Martin Winter

Fully-dynamic aimGraph

Efficient memory management and algorithmic validation of a dynamic graph framework on GPUs

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dr.techn. Markus Steinberger
Institute for Computer Graphics and Vision

Univ.-Prof. Dr.techn. Dieter Schmalstieg
Institute for Computer Graphics and Vision

Graz, Austria, Nov. 2017

Abstract

In this thesis a new, dynamic graph data structure for the Graphics Processing Unit (GPU) is presented. It is built to deliver high update rates while keeping a low memory footprint using autonomous memory management directly on the GPU.
The data structure is fully-dynamic, allowing not only for edge but also vertex updates. By transferring the memory management to the GPU, efficient updating of the graph structure and fast initialization times are enabled as no additional kernel calls or reallocation procedures are necessary since they are handled directly on the device. Edge updates can either be performed sequentially or concurrently due to the exclusive access policy enforced for adjacency data.
In comparison to previous work, our optimized approach allows for significantly lower initialization times (up to 300x faster for tested graphs) and much higher update rates for significant changes (up to 30x faster for tested graphs) to the graph structure and about equal rates for small changes. Furthermore, building on a queuing scheme, currently unused memory can be returned to the memory manager, further reducing the memory requirements. The framework provides different update implementations tailored specifically to different graph properties. This enables over 100 million edge updates per second and permits keeping tens of millions of vertices and hundreds of millions of edges in memory without transferring data back and forth between device and host. We evaluate algorithmic performance using a PageRank and a static triangle counting implementation. Due to their use in a multitude of problem domains, dynamic graphs structures are becoming increasingly important. The presented framework allows users to efficiently operate and work with such massive graphs, harnessing the massively parallel compute capabilities of modern GPUs, at an affordable price compared to their High-Performance Computing (HPC) cluster counterparts.

Keywords. GPGPU, dynamic graphs, independent memory management, graph algorithms

Kurzfassung

Diese Arbeit präsentiert eine neue Datenstruktur auf der GPU zur Repräsentation von dynamischen Graphen. Der Fokus liegt dabei auf hohen Änderungsraten sowie einem geringen Speicherverbrauch mit unabhängigem Speichermanagement direkt auf der GPU. Die Datenstruktur ist dabei voll dynamisch, somit können nicht nur Kanten hinzugefügt bzw. gelöscht werden, selbiges funktioniert auch mit Knoten.
Durch das Verschieben des Speichermanagements auf die GPU werden das effiziente Aktualisieren der Graphstruktur sowie schnelle Initialisierungszeiten ermöglicht, denn dadurch entfallen zusätzliche Kernel-Aufrufe sowie zusätzliche Re-Allokierungsprozeduren. Das Aktualisieren von Kanten kann sowohl sequentiell als auch parallel ablaufen, da durch das Absperren der Kantenspeicherbereiche der exklusive Zugriff gewährt wird.
Verglichen mit vorheriger Arbeit kann dieser optimierte Zugang zu Graphverwaltung auf der GPU viel geringere Initialisierungszeiten (bis zu 300 mal schneller) bieten sowie größere Aktualisierungsraten bei größeren Änderungen (bis zu 30 mal schneller für getestete Graphen) an der Graphenstruktur und annähernd gleiche Raten bei kleinen Änderungen.
Ungenützter Speicher kann außerdem wieder an den Speichermanager übergeben werden. Hier wird ein Queuing-Verfahren genutzt um Indizes zu speichern und wiederzuverwenden. Das Framework bietet unterschiedliche Aktualisierungsmethoden an, welche speziell auf unterschiedliche Grapheigenschaften ausgelegt sind.
Dadurch können (bei den getesteten Graphen) mehr als 100 Millionen Aktualisierungen pro Sekunde ermöglicht werden sowie mehr als 10 Millionen Knoten und mehrere 100 Millionen an Kanten gleichzeitig im Speicher gehalten werden. Zwei Algorithmus-Implementierungen (PageRank und static triangle counting) sind vorhanden um algorithmische Leistungsfähigkeit zu demonstrieren.
In der heutigen Zeit existieren immer mehr Problemstellungen, welche dynamische

Graphen nutzen um wichtige Metriken und Kennzahlen zu berechnen. Die vorgestellte Arbeit ermöglicht es dem Benutzer, effizient mit großen Graphen zu arbeiten und dabei die massiv parallelen Fähigkeiten moderner GPUs zu nutzen. All das, verglichen mit HPC Clustern mit mehreren Central Processing Unit (CPU)s, zu einem vergleichsweise leistbaren Preis.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Date	Signature

Acknowledgments

First and foremost I would like to thank Ass.Prof. Dr.techn. Markus Steinberger, who was my advisor on this thesis. I am especially grateful to be given the opportunity to independently work on this big project. Not only did he frequently offer helpful advise and continued to support me in all situations when working on this thesis, he also gave me the opportunity to develop the project in major parts independently, exploring new directions, approaches and avenues on my own. In weekly meetings we discussed the progress and the outlook and it was an incredible joy working with and for him. But I also am grateful for his incredible dedication to his job, always reachable (even on weekends) and always a few tips or some words of encouragement in line. Nothing better could have happened to me when I decided to apply for a topic in early 2017, getting such a knowledgeable and caring advisor made this work possible in the first place.
I would also like to extend my gratitude to both Markus and Univ.-Prof. Dr.techn. Dieter Schmalstieg for giving me the opportunity to travel to Germany and to Boston, to present the results of my master's project at an international conference which not only was interesting but also rewarding, as the paper received a "Best Student Paper" award in the end.
I also have to thank my long time friends from university, Daniel Freßl, Thomas Neff and Thomas Pichler, not only did I spend most of my time at the university with them in the last 5 years, working on assignments, lunching at a near diner, having fun, they also were a source of encouragement and always provided help for my master thesis and project.

I also want to thank my family, starting with my parents Gerlinde and Walter, who always supported and encouraged me to strive for the best and make the best of every day. Their support allowed me to make the decision to switch my studies after the first year, which definitely was a great idea in retrospect and also focus on my studies without the need to earn a lot of money to sustain every day life. And also my brother Andreas and sister Anna, who both brought so much joy to my life and I can depend on in every situation,

both incredible, young people.
And last but not least, I'd like to thank my girlfriend Vera, who endured my working long hours, often on the weekend, constantly not having enough time to spend with her. She supported me in everything I did in the last years and if I worked too much or something did not go according to plan, she was always there to put me back on my feet, but also when everything was going right, she shared my joy. She is a huge inspiration to me and every second we spend together gives me so much joy and strength, this work also would not have been possible without her.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

In today's world, large, ever-changing data structures are in common use and dynamic graphs are used to represent a multitude of problem domains, including

- Communication network,
- Social-Media network,
- Biological network,
- Intelligence network

As modern graphics cards become ever more ubiquitous and comparatively inexpensive, massively parallel computing devices are available to deal with problems posed in such large-scale domains.
The need to turn to massively parallel architectures like the GPU also has its root in limitations of the hardware. In the past, it was expected that every new hardware generation increases its clock speed and transistor count. But, while the transistor count keeps growing exponentially, the clock speed has hit the so-called *power wall*[18]. This means that clock speed cannot increase or even decreases with a higher transistor count due to power and thermal limitations. Consequently, a significant speed-up is only possible by exploiting parallelism in algorithms. This is especially true as newer generations of GPUs

spend their increased transistor count on more and more processing cores.
In general, the GPU has gained a lot of traction outside the computer graphics market as a general purpose processor. The programming model is tailored to the underlying hardware architecture, which works as a Single Instruction - Multiple Data (SIMD) processor. This is one of the reasons why the GPU is well suited to deal with problems with *high data parallelism*.
High data parallelism refers to problem domains where a lot of identical work has to be performed on large sets of data. As graphs are increasing in size, data parallelism increases subsequently as well.

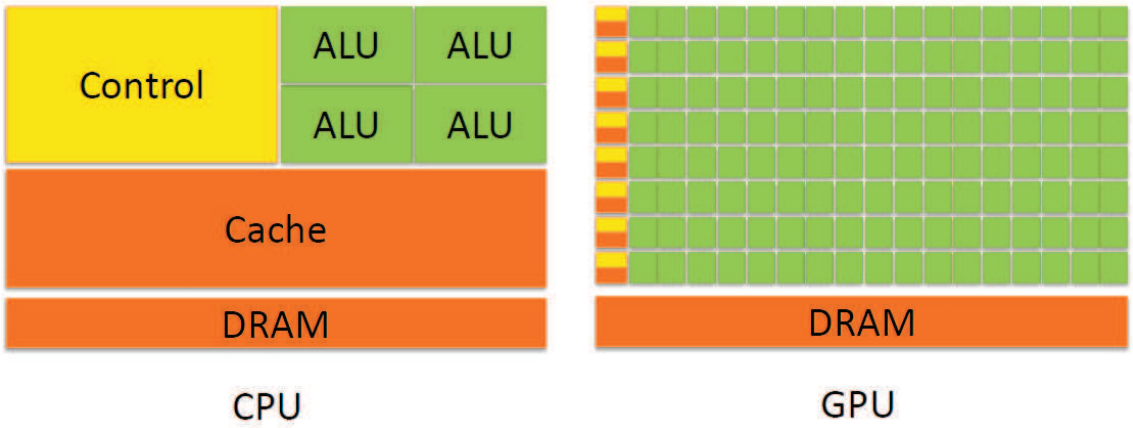

Figure 1.1: CPU hardware model compared to GPU hardware model [13]

Furthermore, the throughput-oriented architecture of the GPU fits the graph domain very well. The GPU works with thousands of threads, drawing from comparatively small caches and relying on much simpler control logic compared to the CPU, as can be seen in Figure 1.1. But since most algorithms and management work on graphs comprises of simple operations that have to be performed on millions of objects, this highlights the benefits of the GPU as a general purpose processor.

The advantages of the GPU are already utilized in many different static graph libraries and algorithms take advantage of this massive parallelism. However, most deal only with the static use case as performance on the GPU is predicated on optimization. This means being able to manage thread divergence, memory locality and optimal work distribution, which becomes highly difficult in a dynamic use case, where the memory layout and work effort are changing all the time.
In a dynamic setting there exist currently two different frameworks (as of writing this thesis), **cuSTINGER**[8] and a previous version of **aimGraph** [21]. Both frameworks are partially dynamic, utilising static vertex management data and offering update functionality for edge data only.

1.2 Outline

In this thesis we present an agile and fast, dynamic graph solution on the GPU with low memory requirements and autonomous memory management. The presented framework is fully dynamic, hence it allows for both vertex and edge updates.
Our current implementation uses the Compute Unified Device Architecture (CUDA) [13] programming language, but could also be implemented using Open Computing Language (OpenCL) [10]. The proposed solution is built from the ground up to achieve maximum performance on the GPU, allocating the free device memory upfront with a single call to `cudaMalloc()` in the beginning and managing all device memory directly on the GPU. Dynamic memory used for adjacency data is managed using variable-size pages, free vertex and page indices are stored in respective queues for later re-use. This alleviates individual reallocation calls needed to increase dynamic memory for individual vertices in the updating stage. Furthermore, it allows the framework to perform insertions and deletions with a single kernel call respectively for edge updates.
The same is true for vertices, but here additional kernel calls are used for maintenance work. The host just provides update data to the framework and the update procedure is handled independently on the GPU. An added benefit of this methodology, in addition to alleviating additional management interventions from the host, is the fact that users do not need to care about memory management themselves. The system provides not only a mechanism to hold and update the graph data structure on the GPU, but also the ability to place temporary data (e.g. edge updates, algorithmic data) on the device without the need to allocate or free this memory segment later on. Requesting a new area of memory is as simple as requesting a pointer to a provided block of memory.

The general memory layout is tailored to the architecture of the GPU, providing a vertex management segment at the beginning of the device memory that even allows locking on a per-vertex basis, if so required. This vertex management data is stored in an AOS approach as even the vertex management data is dynamic. Vertices can be allocated from the memory manager or returned to it to be placed in a queue.

The adjacency per vertex itself is stored in a combination of an edge block array and an edge block list on pages in memory. The actual memory layout can both represent an AOS as well as an SOA approach. The right choice depends on the access characteristics of the task at hand. The page size is flexible and can be tailored to the specific graph application, with smaller pages reducing the memory overhead and larger pages reducing traversal overhead. This paging approach allows for efficient memory access within a block but retains the flexibility to resize the adjacency with little overhead in case of reallocation.
The framework currently provides three different semantic modes for graphs. This permits options to include weight, type and timestamp information into the structure, enabling greater expressiveness at the cost of higher memory requirements. Depending on graph properties (like the average degree per vertex), different update implementations are pro-

vided that are optimized for the different properties. If the average size of an adjacency per vertex is small, a different update algorithm performs better when compared to large adjacencies. These optimizations can be adjusted by the user depending on the use case, different update strategies can also be mixed and matched for insertion and deletion or consecutive calls.
Building upon a queuing approach for both page and vertex indices, the framework has the flexibility to not only increase the memory requirements during run-time but also free up unneeded memory. This contributes further to the low memory requirements, especially over time, as graphs may grow and shrink back in certain regions in a long term use case.
The framework is also thoroughly evaluated using different edge and vertex update strategies. Additionally, two algorithm implementations are provided that show the suitability of the framework even for memory-intensive algorithms.

1.3 Graphs

Before diving into the actual description of the framework, a few fundamental terms are examined to provide context in terms of notation and vocabulary.
A graph G is a structure used to represent objects that may form a pair-wise relationship. The objects are typically referred to as *vertices* (also *nodes*) and a relationship between two vertices is called an *edge*. Each edge consists of exactly two elements from the set of all vertices V.
Hence, we can define a graph G as the set of all vertices V and set of all edges E, given as

$$G = (V, E) \tag{1.1}$$

There exist various forms of graphs, including (a few relevant to this thesis)

- **Undirected graph**:
  Edges do not have an orientation (e.g. edge (x, y) is identical to (y, x) for vertices $x, y \in V$)
- **Directed graph**:
  Edges have an orientation (e.g. edge (x, y) is different from (y, x) for vertices $x, y \in V$)
- **Simple graphs**:
  Undirected graph, multiple edges and loops are disallowed
- **Weighted graph**:
  Edges and/or vertices are assigned a number, referred to as a *weight*

Following the same idea of **weighted graphs**, it is also possible to assign more and different properties to vertices and/or edges. These are referred to as **semantic graphs** in this thesis, the number and kind of properties can vary depending on the application.

1.3.1 Graph types

Different application domains require different graph types, these include

- **Static graph**:
  A static graph is set up and does not change after the initialization
- **Streaming graph**:
  A streaming graph is set up initially but can be updated over time by inserting new edges and/or vertices, changes arrive as a *stream* of updates
- **Dynamic graph**:
  Similar to a **streaming graph**, but here updates are typically released as batches onto the graph structure

This thesis focuses especially on **dynamic graphs**, as this definition also includes **streaming graphs** by simply batching up an incoming update stream.

1.3.2 Scope of application

Graphs are used in a multitude of different application domains.The list below includes a few that benefit from **dynamic graphs**

- Communication networks
  - Vertices can be mobile devices in the network with the connections between devices or cell towers represented by edges
- Social-Media networks
  - Vertices may represent people with edges representing friend relationships between individuals or groups
- Biological networks
  - Modelling Protein-Protein Interaction (PPI), vertices represent proteins and edges the actual interactions between proteins
- Intelligence networks
  - Vertices may represent players in such a network with edges representing the interaction between players

and many more. Such application domains change frequently and also require high update rates to be feasible.

2 Related Work

Contents

This section introduces related work in the area of graph frameworks, both static and dynamic, as well as a few algorithms of interest.

2.1 Static Graph Frameworks on the GPU

There is a variety of static graph data structures available on the GPU, including

- **nvGraph**
  - **nvGraph**[12] (NVIDIA Graph Analytics library) offers implementations of three different, widely-used algorithms, supporting up to 2 billion edges (using a NVIDIA Tesla M40 with 24 GBs of VRAM). The three algorithms are *page rank*, *single source shortest path* as well as *single source widest path* on the GPU and is freely available as part of the *CUDA toolkit*.
- **GasCL**
  - **GasCL**[4] represents a vertex-centric graph model on the GPU. It is built using OpenCL, supporting the "think-like-a-vertex" programming model.
- **BlazeGraph**
  - **BlazeGraph**[19] offers a high-performance graph database, using its own domain-specific language, DASL, to implement advanced analytics with high-level functionality.

- **BelRed**
  - **BelRed**[5] addresses the problem that significant manual effort is required for users to build graph application on the GPU. This is done by offering a library of software building blocks which can be used to build graph applications on the GPU.
- **Gunrock**
  - **Gunrock**[20] is a CUDA library for graph processing using highly optimized operators for graph traversal while achieving a balance between performance and applicability to a wide range of problems.

2.2 Dynamic Graph Frameworks on the GPU

2.2.1 cuSTINGER

cuSTINGER[8] is a new graph data structure focussing on dynamic graphs and represents a GPU implementation of **STINGER**.
Spatio-Temporal Interaction Networks and Graphs Extensible Representation (STINGER)[1] is a high performance data structure designed for efficient handling of dynamic graphs. It uses a blocked linked list approach to store the adjacency data per vertex. In its advancement, **GTSTINGER**[7], an internal memory manager is used for allocating such edge blocks to the individual vertices. **GTSTINGER** shows that high update rates are possible on the CPU, outperforming several leading graph databases, including shared and distributed-memory based approaches.
cuSTINGER addresses the key issues of the **STINGER** data structure on the GPU, switching from edge block lists to edge block arrays, interleaving the vertex management data and allocating individual edge block arrays as a whole.
The initialization process occurs on the host by allocating individual arrays for the individual properties of the vertices, representing the static vertex management data, organised in a SOA way.
After initialization, the adjacency for each vertex is allocated from the host and the adjacency data is copied over to the device. The general memory layout bears similarities to the Compressed Sparse Row (CSR) format in the way edges are stored in edge arrays, similar to the adjacency array of CSR. The difference stems from the fact that individual properties are stored in a SOA fashion as this offers better aligned memory access if just a single feature is required. Different meta-data modes are provided to represent different graph types

- **Simple**: This mode stores the bare graph structure using just the destination vertex in the edge data array
- **Weights**: This mode adds the support for weights to the simple mode

- **Semantic**: Additionally to weights, this mode adds support for type information and also two timestamps per edge, which increases the size required per edge significantly

cuSTINGER offers a single update implementation which is based on starting *warpsized* (32 threads) blocks for concurrent adjacency access, but this approach suffers from low occupancy. Additionally, for small to medium sized adjacencies ($\leq$ 50 vertices per adjacency), this leads to an unnecessarily large number of stalling threads per update.
Furthermore, the management data structures are unnecessarily large and contribute to the significant memory usage. As memory management is primarily performed on the CPU, reallocation becomes a significant performance factor. The device can only set flags if reallocation is required. The actual management then is performed sequentially on the host, with sequential calls to `cudaMalloc()` and `cudaMemCpy()`.
cuSTINGER tries to deal with this issue by over-allocating 50% more edges per adjacency (and also over-allocate with each reallocation). This helps in the beginning or for small update batches, but hampers performance especially in continuous mode or for large update batches. Additionally, this places tighter restrictions on the size of graphs, as this over-allocation becomes significant for larger graphs with a large number of vertices.

2.3 Algorithms

Managing graphs on the GPU is one thing, but most of the time not just the pure representation is desirable. A requirement in a certain problem domain may be to efficiently derive different metrics using algorithms. There exist various kinds of algorithms operating on graphs that report on graph properties or analyse different metrics in a graph, including

- **Triangle Counting**
  - Is mainly used as a building block to find the clustering coefficients, counts triangles within a graph, a triangle in an *undirected graph* $G = (V, E)$ is given as
    $$x, y, z \in V,\ (x, y), (y, z), (x, z) \in E \Rightarrow (x, y, z)_{triangle} \tag{2.1}$$
- **PageRank**
  - Used to rank websites in search results for Google by counting number and quality of links to a page
- Connected Components
  - Finds subgraphs within a graph, where each vertex has a path to other vertices within the subgraph, but not within the supergraph
- Single-source shortest path
  - Tries to find a path between two vertices such that the sum of weights of the edges along the path is minimized
- Betweeness centrality
  - Is a measure of centrality in a graph based on shortest paths
- Community detection
  - Communities (or clusters) are groups of vertices with a higher probability of being connected with members of their group, compared to vertices outside of their group

This thesis will focus on the first two algorithms to validate the algorithmic performance of **aimGraph**, both are described in more detail in the following two paragraphs.

2.3.1 Triangle Counting

Triangle counting in a graph is a useful metric, mainly used to measure how connected a vertex is within a graph. Typically it is used as a building block for **clustering coefficients**, which is a widely used social network analytic for finding key players in a network based on their local connectivity.
A triangle in an *undirected graph* $G = (V, E)$ is given as

$$x, y, z \in V,\ (x, y), (y, z), (x, z) \in E \Rightarrow (x, y, z)_{triangle} \tag{2.2}$$

The global clustering coefficients hence can be defined as the sum of the ratios of the number of triangles over all possible triangles

$$CC_{global} = \frac{1}{|V|} \sum_{v \in V} \frac{tri(v)}{deg(v) \cdot (deg(v) - 1)} \tag{2.3}$$

so the local clustering coefficient per vertex v is defined as

$$CC_{local}(v) = \frac{tri(v)}{deg(v) \cdot (deg(v) - 1)} \tag{2.4}$$

2.3.2 PageRank

PageRank[3] is an algorithm developed by and named after Larry Page, one of the founders of Google. It offers a way of measuring the importance of web pages according to the number (and also quality) of links to each page. This assumes that important websites are more likely to be linked to by other sites.
It assigns a weight to each site E, referred to as the **PageRank** $P(E)$. Compared to other similar algorithms, PageRank does not count all links equally and normalises by the number of links on a page.
The formal definition is given as (assuming we consider Page A that is linked to by pages B, C, and D, $L(B)$ denotes the number of out-bound links of page B and d is a dampening factor that tries to model how likely a person is to continue clicking on a link):

$$PR(E) = (1 - d) + d \cdot \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} \right) \tag{2.5}$$

3 aimGraph

Contents

In the following section we sketch the design and general idea behind **aimGraph** (**a**utonomous, **i**ndependent **m**anagement of dynamic **Graph**s on the GPU) and focus on the initial memory setup and layout, the update implementations as well as performance relevant optimizations. This section is followed by a comparison to **cuSTINGER**, a look at performance differences and memory management optimisations as well as an evaluation using different algorithms.

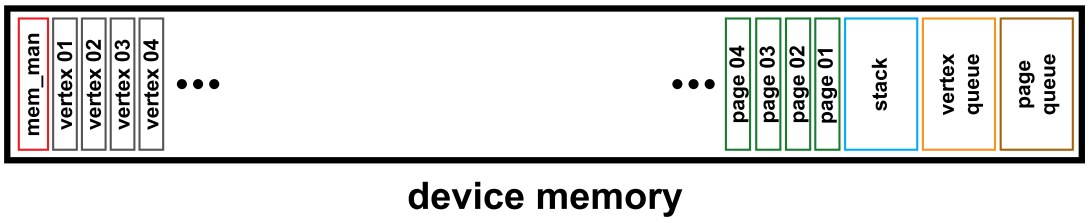

Figure 3.1: Visualization of the device memory layout as managed by the *memory manager*

3.1 Memory Layout

aimGraph initializes the system with a single GPU memory allocation, the size of this memory block is configurable, shown in Figure 3.1. This can encompass as much memory as is freely available on the device or leave memory available for other applications. All other necessary allocation calls are handled internally on the device by requesting memory from the memory manager. Memory is managed using a simple memory manager, which is initialized on the CPU (setting up the edge mode, the block size, kernel launch parameters, etc.) and then placed at the beginning of the large block of memory previously allocated on the device.
It holds a pointer to the beginning and end of the allocated device memory as well as offsets for the stack and queues. Additionally, it also stores all the necessary management data and provides access to the queuing structures holding usable vertex indices or free page indices. Using this autonomous memory management approach, the framework can facilitate all dynamic memory needs directly on the GPU. This significantly reduces the run-time overhead by removing all individual allocation calls from the host.
Calls that otherwise would have been performed sequentially from the CPU can now be parallelised directly on the GPU using **atomic access** to management variables or the respective queues.
Earlier versions of this project [21] offered a similar approach, comparable to traditional memory management as seen in CPU C/C++ programs. Static data was placed at the bottom, the dynamic area was placed right after the static data, while the temporary data on the stack grew from the top downwards.
This has been superseded by the **fully-dynamic memory model**, as now even vertices are considered dynamic and can be inserted/removed from the graph structure. Now even the vertex management data structures build on an AOS approach. Additionally, **queuing structures** are introduced capable of holding free vertex or page indices to allow for more efficient memory management.

3.1.1 Memory Manager

The **memory manager** is the central management unit on the GPU. It is allocated first from the host and takes over control over the allocated block of memory. For this use it is placed directly at the beginning of this large block. As such the memory manager can be passed to kernel calls on the GPU and provide access to the different memory regions as it holds all the necessary memory management information required to place vertices and edges in memory. Furthermore, it also includes graph properties like number of vertices/edges in memory, number of free pages and much more.

3.1.2 Vertex Data

This is an area that was completely redesigned starting from an earlier version [21]. Previously, vertex management data was considered static, similar to **cuSTINGER**[8], and individual vertex properties were stored in adjacent arrays in memory.

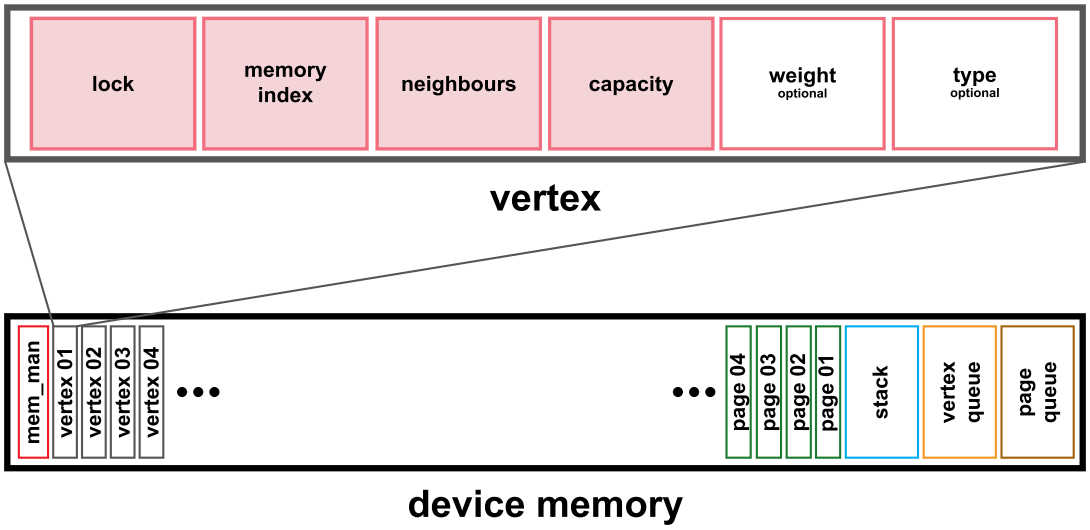

Figure 3.2: Visualization of a vertex management data structure, mandatory properties are shaded

To make this graph structure fully-dynamic and allow for vertices to be inserted/deleted from the structure, this approach switches from a SOA approach to an AOS approach. This enables the dynamic placement of vertices at a free position or at the end of the array and deletion also poses no problem. Free vertex positions can be placed in the **vertex queue** to be reused later by a subsequent vertex insertion procedure. This dynamic area is placed memory-aligned right after the memory manager, each structure holds a certain set of parameters (some of them optional).
The parameters in question are

- **lock**: Algorithms/Updates can restrict access to individual vertices using this lock, also used for concurrent updates to adjacency
- **memindex**: Holds the page index of the first page, where the adjacency data starts
- **neighbours**: Holds the number of neighbours in the adjacency
- **capacity**: Holds the maximum number of neighbours in the adjacency with the current page allocation
- **weight**(*optional*): A weight can be assigned to each vertex

- **type**(*optional*): A type can be assigned to each vertex

3.1.3 Edge Data

The edge data segment remains the same in functionality but changes its position and directionality of growth compared to the previous release. Now it grows from top to bottom, starting right after the stack area, hence vertex and edge data grow towards each other.
Two different memory layouts are provided. If multiple properties per edge are required, the AOS approach, as shown in Figure 3.3, provides better memory access characteristics. On the other hand, if just a single property is needed, using SOA, as shown in Figure 3.4, is advantageous. For **simple graphs**, both approaches are identical.

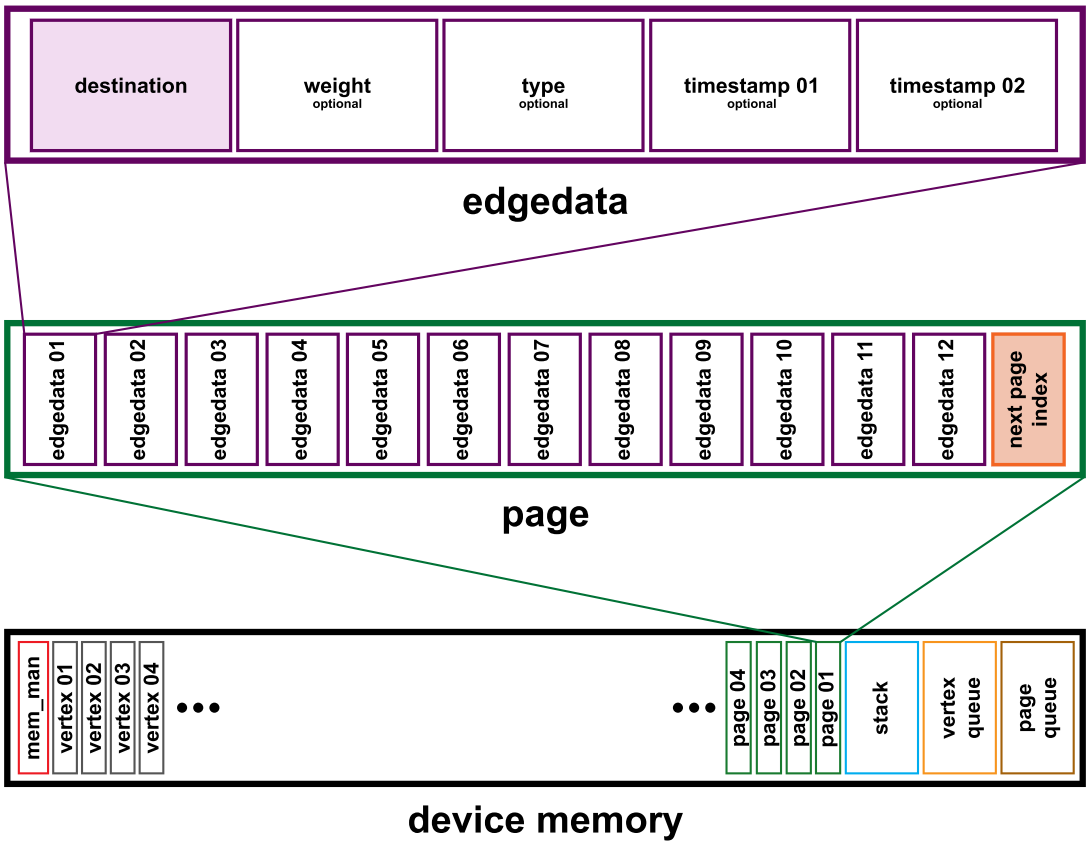

Figure 3.3: Visualization of a page and edge data structure using the AOS approach, mandatory properties are shaded. The last *4 Bytes* on a page are reserved to hold the **next page index**

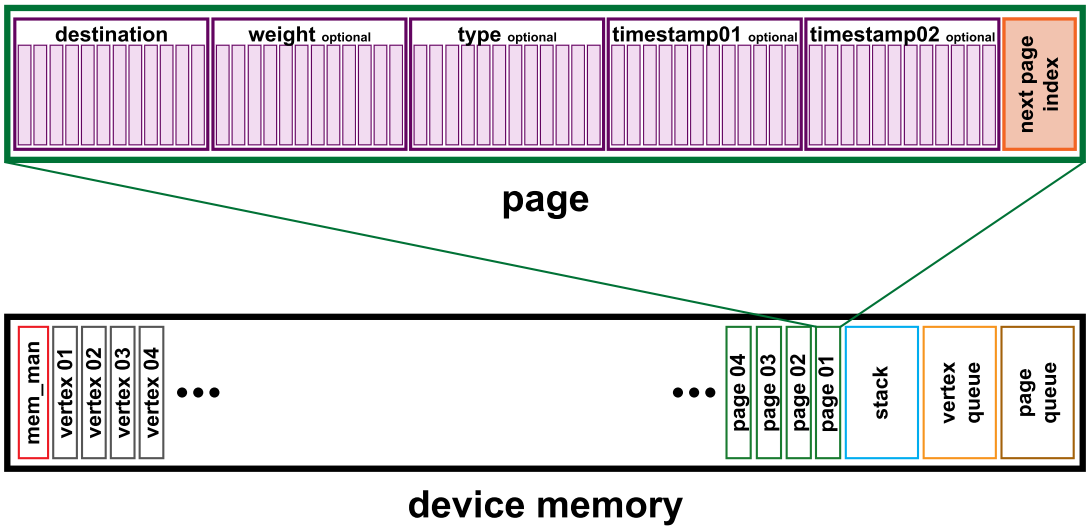

Figure 3.4: Visualization of a page and edge data using the SOA approach. The last *4 Bytes* on a page are reserved to hold the **next page index**

3.1.3.1 Edge Types

- **Simple**: This mode stores the bare graph structure using just the destination vertex
- **Weights**: This mode adds the support for weights to the simple mode
- **Semantic**: Additionally to weights, this mode adds support for type information and also two timestamps per edge

Edge data is managed in pages, the page size depends on the application and the size of individual edge data structures. Each page stores adjacency data and uses the last *4 Bytes* to indicate the location of the following page. This makes this approach a combination of a linked list and an adjacency array and allows for memory locality for vertices within an array. At the same time, this strategy avoids reallocation of the whole page if augmentation is required. Another page can be allocated by simply updating the index at the end of the last page per adjacency. Pages can also be deallocated by simply returning such a page index to the **page queue**.
For a simple adjacency, storing just the destination vertex, *64 Bytes* suffice to hold information for 15 edges per page. For semantic graphs, a larger page size is chosen to accommodate more vertices per page. Additionally, different update mechanisms profit from different memory block sizes. Depending on the update strategy and the average size of the adjacency per vertex, the page size is tuned to strike a balance between performance and memory footprint.
In the initialization step, even this adjacency data can be set up with maximum parallelism using an *exclusive prefix scan* to determine the memory requirements for each adjacency list in a pre-computation step.

3.1.4 Temporary data

In the initialization phase, but also for updating the graph and algorithms running on the graph, additional temporary data may be required. This can be for example edge updates (consisting of source and destination vertex data) or an array holding the triangle count per vertex to calculate the overall triangle count within a graph structure. Temporary data allocation has also changed significantly compared to the last revision [21]. On the one hand there is the stack data area of fixed size, as can be seen in Figure 3.1, on the other hand there is also dynamic area growing from the bottom.
The memory manager holds a stack pointer pointing to a specific position in allocated memory (right before the two queuing structures). It can use this pointer to deal out shares of this memory to algorithms or for pushing updates to the graph structure. On the other hand, if the vertex management data does not change for a certain operation on the framework (like for simple edge insertion/deletion or algorithms), data can also be placed directly after the last vertex data structure in memory. This can help reduce the pre-set size of the stack, as this area is fixed in size with the new memory layout. Keeping this area as small as possible is crucial to keep the memory footprint as small as possible.

3.1.5 Queues

To enable efficient memory management, allowing the framework to return empty page or vertex indices to the system, **index queues** are introduced.

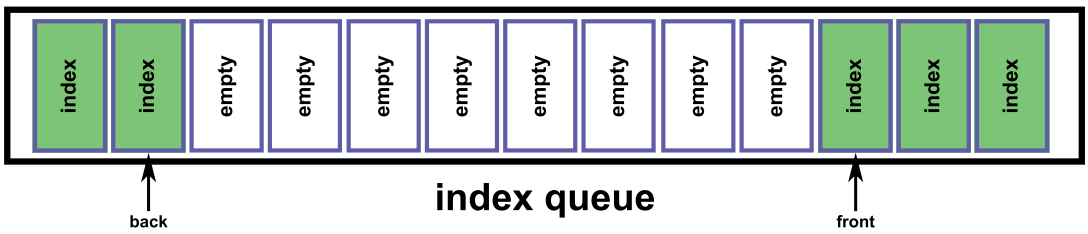

Figure 3.5: Visualization of an index queue with front and back pointer

As shown in Figure 3.5, the queue is set up with a fixed size and acts like a ringbuffer. Hence indices are calculated using the modulus of the size of the given queue.
In that way, it is possible to enqueue elements in the back and dequeue elements from the front. If the queue is empty, a new page index or a new vertex index is acquired from the memory manager directly, otherwise both can be acquired from the respective **vertex queue** or the **page queue**. Using this approach, changes in growth in the graph don't affect the required memory size as much as the previous approach would have, as a graph can grow in specific areas and shrink back and this memory will be available for other vertices later on again.

3.2 Initialization

At the start of the application, a graph is parsed into an intermediary CSR format. Right after that a preprocessing kernel is started to calculate the memory requirements per vertex. This includes computing the number of `neighbours` and from that the `capacity` and `page requirements` per vertex **in parallel**. Using the `block requirements` and an *exclusive prefix sum scan*, the overall memory offsets for all individual edge block lists can be computed. Using all this information, the initialization kernel can be run completely in parallel without regard for locking, while the CSR format is transferred into the **aimGraph** format.

3.3 Graph Types

At this point in time, **aimGraph** supports three different graph types

- Simple
- Weighted (weight per vertex and edge)
- Semantic (weight per vertex and edge and timestamps per edge)

This variety of options is implemented using templated classes and methods. Most functionality is independent of the concrete representation of edges and vertices themselves, just the modification functionality is realized via overloaded functions. Depending on the use case, one of these more advanced modes can be selected at the cost of an increased memory footprint. Choosing a larger sized edge type also increases the basic page size to accommodate a larger number of edges per page.
The framework can easily be extended to more advanced graph types by introducing new vertex and edge structures and providing overloads for the graph traversal functionality as well as accessing the vertex and edge data.

3.4 Edge Insertion

Edge updates in the current setup require a single lock per vertex to combat concurrent reads/writes to the adjacency, neighbours and capacity as shown in Algorithm 1. Access to the *memory manager* on the other hand simply requires *atomic memory access* to get a new page, if the current capacity cannot accommodate the new update and the page queue is empty.

Algorithm 1: Edge insertion using locking

```
Data: edge update batch
Result: Edges inserted into graph
Edge updates put onto stack;
while lock acquired do
    read neighbours & capacity;
    for vertices v in adjacency do
        if v == DELETIONMARKER or index ≥ neighbours then
            remember index;
        if v == edge update then
            found duplicate, ignore;
            break;
        advance in adjacency;
    if !edgeInserted and !duplicateFound then
        if page_queue.dequeue() then
            get page from page_queue;
        else
            get page from Memory Manager;
        update adjacency, index, capacity & neighbours;
    else if !duplicateFound then
        insert element at index;
    release lock;
```

For edge updates that are close to a uniform random distribution, inserting 1.000.000 edges can be achieved in a few milliseconds (detailed results in Section 6.2). Even when accessing the memory manager heavily to adjust the size of individual adjacencies, performance does not suffer significantly. We provide two implementations, optimized for different adjacency list counts. The first, which is the standard insertion mode, is shown in Algorithm 1. It completes each individual update using a single thread. This approach is especially fast for small to medium sized adjacency lists (≤ 50 vertices). If the average size per adjacency grows larger, the traversal of the graph structure becomes the bottleneck. Thus, for larger adjacency list sizes, we use an entire **warp** (32 threads) for the update. In this case, the *for-Loop* is reduced to a loop over pages and the memory access

pattern within pages can be optimized to always request a full **cacheline** per warp at once.

3.5 Edge Deletion

Edge Deletion works in a similar manner to edge insertion, the major difference is that in the most basic form it can be achieved without locking. This approach is detailed in Algorithm 2, showing the standard approach using 1 thread per update without compaction. Previously, the framework did not return empty pages to the memory manager, but simply reused them when edges are inserted for the same node again. In this way, it could avoid access to the memory manager completely during deletion.
As this thesis places an even greater focus on a low memory footprint and introduces a queuing structure to reuse page indices, empty pages are returned to the memory manager. This approach can be seen in Algorithm 3. With this approach, for each deletion update the affected adjacency is compacted and free pages are returned to be used again later on. As with the insertion process, two different implementations are provided, one launching a single thread per update and the other launching a full **warp** per update, depending on the average size of the adjacency. The code listings show the deletion process for the standard launch. When launching a full warp per update, the *for-Loop* is once again reduced and allows for better memory locality as a whole **cacheline** is fetched by the **warp** instead of individual calls to the same region by a thread in a loop.

Algorithm 2: Edge deletion without locking

```
Data: edge update batch
Result: Edges deleted from graph
Edge updates put onto stack;
read capacity;
for vertices v in adjacency do
    if v == edge update then
        atomically update adjacency & neighbours;
        one thread decreases neighbours;
        break;
    advance in adjacency;
```

Algorithm 3: Edge deletion with compaction

```
Data: edge update batch
Result: Edges deleted from graph
Edge updates put onto stack;
while lock acquired do
    read neighbours;
    shuffle_index = neighbours - 1;
    for vertices v in adjacency do
        if v == edge update then
            advance iterator to shuffle_index in adjacency;
            v ← iterator;
            iterator ← DELETIONMARKER;
            decrease neighbours;
            if shuffle_index mod page_size == 0 then
                page_queue.enqueue(page_index);
                decrease capacity;
            break;
        advance in adjacency;
```

3.6 Vertex Insertion

Vertex insertion consists of three separate kernel calls, as compared to edge updates which only require a single kernel call. This is necessary as duplicate checking becomes more difficult in this case, as can be seen in Algorithm 4.

Algorithm 4: Vertex Insertion

```
Data: vertex update batch
Result: Vertex inserted into graph
Copy Vertex updates onto stack;
if sorting_enabled then
    thrust::sort(vertex_updates);
    duplicateCheckingInGraphSorted<<<>>>(vertex_updates, graph);
    duplicateCheckingInBatchSorted<<<>>>(vertex_updates);
else
    duplicateCheckingInBatch<<<>>>(vertex_updates);
    duplicateCheckingInGraph<<<>>>(vertex_updates, graph);
vertexInsertion<<<>>>(vertex_updates, graph);
Copy mapping back to host;
if sorting_enabled then
    Copy vertex updates back to host;
```

The reason for that is that vertex insertion updates arrive on the device with a given host identifier and should receive a new device identifier, detailing the position of the new vertex. This mapping step involved here now mandates the outsourcing of the duplicate checking. This way this check can be performed much more efficiently. Without this outsourcing it would be necessary to either lock the whole graph structure, essentially performing the update sequentially, or would require expensive checks after the insertion process due to the high levels of concurrency. Duplicates can occur within a batch or from batch to graph, both have to be checked before the insertion process.

Algorithm 5: Vertex Insertion Kernel

```
Data: vertex update batch
Result: Vertex inserted into graph and page allocated per new vertex
Retrieve Vertex update from stack;
if update == DELETIONMARKER then
    update already handled;
    return
if vertex_queue.dequeue() then
    get vertex index from vertex_queue;
else
    get fresh vertex index from Memory Manager;
if page_queue.dequeue() then
    get page index from page_queue;
else
    get fresh page index from Memory Manager;
clean page by setting DELETIONMARKERs;
set Vertex Data;
return device identifier and mapping;
```

The actual vertex insertion process is straightforward as can be seen in Algorithm 5. The framework starts by acquiring a new device index and a new page index for each valid vertex update. Both first contact the respective queues for previously deleted indices, if a queue is empty, the memory manager supplies a fresh index.
Each page is cleaned at first (elements are set to an invalid entry, this is only used to stay compatible with the previous implementation not based on the queuing approach) and then the vertex is set up according to the update data. Last but not least, the new mapping (host identifier to device identifier) is reported back to the host.

3.7 Vertex Deletion

Deletion once again does not need to concern itself with duplicates, as this can be enforced using *atomic operations*.

Algorithm 6: Vertex Deletion

```
Data: vertex update batch
Result: Vertex deleted from graph
Copy Vertex updates onto stack;
if sorting_enabled then
    thrust::sort(vertex_updates);
vertexDeletion<<<>>>(vertex_updates, graph);
if graph_is_directed then
    if sorting_enabled then
        deleteVertexMentionsSorted<<<>>>(vertex_updates, graph);
    else
        deleteVertexMentions<<<>>>(vertex_updates, graph);
compaction<<<>>>(graph);
Copy mapping back to host;
```

But contrary to vertex insertion, deleting a vertex not only changes vertex management data but also has implications for adjacency data.
Deleting a vertex also includes deleting all mentions of said vertex from the graph structure and compacting the data structure, as can be seen in Algorithm 6.
In the *undirected case* these vertex mentions can be deleted directly in the deletion procedure, as can be seen in Algorithm 7. This is possible as all vertex mentions are directly known from the adjacency, and as no duplicates are present in the graph, this can even be done without locking.
If the graph is *directed*, only the allocated pages per vertex are returned to the respective queue. An additional kernel call is required as in this case the positions of vertex mentions are not obvious just from the adjacency. This way it is more efficient to start a further kernel specifically dealing with deleting these mentions. To avoid unnecessary locking procedures during this additional deletion step, the actual clean up (compaction step) is also performed in a separate kernel call. Compaction is required to potentially free up pages that then can be returned to the memory manager and placed back in the queue.
Last but not least, the now free vertex index is returned to the respective queue and the mapping change is reported back to the host.

Algorithm 7: Vertex Deletion Kernel

```
Data: vertex update batch
Result: Vertex deleted from graph and pages returned to queue
Retrieve Vertex update from stack;
if update == DELETIONMARKER then
    update already handled;
    return
if atomicExch(identifier, DELETIONMARKER) == DELETIONMARKER
 then
    deletion already handled;
    return
if graph is directed then
    return pages to page_queue;
    mentions handled in separate kernel;
else
    iterate over adjacency;
    foreach vertex in adjacency do
        Find source vertex in adjacency and delete;
vertex_queue.enqueue(identifier);
return mapping update;
```

4 Memory Management

Contents

The following section compares the different approaches to memory management as seen in the previous implementation[21] compared to the current implementation.
Both approaches have in common that management is performed independently on the GPU, only requiring a supply of update data to its update procedures from the host. The memory manager in both cases also serves the same purpose and is near identical. The memory layout has changed significantly (as shown in Section 3.1 in Figure 3.1), but the base functionality is still present and (if needed) can be operated without the improvements added as part of this thesis.
The biggest changes are

- **Fully dynamic**: The framework is now fully dynamic, it is possible to update vertex and edge data in the graph structure, compared to just edge data
- **Concurrent updates**: Due to the exclusive access policy enforced for adjacency access it is possible to perform edge insertion and deletion updates concurrently
- **Efficient memory management**: The framework now employs a queuing approach to manage memory more efficiently

4.1 Fully dynamic updates

Previously, vertex management data was considered static (as is the case with **cuSTINGER**). This has been changed in this version. It is now possible to insert and delete vertices from the structure. This is especially helpful in use cases where the actual nodes or players in such a network can vary significantly.

Telecommunication network A possible use case for fully dynamic graphs presents itself with telecommunication networks. Modelling such a network via a graph structure to retrieve metrics necessary for load balancing for example, the goal would be to derive such metrics from a real-time representation of the graph. And since users not only can move around in the network and change connections (insert/delete edges), but also turn off their device or loose the signal, it becomes necessary to be able to change the active users in this network. Being able to insert or delete vertices from a graph is highly beneficial in this case.

4.2 Concurrent updates

With the new memory management approach and the requirement for compaction in the deletion step, both update procedures utilize an exclusive access policy when updating the adjacency. This enables the framework to perform both procedures in parallel, enabled in two different variations

- **Single kernel**: Especially older hardware generations cannot guarantee concurrent execution of different kernels on the GPU, to actively test concurrent access this single kernel approach is used which, depending on the update data, performs insertion and deletion in a single kernel. Additionally, using a single kernel alleviates kernel launch overhead as the complete update procedure requires just a single kernel launch.
- **cudaStreams**: Using separate streams, each dealing with insertion or deletion respectively, newer hardware generations can also perform concurrent updates on the framework this way.

4.3 Efficient memory management

The goal of the previous implementation of **aimGraph** was to provide maximal update performance (also to be comparable to **cuSTINGER** both started with the same requirements and abilities). But since a big focus of **aimGraph** is not only performance measured in seconds, but also keeping the memory footprint as low as possible, the new approach puts a much bigger focus on **efficient memory management**.
This is especially important in case of dynamic graphs, as these can grow and shrink in

size over time. These varying theoretical memory requirements should be reflected in the actual memory requirements of the framework. Additionally, as graphs can grow in specific regions and shrink back again just to grow somewhere else, this would leave vast areas of memory allocated to parts of the structure unused that would be needed in different sections.
Using the new **queuing approach**, not only pages but also vertex positions can be returned to the memory manager, offering vastly more efficient memory management. This opens up the possibility of long-term usage of larger graphs in memory, which was previously not possible.

4.3.1 Efficiency Comparison

To evaluate the different memory requirements in different usage scenarios, three different testcases were deployed

- **Uniform**: Each iteration performs edge insertion and edge deletion sequentially with a batch size of 1.000.000, the insertion updates are generated by a call to `rand()`, the deletion updates generate the edge source the same way, the edge destination is chosen randomly from the chosen adjacency itself.
- **Sweep**: Each iteration once again performs edge insertion and edge deletion sequentially with a batch size of 1.000.000, deletion updates are generated the same way as in the **uniform** testcase, but insertion updates always focus on a small index region (e.g. the first tenth of vertex indices $\left(0 - \frac{numberVertices}{10}\right)$ as the edge source) for a few rounds and then focusses on the next region $\left(\frac{numberVertices}{10} - 2 \cdot \frac{numberVertices}{10}\right)$ and so on. This means that the insertion updates sweep over the whole index range while the deletion updates always target the complete index range.
- **Random**: Each iteration performs edge insertion OR edge deletion at random, the updates themselves are once again generated by a call to `rand()`.

All approaches have in common that insertion is more likely than deletion. This originates from the fact the batchsize is at least 10 times larger than the number of vertices in all three approaches. This means that the likelihood that edge deletion updates target the same edge multiple times is quite likely, the likelihood that edge insertion updates produce duplicates on the other hand are not particularly high. Hence, graphs tend to grow with these approaches in general.

4.3.1.1 Uniform Updates

Using the **uniform** approach, one can see in Figure 4.1 that both memory management strategies start out with the exact identical page count in the first round. After the initial setup, both grow nearly uniformly as expected, but the queuing approach grows slower as some pages can be returned to the memory manager.

4.3.1.2 Sweep Updates

As shown in Figure 4.2, the starting point is once again identical to the **uniform** approach. After that, the sweep becomes visible, the standard approach grows in size in the targeted regions and does not grow back. Conversely, the queuing approach slowly returns the pages to the memory manager once deletions free up pages. Here the greatest overall difference can be seen as areas grow and shrink much more extensively.

4.3.1.3 Random Updates

Figure 4.3 shows the **random** approach and once again the queuing approach uses significantly less memory compared to the standard approach over time.

Figure 4.1: Page allocation count over 100 update iterations with uniform update strategy. Both approaches grow as insertion is more likely, but the queuing approach grows slower.

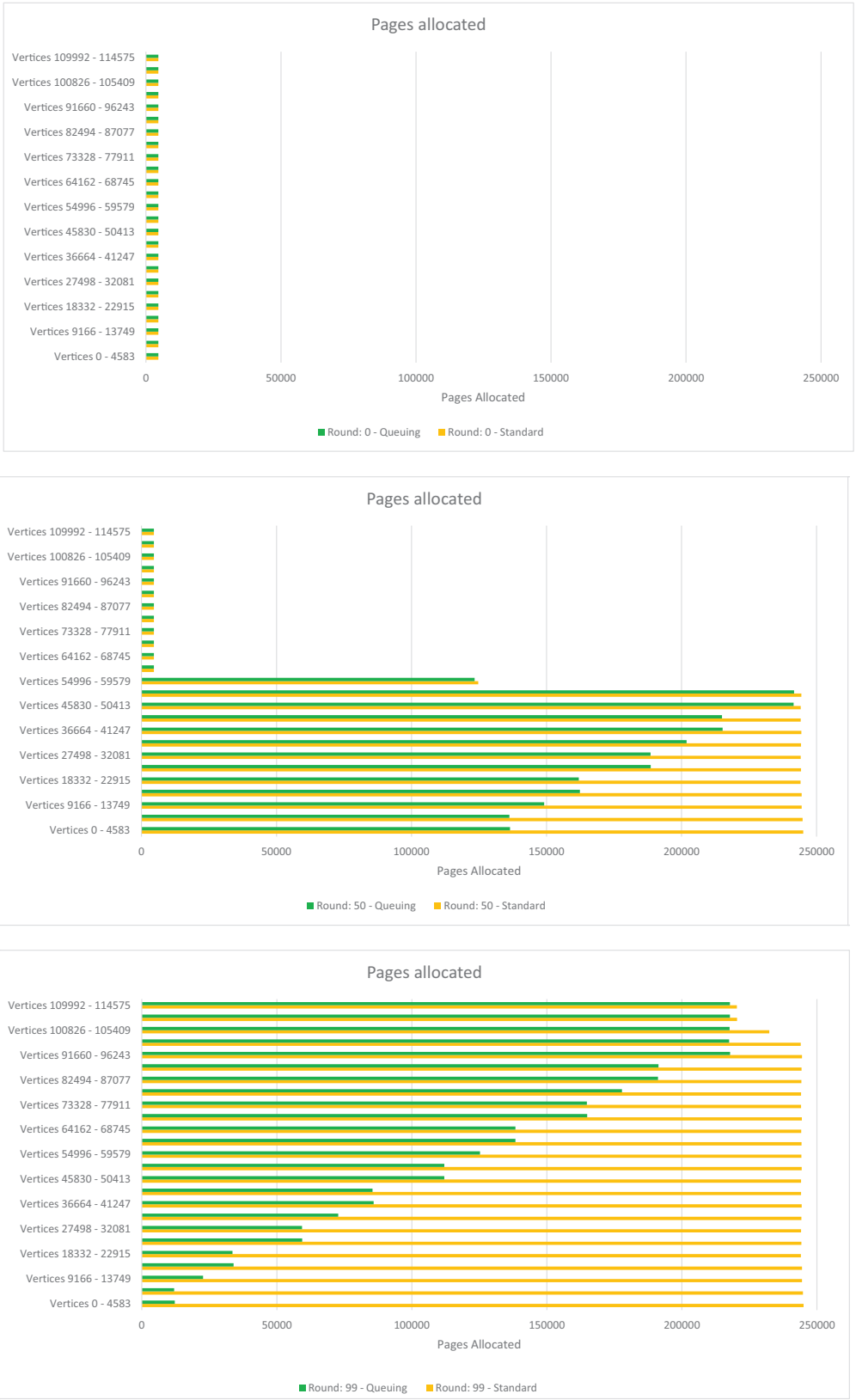

Figure 4.2: Page allocation count over 100 update iterations with sweep update strategy. The queuing approach can return empty pages once the insertions target different sections.

Figure 4.3: Page allocation count over 100 update iterations with random update strategy. Over time the queuing approach uses less memory compared to the standard approach.

5 Comparison to cuSTINGER

Contents

This section provides a comparison between the new version of**aimGraph** and **cuSTINGER** by examining the respective memory footprints and composition, as well as the time spent initializing and updating the graph structure and is followed by an evaluation of the performance differences.

	cuSTINGER	**aimGraph**
Updates	partially-dynamic	fully-dynamic
Memory Management	Primarily CPU	GPU
Reallocation	sequential on CPU	parallel on GPU
Overallocation	50%	not used
Memory Efficiency	Only increases in size	Flexible size
Adjacency access	**Atomic**	**Locking**
	may result in invalid graph	

Table 5.1: Feature comparison between **cuSTINGER** and **aimGraph**

5.1 Memory footprint

One of the biggest differences stems from the way memory allocation is performed in general. **cuSTINGER** performs sequential calls to `cudaMalloc()` from the CPU to allocate the management data and all individual edge blocks. Especially for graphs with more than a million vertices this is a significant overhead, compared to the single allocation in **aimGraph**. Another big difference lies in the memory footprint.
cuSTINGER uses pointers to

- locate attributes
- point to individual edge blocks
- point to data members within an edge block (especially prevalent in *semantic* mode)

This increases the size of the management data and also requires a full block (of *64 bytes*) just to hold member pointers. **aimGraph** on the other hand uses an indexing system (reducing the size per pointer/index from *8 Bytes* to *4 Bytes*). Additionally, it also eliminates the member pointers and additional attribute pointers by combining an efficient indexing scheme and reinterpreting memory on the fly using casts to achieve the same functionality at a fraction of the memory cost.

5.2 Initialization

As previously mentioned, **aimGraph** performs a single device memory allocation and can perform the whole setup in parallel on the GPU with little overhead. In comparison, **cuSTINGER** needs to allocate each individual edge block list from the CPU. This means also performing the pre-computation entirely on the CPU and only the actual setup of the data structure occurs on the GPU. However, as there is no offset-indexing scheme, even this kernel launch cannot utilize the GPU to its full potential, leading to an enormous performance difference in the initialization stage.

5.3 Edge Updates

Here once again, the different strategy in allocating memory pays off for **aimGraph**. Updates can be achieved in a single kernel launch with a single lock per vertex when inserting edges and even without a lock in the deletion process when no compaction is required. With compaction, both update approaches use locking to guarantee graph integrity. **cuSTINGER**, on the other hand, launches at least one kernel, which cannot utilize the whole GPU due to the lack of locking. However, this strategy also incurs a heavy penalty if duplicates are present or reallocation is required. In this case, new space must be allocated using `cudaMalloc()` and the whole edge block array of the given vertex needs to be copied over. In the worst case 5 kernel launches are required to deal with all

eventualities, leading to significantly lower update rates for highly volatile situations.
The performance difference is not as pronounced, as detailed in Section 6.2, in cases where there are no duplicates in the batch, no reallocation is necessary and the average size of an adjacency is large (≥ 50), as just a single kernel launch is required. However, in these cases there is still a chance to produce invalid graphs for duplicates in a batch. This may occur as there is no locking or contention resolution mechanism in place. Depending on the actual behaviour of the hardware thread scheduler as well as number of updates per adjacency, this problem may show up more or less often.

5.3.1 Concurrent Updates

As mentioned in Section 5.3, **cuSTINGER** does not utilize locking for updating the adjacency. Insertion positions are determined atomically, edge deletion also does not take into account additive changes to an adjacency while operating on it.
As **aimGraph** locks the adjacency per vertex to be updated, it can also perform updates (both insertion & deletion) in parallel. This can further reduce the execution time as just a single kernel launch is required, cutting down the overhead associated with additional launches.

5.4 Vertex Updates

Due to the layout of the vertex management data (as mentioned in Section 2.2.1), **cuSTINGER** is only **partially-dynamic** as a graph framework. The SOA approach makes it impossible to insert or delete vertices. In the paper [8], vertex insertions and deletions are mentioned as possible through edge update variations. However, this only allows for deleting the complete adjacency data of a vertex (which does not necessarily imply a deleted vertex), inserting a new vertex is not possible (only updating an existing vertex with new management data).
Contrary to that, **aimGraph** is **fully-dynamic** and allows for vertex insertion as well as deletion, as detailed in Section 3.6 and Section 3.7. This is possible due to the more modular structure of the memory layout, based on an AOS approach, used for the vertex management data.

6 Performance

Contents

The performance measurements were conducted using a NVIDIA® GTX 780 GPU (3 GB V-RAM), an Intel Core™i7 -3770K CPU using 16 GB of DDR3-1600 RAM. The GTX 780 is a Kepler based card with Compute Capability (CC) 3.5, and equipped with 2496 CUDA Cores on 12 Streaming Multiprocessor (SM)s. Although this is considered consumer hardware, the goal is to show differences between **aimGraph** and **cuSTINGER**. Performance on more powerful professional equipment is expected to be even higher.
The graphs used were taken from the *10th DIMACS Graph Implementation Challenge*[2]. They represent a cross section of different problem domains, a selection used for performance testing is highlighted in Table 6.1.
Both frameworks use an identical testing methodology for edge updates, starting with initialization, followed by the generation of random edge updates, which are subsequently added to the graph and then removed again. This is done 10 times and the results are averaged to produce the overall results. Only the calls to the initialization and update functions were measured. This includes copying update data to device, but excludes the update generation. This whole process is repeated 10 times and averaged again, hence the performance numbers shown display the average time of 10 rounds of initialization and 100 rounds of edge insertions and deletions respectively.

Name	Network Type	—V—	—E—
Luxembourg	Road	115k	239k
coAuthorsCiteseer	Citation	227k	815k
coAuthorsDBLP	Citation	299k	1.95M
ldoor	Matrix	952k	45.57M
audikw1	Matrix	943k	76.71M
delaunay_n20	Triangulation	1.04M	3.14M
rgg_n_2_20_s0	Random Geometric	1.04M	6.89M
hugetric-00000	Dynamic Simulation	5.82M	8.73M
delaunay_n23	Triangulation	8.38M	25.16M
Germany	Road	12M	24.74M
nlpkkt120	Matrix	3.5M	93.3M
nlpkkt160	Matrix	8M	221.17M

Table 6.1: Graphs used for performance measurement

6.1 Initialization

As shown in Table 6.2, the different memory setup procedure pays off the most in the initialization step. In all cases **aimGraph** is able to outperform **cuSTINGER** by a significant margin.

Name	Initialization (ms) aimGraph	Initialization (ms) cuSTINGER
Luxembourg	2.16	110.5
coAuthorsCiteseer	3.96	218.5
coAuthorsDBLP	4.57	289.6
ldoor	48.24	1 053.2
audikw1	77.58	1 108.6
delaunay_n20	9.35	1 092.4
rgg_n_2_20_s0	16.42	1 108.5
hugetric-00000	29.85	6 814.6
delaunay_n23	63.55	10 178.3
Germany	45.80	14 010.7
nlpkkt160	228.13	out of memory

Table 6.2: Initialization time in *ms* for **aimGraph** and **cuSTINGER**

The highest advantage is achieved when processing a large number of vertices with a comparatively low number of edges. Analysing the street network `germany`, **aimGraph** is more than 300 times faster. Even for a low number of vertices with a high degree

(referencing the sparse matrices `ldoor` and `audikw1`), the speed up achieved still reaches double digits.
This can be attributed to the fact that **aimGraph** works autonomously on the GPU and can parallelize the setup process. In contrast, **cuSTINGER** performs its setup process from the host with individual initialization calls per vertex, calculating memory requirements and allocating memory from the host directly. Additionally, **aimGraph** has a significantly lower memory footprint. Thus larger graphs can be kept in memory compared to **cuSTINGER** as can be seen for the sparse matrix network `nlpkkt160`.

6.2 Edge insertion

The first eight cases in Figure 6.1 show where **aimGraph** has a clear performance advantage. This is the case if the degree per vertex is not very large, as in those cases the over-allocation strategy of **cuSTINGER** does not provide enough space for the insertion operations. Here, both frameworks have to reallocate which is much faster using **aimGraph**, as everything is done on the GPU in one kernel. **cuSTINGER** has to reallocate from the CPU and also copy over entire edge blocks.

Figure 6.1: Performance measurement for edge insertions, using a batch size of 100.000 and 1.000.000, showing **cuSTINGER** compared to the best **aimGraph** implementation

cuSTINGER has an advantage due to their over allocation policy in the last two cases, as it allocates 50% more to reduce the need for reallocation later on. This becomes a factor if there is a comparatively low number of vertices compared to the number of edges (as seen with the sparse matrices `ldoor` and `audikw1` used in this example). But even in these cases, **cuSTINGER** does not achieve the same update rate as **aimGraph**. This is true even though we actually have to perform memory allocations, which involve more complex traversal mechanisms and locking. More on that in Section 6.2.1.
Additionally, as **cuSTINGER** does not use any form of race condition avoidance, invalid graph structure might arise in some cases. Depending on GPU scheduling, duplicates within batches are not detected and remain in the graph. The behavior of **aimGraph** is independent of scheduling and keeps a more compact memory layout.
Another factor, which becomes performance relevant, is the difference in adjacency traver-

sal. Due to the more modular structure of **aimGraph**, the traversal of individual edge lists takes longer compared to the array traversal of **cuSTINGER**. This results from the fact that the indexing scheme behind connecting multiple pages into a contiguous list requires extra cycles.
For testing purposes, turning off/down the memory overallocation of **cuSTINGER** decreases performance up to a factor of 100 as shown in Section 6.2.1. Similar behaviour can be observed by changing the update strategy by first inserting 10 batches of updates and then removing them again. This also worsens performance for **cuSTINGER** significantly, while **aimGraph** does not see a major effect on its performance.
Overall, it can be noted that **cuSTINGER** only performs well when it works within its overallocation boundaries and for larger sized adjacencies. Otherwise its performance drops significantly.

6.2.1 Overallocation

Figure 6.2: Vertex Insertion with batchsize 1.000.000, **cuSTINGER** is depicted with different overallocation factors (50%, 25%, 15% and 5%), the numbers in the bars represents the average number of additional elements allocated per adjacency with this factor

As mentioned previously, **cuSTINGER** uses an overallocation strategy to mask the performance hits from reallocating individual adjacencies. Figure 6.2 shows the performance numbers for the two sparse matrices (`ldoor` and `audikw1`) for different overallocation factors used by **cuSTINGER**. The numbers in the bars represent the average number of additional elements allocated per adjacency.

For the standard configuration, **cuSTINGER** now allocates on average 22 or 38 additional elements in the given cases. Both graphs host about one million vertices and the update procedure flip-flops between insertion and deletion batches the size of one million. Combined with the large number of additional elements, this reduces the likelihood of reallocation immensely, as it becomes rather unlikely that a single adjacency grows more than the additional element allocated.
If this overallocation factor is reduced from 50% down to 5%(as depicted in Figure 6.2), the performance difference between **aimGraph** and **cuSTINGER** grows significantly.

6.3 Edge Deletion

In case of deletions, the performance difference is slightly less pronounced compared to the insertion process, as can be seen in Figure 6.3. This is due to the fact that deletions always work without rearranging the general memory layout and also there exists no possibility of adding/removing duplicates. This also means that the performance difference for deletion using different graphs is also much smaller and the overall performance difference between **cuSTINGER** and **aimGraph** itself remains nearly unchanged as well.

Figure 6.3: Performance measurement for edge deletions, using batch size 100.000 and 1.000.000, showing **cuSTINGER** compared to the best **aimGraph** implementation

aimGraph employs two different deletion strategies, the first four cases launch a single thread per update, as the average adjacency is comparatively small to medium sized. Under these circumstances adjacency traversal is less important compared to stalling threads.

The performance benefit is therefore greatest for very small adjacencies per vertex and becomes less prominent for larger adjacencies. The other six cases use the **warpsized** approach, launching warpsized blocks, as in those cases the adjacency traversal is crucial to performance. Once again, performance is about 2× faster compared to **cuSTINGER** for the tested graphs. The main difference to **cuSTINGER** is the single kernel launch (compared to two launches for **cuSTINGER**) and the more efficient duplicate checking and update implementation with higher occupancy.

6.4 Overall performance

The following Table 6.3 shows an overall comparison of **cuSTINGER** and **aimGraph**. This provides a listing of the timings for the entire test set for edge insertion and edge deletion with a batchsize of 1.000.000, as well as an overall timing including the initialization time, timing measurements given in *seconds*. Measurements in both cases include transferring update data to the device.

Graphs	Insertion **aimGraph** 1.000.000 in *seconds*	Deletion **aimGraph** 1.000.000 in *seconds*	Insertion **cuSTINGER** 1.000.000 in *seconds*	Deletion **cuSTINGER** 1.000.000 in *seconds*	Overall **aimGraph** 1.000.000 in *seconds*	Overall **cuSTINGER** 1.000.000 in *seconds*
luxembourg	0.585	0.524	20.279	2.882	1.131	24.266
germany	1.014	0.706	10.613	3.041	2.178	153.664
coAuthorsD.	0.685	0.623	14.637	3.001	1.354	20.534
coAuthorsC.	1.062	0.404	7.142	2.929	1.505	12.256
delaunay_20	0.920	0.407	11.213	3.155	1.421	25.292
delaunay_n23	2.038	0.667	15.279	2.995	3.341	118.452
rgg_n.	1.002	0.477	4.499	2.770	1.593	18.354
hugetric.	1.016	0.434	1.976	2.964	1.748	73.086
ldoor	1.976	0.813	2.189	3.073	3.271	16.315
audikw1	2.057	0.981	2.416	3.263	3.813	16.765

Table 6.3: Performance comparison for **aimGraph** and **cuSTINGER** including overall performance

As noted in Table 6.3, **aimGraph** is able to deliver more than 100 million updates per second in more than half of the tested cases. Even though these measurements include the time required for initializing the graph ten times throughout the testing procedure. In the closest cases, **aimGraph** still holds a performance edge of more than 4x which grows to more than 70x due to the parallel initialization procedure.
Overall it can be noted that **aimGraph** performs particularly well for graphs with a large number of vertices and comparatively small to medium sized adjacencies. But even for graphs with large adjacencies the performance difference to **cuSTINGER** remains significant, albeit not as large.

6.5 Concurrent Updates

aimGraph bases its update approach on an exclusive access policy to the adjacency data (except if no compaction is required, then deletion can be done without locking). Due to this setup it is possible to allow for concurrent edge insertion and deletion on a graph, as each adjacency is protected by a lock.
Figure 6.4 shows a comparison between the concurrent update approach (using a single kernel for both insertion and deletion) and sequential edge insertion and deletion with a batchsize of 100.000. The *standard update* procedure with *compaction* enabled is utilized here to examine the performance difference. It is demonstrated that the concurrent approach outperforms the sequential approach in every case, but not by a big margin. This difference can be explained by the reduced overhead of the concurrent approach as just a single kernel launch is required and all updates are passed to the copy engine of the device also just once and not twice.

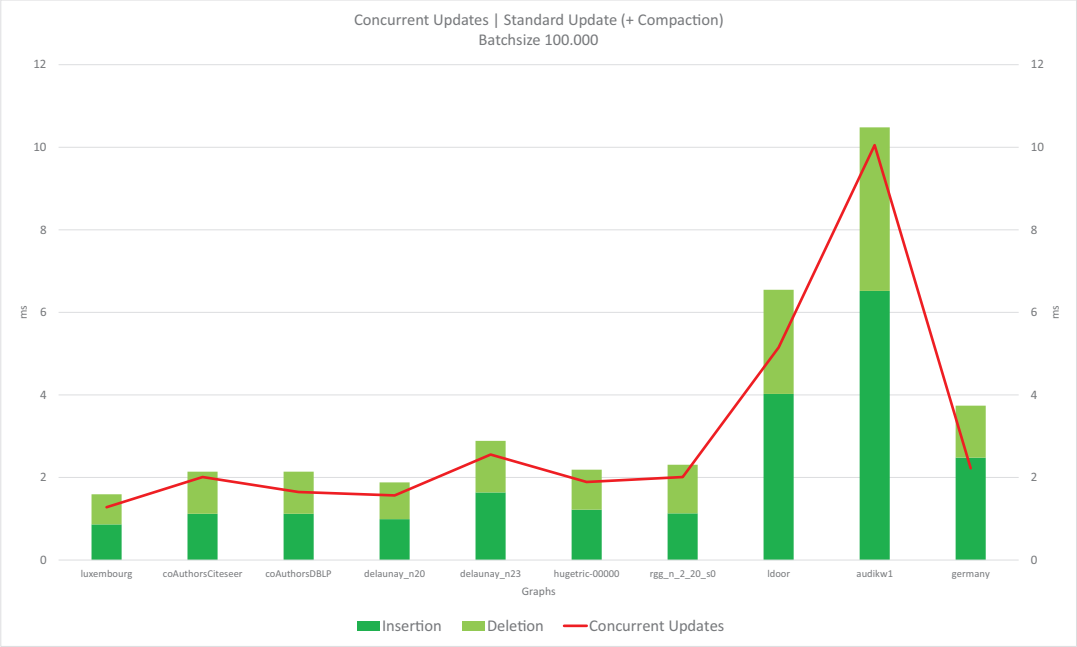

Figure 6.4: Performance comparison between sequential edge insertion and deletion and concurrent updates for a batchsize of 100.000

6.6 Vertex Insertion

The actual process of **vertex insertion** is comparatively straight forward with the given framework, as detailed in Section 3.6. Most time is spent on correctness measures, checking duplicates within the update batch and between update batch and the graph.
Figures 6.5 and 6.6 show the different timings for the given graphs, split up into the different algorithm stages.
The insertion process for all tested batchsizes and graphs always stays below 1 ms, but especially for graphs with a larger number of vertices the duplicate checking between batch and graph becomes the bottleneck. Because of that an additional *sorting stage* is introduced beforehand, this notably speeds up both duplicate checking stages, as both can then utilize binary search to check correctness. Even with this optimization the duplicate checking dominates in graphs with a larger number of vertices. For small graphs the additional stage introduces more overhead than can be gained in performance. Hence in such cases this stage is not executed.
As these correctness stages require most of the processing time, the execution time and the number of vertices in the graph are directly correlated.

6.7 Vertex Deletion

Vertex Deletion is a little bit more complicated compared to vertex insertion, as it does not suffice to delete the existing vertex management data per vertex. Additionally, all mentions to the vertex in the graph must be deleted.
Figures 6.7 and 6.8 show the performance numbers for vertex deletion in case of an *undirected graph*, Figures 6.9 and 6.10 offer performance numbers for *directed graphs*.
In both cases the same graphs are used, the difference results out of the assumptions that can be made in the *undirected* case (each edge in an adjacency can be found in reverse as well). In this case the vertex mentions are deleted directly in the deletion kernel, this prolongs the vertex deletion stage.
For *directed* graphs there is an extra step involved, as it is not directly obvious where the directed edges might reside in memory. This additional kernel once again profits from sorting the updates if the graphs become larger, utilizing binary search in the deletion process. For smaller graphs this optimization is not used, as once again the overhead introduced outweighs the benefits gained.

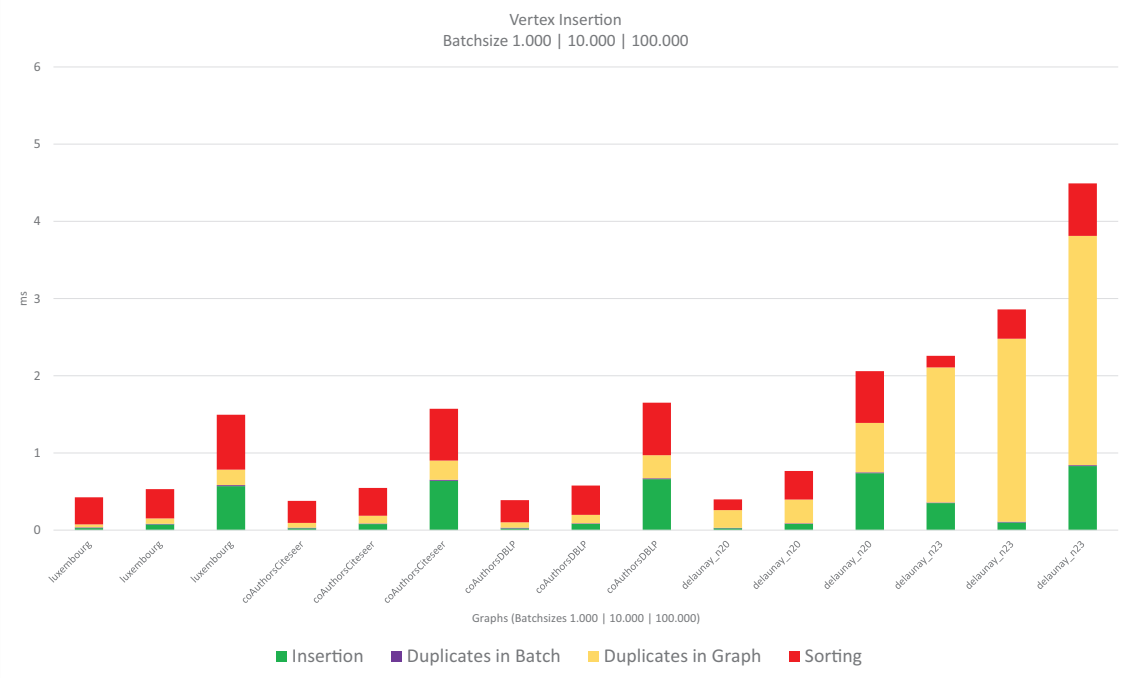

Figure 6.5: Vertex insertion for batchsizes 1.000 — 10.000 — 100.000

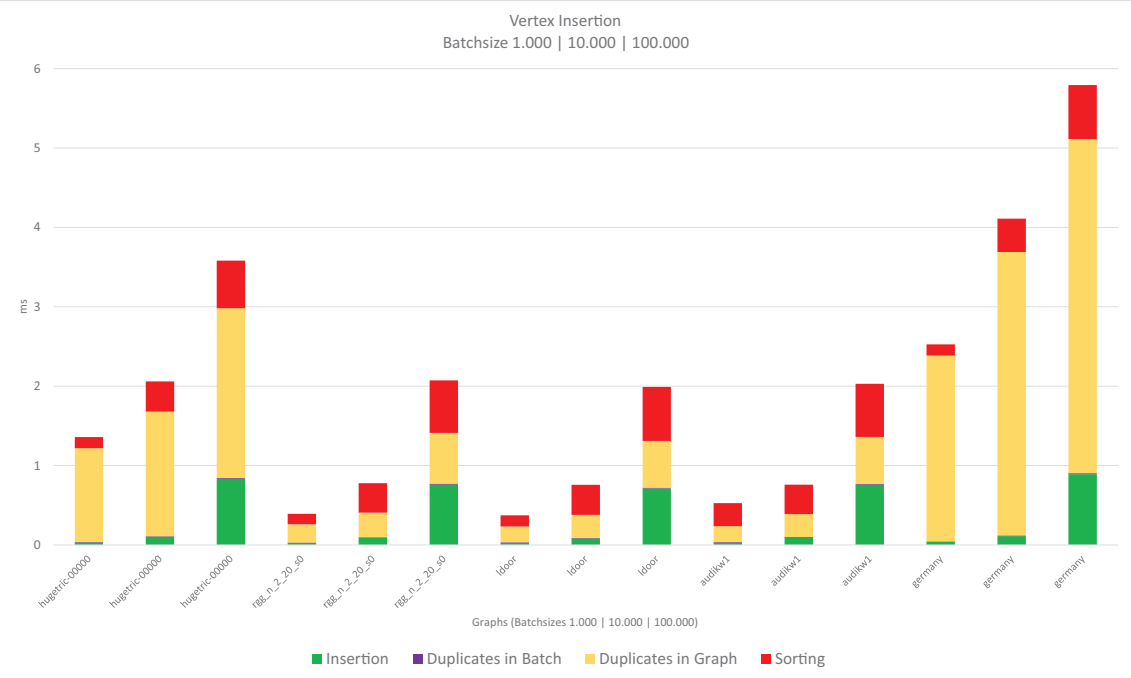

Figure 6.6: Vertex insertion for batchsizes 1.000 — 10.000 — 100.000

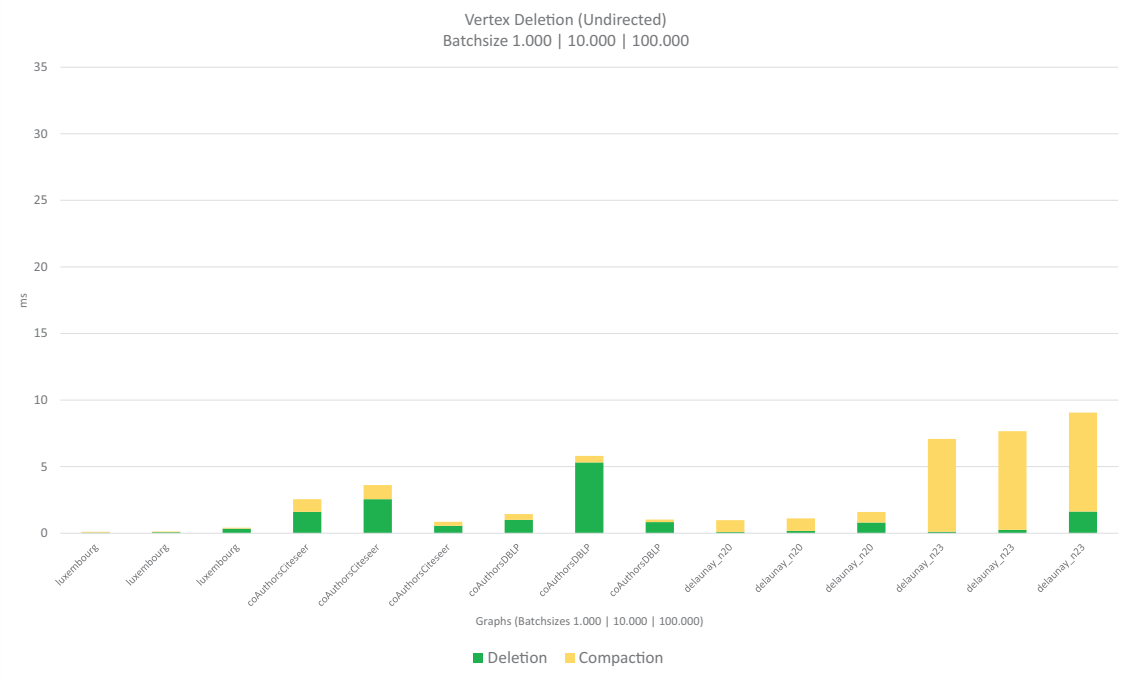

Figure 6.7: Vertex deletion in an undirected graph for batchsizes 1.000 — 10.000 — 100.000

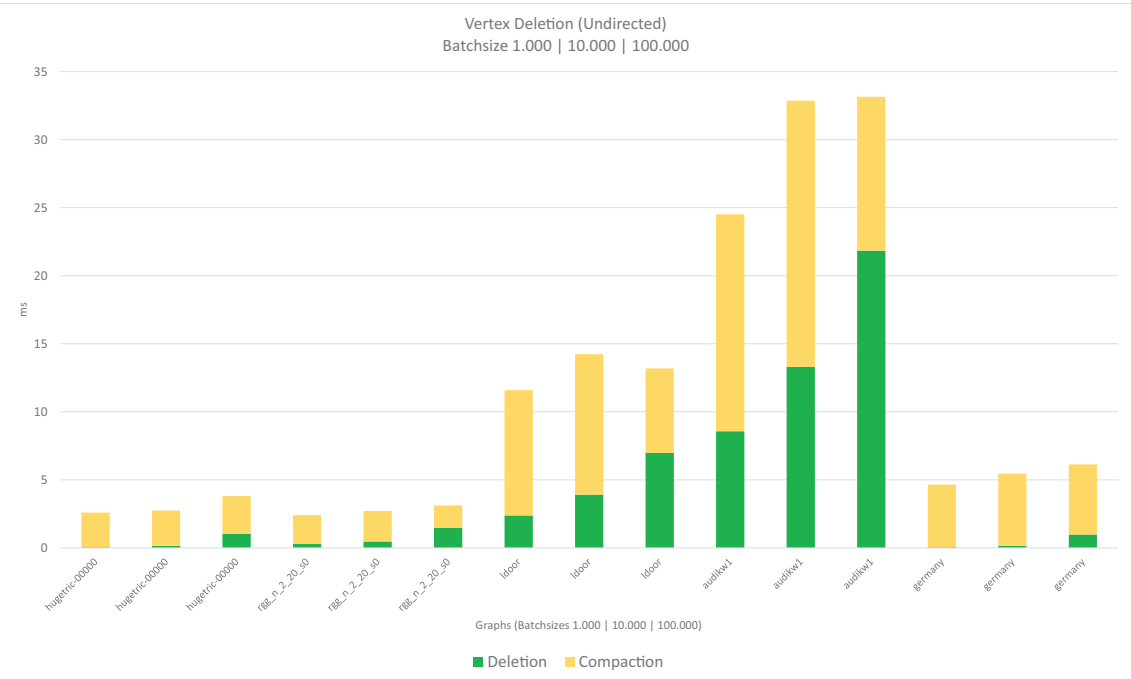

Figure 6.8: Vertex deletion in an undirected graph for batchsizes 1.000 — 10.000 — 100.000

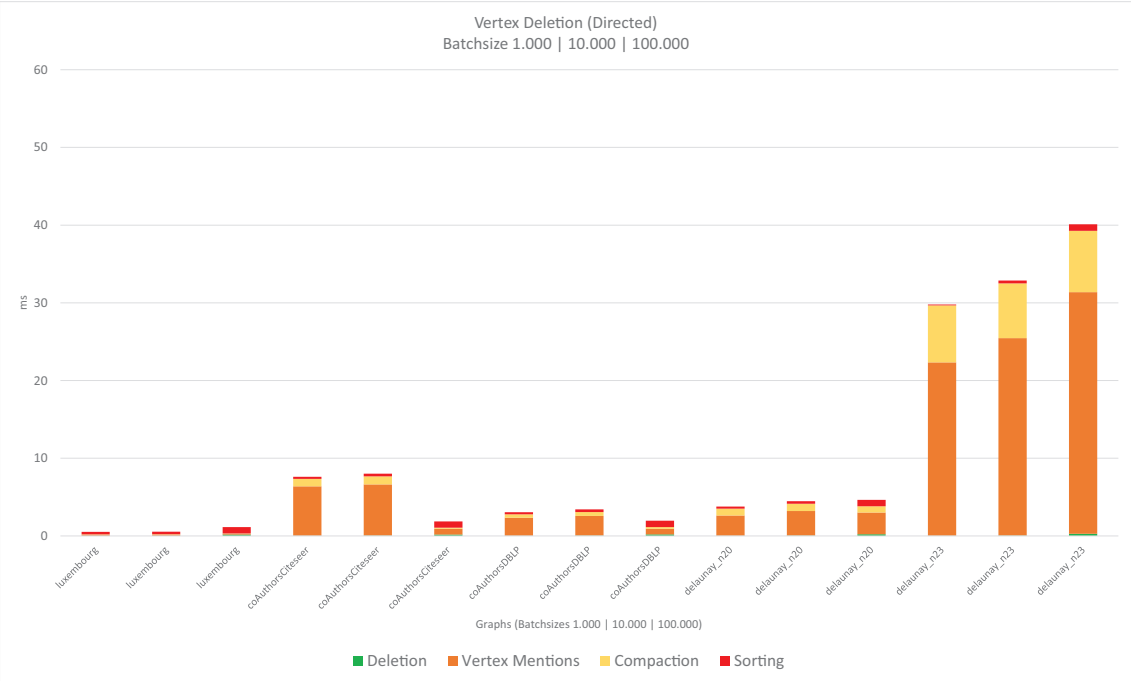

Figure 6.9: Vertex deletion in a directed graph for batchsizes 1.000 — 10.000 — 100.000

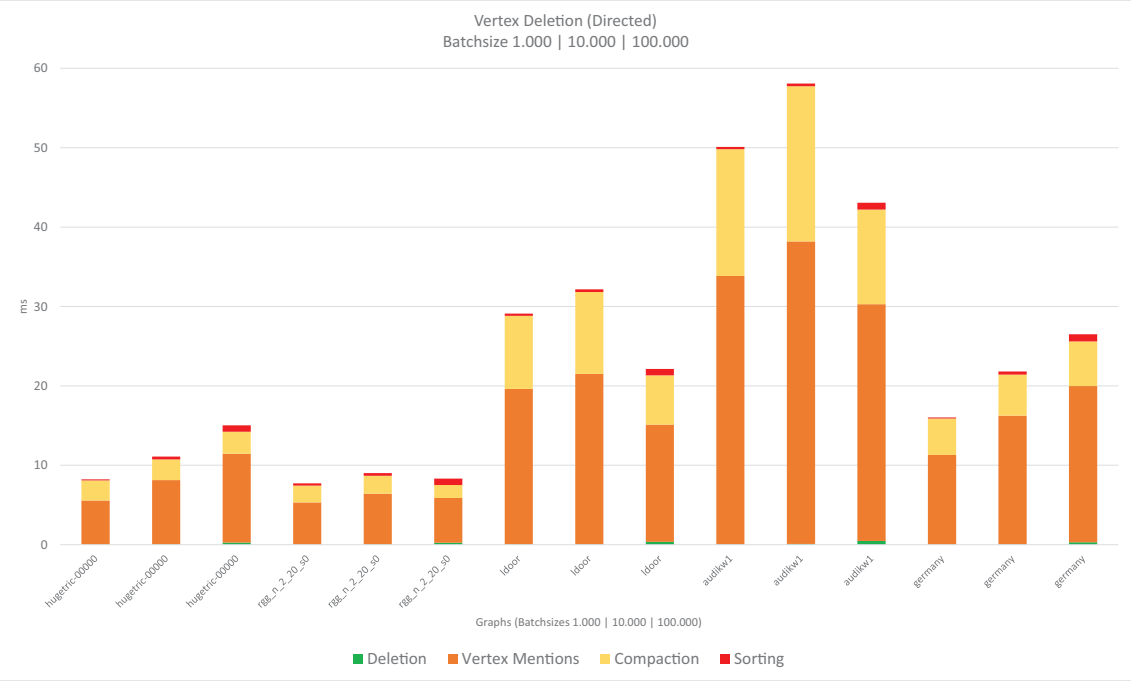

Figure 6.10: Vertex deletion in a directed graph for batchsizes 1.000 — 10.000 — 100.000

7 Algorithms

Contents

The following section starts by introducing a scheme to enable the framework to balance the workload according to the pages allocated, detailed in Section 7.1. In Section 7.2 an algorithmic performance comparison between **cuSTINGER** and **aimGraph** is provided, whereas **aimGraph** provides naive and balanced implementations of the algorithms given.

7.1 Work-Balance Preprocessing

The GPU operates using a **throughput-oriented design**, hence distributing work as well as possible to all threads is highly beneficial to performance. As a lot of operations and algorithms have to traverse the adjacencies of individual vertices, just starting a thread per vertex does not distribute work adequately. This is especially true if those adjacencies become longer or vary highly in size. Due to the underlying architecture utilizing a SIMD approach, it is necessary to keep thread divergence at bay.
Hence, a work balancing scheme is introduced that calculates an offset scheme to locate individual pages in memory. This can then be used to start one thread per page in memory. This is done by providing offset vectors to the threads, so that each thread receives the vertex index in question as well as the page number for that vertex.
Figure 7.1 shows the performance overhead introduced by this preprocessing step as well as the pages allocated in memory.

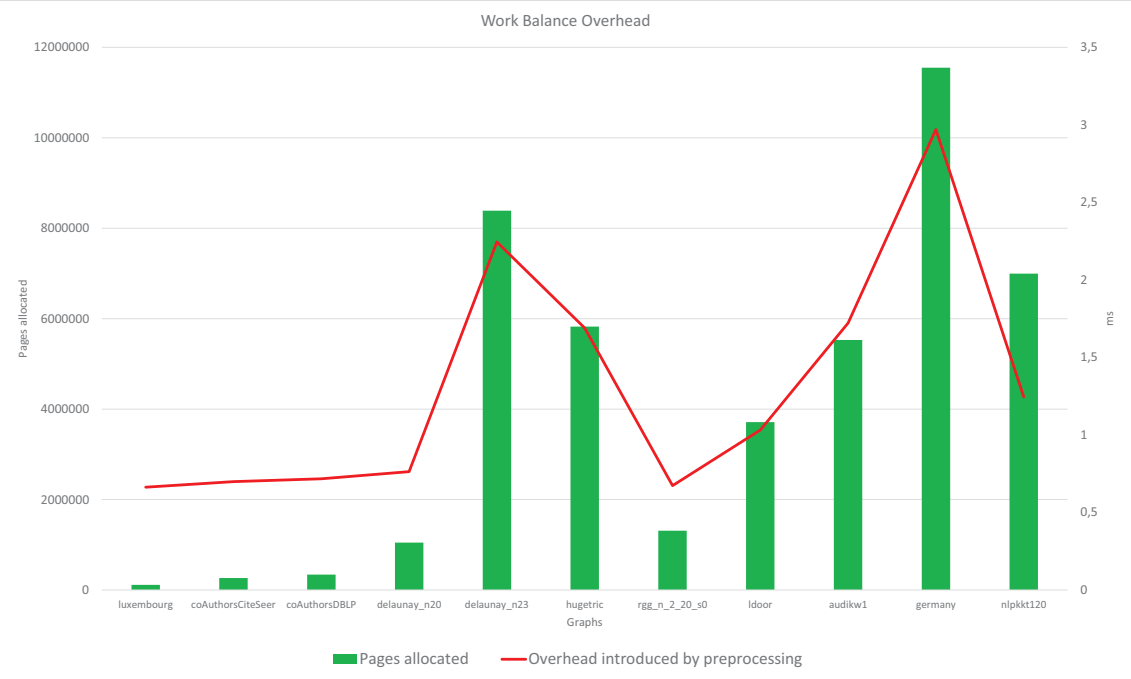

Figure 7.1: Visualization of the overhead introduced by work balancing, bars representing the pages allocated in memory for the given graph, the line represents the time needed for the work balancing

Figure 7.1 shows that there is an obvious and clear correlation between the overhead introduced and the pages in memory. However, the benefit gained also grows much more in cases with more pages in memory. For the tested graphs, the performance overhead falls somewhere between 0.5 *ms* and 3.0 *ms*. Most of this overhead results from the *prefix-sum scan* required to get the total number of vertices as well as the offsets.

7.2 Algorithm Performance

7.2.1 Static Triangle Counting - STC

In this section, six different variants of STC are compared. Two of them are included in the **cuSTINGER** framework [9] and are based on a list intersection algorithm called **Intersect Path**.
The underlying data structures are

- **CSR**: One implementation uses the **CSR** format as an underlying data structure
- **cuSTINGER**: The other implementation is based on the **cuSTINGER** framework

The algorithm operates on two stages of parallelism. The first stage balances the vertices on the multiprocessors and the second stage balances the adjacency access using different block sizes. One drawback of the given algorithm is the requirement that **adjacencies have to be sorted** for it to work.

Figure 7.2: Comparison between three different implementations for **aimGraph** (naive, balanced and warpsized), **cuSTINGER**, **CSR** as well as the performance ratio of **cuSTINGER** to the best **aimGraph** implementation.

The other four implementations are based on **aimGraph**. The actual algorithm implementation is straight forward and does not require any preconditions like sorting, the four implementations differ in the following way:

- **Naive**: Straight forward implementation, one thread per adjacency

- **Balanced**: One thread per page, introduces overhead due to the index computation but distributes work more evenly
- **Warpsized**: One warp (32 threads) per page, more efficient memory access, but also more stalling threads
- **Combined**: Combines the **balanced** and the **warpsized** approach, hence a warp per page is started

Figure 7.3: Comparison between three different implementations for sorted **aimGraph** (naive, balanced and warpsized), **cuSTINGER**, **CSR** as well as the performance ratio of **cuSTINGER** to the best sorted **aimGraph** implementation.

The performance numbers are recorded in Figure 7.2. The line plot represents the performance factor between **cuSTINGER** and the best **aimGraph** implementation for the given use case.
aimGraph is able to outperform **cuSTINGER** in the first five cases, as those cases have comparatively small adjacencies (on average ≤ 15). The next two cases are closer but **cuSTINGER** pulls slightly ahead with a growing adjacency count. In these cases, *work balancing* is required as the adjacency size varies more and there are a few very large adjacencies (multiple 100 and for `coAuthorsCiteSeer` even one adjacency ≥ 1000). The last three cases are clearer, as all have larger adjacencies (45, 76 and 27 edges per adjacency on average), in this case **sorting** significantly reduces the workload for **cuSTINGER** compared to **aimGraph**.

Introducing **sorting** into the **aimGraph** implementations, as can be seen in 7.3, improves performance in all cases, now **aimGraph** is able to outperform **cuSTINGER** in the first six cases with one close case and even the last three cases are now less than 5× slower compared to **cuSTINGER**.

The remaining difference stems from the fact that the underlying memory layout of **cuSTINGER** allows for random access to the adjacency data. This way it is possible to reduce the search overhead from $O(n)$ to $O(ld(n))$ for n elements in an adjacency.
aimGraph on the other hand has to perform page traversal in the case of random access to the adjacency, hence for larger adjacencies decisive overhead is introduced. Currently, implementations that assume sorted adjacencies do the same linear traversal from beginning to end, but may break earlier, if the search index becomes larger than the current element in the adjacency.

7.2.1.1 Discussion

- **aimGraph** is well suited even for memory intensive algorithms, if the average size of an adjacency does not grow incessantly (average size ≤ 20 edges per adjacency)
- For unbalanced graphs, using **work balancing** can significantly reduce execution times
- **aimGraph** performs very well for graphs with smaller adjacencies due to the focus on a smaller memory footprint
- **aimGraph** is not well suited for random adjacency access
  - If random adjacency access is required, page traversal should be limited as much as possible
    * Try to re-use adjacency iterators
    * Increase page size to limit need for traversal

7.2.2 PageRank

As detailed in Section 2.3.2, **PageRank** is a fairly straightforward algorithm concerning its implementation. The algorithm has to traverse the adjacencies of all vertices and compute the contributions of all relationships for each vertex.
This means that every edge is touched exactly once, the same is true for every vertex, the only point of convergence remains the *PageRank vector* itself used to sum up the contributions.
Figure 7.4 shows the direct comparison between **cuSTINGER**, a **naive implementation** (one thread per vertex, traverse adjacency and compute contributions) and a **balanced implementation** (preprocessing step required, one thread per page, only traverse page and compute contributions) using **aimGraph**.

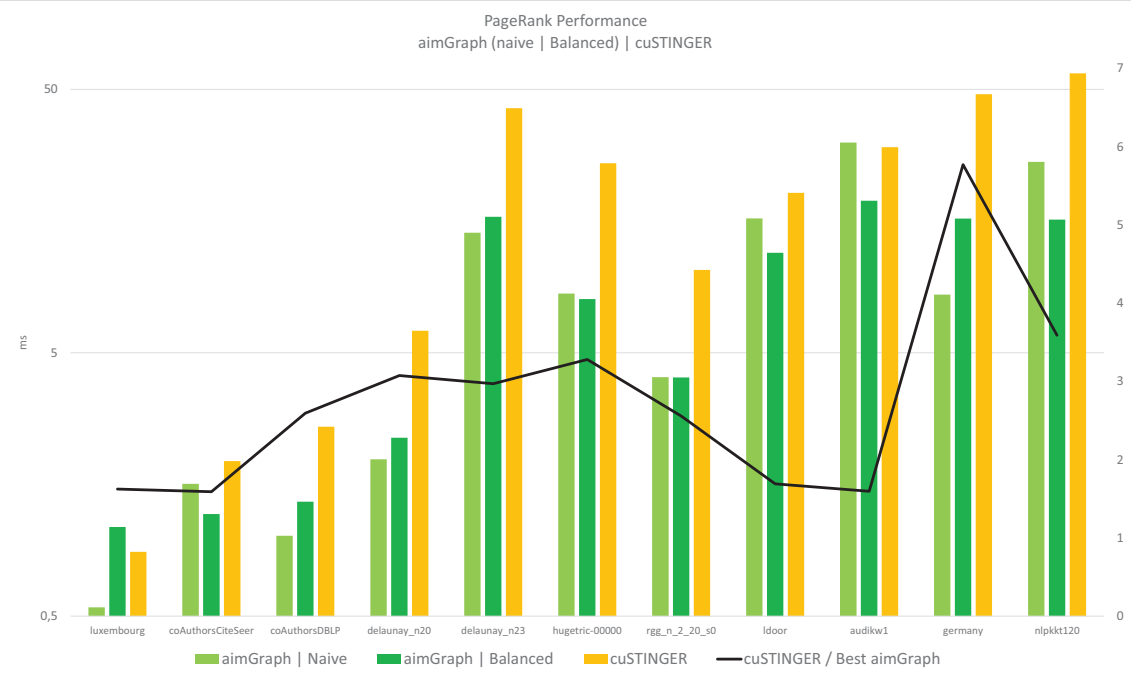

Figure 7.4: PageRank Calculation Performance comparison

Even the **naive implementation** using **aimGraph** is able to outperform **cuSTINGER** for the given graphs, but the performance difference narrows for larger graphs as **cuSTINGER** employs a balanced approach as well. The **balanced implementation** delivers better performance in cases where on average multiple pages are used per adjacency, this is especially noticeable for the two sparse matrices, `ldoor` and `audikw1`. For graphs with small adjacencies, especially the two street networks, `luxembourg` and `germany`, the preprocessing step introduces more overhead as a disadvantage as what could be gained from a more balanced kernel, as the launch will be near identical to the naive approach in these cases.

7.2.2.1 Discussion

- As PageRank has moderate memory access requirements, **aimGraph** does not require optimisations to perform well
- Unbalanced and larger graphs once again profit from using **work balancing**
  - For smaller graphs the overhead negates the performance benefits
- **aimGraph** is able to outperform **cuSTINGER** in every tested scenario

8 Conclusion & Future Work

Contents

8.1 Conclusion

aimGraph is a memory-efficient, fully dynamic graph solution with autonomous memory management directly on the GPU. Compared to previous work [21], the framework is now fully-dynamic. This means that not only edge updates, but also vertex updates are supported at very high update rates. All of that is possible without transferring the graph data structure to and from the host for updating.
The solution is purpose-built for the GPU, reducing memory requirements as much as possible. This is achieved by using an indexing structure instead of pointers and managing the device memory directly on the device, without the need for copying and allocating new pages from the host. In this way, edge updates can be achieved with a single kernel call. Additionally, now both insertions and deletions can be performed concurrently in a single kernel call. Vertex updates can also be achieved with little overhead. Vertices and all references to them can be deleted from the structure or new vertices added to it. Furthermore, concurrent initialization of the graph structure is possible due to the direct management of memory on the GPU, reducing the setup time significantly.
The current implementation supports different semantic modes for vertices and edges (including simple, weighted and semantic graphs). Additionally, different update methodologies are provided to developers that can be selected for specific workloads or different graph types to tune performance in every scenario. To verify new implementations, different verification methods are included as well. To validate algorithmic performance on the data structure, two algorithm implementations (PageRank and static triangle counting)

are provided.
In its current state, **aimGraph** is able to outperform its competitor in all tested graphs (tested on a NVIDIA® GTX 780 with 3GB VRAM) in edge update rate (2x to 20x higher update rate) as well as initialization time (10x to 300x faster initialization time). Furthermore, it adds the ability to update vertices as well, while remaining more memory efficient by using an advanced memory management scheme based on a queuing approach. The framework can hold tens of millions of vertices and hundreds of millions of edges in memory (on the given consumer-level GPU and depending on the semantic mode).
It is able to process 20 - 100 million edge insertion per seconds and between 50 - 150 million edge deletions per second. Vertex updates can also surpass a few million updates per second, but depend more on the size of the graph due to verification methods required. As shown in Section 7.2, **aimGraph** also performs very well in memory-intensive algorithms, the only caveat being that random access to the adjacency does not perform as well.

Overall, **aimGraph** offers an efficient and fast, fully dynamic graph framework with a focus on efficient and autonomous memory management directly on the GPU. Using different update implementations, the framework delivers high update rates tailored to different use cases and graph properties, for both vertex and edge updates.

8.2 Future Work

To further update and expand the capabilities, the objective is to investigate the use of a **mega-kernel approach**, launching just a single kernel and distributing the resources on the fly, directly on the GPU. Doing so would eliminate the need for separate kernel launches from the host all together, reducing the overhead associated with a kernel launch. Additionally, this would allow the usage of the framework in an even more autonomous way, frequently reporting back different metrics that are derived from algorithms running in constant time intervals. Furthermore, this would also incorporate the possibility of updating the graph structure and running algorithms simultaneously.
Another objective would be to assess the possibility of using **multiple GPUs** for **aimGraph**. This would allow the framework to access significantly more memory which in turn enables bigger graphs to be kept in memory. However, this would require challenging changes to the memory layout to be able to distribute the work across multiple GPUs. The goal would be to find a beneficial work distribution while also focussing on adjacency locality as much as possible.
Another area of interest would be the usage of **out-of-core** graphs, referring to graphs that cannot be held in memory in their entirety. This would allow for even bigger graphs to be used with **aimGraph**.
Last but not least, the goal would be to provide **further algorithm implementations**, currently the framework offers multiple variants of triangle counting, comparing to versions by **cuSTINGER** [9],[14], as well as a PageRank implementation. Further algorithms of

interest would include connected components [16], single-source shortest path[6], betweeness centrality for static graphs[15], betweeness centrality for dynamics graphs[11] and community detection[17].

A

List of Acronyms

AOS	Array of Structures xv, 3, 14–16, 37, 61
CC	Compute Capability 39, 61
CPU	Central Processing Unit vi, xv, 2, 8, 9, 14, 35, 36, 39, 42, 61
CSR	Compressed Sparse Row 8, 19, 61
CUDA	Compute Unified Device Architecture 3, 7, 8, 61
GPU	Graphics Processing Unit iii, v, vi, xv, 1–3, 7, 8, 10, 13, 14, 28, 35, 36, 39, 41, 42, 51, 57, 58, 61
HPC	High-Performance Computing iii, vi, 61
OpenCL	Open Computing Language 3, 7, 61
PPI	Protein-Protein Interaction 5, 61
SIMD	Single Instruction - Multiple Data 2, 51, 61
SM	Streaming Multiprocessor 39, 61
SOA	Structure of Arrays xv, 3, 8, 15–17, 37, 61
STINGER	Spatio-Temporal Interaction Networks and Graphs Extensible Representation 8, 61

List of Publications

My work at the Institute for Computer Graphics and Vision led to the following peer-reviewed publications.

B.1 2017

Autonomous, Independent Management of dynamic Graphs on GPUs

Martin Winter, Rhaleb Zayer and Markus Steinberger
In: *Proceedings of IEEE High Performance Computing Conference (HPEC'17)*
September 2017, Boston, USA
(Accepted for oral presentation, received *Best Student Paper Award*)

Abstract: In this paper, we present a new, dynamic graph data structure, built to deliver high update rates while keeping a low memory footprint using autonomous memory management directly on the GPU. By transferring the memory management to the GPU, efficient updating of the graph structure and fast initialization times are enabled as no additional kernel calls or reallocation procedures are necessary since they are handled directly on the device. In comparison to previous work, this optimized approach allows for significantly lower initialization times (up to 300x faster) and much higher update rates for significant changes to the graph structure and equal rates for small changes. The framework provides different update implementations tailored specifically to different graph properties, enabling over 100 million of updates per second and keeping tens of millions of vertices and hundreds of millions of edges in memory without transferring data back and forth between device and host.

Bibliography

[1] Bader, D., Berry, J., Amos-Binks, A., Chavarria-Miranda, D., Hastings, C., Madduri, K., and Poulos, S. (2009). Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. In *Tech. Rep.* Georgia Institute of Technology. (page 8)

[2] Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D. (2013). Graph partitioning and graph clustering. 10th dimacs implementation challenge workshop. In *ser. Contemporary Mathematics, no. 588.* (page 39)

[3] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems.* Stanford University. (page 11)

[4] Che, S. (2014). Gascl: A vertex-centric graph model for gpus. In *IEEE High Performance Embedded Computing Workshop (HPEC).* (page 7)

[5] Che, S., Beckmann, B. M., and Reinhardt, S. K. (2014). Belred: Constructing gpgpu graph applications with software building blocks. In *IEEE High Performance Embedded Computing (HPEC).* (page 8)

[6] Davidson, A., Baxter, S., Garland, M., and Owens, J. D. (2014). Work-efficient parallel gpu methods for single-source shortest paths. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS).* (page 59)

[7] Ediger, D., McColl, R., Riedy, J., and Bader, D. A. (2012). Stinger: High performance data structure for streaming graphs. In *IEEE High Performance Extreme Computing Conference (HPEC).* Georgia Institute of Technology. (page 8)

[8] Green, O. and Bader, D. (2016). custinger: Supporting dynamic graph algorithms for gpus. In *Conference Paper.* Georgia Institute of Technology. (page 2, 8, 15, 37)

[9] Green, O., Yalamanchili, P., and Munguia, L. (2014). Fast triangle counting on the gpu. In *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms.* (page 53, 58)

[10] Group, K. (2017). OpenCL - The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`. [Online; accessed 21-May-2017]. (page 3)

[11] McLaughlin, A. and Bader, D. (2014). Revisiting edge and node parallelism for dynamic gpu graph analytics. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* (page 59)

[12] NVIDIA (2016). nvGraph. `https://developer.nvidia.com/nvgraph`. [Online; accessed 12-May-2017]. (page 7)

[13] NVIDIA (2017). NVIDIA CUDA Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. [Online; accessed 01-May-2017]. (page xv, 2, 3)

[14] Polak, A. (2015). Counting triangles in large graphs on gpu. In *arXiv preprint*. (page 58)

[15] Sariyüce, A. E., Kaya, K., Saule, E., and Catalyürek, . V. (2013). Betweenness centrality on gpus and heterogeneous architectures. In *6th Workshop on General Purpose Processor Using Graphis Processing Units*. (page 59)

[16] Soman, J., Kishore, K., and Narayanan, P. (2010). A fast gpu algorithm for graph connectivity. (page 59)

[17] Soman, J. and Narang, A. (2011). Fast community detection algorithm with gpus and multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. (page 59)

[18] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, pages 202–210. (page 1)

[19] SYSTAP, L. (2017). BlazeGraph. `https://www.blazegraph.com/`. [Online; accessed 01-May-2017]. (page 7)

[20] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2015). Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices, vol. 50*. (page 8)

[21] Winter, M., Zayer, R., and Steinberger, M. (2017). Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC '17)*. Graz University of Technology. (page 2, 14, 15, 18, 27, 57)