	✓	✓	✓	✓
Clustering Coefficient		✓	✓	
Connected Components		✓	✓	
Graph Coloring	✓			imp.
Geolocation	✓			imp.
Hyperlink-Induced Topic Search	✓			imp.
Katz Centrality		✓		
K-Core Decomposition	✓			imp.
Minimum Spanning Tree	✓			
PageRank	✓	✓	✓	✓
Local Graph Clustering	✓			
Sparse-Matrix Vector Multiplication	✓	✓	✓	✓
Single-Source Shortest Path	✓	✓		✓
Triangle Counting	✓	✓	✓	✓

Table 4.1: Implemented graph algorithms (imp: implementable).

4.2.2 Results

Regarding expressiveness, Table 4.1 contains a list of common graph algorithms and the frameworks that implement them. As we can see, Gunrock is currently the framework that supports the largest number of algorithms. Implementing 12 of the 15 algorithms considered. Hornet and FaimGraph implement 9 and 7 respectively, supporting some algorithms that are not supported by Gunrock, such as Clustering Coefficient, Connected Components, and others. Marrow-Graph can potentially implement at least 10 algorithms. Because of time constraints, only 5 have indeed been implemented. However, taking into account that Marrow-Graph and Gunrock share a very similar programming model, looking at Gunrock's implementations of these algorithms, we can safely assume that the Betweenness Centrality, Graph Coloring, Geolocation, Hyperlink-Induced Topic Search, and K-Core Decomposition algorithms can theoretically also be implemented in Marrow-Graph. All of these algorithms are implemented using operations that are supported by either Marrow-Graph or marrow. The only algorithm whose implement-ability in Marrow-Graph is uncertain is Minimum Spanning Tree, as it uses a specialized Gunrock operator `parallal_for_each_t::element`. Additionally, the Local Graph Clustering algorithm is no longer available in the public Gunrock repository. Regarding the algorithms implemented in Hornet and FaimGraph, but not in Gunrock, it is difficult to confidently say whether or not these algorithms are easily implementable in Marrow-Graph. Given their differences in programming models, a more in-depth analysis of these algorithms would be necessary. Regardless, we can conclude that Marrow-Graph already shows a very high level of expressiveness.

Regarding conciseness, we believe that we provide the solution with the most concise, and yet readable, algorithm implementations. While line count is not an entirely objective measure of conciseness, it offers valuable insights. To support our claim, Table 4.2 displays the total number of useful lines in each framework's implementations of the common

Algorithm	Gunrock	Hornet	FaimGraph	Marrow-Graph
Breadth-First Search	104	64	68	44
PageRank	150	73	134	79
Sparse-Matrix Vector Multiplication	116	38	59	18
Single-Source Shortest Path	125	48	-	55
Triangle Counting	129	95	106	40

Table 4.2: Number of lines of code composing graph algorithms.

algorithms. We can immediately see that Marrow-Graph requires significantly less lines of code than Gunrock and FaimGraph, and slightly less lines of code than Hornet. Marrow-graph only shows a very slightly higher number of lines of code than Hornet in 2 algorithms (PR and SSSP). In general, Marrow-Graph requires almost no boilerplate code to express these algorithms. Gunrock require a significant amount of boilerplate code, and Hornet algorithms must be defined inside inheriting classes, which require overriding a set of common methods. FaimGraph does not provide a high-level programming model like the other solutions, requiring a more verbose and low-level implementation of the various algorithms.

Besides conciseness, we also believe that we offer the best readability. Although the implementations between Marrow-Graph and Gunrock share many similarities, as already stated, Gunrock requires dealing with a significant amount of boilerplate, but also with a lot of Gunrock-specific concepts, such as contexts, enactors, load-balancing-methods, execution policies, and more. All of these concepts of course allow for more flexibility and control over these algorithms, but also make them less readable, in the sense that the actual algorithm logic becomes more obfuscated. Additionally, Gunrock makes heavy usage of the thrust library, while Marrow-Graph uses marrow. We believe marrow offers a more concise and simpler syntax supported by the adoption of a unified address space, which does not require the programmer to worry about host-device communication and synchronization. One notable superiority of Gunrock over Marrow-Graph lies in its utilization of `std::functions` and lambdas, which undergo rigorous type-checking. This contrasts with Marrow-Graph's reliance on templated functors, potentially resulting in less transparent compilation errors. Hornet in general also offers algorithms with a good level of readability. Most of the code is directly related to the algorithm's logic, and a set of high-level operators, such as `forAllEdges` and `foraAllVertices`, are provided to easy development. Hornet makes use both of functors and lambdas to express operations. Similarly to Gunrock, Hornet has the disadvantage of leaving most device memory management (of problem data) to the programmer using operators such as `host::copyToDevice`, `gpu::allocate`, and `gpu::free`. FaimGraph's main focus is to provide a novel efficient dynamic-graph data structure [46]. For this reason, as already stated, it does not include a high-level programming model that allows for simple and accessible algorithm implementations.

4.2.3 Conclusion

From our analysis, we can conclude that Marrow-Graph already demonstrates a high level of expressiveness, and allows for the development of concise and very readable algorithms. The integration of a simple, yet powerful interface inspired by Gunrock, with the expressive and user-friendly marrow library, results in a cohesive library that maintains ease of use but also achieves an expressive capability comparable to Gunrock.

4.3 Performance

To assess Marrow-Graph's performance, we will try to answer the following questions

- Is there an optimal block size for marrow-graph?
- Does Marrow-Graph offer competitive algorithmic and mutative performance against the state-of-the-art GPU-accelerated graph processing frameworks?
- How does Marrow-Graph's GPU usage compare to the state-of-the-art GPU-accelerated graph processing frameworks?

4.3.1 Evaluation Methodology

For this evaluation, as a baseline, we will use Gunrock [42], the state-of-the-art static GPU-Accelerated graph framework, and Ligra [39], the state-of-the-art static CPU-based graph framework. As our main competitors we will consider Hornet [6] and FaimGraph [46], as these are the current state-of-the-art GPU-Accelerated dynamic graph frameworks. As metrics to answer the previous questions, we will consider:

Graph initialization time: Time it takes to load a graph from a file and store it on the device.

Graph update rates: Number of edges/vertices that can be updated per time unit.

Algorithmic Time: Time it takes to execute an algorithm over a graph after it has been loaded and initialized.

GPU percentile usage: The amount of processing power that is being consumed by a graph processing application on the GPU.

Video Random Access Memory (VRAM) Usage: The amount of VRAM that is being consumed by a graph processing application on the GPU.

Optimal Block-Size. In order to determine the optimal block size for Marrow-Graph, we developed a benchmark that measures both graph initialization time, edge insertion time, and algorithmic time, using different block sizes. This benchmark performs a fixed-sized number of small/medium batch insertions, followed by a set of iterations of the SSSP

algorithm. We chose the SSSP algorithm given that it is a fairly complex algorithm that requires multiple iterations, and uses most of Marrow-Graph's functionalities.

Graph Initialization. In order to measure graph initialization times, we measure the time it takes to load the `mtx` or `pbbs` file into memory and upload the constructed graph to the device. All the frameworks use the `mtx` file format to load the graphs, with the exception of Ligra which uses `pbbs`.

Graph Updates. In order to measure update rates, we developed a set of benchmarks that measure the time it takes for a set of update bathes to be loaded into the graph. Both Hornet and FaimGraph allow batches of any size to be loaded onto the graph. We could simply measure the update rates of single large batch insertions using the different frameworks. But in a real-case scenario, updates of different sizes can arrive dynamically. For this reason, instead of only measuring the optimal case of inserting single large batches of size N, we measure the update rates obtained while performing N update operations using multiple batches of varying sizes. Another reason such measurements are more useful is related to the manner in which Marrow-Graph performs its graph updates. Contrary to FaimGraph and Hornet, Marrow-Graph performs updates on the host and synchronizes them to the device whenever an algorithm or operator is executed. This means Marrow-Graph can handle real-time updates of small to medium sizes very efficiently on the host. FaimGraph and Hornet, on the other hand, perform updates directly on the device, meaning that they benefit from using single large batches. In order to fairly portray both the benefits and shortcomings of both methods of performing updates, we measure the update rates using both single large batches, and multiple smaller batches. Ideally, we would like to measure edge and vertex insertion and deletion rates. However, given that Hornet does not directly support vertex updates and that we were not able to perform edge deletions in either framework (runtime errors were always encountered), we only measure edge insertion rates.

Algorithms. For algorithmic performance, we will run a set of common graph analytics algorithms, namely: BFS, SSSP, TC, SpMV and PR. Most of the frameworks implement all of these algorithms, with the exception of FaimGraph which does not implement SSSP, and Ligra wich does not implement SpMV. However, we did not consider all of the implementations for our benchmarks. FaimGraph's PR implementation is different from Marrow-Graph's, Hornet's and Gunrock's, as it uses a fixed size number of iterations, rather than a metric based on the absolute differences between the results of the last two iterations. Additionally, FaimGraph's SpMV and Hornet's TC both resulted in runtime errors when we tried to run them. In order to achieve fair and comparable results between the frameworks, some small modifications were performed to some of the algorithms, namely:

- Ligra's PR was modified to use the same alpha and tolerance values as Marrow-Graph.
- Hornet's PR was modified to use the same alpha and tolerance values as Marrow-Graph, its reduction was changed to use `max` instead of `add` (like Marrow-Graph and Gunrock), and an iteration counter was incorporate that could be queried by the benchmark suite.
- Gunrock's PR was modified to use the same alpha and tolerance values as Marrow-Graph and an iteration counter was incorporated that could be queried by the benchmark suite.
- FaimGraph's SpMV was edited to use the non-warped-sized implementation given that the warped-sized implementation resulted in runtime errors.

The PR algorithm loops until a given condition is met (the tolerance and alpha values we used were $1.0 * 10^{-6}$ and 0.85 respectively). To ensure a fair comparison between the different frameworks, we logged the number of iterations performed by each implementation and ensured that they were the same. The other algorithms either don't require iterating (SpMV and TC), or have a deterministic number of iterations that is not tied to a heuristic (BFS and SSSP). For all the algorithms that require an initial source vertex, we used the source vertex with ID 0.

It is worth noting that from the list of algorithms that we chose to use in this benchmark, only PR has been implemented by Hornet using the dynamic data structure. Unfortunately, we were not able to use it as it often resulted in runtime errors. This means that all of Hornet's results are not fairly comparable to Marrow-Graph and FaimGraph, as all the algorithms run on the static version.

Updates + Algorithms. Besides measuring algorithmic and mutative performance separately, we also developed a set of benchmarks that measure the performance of doing both operations interchangeably. This means that we load a graph to the device, and then perform N iterations, where we load a single update batch and execute the SpMV algorithm in each iteration. We chose SpMV because it was the algorithm that displayed the most similar performance across all implementations. Besides Marrow-Graph and FaimGraph, we also included Gunrock in these benchmarks. Although Gunrock does not support graph mutations, we developed a function that emulates the workload of reloading an updated graph. We do so by re-uploading the whole graph from the host to the device. This is somewhat of a conservative approach, given that updating the graph would require reconstructing the CSR data structure before re-uploading it. Regardless, we can still obtain a rough estimation of the performance gains of using dynamic graph frameworks versus static graph frameworks, such as Gunrock, when dealing with both algorithms an mutations. We were not able to include Hornet in this benchmark, given that very few algorithms have been implemented in Hornet using the dynamic data structure

Name	N. Vertices	N. Edges	Min. Deg.	Max. Deg.	Avg. Deg.
roadNet-CA	1.9M	2.7M	1	12	2
belgium_osm	1.4M	1.5M	1	10	1
delaunay_n21	2.1M	6.3M	3	23	5
hollywood-2009	1.1M	57.5M	1	11.5K	105
road_usa	23.9M	28.9M	1	9	2

Table 4.3: Graph statistics.

	Machine 1	**Machine 2**
CPU	Intel Core i5-3570 @ 3.40GHz	AMD Ryzen 5 3600 @ 3.60GHz
GPU	NVIDIA GTX 980 4GB	NVIDIA RTX 3060 12GB
RAM	8GB DDR3 @ 1600MHz	16GB DDR4 @ 3200MHz
OS	Ubuntu 22.04.3 LTS	Ubuntu 22.04.3 LTS
Linux kernel	6.5.0-21	6.5.0-14
NVIDIA driver	535.154.05	545.23.08
CUDA	12.0	12.3
G++	11.4.0	11.4.0

Table 4.4: Experimental setups.

(most algorithms only run on the static version), and the few that were, often result in runtime errors.

GPU Usage. In order to compare the GPU usage of the different frameworks, we measured the total GPU percentile usage, and VRAM usage, during the execution of the `update_spmv` benchmark. These measurements were performed using the Nvidia-smi tool, with a sampling frequency of 100Hz, or every 10 milliseconds. We chose the `update_spmv` benchmark given that it is the most complex and complete benchmark that deals with both graph initialization, graph updates, and algorithm execution.

Experimental Setup. For our benchmarks, we utilized a subset of the real-life-large-graphs proposed by Gunrock, which are available at the Network Repository [33]. Some of the larger graphs weren't utilized given the limited VRAM of one of the machines used for the execution of the benchmarks. The graph's relevant statistics can be found in Table 4.3.

All time measurements were conducted utilizing the standard C++ chrono library for accuracy (in the microseconds range) and consistency. To ensure robustness, each algorithm underwent ten iterations, with the average execution time serving as the basis for our results. Regarding compilation, all frameworks use the GCC and NVCC compilers. Additionally, similarly to most of the other frameworks, all Marrow-Graph benchmarks were compiled using the -O3 flag. All benchmarks were executed on the machines detailed in Table 4.4.

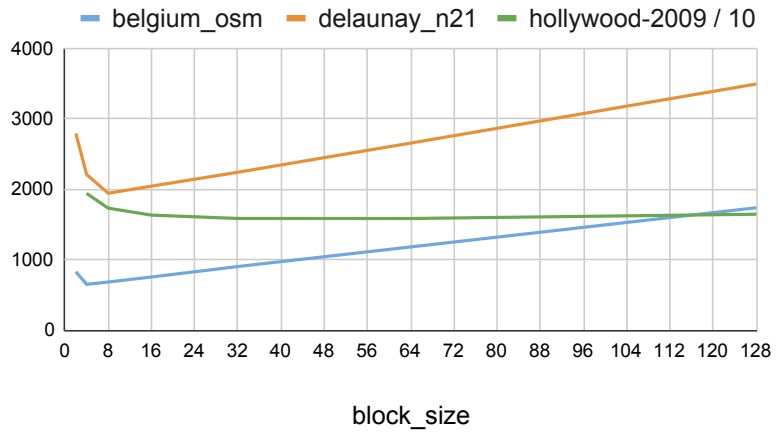


Figure 4.1: Block size benchmark: Total graph load time in milliseconds.

4.3.2 Results

Optimal Block-Size. Regarding optimal block-size, we measured the graph load time (Figure 4.1), edge insertion time (Figure 4.2) and SSSP average time (Figure 4.3) using block sizes of 2, 4, 8, 16, 32, 64 and 128. These plots contain the results obtained on Machine 1. Note that the displayed graph load times for the hollywood-2009 graph have all been divided by 10 for improved readability.

As we can see in Figure 4.1, the time it takes to build and upload both the belgium_osm and delaunay_n21 graphs, with block-sizes larger than 4, increases linearly with the increase in block size. This makes sense since the adjacency lists of most nodes fit in a block of size 8 (the graphs have average degrees of 1 and 5 respectively as seen in Table 4.3). For block sizes over 8, the larger the block, the more data has to be allocated and synchronized, leading to higher graph construction times. Using smaller block sizes below 8, we can see the effects of the overheads associated with allocating multiple blocks per adjacency list. Regarding hollywood-2009, contrary to the other two graphs, the graph construction time keeps decreasing, even for block sizes above 8. This is related to the length of the adjacencies found in this specific graph. As we can see in Table 4.3, the average vertex degree is 105, and the maximum degree is 11.5K. This means the larger the block size, the better, as we can store more adjacencies in a single block.

Regarding edge insertion times, we see a decrease in insertion times (Figure 4.2) when increasing the block size up to 16. This makes sense given that small block sizes result in frequent block allocations while inserting edges. For block sizes exceeding 16, insertion times remain relatively constant. This is expected since the newly inserted edges are expected to fit entirely within the last block of the corresponding adjacency list. The same applies to the denser hollywood-2009 graph. Although graph construction times benefit from larger block sizes, the key consideration for edge insertion efficiency is ensuring that there is enough space within the last block to accommodate new edges, thereby avoiding the need for frequent block allocations.

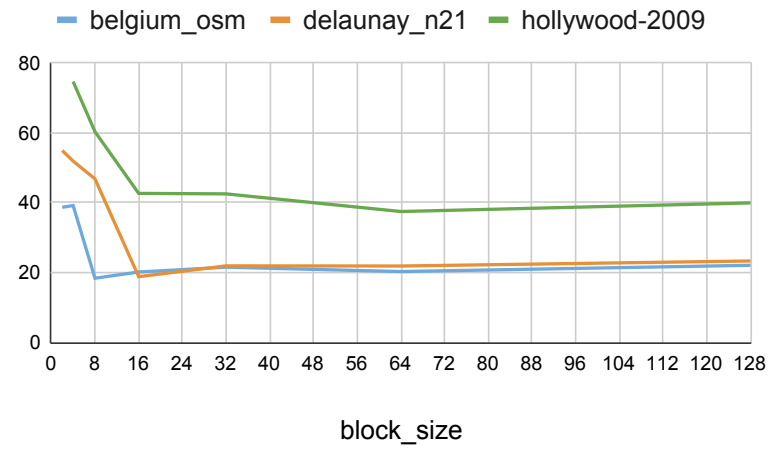


Figure 4.2: Block size benchmark: Total edge insertion time in milliseconds.

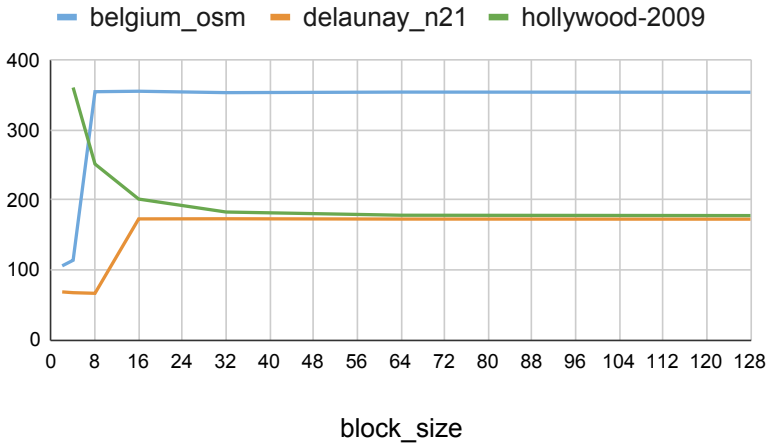


Figure 4.3: Block size benchmark: Mean SSSP execution time in milliseconds.

As we can see in Figure 4.3, the average SSSP execution times see a significant increase when we increase the block size from 4 to 8 and from 8 to 16 while processing the belgium_osm and delaunay_n21 graphs respectively. One possible explanation is related to the manner in which GPUs fetch global data in blocks of 128 bytes. The SSSP makes extensive use of the advance operator. This operator performs a lot of consecutive reads to the graph's data structure. Consecutive threads search for consecutive neighbors of a given source vertex. This means that if an ideal block size is used (2/4 for belgium_osm and 8 for delaunay_n21, given their respective average degrees of 1 and 5), we can take maximum advantage of the 128-byte cache lines. The same is also true for even smaller block sizes. Given that, typically, blocks comprising a vertex's adjacency list are stored consecutively, they also benefit from coalesced memory reads. If we double the block size from the ideal value, this leads to a lot of unused positions in each block. In turn, each global memory access fetches half as much useful data. This can result in half (or less) as many cache hits, leading to significantly worse performance. The hollywood-2009 graph

	delaunay_n21	roadNet-CA	hollywood-2009	belgium_osm	road_usa
marrow-graph	1643+118	853+112	14260+378	553+97	11031+778
gunrock	2191	1038	19515	512	10082
hornet	1300	671	10321	455	6557
ligra	1730	855	13300	518	11100
faimgraph	1464	130	14600	71	-

Table 4.5: Machine 1: Graph initialization times in milliseconds (red: slower than marrow-graph, marrow-graph: construction time + upload).

	delaunay_n21	roadNet-CA	hollywood-2009	belgium_osm	road_usa
marrow-graph	1356+100	733+89	11468+161	491+82	9146+261
gunrock	1717	741	15358	419	8141
hornet	969	535	7314	364	4943
ligra	384	193	2960	117	2360
faimgraph	362	167	2896	87	-

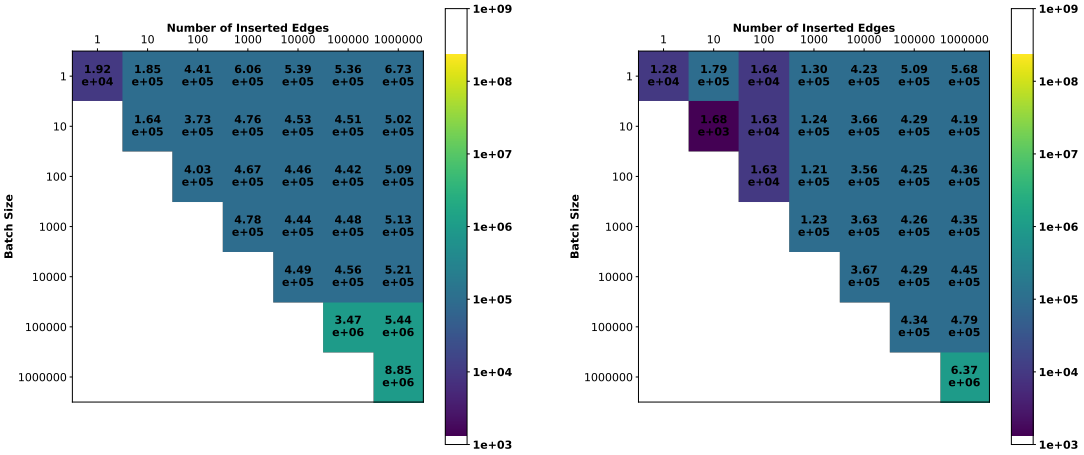
Table 4.6: Machine 2: Graph initialization times in milliseconds (red: slower than marrow-graph, marrow-graph: construction time + upload).

on the other hand, benefits from using large block sizes. Given its large average degree of 105, using larger blocks leads to less time spent iterating over multiple blocks.

We can conclude that there is not an ideal block size, as it depends on the characteristics of the graph we are processing. The ideal block size seems to be tied to the average and maximum degrees of a given graph. Regardless, for the types of graphs we'll be using in this evaluation, we can see that a block size of 8 or 16, seems to yield good general performance. For this reason, we will adopt a default block size of 8, which we may adjust as needed based on specific graph characteristics and performance considerations.

Graph Initialization. From Table 4.5 (containing the results from Machine 1), we can see that Marrow-Graph has overall faster graph initialization times than Gunrock and just slightly slower times than Ligra. Hornet shows relatively faster initialization times than Marrow-Graph, Gunrock, and Ligra. FaimGraph has some of the best initialization times for graphs like belgium_osm and roadNet-CA, but shows similar initialization times to Marrow-Graph for the dense graph hollywood-2009, and fails to load road_usa. Overall, Marrow-Graph offers quite satisfactory graph initialization times. The results obtained using Machine 2 (Table 4.6) are very similar, with the exception of Ligra, which performs significantly better due to a faster processor.

Graph Updates. Figures 4.4, 4.5, and 4.6 show the edge insertion rates obtained by each framework on Machine 2 (we achieved similar results on Machine 1). Each heat plot contains the insertion rates for inserting N edges, where $N = \{10^k : k \in \{0, 1, 2, ..., 6\}\}$, using batches of size B, where $B = \{10^k : k \in \{0, 1, 2, \ldots, 6\} \wedge 10^k \leq N\}$. This means that we measure both the insertion rates for inserting single large batches, as well as multiple smaller batches.



(a) belgium_osm (b) delaunay_m21

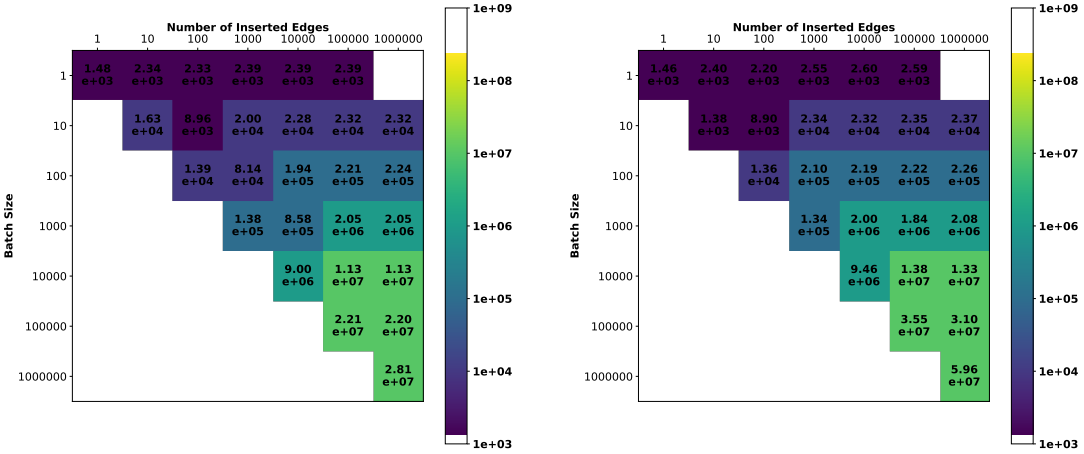
Figure 4.4: Machine 2: Marrow-Graph edge insertion rates (edge/second).

As we can see in Figure 4.4, Marrow-Graph has a fairly consistent edge insertion rate, for any size of batch and any number of inserted edges, around $4.0 * 10^5$ *edges/second*. For larger batches such as 10^6, it achieves rates up to $8.85 * 10^6$ *edges/second*.

Hornet (Figure 4.5), displays significantly lower insertion rates for batch sizes up to 10^2. Using a batch size of 10^3, we see similar insertion rates for a number of inserted edges up to 10^4, and better insertion rates by Hornet for larger numbers of inserted edges. For batch sizes over 10^3, Hornet offers better insertion rates than Marrow-Graph. This makes sense given that Marrow-Graph performs the updates on the host and synchronizes the changes to the device, while Hornet performs the updates directly on the device. This means that smaller batch sizes can be more efficiently updated on the host and later synchronized to the device, while larger batches can be inserted in parallel directly on the device more efficiently.

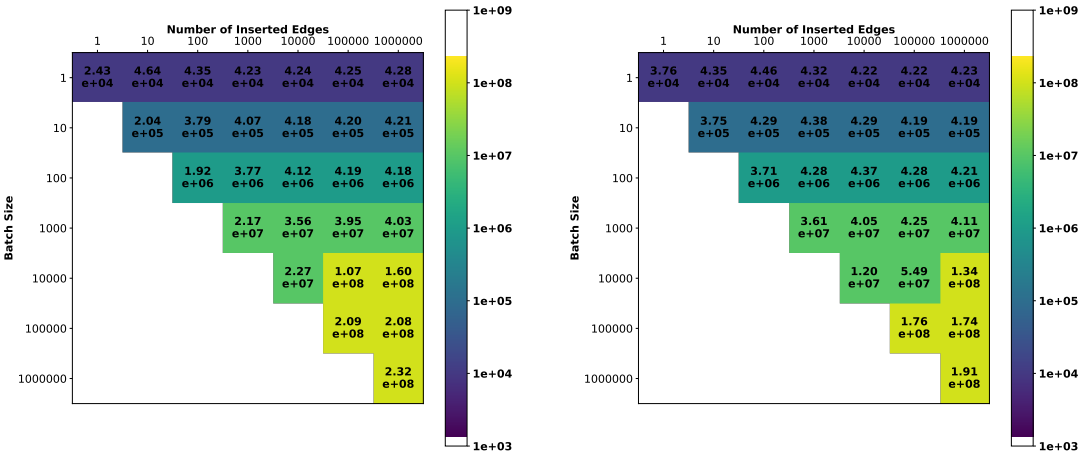
FaimGraph (Figure 4.6), offers the best insertion rates for medium to large batch sizes. For small batch sizes of 1 and 10, Marrow-Graph achieves comparable, and in some instances, better insertion rates than FaimGraph. For batch sizes of 10^2 and bigger, FaimGraph outperforms both Marrow-Graph and Hornet. This is most likely related to FaimGraph's optimized memory manager.

Algorithms. Looking at Tables 4.7 and 4.8, we can compare algorithmic performance of the various state-of-the-art frameworks. For the SpMV algorithm, Marrow-Graph shows similar performance to the other frameworks, even outperforming FaimGraph on Machine 1 and Hornet and Gunrock (partially) on Machine 2, falling only behind more significantly when using the road_usa graph. Regarding the BFS and SSSP algorithms, Marrow-Graph exhibits roughly double the execution time compared to the other GPU-based solutions



(a) belgium_osm (b) delaunay_m21

Figure 4.5: Machine 2: Hornet edge insertion rates (edge/second).



(a) belgium_osm (b) delaunay_m21

Figure 4.6: Machine 2: FaimGraph edge insertion rates (edge/second).

	delaunay_n21	roadNet-CA	hollywood-2009	belgium_osm	road_usa
			SpMV		
marrow-graph	3.85	1.63	46.79	1.03	30.18
faimgraph	-	1.82	-	1.31	-
hornet	2.66	1.99	40.21	1.01	16.36
gunrock	2.52	1.51	25.96	1.07	12.79
			BFS		
marrow-graph	160.87	152.43	240.69	338.05	5283.12
faimgraph	84.60	70.73	88.89	134.12	-
hornet	39.27	38.78	21.10	95.66	470.18
gunrock	79.50	73.12	112.51	167.90	802.97
ligra	41.40	34.70	49.80	30.20	519.00
			SSSP		
marrow-graph	167.13	156.49	252.31	337.76	5398.88
hornet	46.25	40.93	27.59	96.26	496.75
gunrock	85.52	78.49	108.98	188.52	880.10
ligra	70.20	65.40	201.00	40.80	689.00
			PR		
marrow-graph	18.49	27.82	815.60	14.84	123.09
hornet	6.75	20.70	130.31	9.88	50.34
gunrock	9.04	13.61	266.05	6.02	46.45
ligra	1280.00	1710.00	15 700.00	1640.00	32 100.00
			TC		
marrow-graph	58.78	7.85	12 915.90	4.24	136.00
faimgraph	7.03	2.05	-	1.24	-
gunrock	10.58	2.79	-	1.66	28.92
ligra	128.00	59.40	52 600.00	46.10	782.00

Table 4.7: Machine 1: Algorithmic execution times in milliseconds (red: slower than marrow-graph).

on Machine 1. It is also interesting to note that Ligra (the only Central Processing Unit (CPU)-based solution) performs exceptionally well in these two algorithms, but falls significantly behind in all others. This is most likely due to the fact that BFS and SSSP are the algorithms that require the largest number of iterations, and therefore, the largest number of host-device communications for the GPU-based solutions. For the PR algorithm, Marrow-Graph, once again, exhibits roughly double the execution time compared to the GPU-based solution on Machine 1. On Machine 2 however, Marrow-Graph outperforms Hornet for most graphs. Compared to Ligra, Marrow-Graph achieves significant speedups up to x260. Finally, regarding the TC algorithm, Marrow-Graph once again is slower than the GPU-based solution but is significantly faster than Ligra.

In order to better understand the reasoning for the performance discrepancies between Marrow-Graph and the rest, we analyzed the execution of some of the benchmarks using NVIDIA's Nsight profiler. We found out, that the BFS and SSSP algorithms' poorer performance is mainly tied to the regular advance operator. The regular advance operator is composed of the execution of five separate kernels. Gunrock on the other hand, performs most of the advance's logic using a single kernel. Additionally, each Marrow-Graph advance performs a `memset` over a container with a size equal to the graph's number of vertices. The extra host-device communication hinders the performance of Marrow-Graph. Regarding the TC and PR algorithms, which utilize the frontier-less unbalanced advance,

	delaunay_n21	roadNet-CA	hollywood-2009	belgium_osm	road_usa
			SpMV		
marrow-graph	1.203	0.433	18.918	0.395	8.658
faimgraph	-	-	-	-	-
hornet	1.173	4.06	26.139	3.939	14.211
gunrock	1.106	0.754	11.626	0.647	5.430
			BFS		
marrow-graph	104.478	100.664	130.023	233.658	2763.38
faimgraph	44.480	38.394	22.711	84.602	-
hornet	65.503	78.217	9.665	177.849	997.135
gunrock	25.939	23.754	27.636	52.818	257.659
ligra	28.0	25.6	17.2	29.9	303.0
			SSSP		
marrow-graph	107.062	101.855	135.285	236.958	2795.07
hornet	77.0035	88.643	15.863	193.334	1093.71
gunrock	30.127	27.333	35.425	62.584	299.02
ligra	33.4	31.2	-	35.8	361.0
			PR		
marrow-graph	5.277	9.261	319.25	5.286	33.614
hornet	8.263	14.878	51.627	8.868	28.014
gunrock	2.858	4.884	78.519	2.554	15.535
ligra	454.0	477.0	5180.0	410.0	12 000.0
			TC		
marrow-graph	18.191	1.866	59 614.4	1.272	32.490
faimgraph	2.198	0.883	-	0.598	-
gunrock	3.170	1.132	17 098.0	0.843	7.998
ligra	51.3	20.3	22 900.0	15.5	303.0

Table 4.8: Machine 2: Algorithmic execution times in milliseconds (red: slower than marrow-graph).

these exhibit performance closer to the competitors, but also seem to suffer from slightly slower kernels. This might be related to worse memory locality or the unbalanced nature of the advance being used. Even with these shortcomings, which can be mitigated in the future, Marrow-Graph is able to achieve significant speedups when compared to a CPU-based solution like Ligra, an outperform GPU-based solutions in some algorithms.

Updates + Algorithms. Table 4.9 and Table 4.10 show the results of the update_spmv benchmark, which measures the performance of interchangeably inserting edges and running the SpMV algorithm. As we can see, for small to medium edge insertion batch sizes, Marrow-Graph outperforms FaimGraph by a large factor, achieving speedups up to x16. For larger batch sizes, FaimGraph outperforms Marrow-Graph, which is expected given the previous edge insertion rate analysis. It's worth noting that Marrow-Graph exhibits superior performance compared to FaimGraph in this benchmark, in contrast to the edge insertion benchmark where its performance was less pronounced. Considering the similar average execution times of the SpMV algorithm on both Marrow-Graph and FaimGraph (refer to Table 4.8), this observation suggests that Marrow-Graph experiences a lesser loss in performance than FaimGraph when transitioning between graph updates and analytics tasks. Regarding the estimated Gunrock execution times, the benefits of supporting a fully dynamic graph data structure are very apparent. Marrow-graph

	10^0	10^1	10^2	10^3	10^4	10^5
	belgium_osm					
marrow-graph	17.968	16.824	11.99	17.929	169.286	674.462
faimgraph	68.316	68.549	68.774	68.129	70.537	83.193
gunrock	176.374	176.883	178.479	176.923	177.849	185.608
	roadNet-CA					
marrow-graph	27.933	48.354	85.748	234.513	390.803	840.588
faimgraph	95.249	94.061	94.297	93.824	95.458	109.704
gunrock	297.400	299.162	297.961	299.017	297.501	307.905

Table 4.9: Machine 1: Update SpMV execution times in milliseconds (red: slower than marrow-graph).

	10^0	10^1	10^2	10^3	10^4	10^5
	belgium_osm					
lmarrow-graph	5.788	-	2.395	22.156	218.282	299.821
faimgraph	24.853	24.101	24.153	24.271	25.030	28.502
gunrock	81.986	81.762	83.645	81.625	82.385	85.352
	roadNet-CA					
lmarrow-graph	6.552	11.632	19.271	77.877	266.171	351.084
faimgraph	26.989	26.703	27.715	27.113	28.036	35.631
gunrock	132.708	133.32	134.461	133.203	133.458	136.335

Table 4.10: Machine 2: Update SpMV execution times in milliseconds (red: slower than marrow-graph).

achieves speedups up to x11 compared to Gunrock, given that a static solution such as Gunrock, requires re-uploading the whole graph whenever we perform an update. However, we can observe that Gunrock's execution times remain fairly constant. Given that in this estimation we are only accounting for the time spent re-uploading the graph, the execution time should only increase significantly when the number of inserted edges is proportionate to the size of the Compact Sparse Rows (CSR) data structure. We can indeed see a slight increase in execution time when inserting 10^5 edges. This increase becomes more prominent for even larger batch sizes. On Machine 2 (refer to Table 4.10) we obtained very similar results. We can observe a fairly uniform speedup in all benchmarks given the more powerful hardware. The only significant difference is the performance of Marrow-Graph using large batch sizes. As we can see, the execution time difference from a batch size of 10^4 to a batch size of 10^5 isn't as large using Machine 2 compared to Machine 1. This is most likely related to the hardware limitations of Machine 1.

GPU Usage. While executing the update_spmv benchmark with a batch size of 100, we logged both the total GPU and VRAM usage. The results obtained on Machine 1 can be found in Figures 4.8, 4.7, and 4.9 (we obtained similar results on Machine 2). Given that this benchmark is performed over two graphs (roadNet-CA and belgium_osm) we can roughly see two large spikes in all plots. Additionally, Marrow-Graph exhibits two small GPU usage spikes before each larger spike, which are most likely related to the

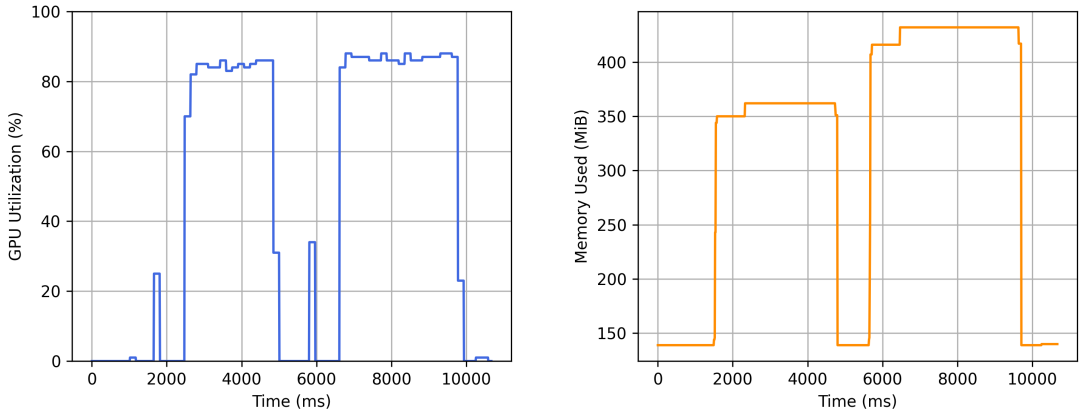


Figure 4.7: Marrow-Graph GPU utilization.

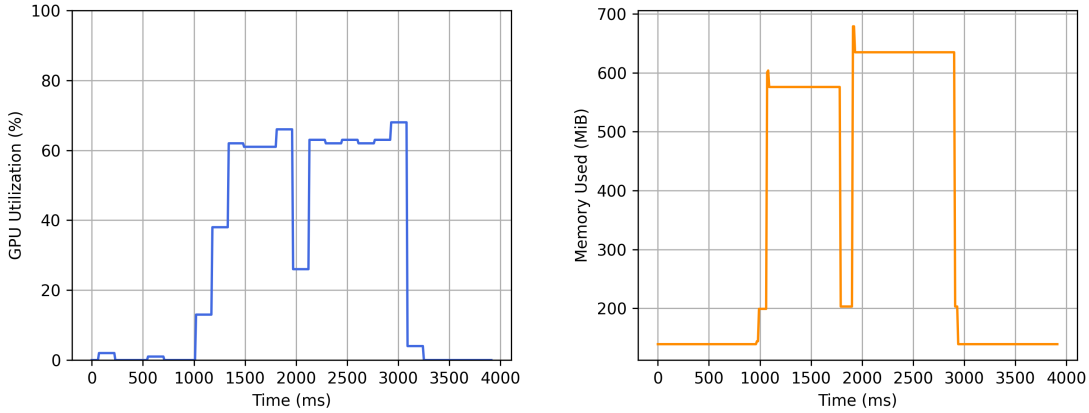


Figure 4.8: FaimGraph GPU utilization.

construction of each graph, before proceeding with the algorithm executions and graph updates. Marrow-graph exhibits overall slightly higher (around 10%) GPU utilization than Gunrock, and around 20% more utilization than FaimGraph. Regarding VRAM usage, FaimGraph exhibits the largest values, using around 600Mb and 630Mb during the processing of each graph. Gunrock uses around 250-275Mb and 275-325Mb respectively, and Marrow-Graph around 350Mb and 470Mb respectively. FaimGraph's larger memory usage is due to FaimGraph's memory manager, which initially allocates a single large block of memory, which is then managed on the device as needed. Gunrock offers slightly lower memory usage than Marrow-Graph because the CSR data structure is more compact than the blocked adjacency list. Note that we used a block size of 8 for Marrow-Graph.

4.3.3 Conclusion

Returning to our initial questions, we can say that although there is not a general optimal block size, as this depends on the characteristics of the graph being processed, a block size of 8 shows an overall good performance for graphs with average degrees below 16.

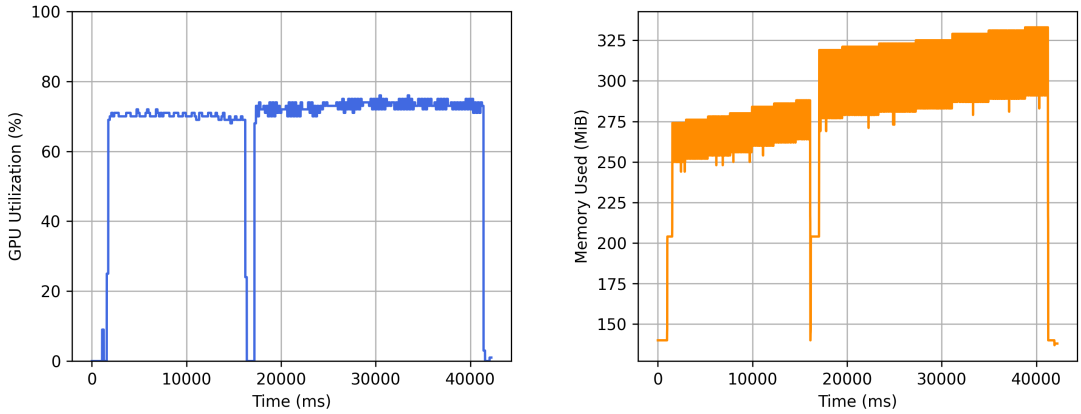


Figure 4.9: Gunrock GPU utilization.

In terms of algorithmic performance, Marrow-Graph has demonstrated superiority over its primary competitor, FaimGraph, in the SpMV algorithm, but falls behind in other algorithms such as BFS and TC. When compared to the static GPU-based solutions, Marrow-Graph is often outperformed, although its execution times consistently remain within the same order of magnitude. Despite this, there are instances where Marrow-Graph outperforms Hornet. Notably, Marrow-Graph also showcases remarkable speedups when compared with the only CPU-based solution considered, Ligra. Given that there is still a lot of room for performance tweaking, we believe that it is realistic to expect comparable performance to FaimGraph in all benchmarks in the future.

Regarding graph updates, marrow-graph offers excellent edge insertion rates for small to medium batch sizes. For large batch sizes, the competitors offer better insertion rates given that the batches are inserted in parallel directly on the device. Regardless, marrow-graph achieves very high and consistent insertion rates in the order of 10^5 and 10^6 *edges/second*. Additionally, when performing graph updates and analytics interchangeably, Marrow-Graph exhibits excellent performance for small to medium insertion batch sizes, outperforming FaimGraph by large factors.

Finally, in terms of GPU utilization, both FaimGraph, Gunrock, and Marrow-Graph show mostly similar results. FaimGraph has some overall higher VRAM usage due to its memory manager, and Marrow-Graph exhibits slightly higher GPU usage.

5

Conclusion and Future Work

5.1 Conclusion

In this dissertation we aimed to develop a novel GPU-accelerated dynamic graph processing library that provided both ease of use, using a high-level programming model inspired by Gunrock, and competitive performance against the limited state-of-the-art, using a data structure inspired by FaimGraph. Although the proposed solution still requires some performance tweaking, we believe that we have achieved a library that provides better usability than the current state-of-the-art, and promising performance.

In terms of usability, marrow-graph stands out with its intuitive and expressive interface. Paired with marrow's high-level functionalities, it facilitates the development of very readable and concise graph processing algorithms and applications. In contrast, FaimGraph lacks a high-level interface for such development, and Hornet provides a notably more complex interface compared to marrow-graph. Furthermore, both FaimGraph and Hornet exhibit recurring runtime errors, and their apparent lack of recent updates—more than three years in both cases—suggests that their development and support have stagnated. Hornet also notably provides a very limited assortment of algorithms that can run using the dynamic data structure.

Regarding performance, marrow-graph shows promising results in algorithms such as SpMV and PR, provides decent graph initialization times, and offers fast graph updates. In other algorithms, marrow-graph falls behind the competition, but not by a larger order of magnitude. This can most likely be overcome with the transition to the original marrow runtime (which offers more parallelization), and with some additional performance tweaking. Compared to the state-of-the-art CPU-based framework Ligra, it offers better performance in most benchmarks by a large factor.

We can conclude that the goals set out for this thesis were met, and that marrow-graph shows potential to become one of the leading state-of-the-art GPU-accelerated dynamic graph processing frameworks.

5.2 Future Work

In this section we discuss relevant future work to improve marrow-graph.

Marrow & Performance. Switch to the original complete marrow runtime, in order to benefit from all its features, and allow for increased parallelization. Additionally, continue tweaking the performance of Marrow-Graph's operators and algorithms. Especially the advance operator which currently requires launching multiple kernels.

Device Batch Updates. Support batched updates directly on the device as well as on the host. Currently, host updates offer excellent update rates for small to medium batch sizes. Support for updates directly on the device would allow marrow-graph to achieve higher insertion rates for larger batches.

Algorithms. Implement more graph-analytics algorithms and study their performance and accuracy.

Multi-GPU Processing. Given marrow's support for multi-GPU environments, study how marrow-graph could benefit from using multiple GPUs, especially when dealing with very large graphs that don't fit in a single GPU's memory.

Hybrid Processing. Given that marrow-graph stores the graph's data structure both on the host and device, it is possible to process the graph and execute algorithms not only on the device (as we currently do), but also on the host. It would be interesting to study the performance implications of allowing for such hybrid processing.

Bibliography

[1] I. C. committee (WG21). *std::numeric_limits::epsilon - cppreference.com*. URL: https://en.cppreference.com/w/cpp/types/numeric_limits/epsilon (cit. on p. 63).

[2] F. Alexandre, R. Marqu'es, and H. Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. 2014, pp. 880–885. DOI: 10.1145/2554850.2555018 (cit. on p. 11).

[3] S. Ashkiani, M. Farach-Colton, and J. D. Owens. *A Dynamic Hash Table for the GPU*. 2018. DOI: 10.1109/IPDPS.2018.00052 (cit. on p. 28).

[4] M. A. Awad et al. "Dynamic Graphs on the GPU". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. 2020, pp. 739–748. DOI: 10.1109/IPDPS47924.2020.00081 (cit. on pp. 28, 31).

[5] G. E. Blelloch. *Vector models for data-parallel computing*. Vol. 2. Citeseer, 1990 (cit. on p. 18).

[6] F. Busato et al. "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs". In: *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547541 (cit. on pp. 2, 3, 25, 27, 28, 30, 31, 66).

[7] F. Busato et al. "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs". In: *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547541 (cit. on pp. 17, 18).

[8] M. E. Coimbra, A. P. Francisco, and L. Veiga. "Distributed graphs: in search of fast, low-latency, resource-efficient, semantics-rich Big-Data processing". In: *CoRR* abs/1911.11624 (2019) (cit. on pp. 1, 22).

[9] N. Corporation. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#cuda-a-general-purpose-parallel-computing-platform-and-programming-model (cit. on pp. 5, 7).

[10] N. Corporation. *NVIDIA GeForce GTX 1080. Gaming Perfected*. URL: https://www.es.ele.tue.nl/~heco/courses/ECA/GPU-papers/GeForce_GTX_1080_Whitepaper_FINAL.pdf (cit. on p. 6).

[11] N. Corporation. *NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU*. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (cit. on p. 6).

[12] R. S. Dehal et al. "GPU Computing Revolution: CUDA". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2018, pp. 197–201. DOI: 10.1109/ICACCCN.2018.8748495 (cit. on p. 5).

[13] D. Ediger et al. "STINGER: High performance data structure for streaming graphs". In: *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. 2012, pp. 1–5. DOI: 10.1109/HPEC.2012.6408680 (cit. on p. 17).

[14] A. Fender, B. Rees, and J. Eaton. "RAPIDS cuGraph". In: *Massive Graph Analytics*. 2022, pp. 483–493. DOI: 10.1201/9781003033707-22 (cit. on p. 2).

[15] S. Firmli et al. "CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure". In: *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*. Vol. 184. LIPIcs. 2020, 17:1–17:16. DOI: 10.4230/LIPICS.OPODIS.2020.17 (cit. on p. 17).

[16] Z. Fu, B. B. Thompson, and M. Personick. "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs". In: *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*. Ed. by P. A. Boncz and J. L. Larriba-Pey. CWI/ACM, 2014, 2:1–2:6. DOI: 10.1145/2621934.2621936 (cit. on p. 26).

[17] J. E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 2014, pp. 599–613 (cit. on p. 23).

[18] J. E. Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. by C. Thekkath and A. Vahdat. 2012, pp. 17–30 (cit. on pp. 20, 22, 23).

[19] O. Green and D. A. Bader. "cuSTINGER: Supporting dynamic graph algorithms for GPUs". In: *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. 2016, pp. 1–6. DOI: 10.1109/HPEC.2016.7761622 (cit. on pp. 2, 27, 30).

[20] C. Gui et al. *A Survey on Graph Processing Accelerators: Challenges and Opportunities*. 2019. DOI: 10.1007/S11390-019-1914-Z (cit. on p. 1).

[21] W. Han et al. "TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC". In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. Ed. by I. S. Dhillon et al. 2013, pp. 77–85. DOI: 10.1145/2487575.2487581 (cit. on pp. 18, 19, 30).

[22] V. Kalavri, V. Vlassov, and S. Haridi. "High-Level Programming Abstractions for Distributed Graph Processing". In: *IEEE Trans. Knowl. Data Eng.* 30.2 (2018), pp. 305–324. DOI: 10.1109/TKDE.2017.2762294 (cit. on p. 21).

[23] A. Karunarathna. *CUDA Thread Indexing*. URL: https://anuradha-15.medium.com/cuda-thread-indexing-fb9910cba084 (cit. on p. 7).

[24] F. Khorasani et al. "CuSha: vertex-centric graph processing on GPUs". In: *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. 2014, pp. 239–252. DOI: 10.1145/2600212.2600227 (cit. on p. 26).

[25] C. E. Leiserson. "The Cilk++ concurrency platform". In: *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*. 2009, pp. 522–527. DOI: 10.1145/1629911.1630048 (cit. on p. 24).

[26] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[27] Y. Low et al. "GraphLab: A New Framework For Parallel Machine Learning". In: *CoRR* abs/1408.2041 (2014) (cit. on pp. 20, 24).

[28] P. Macko et al. "LLAMA: Efficient graph analytics using Large Multiversioned Arrays". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by J. Gehrke et al. 2015, pp. 363–374. DOI: 10.1109/ICDE.2015.7113298 (cit. on p. 23).

[29] G. Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. 2010, pp. 135–146. DOI: 10.1145/1807167.1807184 (cit. on pp. 20, 21).

[30] R. Marqués et al. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Vol. 8097. Lecture Notes in Computer Science. 2013, pp. 874–885. DOI: 10.1007/978-3-642-40047-6_86 (cit. on pp. 3, 11).

[31] R. Martinho. "Grafos Dinâmicos em GPU". MA thesis. NOVA School of Science and Technology, 2021 (cit. on p. 30).

[32] Y. Perez et al. "Ringo: Interactive Graph Analytics on Big-Memory Machines". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by T. K. Sellis, S. B. Davidson, and Z. G. Ives. 2015, pp. 1105–1110. DOI: 10.1145/2723372.2735369 (cit. on pp. 23, 24).

[33] R. A. Rossi and N. K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, pp. 4292–4293. DOI: 10.1609/AAAI.V29I1.9277 (cit. on p. 69).

[34] A. Roy, I. Mihailovic, and W. Zwaenepoel. "X-Stream: edge-centric graph processing using streaming partitions". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. 2013, pp. 472–488. DOI: 10.1145/2517349.2522740 (cit. on pp. 20, 22, 24).

[35] A. Roy et al. "Chaos: scale-out graph processing from secondary storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 2015, pp. 410–424. DOI: 10.1145/2815400.2815408 (cit. on pp. 20, 22).

[36] D. Sengupta et al. "GraphIn: An Online High Performance Incremental Graph Processing Framework". In: *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. Ed. by P. Dutot and D. Trystram. Vol. 9833. 2016, pp. 319–333. DOI: 10.1007/978-3-319-43659-3_24 (cit. on p. 22).

[37] M. Sha et al. "Accelerating Dynamic Graph Analytics on GPUs". In: *Proc. VLDB Endow.* 11.1 (2017), pp. 107–120. DOI: 10.14778/3151113.3151122 (cit. on p. 30).

[38] X. Shi et al. "Graph Processing on GPUs: A Survey". In: *ACM Comput. Surv.* 50.6 (2018), 81:1–81:35. DOI: 10.1145/3128571 (cit. on pp. 1, 7).

[39] J. Shun and G. E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. 2013, pp. 135–146. DOI: 10.1145/2442516.2442530 (cit. on pp. 24, 66).

[40] F. Soldado, F. Alexandre, and H. Paulino. "Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments". In: *Concurr. Comput. Pract. Exp.* 28.3 (2016), pp. 768–787. DOI: 10.1002/CPE.3612 (cit. on p. 11).

[41] *Using Shared Memory in CUDA C/C++*. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc (cit. on p. 10).

[42] Y. Wang et al. "Gunrock: GPU Graph Analytics". In: *ACM Trans. Parallel Comput.* 4.1 (2017), 3:1–3:49. DOI: 10.1145/3108140 (cit. on pp. 1–3, 21, 24–28, 30, 33, 66).

[43] t. f. e. Wikipedia. *Graph theory*. 2023. URL: https://en.wikipedia.org/wiki/Graph_theory (cit. on p. 1).

[44] Wikipedia, the free encyclopedia. *Bulk synchronous parallel*. 2022. URL: %7Bhttps://en.wikipedia.org/wiki/Bulk_synchronous_parallel%7D (cit. on p. 21).

[45] M. Winter, R. Zayer, and M. Steinberger. "Autonomous, independent management of dynamic graphs on GPUs". In: *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. 2017, pp. 1–7. DOI: 10.1109/HPEC.2017.8091058 (cit. on pp. 27–30).

[46] M. Winter et al. "faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. 2018, 60:1–60:13 (cit. on pp. 2, 3, 17, 25, 28, 29, 40, 65, 66).

[47] K. Zhang, R. Chen, and H. Chen. "NUMA-aware graph-structured analytics". In: (2015). Ed. by A. Cohen and D. Grove, pp. 183–193. DOI: 10.1145/2688500.2688507 (cit. on p. 23).

[48] J. Zhong and B. He. "Medusa: Simplified Graph Processing on GPUs". In: *IEEE Trans. Parallel Distributed Syst.* 25.6 (2014), pp. 1543–1552. DOI: 10.1109/TPDS.2013.111 (cit. on p. 21).

A

Listings

```
marrow_function
idx_t binary_search(std::size_t tid, std::size_t frontier_size, std::size_t* degrees_scan) {
    int frontier_vertex_n = -1;
    int low = 0, high = frontier_size - 1;
    while (frontier_vertex_n == -1 && low <= high) {
        int mid = (low + high) / 2;
        if (tid < degrees_scan[mid] && (mid==0 || tid >= degrees_scan[mid-1]))
            frontier_vertex_n = mid;
        else if (tid < degrees_scan[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return frontier_vertex_n;
}
```

Listing 21: Graph Bal Binary Search.

```
template<typename vertex_attributes, typename edge_attributes,
  typename on_intersection_fun, typename... on_intersection_arguments>
__device__
int segmented_intersection(
        graph_bal_t<vertex_attributes, edge_attributes> &graph,
        vertex<vertex_attributes>& vertex_a,
        vertex<vertex_attributes>& vertex_b,
        on_intersection_fun on_intersection,
        on_intersection_arguments&... on_intersection_args) {

    using Vertex = vertex<vertex_attributes>;
    using Edge  = edge<edge_attributes>;

    int nintersections = 0;
    int degree_a = graph.vertices[vertex_a.idx].degree;
    int degree_b = graph.vertices[vertex_b.idx].degree;
    int block_index_a = graph.vertices[vertex_a.idx].adjacency_list;
    int block_index_b = graph.vertices[vertex_b.idx].adjacency_list;
    int offset_a = 0;
    int offset_b = 0;
    int index_a = 0;
    int index_b = 0;

    while (index_a < degree_a && index_b < degree_b) {

        int edges_index_a = block_index_a * graph.block_size + offset_a;
        int edges_index_b = block_index_b * graph.block_size + offset_b;
        Vertex& neighbor_a = graph.vertices[graph.edges[edges_index_a].dst];
        Vertex& neighbor_b = graph.vertices[graph.edges[edges_index_b].dst];

        if (neighbor_a.idx == neighbor_b.idx) {
            nintersections++;
            on_intersection(vertex_a, vertex_b, neighbor_a, on_intersection_args...);
            index_a++;
            offset_a++;
            index_b++;
            offset_b++;
        }
        else if (neighbor_a.idx > neighbor_b.idx) {
            index_b++;
            offset_b++;
        }
        else {
            index_a++;
            offset_a++;
        }

        if (offset_a >= graph.block_size) {
            offset_a = 0;
            block_index_a = graph.block_links[block_index_a];
        }
        if (offset_b >= graph.block_size) {
            offset_b = 0;
            block_index_b = graph.block_links[block_index_b];
        }
    }

    return nintersections;
}
```

Listing 22: Segmented Intersection.

```cpp
template <typename vertex_attributes, typename edge_attributes, std::size_t block_size>
struct graph_bal_sort_adjacency_list_fun {
    using Block = array<edge<edge_attributes>, block_size>;
    vertex<vertex_attributes>& src_vertex;
    vector<Block>& blocks;
    vector<idx_t>& block_links;

    graph_bal_sort_adjacency_list_fun(vertex<vertex_attributes>& src_vertex,
      vector<Block>& blocks, vector<idx_t>& block_links) :
        src_vertex(src_vertex), blocks(blocks), block_links(block_links) {
    }

    void operator()() {
        quick_sort(0, src_vertex.degree-1);
    }

    void quick_sort(int start, int end) {

        if (start >= end)
            return;

        int p = partition(start, end);
        quick_sort(start, p - 1);
        quick_sort(p + 1, end);
    }

    edge<edge_attributes>& get(int i) {
        int block_n = i/block_size;
        int offset = i%block_size;
        int block_index = src_vertex.adjacency_list;
        for(int j = 0; j < block_n; j++)
            block_index = block_links.read(block_index);
        Block& block = blocks.read(block_index);
        return block.read(offset);
    }

    void swap(edge<edge_attributes> *a, edge<edge_attributes> *b) {
        edge<edge_attributes> t = *a;
        *a = *b;
        *b = t;
    }

    int partition(int low, int high) {
        int pivot = get(high).dst;
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (get(j).dst <= pivot) {
                i++;
                swap(&get(i), &get(j));
            }
        }
        swap(&get(i + 1), &get(high));
        return (i + 1);
    }
};

template <typename vertex_attributes, typename edge_attributes, std::size_t block_size>
void graph_bal_sort_adjacency_list(/*...*/) {

    graph_bal_sort_adjacency_list_fun(/*...*/)();
}
```

Listing 23: Sort Adjacency List.

```
struct graph_bal_frontierless_advance_fun : function_with_coordinates</*...*/> {

    template<typename vertex_attributes,
      typename edge_attributes, typename compute_fun, typename... compute_args>
    marrow_function
    void operator()(coordinate_t* coordinate,
                    idx_t *frontier,
                    vertex<vertex_attributes> *vertices,
                    edge<edge_attributes> *edges,
                    std::size_t block_size,
                    idx_t *block_links,
                    std::size_t frontier_size,
                    compute_fun &compute,
                    compute_args &... comp_args) {

        using Vertex = vertex<vertex_attributes>;
        using Edge = edge<edge_attributes>;
        using Graph = graph_bal_t<vertex_attributes, edge_attributes>;

        const std::size_t tid = coordinate[0];
        idx_t vertex_idx = frontier[tid];
        Vertex &vertex = vertices[vertex_idx];

        int neighbour_n = 0;
        for (idx_t block_idx = vertex.adjacency_list;
             block_idx != -1 && neighbour_n < vertex.degree; block_idx = block_links[block_idx]) {

            for (std::size_t block_offset = 0;
                 block_offset < block_size && neighbour_n < vertex.degree; block_offset++) {

                Edge _edge = edges[block_idx * block_size + block_offset];

                if constexpr (std::is_invocable<decltype(compute),
                  Graph &, Vertex &, Vertex &, Edge &, compute_args &...>::value) {

                    Graph graph = {
                            .block_size = block_size,
                            .vertices = vertices,
                            .edges = edges,
                            .block_links = block_links
                    };

                    compute(graph, vertex, vertices[_edge.dst], _edge, comp_args...);
                } else {
                    compute(vertex, vertices[_edge.dst], _edge, comp_args...);
                }

                neighbour_n++;
            }
        }
    }
};
```

Listing 24: Graph Bal Unbalanced Frontierless Advance Marrow Function.

```
template <typename vertex_attributes, typename edge_attributes>
class graph {

    using Vertex = vertex<vertex_attributes>;
    using Edge = edge<edge_attributes>;
public:
    virtual idx_t add_vertex(vertex_attributes attributes) = 0;
    virtual bool remove_vertex(idx_t idx) = 0;
    virtual bool add_edge(idx_t src, idx_t dst, edge_attributes attributes) = 0;
    virtual bool add_edge_batch(edge_insertion_batch<edge_attributes>& batch) = 0;
    virtual bool remove_edge(idx_t src, idx_t dst) = 0;
    virtual bool remove_edge_batch(edge_deletion_batch<edge_attributes>& batch) = 0;
    virtual bool edit_edge(idx_t src, idx_t dst, idx_t new_dst) = 0;
    virtual vector<idx_t> get_connection(idx_t idx) = 0;
    virtual std::size_t get_degree(idx_t idx) = 0;
    virtual std::size_t get_number_of_vertex() = 0;
    virtual std::size_t get_number_of_edges() = 0;
    virtual void sort() = 0;

    template <bool return_frontier = true, bool balanced = true>
    template <typename compute_fun, typename... compute_args>
    auto advance(vector<idx_t>& frontier, compute_fun& cfun, compute_args&... cargs);

    template <typename filter_fun, typename... filter_args>
    auto filter(vector<idx_t> &frontier, filter_fun& ffun, filter_args&... fargs);
};
```

Listing 25: Graph interface.