Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Path-finding on GPUs for Database Systems

Author: Thijs Dreef (2764266)

1st supervisor: Peter Boncz
daily supervisor: Daniel ten Wolde (Centrum Wiskunde & Informatica)
2nd reader: Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 8, 2025

"Never judge a function by its size"
by Casey Muratori

Abstract

Graphs are becoming an increasingly popular way to model real-world relationships due to their ability to represent complex connections in data. As the volume of graph-structured data continues to grow, it has captured the attention of database researchers and developers. DuckPGQ is an extension for DuckDB that introduces support for SQL/PGQ, enabling users to construct graphs directly from relational data and perform advanced graph operations such as shortest path computation, pattern matching, and more.

DuckPGQ uses MS-BFS (Multi-Source Breadth-First Search) to compute the length of the shortest path between large sets of source-destination pairs. Executing this algorithm takes up the majority of the query execution time, making it a target for optimization. The CPU version of this algorithm is bandwidth-bound. GPUs have a higher memory bandwidth; thus, offloading the execution to the GPU is expected to result in a significant speedup.

In this thesis, we develop a GPU implementation of the MS-BFS algorithm using WebGPU, enabling our implementation to run on any system that supports a modern web browser. We implement optimizations using GPU subgroup operations, direction switching, and different parallelization schemes. The GPU algorithm is built into a library, which allows for easy integration with DuckPGQ

We evaluate the performance and scalability of the GPU version of the algorithm using the Linked Data Benchmark Council's Social Network (LDBC) Social Network Benchmark (SNB) dataset. The experiments demonstrate that, depending on the available hardware, a GPU version of the algorithm can outperform the CPU version by up to 2.25 times. Using subgroup operations allows for better scaling. Direction optimization can provide minor speedups on larger graphs, but it can slightly degrade performance on smaller graphs. The GPU version does not perform orders of magnitude faster due to the low computational cost. Without computation, there is no way to perform latency hiding, resulting in the GPU stalling while waiting for data to arrive, which leads to underutilization of the GPU.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Introduction

1.1 Context

There is an increasing amount of data that needs to be analyzed, and this data is often highly interconnected, allowing it to be represented as a graph (80). Executing graph workloads in traditional Relational Database Management Systems (RDBMS) requires users to write large and complex SQL queries (28). In contrast, Graph Database Management Systems (GDBMS) offer a more user-friendly approach. This difference has led to the rise of specialized databases, such as Neo4j (2) and ArangoDB (1). With the introduction of Property Graph Queries (PGQ) in SQL:2023 (28), formulating graph queries within an RDBMS has become simpler. DuckPGQ implements the SQL PGQ within DuckDB.

DuckDB (74) is an in-process, high performance analytical database system known for its efficient columnar storage (3) and vectorized execution engine (83). DuckPGQ is an extension for DuckDB that implements the execution of various graph-based operations, including traversals, shortest path calculations, and pattern matching.

Path-finding, also known as the shortest path problem, is a fundamental operation in graph databases (73), required for shortest path queries, network analysis, and recommendation systems. Computing the shortest path in DuckPGQ utilizes the Multi-Source Breadth-First Search) (MS-BFS) algorithm, a specialization of the Breadth-First Search (BFS) algorithm designed to efficiently handle multiple source-destination pairs, as proposed by Then (86). DuckPGQ's initial implementation of the MS-BFS algorithm was sequential and utilized a User Defined Function (UDF) (85). The work by Ren (76) enables the use of a custom operator, which allows for a custom multi-threading model (76). This work was further continued by Daniël ten Wolde using thread-local CSRs and MS-PBFS[1]. These efforts significantly improved the performance of path-finding operations in DuckPGQ. However, path-finding remains a costly operation for DuckPGQ.

A Graphics Processing Unit (GPU) is available on most systems in the form of a discrete GPU or an Integrated Graphics Processing Unit (IGPU). The intended use of a GPU is to render computer graphics, but it has evolved to also accelerate compute-intensive tasks such as machine learning (23). The GPU is better suited to handle compute-intensive tasks than the CPU, as it has more cores and higher bandwidth memory. These higher core counts come at the tradeoff of having less complex cores and a different execution model. The execution model used by GPUs is Single Instruction Multiple Threads (SIMT), which executes the same instruction on multiple cores, sacrificing flexibility for higher throughput.

[1] https://github.com/cwida/duckpgq-extension/tree/pathfindingoperator-two-phase-csr-lock-free

Research projects that aim to run algorithms on the GPU often utilize NVIDIA's GPU programming model named CUDA (18, 49, 56, 95), which limits the algorithm to run only on NVIDIA hardware. WebGPU is a new GPU programming model originating from a web specification, intending to bring GPU compute to the web (16). Since it is used for the web, it should run on all hardware that can run a browser. Thus, a WebGPU-based algorithm can run on nearly all devices that have a GPU.

Several studies have been conducted on accelerating path-finding algorithms using the GPU, such as A* (18) and IBFS (56). These studies report a significant performance improvement when using a GPU to run the algorithm instead of a CPU, indicating that utilizing a GPU to the path-finding operator in DuckPGQ could be worthwhile.

Accelerating path-finding operations on the GPU has been done before, but integrating these GPU-accelerated operations into a complete system's execution engine remains an open problem, according to Bonifati et al. (20). The primary aim of this thesis is to improve the efficiency of path-finding operations in DuckPGQ by implementing a GPU-based path-finding operator using WebGPU. The research involves developing and optimizing a GPU path-finding algorithm that executes on a multitude of devices, evaluating the performance of different systems and operating systems. By addressing these objectives, this thesis aims to provide insight into the performance characteristics of different hardware and when to offload computation to the GPU.

1.2 Goals

The goal of this thesis is to improve the performance of path-finding in DuckPGQ by developing a GPU implementation of the MS-BFS path-finding algorithm. DuckPGQ utilizes this algorithm to compute both the shortest path and its length. These two objectives produce different outputs, allowing for use-case optimizations. Developing an algorithm using CUDA would eliminate a significant portion of the DuckDB/DuckPGQ user base, as it is utilized in various setups, ranging from laptops and desktops to server hardware. Not all of these devices have access to an NVIDIA GPU, but most have access to some form of a GPU, ranging from integrated graphics to discrete cards from any vendor. To support the wide range of hardware used by DuckDB users, WebGPU will be used. WebGPU enables all GPU and OS combinations that support either DirectX12, Vulkan, or Metal to execute the GPU-accelerated MS-BFS algorithm.

The secondary goal of this thesis is to provide insight into the performance characteristics of different GPU hardware. Most GPU acceleration studies focus on performance insights using server-grade NVIDIA hardware using CUDA. By benchmarking these multiple systems with varying hardware configurations, we aim to provide insight into the performance of these algorithms on consumer-grade hardware. Slower hardware can also exacerbate performance bottlenecks in a GPU implementation, such as synchronization costs, data transfers, and driver overhead.

By documenting the bottlenecks identified during the development of a WebGPU path-finding algorithm, we aim to provide insights into the challenges and best practices when developing GPU-based algorithms.

1.2.1 Research questions

We aim to answer the following research questions with this thesis

1. How to design a GPU based path-finding algorithm?
2. How can GPUs be used to accelerate the path-finding algorithm in DuckPGQ?
    - What are the bottlenecks of a GPU path-finding algorithm?
    - When does the GPU provide a speedup compared to a CPU?
    - How to optimize a GPU based path-finding algorithm?
3. How do different system configurations influence performance?
    - How do performance characteristics differ between integrated and discrete cards?
    - How do performance characteristics differ between Operating Systems?
    - How does WebGPU compare to a CUDA implementation?

Background

2.1 DuckDB

DuckDB (74) is an open-source, in-process high-performance analytical database that runs on hardware ranging from server-grade to consumer-grade hardware. The design goals are to be fast, reliable, portable, and user-friendly. It uses a columnar storage layout (3) and vectorized (19) execution model, to allow for high throughput and low-latency query execution. Being in-process allows DuckDB to avoid data transfers between the host and database processes, unlike a typical client-server architecture. These properties make DuckDB ideal for embedded analytics and interactive data analysis. These design choices simplify deployment and enable better data locality, resulting in faster query evaluation.

DuckDB allows for extensions to add functionality; these extensions are separate components that provide new data types, functions, or storage formats. These extensions can be dynamically loaded at runtime by the user.

2.1.1 Morsel-driven parallelism

Traditional databases processes records one tuple at a time, Boncz et al. (19) popularized vectorized execution. With vectorized execution, multiple tuples are processed simultaneously, enabling the CPU to apply loop pipelining and other optimizations. By storing tuples in a columnar layout, it becomes faster to read a subset of a row, but more costly to update a row (3).

DuckDB employs a technique called morsel-driven parallelism (55) to distribute query execution efficiently across multiple CPU cores. By partitioning the workload into small chunks called "morsels", each core can work on one morsel independently. Using these morsels enables fine-grained parallelism and dynamic load balancing, even on NUMA machines.

DuckDB's vectorized query engine treats 2048 tuples as one vector. Morsels have a size of 60 * 2048 ≈122,000 tuples. It then follows that a thread works on 122,000 tuples at a time. The scheduler ensures that each thread receives a new morsel as soon as it completes its current one, providing a constant flow of work and minimizing idle time.

2.1.2 DuckPGQ

DuckPGQ (91) is an extension for DuckDB that brings the SQL:2023 PGQ (28) (Property Graph Queries) into DuckDB. DuckPGQ supports complex graph queries typically found in graph databases by utilizing path-finding algorithms to handle shortest paths and cheapest

paths efficiently. Allowing for analytical querying and management of graph data within a unified SQL-based system.

DuckPGQ utilizes Batched Bellman-Ford (86) for weighted graphs and MS-BFS (86) for unweighted graphs (i.e., where the weight is 1 for all edges). Combined with the morsel-driven parallelism model, executing the algorithm on the tuples within a morsel allows users to benefit from the multiple CPU cores in their system when the workload is sufficiently large.

2.1.3 Compressed sparse row (CSR)

DuckPGQ uses the Compressed Sparse Row (CSR) data structure to represent graphs. The CSR data structure consists of two arrays: the edge array and the vertex array. The vertex array contains the offset into the edge array for each vertex ID, while the edge array contains the edges for each vertex, as illustrated in Figure 2.1. To associate a tuple with its vertex, the row ID is used as a dense, sequential identifier.

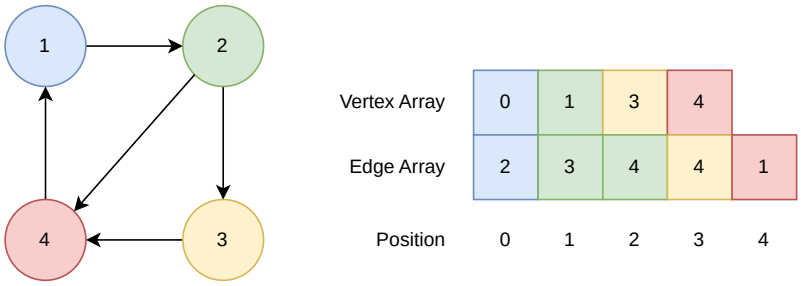


Figure 2.1: CSR example

To loop over the outgoing edges of a vertex n we use the algorithm depicted in Algorithm 1 where E is the edge array and V is the vertex array. The benefit of using a CSR is that we benefit from cache locality, as $V[n]$ and $V[n+1]$ are sequential, as well as the loop over the edge array.

Algorithm 1 Looping over edges of CSR

```
function OUTGOINGEDGES(V, E, n)
    edges ← ∅
    offset ← V[n]
    while offset < V[n + 1] do
        edges ← edges ∪ E[offset]
        offset ← offset + 1
    return edges
```

2.1.4 Multi-Source Breadth-First Search

DuckPGQ uses the MS-BFS algorithm as proposed by Then (86). This algorithm expands upon the single source BFS by executing it on multiple source-destination pairs

concurrently. The number of pairs evaluated by the algorithm at once is denoted by ω. This factor depends on the maximum bit width of the available Single Instruction Multiple Data (SIMD) instructions. For example, with AVX-512 (48) we can scale up ω to 512 as we can execute bit operations of width 512 efficiently. Reducing the number of times the path-finding algorithm needs to be run by a factor ω. The benefit of using MS-BFS can be observed when doing path-finding for x sources, where a single source BFS needs to run x times, where MS-BFS runs $ceil(x/\omega)$ times. As long as MS-BFS is not slower than a factor ω compared to the single-source algorithm, we will see a performance benefit.

The algorithm works like a regular BFS except that the frontier contains the nodes which are to be visited by **any** active BFS, effectively only exploring nodes once at each level. We start by adding ω source nodes to our visit and seen set in their respective BFS instance, using the search index to determine the bit to set. We then loop over the visit set, where we mark the nodes corresponding to the outgoing edges as seen. When such a node has not been seen previously by **any** search, it is added to the visitNext set, marked as seen, and the BFS callback function is called with this node, which can be used to construct the BFS search tree. The algorithm is depicted in Algorithm 2.

Algorithm 2 MS BFS

```
function MSBFS(V, E, sources, func)
    visit ← (0_0, ..., 0_|V|)
    seen ← (0_0, ..., 0_|V|)
    visitNext ← (0_0, ..., 0_|V|)
    for each source_i ∈ sources do
        visit[source] ← 1 << i
        seen[source] ← 1 << i
    while visit ≠ ∅ do
        for i = 0, ..., |V| − 1 do
            if visit[i] = ∅ then continue
            for each v ∈ outGoingEdges(V, E, source) do
                d ← visit[source] & ∼ seen[v]
                if d ≠ ∅ then
                    visitNext[v] ← visitNext[v] | d
                    seen[v] ← seen[v] | d
                    func(v)
        visit ← visitNext
        visitNext ← (0_0, ..., 0_|V|)
```

2.1.5 IBFS

IBFS, as proposed by Liu et al. (56) is a specialization of the MS-BFS algorithm for GPUs. IBFS aims to minimize branch divergence by moving the $visit[i] \neq \varnothing$ to a separate step. In the identification step, the algorithm determines the frontier for all concurrent searches and adds it to a queue, which we refer to as the joint frontier queue. During the expand step, it propagates the seen state of the frontier to all neighboring nodes. The algorithm

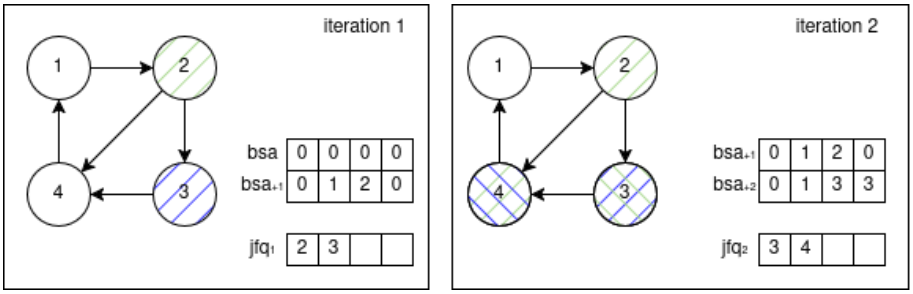


Figure 2.2: IBFS example

alternates between these steps until the joint frontier queue is empty. Once the queue is empty, it indicates that all traversals have visited all nodes, and the algorithm terminates.

IBFS maintains the status of all nodes in Bitwise Status Arrays (BSA); these arrays contain the seen status of all nodes for a given iteration. To find the frontiers for the next iteration, the identify step does an $\oplus$ (XOR) between the status arrays of the current level and the last. Any node that differs between two iterations is in the current frontier and is thus appended to the joint frontier queue (JFQ). This algorithm is depicted in Algorithm 3; an illustration of the algorithm is shown in Figure 2.2.

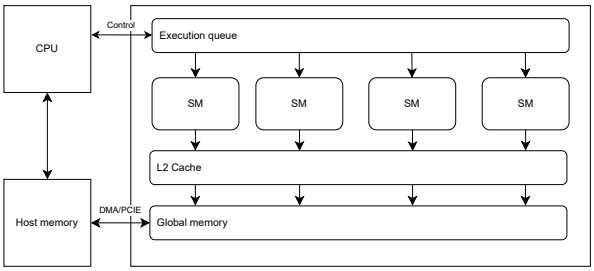
Algorithm 3 Top down IBFS

function IDENTIFY(V, bsa, bsa_{+1}, jfq)
  jfq $= \varnothing$
  for $i = 0, ..., |V| - 1$ **do**
    if $bsa[i] \oplus bsa_{+1}[i]$ **then**
      jfq.enqueue(i)
function EXPAND(V, E, bsa, bsa_{+1} jfq)
  for source $\in$ jfq **do**
    for neighbour $\in$ $outGoingEdges(V, E,$ source$)$ **do**
      bsa$_{+1}$[source] $\leftarrow$ bsa$_{+1}$[source] **OR$_{\text{atomic}}$** bsa[neighbour]
function IBFS(V, E, sources, destinations)
  bsa $\leftarrow (0_0, ..., 0_{|V|})$
  bsa$_{+1}$ $\leftarrow (0_0, ..., 0_{|V|})$
  jfq $\leftarrow$ sources
  while jfq $\neq \varnothing$ **do**
    bsa $\leftarrow$ bsa$_{+1}$
    Expand(V, E, bsa, bsa$_{+1}$, jfq)
    Identify(V, bsa, bsa$_{+1}$, jfq)

2.2 Graphics processing unit (GPU)

The graphics processing unit has become an integral part of today's computing systems (71). The modern GPU is more than just a graphics accelerator, as it is also used for general-purpose compute tasks (GPGPU) (66). Where CPUs are task-parallel latency-

(a) GPU SM architecture as described for the Tesla V100 (NVIDIA). Each SM contains four execution units, which share L1 Cache and texture units. Each execution unit has its own ALUs, scheduler, register file, instruction cache and dispatch unit.



(b) Schematic overview of GPU device architecture. A CPU has control over an execution queue from which tasks get executed on the GPU. Memory transfers to GPU local memory are performed from CPU main memory using DMA or PCIe transfers. The L2 cache is shared between streaming multiprocessors.

Figure 2.3: Two figures about GPU architecture, figure (a) shows the internal workings of a Tesla V100 GPU. Figure (b) shows a schematic overview of the GPU architecture, which shows how different components interact with each other.

oriented processors, GPUs are data parallel and throughput-oriented (61). Therefore, GPUs specialize in data- and compute-intensive tasks, such as image processing, computational physics, protein folding, graphics, machine learning, and more (58).

To achieve a speedup for these data-parallel tasks, the Single Instruction Multiple Threads (SIMT) execution model is used. For NVIDIA hardware, the control for multiple threads is placed in an Streaming Multiprocessor (SM), which is responsible for scheduling, dispatching, and caching. NVIDIA's hardware architecture is depicted in Figure 2.3. All GPUs have a certain number of cores that share the same control logic, which we refer to as a workgroup for AMD and WebGPU, or a warp for NVIDIA. These groups share a small amount of shared memory that is significantly faster than global memory, but it is only accessible by cores within a workgroup.

2.2.1 Programming model

Programmable GPUs were first introduced to give programmers control over the vertex and fragment stages of the fixed-function pipeline (57). These programmable features could be addressed by using graphics APIs such as OpenGL (52) or DirectX (60). As more functionality was exposed to programmers, the GPU began to be used for compute workloads.

OS	OpenGL	DirectX	Vulkan	Metal	OpenCL	CUDA	WebGPU
Windows	x	x	x	-	x	x	x
Linux	x	E	x	-	x	x	x
MacOS	-	-	E	x	-	-	x
Web	E	-	-	-	-	-	x

Table 2.1: Support table for the different Graphics/Compute APIs, where X shows support, E means emulated support

Due to the requirements of GPGPU tasks, the Compute Unified Device Architecture (CUDA) was launched in 2006 by NVIDIA (66). The goal of CUDA was to make it easier to use GPUs for general compute tasks by integrating with programming languages such as C, C++, Fortran, and Python. The major downside is that it only works on NVIDIA GPUs. CUDA is used extensively by both academia and industry (18, 49, 56, 95). Giving a biased view of the performance results, as most studies only have metrics for NVIDIA hardware (49, 56, 95).

OpenCL is the Open Computing Language, which specifies a C-like programming language for writing GPU compute kernels. The benefit of OpenCL is that it supports computation on all GPU vendors, including integrated GPUs, and even execution on a CPU. There are tools to convert CUDA to OpenCL, such as ZLUDA (88), chipStar (25), and Orochi (6). The downside of using OpenCL is that it is no longer supported on Apple systems since macOS 10.14 (11). The OS vendors all have their preferences for different Graphics APIs, as shown in Table 2.1. WebGPU is a new GPU programming model intended for the web, thus requiring support for all major operating systems and system configurations.

2.2.1.1 GPU programming paradigms

The programming paradigms between CPU and GPU differ the most in the utilization of threads and memory accesses. When multithreading on a CPU you have to avoid modifying data that is in the cache of other cores thus each core works on a seperate part of sequential data. Where a GPU cache is shared among multiple cores thus we want data access to be coalesced to best utilize the cache. For example, if four threads want to access eight elements, a GPU would let thread one access elements 1 and 5, thread two access elements 2 and 6, and so on. While a CPU would have thread one access elements 1 and 2, thread two accesses elements 3 and 4, and so on. The previous example is illustrated in Figure 2.4.

Each thread in a CPU is capable of branching individually. The control logic of GPU threads is much more limited, as this logic is shared within a workgroup, which can result in thread divergence when different threads execute different branches. When a specific branch is executed, all threads in a workgroup that should not execute that branch become inactive. These threads are reactivated when the control flow converges. Divergence results in low GPU utilization, as threads are stalled waiting for the other threads to converge again. An example of thread divergence is given in Figure 2.5.

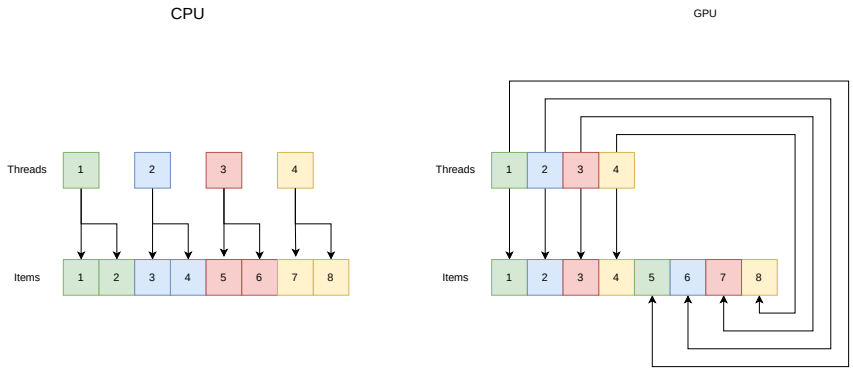


Figure 2.4: On the left, we illustrate the partitioned access for the CPU, while on the right, we demonstrate the coalesced access, which is optimal for the GPU.

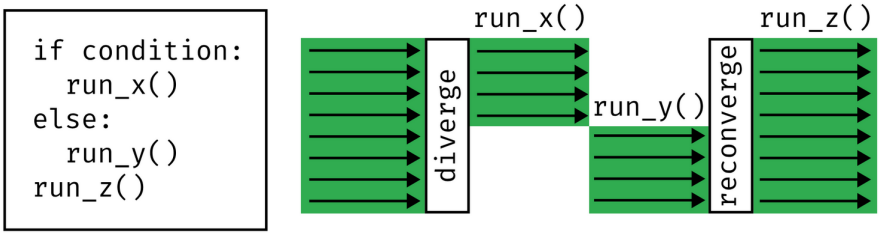


Figure 2.5: Shows the possible impact of thread divergence on active threads within workgroup

2.2.1.2 WebGPU

WebGPU (16) was designed efficiently map to post-2014 native GPU APIs, such as DirectX12 (60), Vulkan (53), and Metal (10). The goal of WebGPU is to bring the power of GPUs to the web for both rendering and compute. WebGPU is a specification of how the API should work, while the implementation is left to the browser vendors. Google has its implementation, used by Chrome, called Dawn. Firefox relies on the open-source Rust implementation called WGPU. Both implementations support the WebGPU specification through a C header, which is exposed by the browser through JavaScript. Due to Dawn and WGPU being developed as a standalone library, one can link their native application to either WGPU or Dawn as a native graphics/compute API without requiring a browser. WGPU or Dawn effectively translates the WebGPU calls to the graphics/compute API that is available natively on the platform.

```
@group(0)
@binding(0)
var<storage, read_write> to_reduce: array<u32>;

@group(1)
@binding(0)
var<uniform> stride: u32;
```

```
@compute
@workgroup_size(32)
fn main(@builtin(global_invocation_id) global_id: vec3<u32>) {
  var first : u32 = global_id.x * (stride * 2);
  var second : u32 = first + stride;
  to_reduce[first] = to_reduce[first] + to_reduce[second];
}
```

Source Code 2.1: Example of a WGPU reduce sum kernel

2.2.1.3 Compute shaders

Programming the GPU is done using shaders written in WGSL (Web GPU Shading Language). An example of such a shader is given in Source Code 2.1. This shader requires two resources to be provided, the `stride` and `to_reduce` array. Both resources are annotated with both the group and binding annotations. These annotations provide an index-based grouping, which is used by the driving C code to provide the correct resource. All resources in the same group must be provided at once; it is not possible to update individual bindings within a group. For the use case in this example, the `to_reduce` array is not changed between invocations of this shader, while the `stride` value is updated between invocations, hence it is in a different group. WebGPU optimizes for the order of changes to these bind groups. WebGPU expects groups with a higher ID to change more frequently than those with a lower ID.

Both these resources have qualifiers for their storage; the ones used in this example are `uniform`, `storage`, and `read_write`. These flags signal usage to the driver allowing for optimizations. The `uniform` qualifier is a faster read-only type of storage that has a lower maximum size and no write access. The `storage` qualifier has a high maximum size and can have write access. WebGPU validates the flags of a buffer before it is used. For example, a GPU buffer that is mappable on the CPU cannot be used by a shader.

The main function is annotated with two properties, `@compute` and `@workgroup_size`. The `@compute` is used to signal that it is an entry point for a compute pipeline. The `@workgroup_size` signals how many threads should execute this shader at a time. The example demonstrates the use of a shorthand for specifying the work group size. Workgroup size is specified using a 3D vector, so the example is interpreted as `@workgroup_size(32, 1, 1)`. The work group size signals to the SM how many threads should be executed in lockstep. A lower *workgroup size* provides more flexibility but sacrifices performance as not all workgroup sizes can be efficiently executed by an SM. A low number of threads can result in threads being underutilized, while too high a number of threads causes the shader execution to be split up across multiple SM's.

In the function signature, we see another annotation `@builtin`, the possible values for this annotation in compute shader the following:

- `local_invocation_id` is a vector containing the position of the executing thread within the current running workgroup.
- `local_invocation_index` is a linearized index of the invocation's position within the wokrgroup

- `workgroup_id` contains the current workgroups position in the compute shader grid.
- `num_workgroups` contains the size of the dispatch call from the driver side.

2.2.1.4 The C++ driver code

To use the compute shader in Source Code 2.1, we need some driver code that initializes the resources and instructs the GPU on what to execute. This driver code is shown in Source Code 2.2. The initialization of resources is not shown in this listing for simplicity.

Submitting commands to the GPU is done using command buffers. These buffers cannot be created by the user but are made using a command encoder. A command encoder can start a compute pass or execute copy operations. A compute pass contains zero or more invocations of a pipeline, where a pipeline describes the shader to be executed and the types of resources used. The actual invocation of the shader is done using the function `dispatchToWorkgroups`. This function takes three arguments: x, y, and z. The arguments denote the dimensions of the grid of workgroups that we want to dispatch. Recall that in our example in Source Code 2.1 we have defined our `workgroup_size` as 32, which is shorthand for (32, 1, 1). Thus, for 32 invocations of our shader, we call `dispatchToWorkGroup(1, 1, 1)` which will be run on one SM with 32 cores in lockstep (depending on the hardware). To call 64 invocations on two SMs, we call `dispatchToWorkGroup(2, 1, 1)`. Due to this behavior, developers need to consider how to partition the work to maximize the use of the hardware.

After the compute pass ends, we encode a copy command to copy the results from the writable storage buffer to the staging buffer. The staging buffer can be CPU-mapped, allowing the CPU to read the results of our compute shader.

```
wgpu::CommandEncoder encoder = device.createCommandEncoder({});
wgpu::ComputePassEncoder compute_pass = encoder.beginComputePass({});
compute_pass.setPipeline(pipeline);
compute_pass.setBindGroup(0, bind_groups[0], 0, nullptr);
compute_pass.setBindGroup(1, bind_groups[1], 0, nullptr);
compute_pass.dispatchWorkgroups(length / 2 / WORKGROUP_SIZE, 1, 1);
compute_pass.end();
compute_pass.release();
encoder.copyBufferToBuffer(storage_buffer, 0, staging_buffer, 0, sizeof(uint32_t) * length);
wgpu::CommandBuffer commands = encoder.finish();
queue.submit(commands);
```

Source Code 2.2: Simplified example of WebGPU reduce sum kernel driving code in C++

2.2.1.5 Features and limitations

Since WebGPU was designed to work on a multitude of devices with varying performance characteristics, not all features are available on every device. To allow the use of features that may not be present, one can request specific features when requesting an `Adapter`. These features range from support for timestamping on the GPU to reading compressed textures. A WebGPU context is aware of a set of limits, such as the maximum buffer size, the maximum number of bind groups, and the maximum dimension for compute workgroups. When the WebGPU adapter is requested, you can specify the limits required

for your application to function. If no adapter can satisfy the requirements, none will be returned.

WGSL has a set of supported types `i32`, `u32`, and `f32`. There is also support for `f16`, but this requires the `shader-f16` feature to be available. There are additional WGSL limits, such as the maximum combined byte size of all variables within a shader and the maximum number of members in a structure type. A complete overview of all these limits can be found in Section 2.4 of (89).

The usage of synchronization barriers requires a WGSL shader to be uniform in terms of control flow (meaning no divergence can occur up until the barrier). When a WGSL shader is transpiled, it is automatically checked for uniformity. The automatic check is overly strict to ensure uniformity between invocations in the same workgroup. As a developer, we can ensure that a load will always be uniform (meaning all threads in a workgroup receive the same value); to enforce this, the `workgroupUniformLoad` is used.

2.2.2 Data transfers

The initial cost of executing any workload on the GPU is copying the data from the host to the GPU (32). The cost of copying data means that before we can observe any speedup, the cost of computation must be higher than the cost of copying to the GPU. Not all workloads that are suited to GPU computation will see any speedup, as the computational complexity might be too low.

The limiting factor in copying data to the GPU is the PCIe link that connects the host system to the GPU. In the case of an integrated GPU, no PCIe link separates the CPU and the GPU, which means the driver does not have to transmit data over the PCIe link. When an integrated GPU is sufficiently fast, it can be slower to execute on a discrete graphics card due to these transfer costs.

Related Work

3.1 GPUs in database systems

GPUs offer parallelism and high-bandwidth memory access, making them an attractive option for accelerating data analytics in database systems. In this section, prior work using GPUs for database systems is discussed.

3.1.1 Accelerating GPU operators

GPU-accelerated database operators such as Selection (82), Joins (38, 50, 77, 81, 90, 93), and Sorts (35, 84) work by executing one or multiple GPU kernels to produce the desired result. An example of this is the work done by He et al. (38), which proposes using four relational algebra primitives, implemented as GPU kernels, to design join algorithms. Executing a sequence of such primitives incurs overhead due to the need to store intermediate results in memory. To address this, Wu et al. (92) developed a system that automatically fuses separate GPU kernels into a single operator. The fusing of these kernels reduces PCIe traffic, eliminates intermediate data, frees up GPU memory, and enables more effective compiler optimizations. The drawback of fusing kernels is that it increases register pressure, which can reduce the number of concurrently active threads. The system fuses operators until the required registers go above a certain threshold.

One of the first GPU query co-processing systems was proposed by He et al. (39). They used the algorithms from He et al. (38) to create a database system called GDB (GPU Database). Their system utilizes a cost estimation model to determine whether a query should be executed on the GPU, CPU, or both. Their cost model is based on three factors: transfer time to the GPU, GPU computation time, and transfer time to the CPU. The limiting factor in their system is GPU memory; when GPU memory requirements become too large, the workload is partitioned. They claim that partitioning this workload on the CPU is a significant performance cost.

The study by He et al. (39) got revisited in 2015 by Rui et al. (78), where it is shown that even greater performance gains can be obtained by utilizing new hardware features (such as shuffle instruction). This results in older GPUs not being able to run the modified algorithms due to a lack of features. They show that compute grew faster than bandwidth, 13 times vs 3 times, respectively.

3.1.2 Crystal

Crystal is a library of data processing primitives that can be composed to implement SQL queries on the GPU (79). Crystal claims to be faster than its predecessors by abandoning

the co-processor model and keeping the complete working set on the GPU. Crystal requires a developer to implement SQL queries using its primitives, which currently limits Crystal to just executing the Sample Star Schema Benchmark (SSB) queries (70).

Traditional GPU query execution executes multiple kernels, which produce and consume intermediary results. An example of this is a selection query that uses three kernels: the first to evaluate a predicate and count the number of matches, a second to perform a sum over all counts, and a third to produce the output vector. This results in multiple scans over global GPU memory. Crystal aims to avoid these scans by using a Tile-based execution model. A tile-based execution model expands upon vector-based processing on the CPU by assigning each thread a vector instead of a single value, creating a tile, as illustrated in Figure 3.1. Instead of data being stored in global memory, tiles are stored in shared memory, which provides higher bandwidth and enables synchronization between threads working on the same tile. After executing on such a tile, the results are written back to global memory.

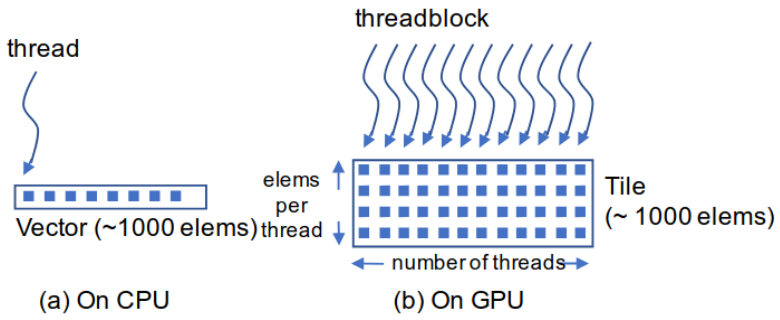


Figure 3.1: Illustration of tile based execution as shown in (79)

The primitives provided by Crystal either fetch or store from global memory or operate on a tile in shared memory. Coordination between blocks is done by combining Crystal's primitives with an atomic operation on some global structure. Instead of synchronization between threads, there is synchronization between each tile, which significantly reduces overhead.

Crystal demonstrated that running queries on the GPU, compared to the CPU, yields roughly 1.5 times the bandwidth ratio as a speedup. The cost ratio of renting a top-end GPU compared to a CPU is approximately six times as expensive, while the performance gap is about 25x. According to their evaluation, executing queries on a GPU could not only be faster but also more cost-efficient. Future work considered by Crystal includes multi-GPU execution, compression, handling strings, and non-scalar data types.

3.1.3 PG-Strom

PG-Strom is an extension for PostgreSQL which provides GPU-accelerated operators for `WHERE`, `JOIN`, `SORT`, and `GROUP BY` (45). Where applicable, these operators provide transparent speedups for PostgreSQL users. A hybrid CPU/GPU query plan is used to ensure that operators are executed on the fastest hardware available.

GPUDirect storage enables GPUs to access NVMe SSDs directly without CPU intervention, as shown in Figure 3.2. PG-Strom utilizes GPUDirect storage to implement "GPU Direct SQL" execution. During normal execution of analytical workloads, the `WHERE` clause in filtering, joins, or groupings limits the size of the output set. To filter the input set, a significant amount of irrelevant data needs to be transferred from disk to RAM, to the GPU, and back to RAM. GPU Direct SQL execution changes this flow to load data blocks directly to the GPU. Allowing the GPU to filter the input set and move the relevant data back to the CPU to execute more complex SQL operators (44). GPU Direct SQL is essentially using the GPU as a data preprocessor for CPU execution. Using GPU Direct storage is only supported on a limited set of systems due to the software and hardware requirements.

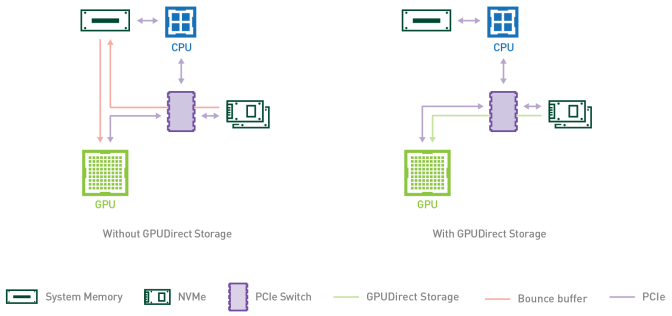


Figure 3.2: Illustration of GPU direct storage access as shown by NVIDIA (65)

3.1.4 HeavyDB (MapD, OmniSciDB)

MapD was introduced by Mostak (62) to perform data processing and visualization on the GPU, avoiding the need for data reformatting and data transfers in and out of GPU memory. MapD is a hybrid multi-CPU/GPU columnar relational database with a three-level memory model. The three-level memory model is arranged in a pyramid, where each successive level of memory is slower but has a larger capacity. The fastest and smallest level is GPU memory, followed by CPU memory, and the last level is data on disk. The unit of memory management in MapD is a chunk, which is a user-definable subsection of a column. The size of the chunk determines the rate at which a query can be partitioned across multiple devices; smaller chunks allow for better partitioning but result in higher overhead.

Mostak (62) claims that previous GPU database research has only seen modest speedups due to data being transferred mindlessly to and from GPU memory, resulting in disproportionate time spent on transfers over the PCIe bus. MapD aims to minimize this by only sending compressed bitmaps, index maps, or post-filtered results across the PCIe bus.

Later, MapD was rebranded to OmniSciDB (41), during which it gained support for

other architectures, as well as a JIT query compilation framework using LLVM. For CPU code generation, LLVM MCJIT is used; for GPU, NVIDIA PTX is generated instead. The PTX code is handed to the CUDA driver to produce an executable kernel. This code generation is combined with a cache for both the GPU and CPU to avoid regenerating the same code too often.

OmniSciDB was then rebranded to HeavyDB in 2022 (42). The rebrand into HeavyDB sparked the creation of new commercial products utilizing HeavyDB, such as HeavyImmerse, HeavyRender, HeavyIQ, HeavyConnect, and HeavyML. Where they claim to be 50x faster than Snowflake and being able to query 10 billion rows in less than 100ms (43)

3.1.5 BlazingSQL / RAPIDS

BlazingSQL (17) is a GPU-accelerated SQL engine built on top of the RAPIDS (75) ecosystem. RAPIDS is an open-source GPU-acceleration platform for large-scale data analytics and machine learning, introduced by NVIDIA (64). It consists of GPU-accelerated Python packages, including cuDF, cuML, and cuGraph. These packages help speed up existing data science tooling by either serving as a drop-in replacement or providing transparent integration. The GPU acceleration for these packages is implemented using CUDA and is compatible with NVIDIA GPUs on both Linux and Windows, utilizing WSL 2.0 (Windows Subsystem for Linux).

These data science libraries also offer GPU-accelerated UDFs and UDAFs (user-defined aggregate functions). The implementation of UDAF suffers from a large number of kernel launches. Yogatama et al. (94) propose switching to a block-oriented approach, as seen in Shanbhag et al. (79), which reduces the number of kernel invocations and lowers the amount of synchronization required.

3.1.6 TQP

TQP is a system developed by He et al. (40) which transforms SQL queries into tensor programs and executes them in a tensor computation runtime such as PyTorch. TQP can run the complete TPC-H benchmark on a wide range of hardware due to its use of portable and optimized tensor routines. They claim that TQP's performance is comparable or superior to that of specialized CPU and GPU query processing systems.

The workflow consists of two phases: compilation and execution. First, an input query is transformed into an executable tensor program. Then, during execution, the data is converted into tensors that are fed into the compiled program to generate the query result. The compilation step allows TQP to apply operator fusion to minimize data materialization across operators. TQP uses either interpreted PyTorch or compiled TorchScript to execute the tensor programs.

3.1.7 HetExchange

Query parallelization techniques used by analytical database engines are designed for homogeneous multicore servers. The state-of-the-art approach utilizes the *Exchange* framework, as proposed by Volcano (36), to enable parallelization. By using the Exchange operators injected into a query plan, it is possible to achieve horizontal, vertical, and

bushy parallelism. HetExchange, as proposed by Chrysogelos et al. (26) improves upon *Exchange* by encapsulating the heterogeneity of multi-CPU and GPU systems, providing a uniform interface to connect producers and consumers in a pipelined plan together with memory infrastructure. HetExchange extends the *Exchange* framework by adding control flow operators and data flow operators.

The control flow operators enable efficient parallelization, allowing for seamless movement of execution between the CPU and GPU, and vice versa. The control flow operators are the following:

- **Device crossing operators** enable pipelining across heterogenoeous hardware. Apart from these operators, all other operators are unaware of hardware heterogeneity and execute on a single device. HetExchange has two of these device crossing operators being: **gpu2cpu** and **cpu2gpu**. These operators are responsible for copying data between the CPU and GPU.

- The **router operator** encapsulates parallelism across multiple processors. Vertical parallelism is achieved by creating an asynchronous queue between a producer and a consumer. Horizontal parallelism is achieved by instantiating multiple instances of consumers and producers. Each router's parent and child are instantiated numerous times to achieve the required degree of parallelism for each device type.

Combining these operators provides all the necessary control flow manipulations required for all three forms of parallelism across multiple heterogeneous compute units. Device crossing operators are placed between heterogeneous producers and consumers to move execution across device types. Routers are placed at strategic points before device crossing operators to parallelize query plans.

To handle the availability of data in a heterogeneous system, data flow operators are used. By placing these operators after routers, it is ensured that data is in the correct form and available on the executing node. The data flow operators available are the following:

- The **mem-move operator** is responsible for ensuring that data is transferred and accessible before its client, the consumer, is executed. Encapsulating the logic to drive the transfers over the different interconnects, as well as to make decisions based on topology and the initial location of the data. When a transfer completes, it pushes the block to the consumer.

- The **pack and unpack operator** reduces the cost of moving data. It packs and unpacks tuples into blocks. Tuples can be packed to create blocks with specific properties. When a block is consumed by a GPU operator, properties such as coalesced access can be enforced.

Query execution on heterogeneous hardware has four fundamental traits: target device, degree of parallelism, data locality, and data packing. HetExchange gives a query optimizer the tools to influence all four. The device crossing operators change the target device, the router changes the degree of parallelism, the mem-move operator changes the data locality, and the pack/unpack operator changes the packing. HetExchange can be used for both interpreted and compiled engines. The work by Chrysogelos et al. (26) integrates with

Proteus (27) a compiled database engine. The generated physical plan is extended using the HetExchange operators to create a heterogeneous-aware plan. Based on this extended plan, code is generated to execute the query. These changes result in a 2 - 10x improvement over CPU and GPU-based alternatives.

HetExchange provides three key insights into the design of heterogeneous database systems. Due to the vast design space that a heterogeneous system provides, separating concerns helps manage complexity. HetExchange can be used in both vectorized execution models and compiled execution models. However, it lends itself more towards code generation infrastructure as it allows the system to have a single unified code base of pipelined operators. The compiler sometimes knows better; writing code to be executed on the GPU must account for thread block size, thread divergence, and atomic operations, among others. Implementing a code-generating engine that has to fine-tune all these device-specific numbers is a burden for a developer.

3.2 Breadth-First Search

3.2.1 Single Source

BFS is a fundamental building block in many other graph algorithms, commonly used to test for connectivity or compute the single-source shortest path in an unweighted graph. The algorithm operates by maintaining two key data structures: a visited set to track nodes that have already been explored, and a frontier queue that stores the nodes to be examined in the upcoming iteration. Initially, the frontier contains only the source node. In each subsequent iteration, the algorithm finds all unvisited neighbors of the current frontier nodes, marks them as visited, and appends them to the next frontier. This process continues until the frontier remains empty. BFS guarantees that each node is expanded exactly once, in order of increasing distance from the source node.

Due to a lack of locality, graph applications are often memory-bound on shared-memory systems or communication-bound on clusters (13). Due to the lack of computational complexity in BFS, these issues are exacerbated (13). Nevertheless, BFS has been extensively optimized (13) and adapted for various computational environments, with parallel (12, 54), distributed (21), and GPU-based (31, 59) implementations to enhance performance on large-scale graphs.

In standard BFS, the search begins from a single source node and explores its neighbors level by level, a process known as top-down. Beamer et al. (13) shows that there are cases where performing a bottom-up iteration is faster than the traditional top-down approach. A bottom-up iteration is performed by checking all unvisited nodes and determining if any of their incoming edges have been encountered. Performing a bottom-up iteration can reduce the number of edges that need to be checked when the frontier is larger than the unvisited set of nodes, depending on the graph's structure (13). Deciding when to switch between top-down and bottom-up is done using a heuristic. Beamer et al. (13) shows that using direction switching a speedup of 2.5x - 8x can be gained.

Parallel and distributed BFS implementations utilize multiple threads or machines to process the same graph. This requires coordination and synchronization between these workers. Work is often distributed using a partition scheme where each worker expands a subset of the graph. A naive way to partition this on a distributed system is a 1D

partition where each worker owns $n/|V|$ vertices as described by (21). Beamer et al. (14) use a simple 2D partition scheme on an adjacency matrix to partition a matrix A into $R \times C$ sub-matrices, which are assigned to a worker. Buluç and Madduri (22) shows that this does not scale up to the extreme scale of supercomputers, due to communication and synchronization overhead. To scale up to supercomputers, they propose using a Bitmap-Based Sparse Matrix representation instead of adjacency matrices or a CSR to reduce the memory footprint. This approach also reorders the vertex IDs for better load balancing. A block-cyclic distribution is used to reduce the required communication.

3.2.2 Multi-Source

Most efforts to improve the speed of a BFS were done using parallel methods (21, 54). Parallel methods focus on utilizing multiple threads or machines to accelerate a single BFS over a graph. Then (86) proposed to use Multi-Source Breadth-First search (MS-BFS) when an algorithm executes more than one BFS on the same graph. An example of such a problem is computing the closeness centrality for a graph. This algorithm executes a BFS for each vertex in the graph as shown in Algorithm 4.

Algorithm 4 Simplified closeness centrality for unweighted graphs with a single connected component without normalization

```
function ClosenessCentrality(Graph)
    for v ∈ Graph do
        for (x, distance) ∈ bfs(v) do
            Sum[v] += distance
```

A small-world network is a graph where the typical distance between two nodes grows proportionally to the logarithm of the network's node count. Many real-world networks are classified as small-world networks, such as social networks. MS-BFS accelerates BFS in the case of multiple searches by leveraging two key characteristics of small-world networks. First, the distance between any two vertices is typically very short, even in large graphs. Second, the number of vertices discovered in each iteration increases rapidly. As a result, most vertices are discovered in just a few iterations, making it likely that concurrent BFS processes will overlap in terms of the vertices they explore during the same iteration. MS-BFS capitalizes on this overlap by sharing the access cost of the edges for multiple BFSs. The cost of accessing these edges is thus amortized over all shared vertices in an iteration. Then (86) shows that for the SNB LDBC (8) graph with 1 million vertices, a significant amount of vertices are shared in the 3rd and 4th iteration, as seen in Figure 3.3.

The algorithm proposed by Then (86) performs multiple BFS traversals on the same graph at the same time to optimize performance for single-server, in-memory processing. MS-BFS employs bitsets of size ω to represent the state of concurrent BFSs. It uses three arrays `visit`, `seen`, and `visitNext`, each of size $|V|$, where each array entry is a bitset of size ω.

The algorithm begins by initializing the `seen` and `visit` arrays by setting the seen bit using its respective search index for all searches up to ω. Both `seen` and `visit` have their bitset set by the respective index of that source. The algorithm then scans the visit array for any active vertices, where non-zero entries represent vertices that are currently active

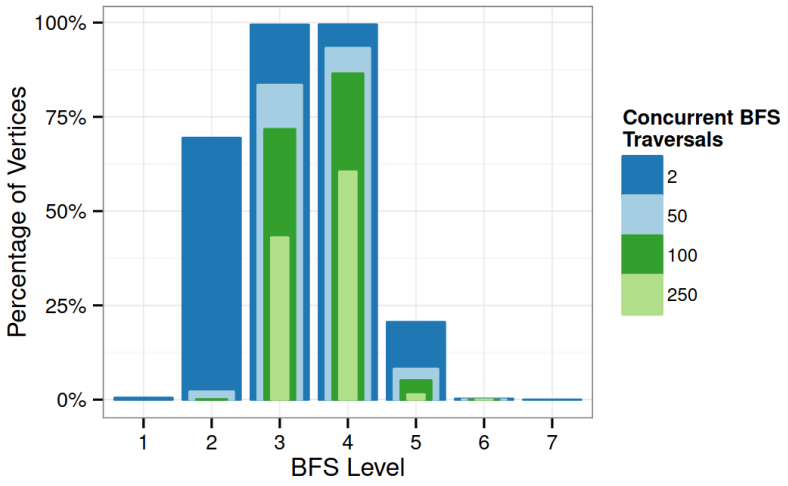


Figure 3.3: Percentage of vertex exploration that is shared per iteration across 512 concurrent BFSs (86)

in one or more searches. The outgoing edges of the active vertices should remain active in the next iteration; thus, we loop over the outgoing edges and check if this outgoing edge has not yet been seen in any active search in the source vertex. If this is the case, we mark the vertex in both the seen and the `visitNext` with the searches for which this outgoing edge should be active. Finally, we switch the `visit` and the `visitNext` arrays and clear the `visitNext`. These iterations continue until there are no remaining active bitsets in the `visit` array. A visual illustration of this algorithm is shown in Figure 3.4, and a code listing is provided in Algorithm 5.

Algorithm 5 MS-BFS using bit operations(86)

```
for source, index ∈ Sources do
    seen[source] ← 1 << index
    visit[source] ← 1 << index
while visit ≠ ∅ do
    for i = 1, ..., |V| do
        if visit[i] = ∅ then
            continue
        for n ∈ neighbors[i] do
            state ← visit[i] & (∼ seen[n])
            if state ≠ ∅ then
                visitNext[n] ← visitNext[n] | state
                seen[n] ← seen[n] | state
    visit ← visitNext
    reset visitNext
```

Cache usage can be further improved by utilizing a technique called aggregated neighbor processing (ANP). Instead of updating the seen state of outgoing edges in the inner loop,

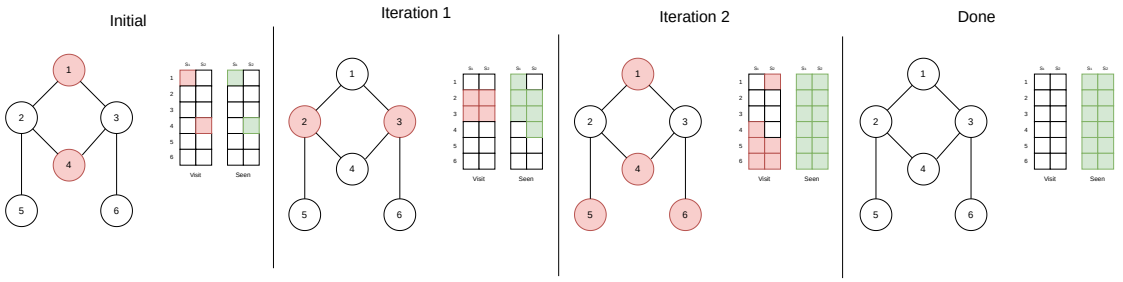


Figure 3.4: MS-BFS example where ω is 2, note that the expansion of node 2, 3 and 5, 6 are shared

only the visitNext array is updated. A second loop over `visitNext` is used to mask out any search that has already been seen by this search, updates the `seen` array and calls the BFS traversal callback. All accesses in the scan over `visitNext` are sequential, which improves the utilization of the low-level cache. This optimization improves the runtime of MS-BFS by 60-110% (86). Additionally, the use of direction switching in MS-BFS was tested, which provided a further improvement of up to 30%, significantly lower than the 2.5-8.5x claimed by Beamer et al. (13).

3.2.2.1 MS-PBFS

Since MS-BFS is limited to sequential execution, the only way to saturate a multicore system is to run one instance of the algorithm per core. Scaling this way requires a large number of source nodes to run efficiently. To overcome this limitation Kaufmann et al. (51) proposes an extension of the algorithm called Multi-Source Parallel Breadth-First search (MS-PBFS), which aims to speed up a single instance of MS-BFS using multiple cores.

The parallelization strategy behind MS-PBFS involves partitioning the vertices into a disjunct subset and processing them in parallel. The top-down traversal of MS-PBFS is based upon the ANP method from MS-BFS, which results in two separate loops. These two loops are divided into two phases for MS-PBFS, with a barrier separating them. Atomic operations are required in the first loop, as the neighbors of active vertices are used to set the next array, which results in random writes. The second phase does not require any atomic operations, as the writes are only to the vertex that is currently being processed, which means no synchronization between threads is needed. With the bottom-up version of MS-PBFS, there is only one phase, as writes are always directed to the vertex being processed, and reads from neighbors are simply random reads.

MS-PBFS divides the graph's vertices into partitions of at least 256 vertices. With partitions smaller than 256 vertices Kaufmann et al. (51) claim they encounter scheduling overhead. Each worker has its own queue of partitions, assigned using a round-robin scheduler, which means there is at most a one-task difference between workers. When a worker finishes, they can steal work from other workers. When all the queues are exhausted, the main thread is signaled, and the next phase or iteration can start.

The partitions are distributed based on the number of outgoing edges in a round-robin fashion. The highest out-degree vertex is placed at the start of the first task for worker one. The second-highest out-degree vertex is at the beginning of worker two, and so on.

With this approach, each worker has approximately the same amount of work. Because the highest-degree vertices are assigned first, the most expansive tasks will be executed first, resulting in less wait time when no more work stealing is possible.

3.3 BFS on the GPU

3.3.1 IBFS

Iterative Breadth-First Search (IBFS) (56) is a variant of MS-BFS, specifically designed for GPUs. IBFS uses three unique techniques: joint traversal, GroupBy, and bitwise-optimizations. The data structures used by IBFS are shared among all ω BFS instances. The Joint Status Arrays (JSA) store the lowest iteration number at which a vertex was encountered for **any** given search. The Joint Frontier Queue (JFQ) contains the frontiers of all concurrent BFS instances.

IBFS consists of two distinct steps: identify and expand. In the identify step, the JFQ is generated. A scan over all vertices is used to check if there is a difference between the current JSA and the previous JSA; if so, that vertex is added to the JFQ. In the expand step, each outgoing edge of all entries in the JFQ is traversed, and the lowest iteration number for that edge is stored in the JSA for this iteration.

Ideally, the best performance for IBFS would be achieved by running all instances of BFS concurrently without requiring GroupBy rules. However, GPU resources limit the number of concurrent BFS instances that can be run in a single instance of IBFS. To optimize sharing IBFS groups, source vertices with similar outdegrees, which improves the sharing ratio up to 10 times.

To further optimize, the JSA is replaced with a Bitwise Status Array (BSA) that contains a bitset of size ω. The bitset is used to store the seen state of a vertex given an index. The bitset enables the algorithm to use a binary `OR` operation instead of a minimum to set the status arrays. The speed up from this approach ranges from 11x to 36x as claimed by Liu et al. (56).

3.3.2 Fused Probabilistic Breadth-First Search

Probabilistic Breadth-First traversals (BPT) are used in network science and graph machine learning applications. Instead of adding all unseen neighbors to the frontier, BPT only adds the neighbor with a given probability. Neff et al. (63) shows that the significant sampling complexity for these traversals makes it hard to parallelize efficiently. They present a new algorithm to fuse BPTs by combining separate traversals executing on distributed multi-GPU systems. The fusing of different BPT traversals is based upon the work of Then (86) and Liu et al. (56). Due to the probabilistic nature of a BPT, no direction-switching or early-abort techniques can be used to speed up traversal, as this would compromise correctness.

When working with heterogeneous CPU-GPU systems, Neff et al. (63) found that performance was lacking compared to a GPU-only setup. CPU workers caused workload starvation at the end of an iteration. Their CPU implementation could be up to 16 times slower than the GPU implementation. To solve this, they execute microbenchmarks to dynamically adjust the partition size for the CPU workers based on their results. With

large graphs, this could leave the CPU with no work. To solve this, they group CPU workers in clusters that share L3 cache regions to collaborate on one BPT group. To further speed up the execution, they apply vertex reordering to ensure the vertices with a high out-degree are spread evenly among workers.

3.3.3 Mix and match A model driven runtime optimization for BFS on GPU

Deciding which BFS variant to execute can heavily impact performance. Picking the BFS variant that performs best on a specific graph remains challenging to predict. Verstraaten et al. (87) proposes using a machine learning model to select from 5 GPU BFS implementations dynamically. Their five implementations are Edge List, Reverse Edge List, Vertex Pull, Vertex Push, and Vertex Push Warp.

The Edge list methods launch one CUDA thread per edge to check if the depth of the origin vertex equals the current BFS level. The level of the destination vertex is then updated to the minimum of its current depth and the BFS level plus one. Vertex Push & Vertex Pull use a vertex-centric parallelization scheme; for each vertex, one CUDA thread is launched. The push method is comparable to the top-down approach (13), and the pull method is similar to the bottom-up approach (13). For the Vertex Push Warp method, the concept of virtual warps (46) is employed to mitigate the workload imbalance between threads.

Verstraaten et al. (87) claim that instead of relying on heuristics to pick a variation, as done in (13, 56, 86), instead the choice of algorithm should be left to a predictive model. The features used by their predictive model include: Graph size, Frontier size, Discovered vertex count, and degree distribution. Using this model, they claim that a 40% performance improvement can be obtained compared to the fastest non-switching implementation. The speed-up could be even greater in practical applications, as it is not guaranteed that the optimal non-switching implementation is used.

3.4 WebGPU usage in science

3.4.1 Security

WebGPU provides more access to GPU resources compared to the previous standard for GPU acceleration on the web (WebGL)(30). Providing more access results in a larger attack surface for malicious software, allowing untrusted web content to be passed to the GPU driver stack, which is optimized for performance rather than security. WebGPU cannot be run in a tightly sandboxed process, worsening the problem. The work by Ferguson et al. (30) showcases this using a GPU cache attack to fingerprint devices. Fingerprinting on the web is used to identify a user uniquely. The key components for identifying a user include device attributes, cookies, browser, and plugins.

The attack vector used by Ferguson et al. (30) uses the GPU's internal L3 cache. Intel's integrated GPUs have three levels of cache, as well as an LLC shared with the CPU. The internal L3 cache is shared across all sub-slices within the slice and supports caching all memory accesses. Their method, Compute Spy, allows monitoring of the L3 cache from a compute context. By executing a compute shader and atomically incrementing a counter in

one set of workgroups while another workgroup with one thread reads the counter. Using these counter values, they were able to identify which integrated GPU the website is visited on. To mitigate this, they propose partitioning the L3 cache between security domains for different processes.

The work by Bernhard et al. (15) raises attention for attack vectors in the processing of WebGPU shaders. They built a fuzzer for WGSL shaders, which has been used to find 39 bugs in the two WGSL transpilers Tint (Dawn) and Naga (WGPU).

3.4.2 Portability

WebGPU enables users to target GPU capabilities in a portable manner, making it an attractive target for development. The work by Paarmann (72) illustrates this by implementing a WebGPU backend for the Futhark language. Futhark is a functional array programming language with the goal of compiling to efficient parallel code.

Directly translating these Futhark kernels to WebGPU is not as straightforward due to WebGPU's limitations. WGSL explicitly deviates from the typical IEEE-754 (47) floating-point standard, as it does not require support for exceptions. Any expression resulting in a floating-point exceptionsuch as, infinity or NaN is allowed to produce incorrect results. Implementations of WGSL may assume that no exceptions are present at runtime. Any expressions that are evaluated before runtime, such as constant expressions resulting in any of floating-point exceptions, should throw an error during shader creation.

They also note the following limitations to support Futharks concepts in WebGPU.

- Primitive types
  - f64, i64, u64 types
  - Built-in arithmetic and conversion functions
  - Full support floating point exceptions
- In-kernel error handling
- Atomic for types other than 32-bit integers
- Uniform control flow analysis violations
- Memory fences

The benchmarks run by Paarmann (72) show that CUDA is 46 times faster than their WebGPU implementation for a mapping operation. They do state that their backend is not production-ready. Futhark programs that contain fences are often correct but are not deterministic. Some correct Futhark programs produce internal compiler issues, while others create a WGSL shader that does not compile.

Han et al. (37) demonstrates that by dynamically mapping WebGL to WebGPU, a significant performance increase can be achieved. They create an intermediate JavaScript layer to perform the mapping between WebGL and WebGPU. Their motivation is to avoid manually migrating old WebGL code to WebGPU, as this is a labor-intensive task. With this approach, they see a decrease in frame times compared to running the code in WebGL. Their approach can accelerate existing WebGL programs up to five times.

The work by Chen et al. (24) brings LLM inference to web clients using WebGPU. They propose two innovations: buffer reuse strategies that reduce the overhead associated with resource preparation and an asynchronous pipeline that decouples resource preparation from GPU execution, enabling parallelized computation with deferred result fetching. They implement these optimizations on top of WebLLM (4). With these optimizations, they see up to a 3x performance increase. Additionally, they use up to 20% less GPU memory.

Design & Implementation

The goal of this project is to design and implement a GPU-based path-finding algorithm using WebGPU. We utilize WebGPU to support a wide range of graphics hardware, from integrated GPUs to discrete GPUs. We implement our algorithms in both CUDA and WebGPU, allowing us to compare WebGPU's performance with CUDA. We expect that a GPU implementation outperforms the current DuckPGQ implementation. This chapter details how we built a library that performs path-finding operations using CUDA, WGPU, and Dawn. First, we will discuss why we do not implement the CPU-optimized MS-BFS on the GPU. We then discuss some of the parallelization schemes we have tried, and finally, we discuss the implementation differences between WebGPU and CUDA.

4.1 Moving away from MS-BFS

The current implementation used by DuckPGQ is based on MS-BFS (86). MS-BFS enables DuckPGQ to execute searches from multiple sources within a single traversal of the graph. Porting MS-BFS to the GPU using a thread-per-vertex parallelization scheme would underperform due to thread divergence. The divergence occurs within the `if` branch of the `for all` vertices loop, causing the entire workgroup to stall until the threads currently expanding are finished. This problem is further exacerbated by the cost of iterating over all neighbors, as each thread works on a different set of neighbors. As a result, all threads will wait for the longest-running loop to complete. An illustration of this top-level thread divergence in MS-BFS is shown in Figure 4.1. A GPU would not perform to its best ability when executing MS-BFS due to this thread divergence, so instead, we shift our focus and optimization efforts towards IBFS.

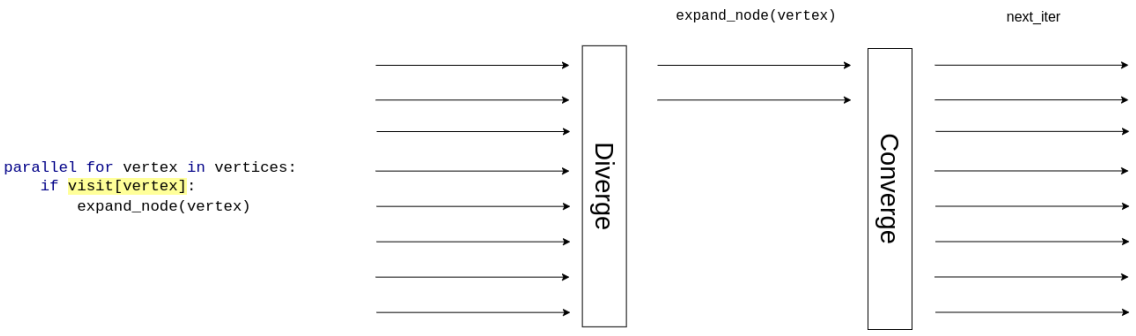


Figure 4.1: Showcase of thread divergence within MS-BFS

Recall that IBFS is a specialized version of MS-BFS (56) optimized for the GPU, which

executes ω BFSs concurrently. IBFS (Algorithm 3) splits MS-BFS into two steps `identify` and `expand`. In the `identify` step, the frontier is identified and appended to the JFQ. The cost of divergence is minimized as the work to append a vertex to the queue is significantly lower than traversing all outgoing edges. In the `expand` step, we loop over the queue to expand each frontier node. This expansion is done by propagating the seen state of the frontier node to all outgoing edges. The `expand` step still suffers from thread divergence as not all vertices have the same number of outgoing edges.

IBFS with bitwise operations requires two data structures, the `BSA` (Bitwise Status Array) and the `JFQ` (Joint Frontier Queue). We need to store two instances of the `BSA` array, one for the previous iteration and one for the current iteration. The size of the `BSA` array is determined by $|V|$ and the size of the bitset ω. The length of the JFQ array is bound by the length of the V array, as at most all vertices can be in the frontier.

4.2 Implementing IBFS

We implement our IBFS version as a library to integrate our efforts with correctness testing, benchmarking, and possibly DuckPGQ. Our implementation is limited to unweighted path length. The signature of our library is depicted in Source Code 4.1. We provide both a WebGPU implementation and a reference CUDA implementation for this library. Selecting a backend can be done at compile time using the cmake flag `-DBACKEND=CUDA|WGPU|DAWN`. A web version can also be compiled using `-DBACKEND=EMDAWNWEBGPU`, although this requires additional JavaScript glue code to be used.

```
struct IterativeLengthResult {
  uint32_t src;
  uint32_t dst;
  uint32_t length;
};
struct PathFindingRequest {
  uint32_t *src;
  uint32_t *dst;
  uint64_t length;
};
struct CSR {
  uint32_t *v;
  uint32_t *e;
  uint64_t v_length;
  uint64_t e_length;
};
std::vector<IterativeLengthResult> iterative_length(PathFindingRequest request, CSR csr);
```

Source Code 4.1: Interface implemented by our library

Before a kernel can be executed on the GPU, its resources must be allocated. We make a distinction between resources that require data from the CPU and GPU local resources. The GPU local resources are the `JFQ`, `BSA`, `BSAK`, and `path_lengths`; all of these are allocated as zero-initialized memory. The `CSR`, `sources`, and `destinations` need to be transferred from CPU to GPU memory over the PCIe link. The data that needs to be pushed to the GPU is thus limited to the total size of `CSR`, `sources`, and `destinations`. The only data that needs to be transferred back to the CPU is the `path_lengths` array.

Our IBFS implementation uses three GPU kernels to execute the algorithm. Both `identify` and `expand` are implemented in a separate kernel, and the initial state of the algorithm is set using a third kernel. The implementation of these kernels varies depending on which parallelization scheme is used, which we discuss in Section 4.3. We provide a pseudocode version of all three kernels in the following sections. These pseudocode versions need to be modified to fit a parallelization scheme.

4.2.1 Set first BSAK

The initial step of the algorithm is to set the `seen` state of all ω sources in BSAK. A kernel is used to set the initial state, thereby avoiding multiple small transfers from the CPU to the GPU. Pseudocode for this kernel is given in Algorithm 6, where `threadIndex` depends on the parallelization scheme used. The atomic OR is used because there is no requirement for sources to be unique. When two non-unique sources are set at the same time, an update could get lost without the atomic operation.

Algorithm 6 Pseudocode of set first bsak kernel

```
function SetFirstBSAK(sources, bsak)
    atomicOr(&bsak[sources[threadIndex]], 1 << threadIndex)
```

4.2.2 Identify

The original version of IBFS only traverses the graph in a breadth-first manner. As we compute the path length for multiple source-destination pairs, we need to check if a destination has been reached. In the identify step, each node is appended to the search exactly once for each search. We use this observation to extend the identify step to write out the path length when the destination is reached. After the destination of a search is reached, the search should become inactive. To accomplish this, we use a mask to determine which searches are still active. By masking out the bits in the difference found in the identify step, we avoid further expanding inactive searches.

The inner loop shown on line 8 in Algorithm 7 uses bitwise operations to gain information about which search is active. Counting the one bits tells us how many searches have first discovered this vertex. We can then check for each search if it is the destination for that search. On line 9, we count the trailing zeros in the current `diff`. Counting the number of trailing zeros gives us the index of the search. We then turn off that bit in the `diff` and use the index to check if this vertex was the destination for that search. If this node is the destination, we modify our mask and write out the iteration number as the path length.

These bitwise operations are available in WebGPU through WGSL functions of the same name. CUDA has the `__popc` compiler intrinsic, which returns the number of one bits in a variable. Finding the trailing zeros in CUDA can be done using the `__clz` intrinsic, which returns the number of leading zeros. By subtracting 31 from the result of `__clz`, we have the index for the search.

Algorithm 7 Pseudocode of identify kernel

```
1: function IDENTIFY(jfq, destinations, bsak, bsa, mask, lengths, iteration)
2:     for each vertex ∈ V in parallel do
3:         diff ← bsak[vertex] ⊕ bsa[vertex]
4:         if diff == 0 then continue
5:         bsak[vertex] ← bsak[vertex] | bsa[vertex]
6:         jfq.enqueue(vertex)
7:         activeSearches ← countOneBits(diff)
8:         for x ∈ activeSearches do
9:             index ← countTrailingZeros(diff)
10:            diff ← diff ⊕ (1 << index)
11:            if destinations[index] != vertex then continue
12:            lengths[index] ← iteration
13:            mask ← mask ⊕ (1 << index)
```

4.2.3 Expand

We leave the expand step unmodified as compared to the IBFS implementation (56). Do note that we use a CSR to represent our graph, so our accesses over E are sequential. To find the outgoing edges of a given vertex, we do two lookups in the V array, one for $V[v]$ and one for $V[v+1]$. These lookups give us the offset into the E array for the starting edge of v and the last edge of v. The pseudocode for the expand kernel is shown in Algorithm 8.

Algorithm 8 Pseudocode of expand kernel

```
function EXPAND(jfq, V, E, bsak, bsa)
    for each vertex ∈ jfq in parallel do
        first ← V[vertex]
        last ← V[vertex + 1]
        val = bsa[vertex]
        for id ∈ (first...last) do
            atomicOr(bsak[E[id]], val)
```

4.2.4 Combining all steps

The driver program is responsible for calling the kernels, managing GPU resources, and fetching the output from GPU memory. In our library, this driver program is the implementation of our `iterative_length` function. The number of source-destination pairs passed to the driver program can exceed the number of concurrent searches supported by our implementation of IBFS. Due to the limited number of concurrent searches in IBFS, we process the source-destination pairs in batches. By batching the execution of the algorithm, we avoid having to delete and recreate GPU resources. After each iteration, we swap BSA and BSAK to avoid having to copy or set either buffer. A pseudocode implementation is given for this driver program in Algorithm 9

When implementing these kernels, some state needs to be preserved between invocations.

Algorithm 9 Pseudocode of driver program

```
function IBFS(V, E, sources, destinations)
    (jfq, bsa, bsak, pathLengths) ← allocateGPULocal()
    (v, e, s, d) ← allocateGPUCopied(V, E, sources, destinations)
    for i ∈ 0...(ω/|sources|) do
        iteration ← 0
        offset ← i * ω
        SetFirstBSAK(s + i * ω, bsak)
        while readFromGPUMemory(|jfq|) > 0 do
            if iteration % 2 == 0 then
                identify(jfq, d + offset, bsak, bsa,  0, pathLengths + offset, iteration)
                expand(jfq, v, e, bsak, bsa)
            else
                identify(jfq, d + offset, bsa, bsak,  0, pathLengths + offset, iteration)
                expand(jfq, v, e, bsa, bsak)
            iteration ← iteration + 1
    return readFromGPUMemory(pathLengths)
```

The state that we wish to store is `mask`, `iteration`, and the `JFQ length`. We group these variables into a structure we call `SearchInfo`. The GPU kernels can modify this state, eliminating the need for copies from the CPU to the GPU. The enqueue operation of the JFQ uses an `atomicAdd` on the length stored in state. Since `atomicAdd` returns the previous value, we can directly use the returned value as the index for the item to enqueue. Such an atomic operation is found to be faster than other alternatives, as shown by Gaihre et al. (33).

Getting the length of the JFQ from GPU memory forces the GPU and the CPU to synchronize and transfer data. Such a synchronization is a costly operation. To avoid synchronizing after each invocation, we want to be able to tune this parameter. So we execute N iterations before checking the JFQ length. A high value for N results in running empty iterations, while a low value for N results in more synchronization. During development we have seen that changing the value from one to two gave an improvement, any higher did not result in any gains for native execution. The overhead of synchronizing differs between CUDA, WGPU, and Dawn. These costs are even higher when executing on the web, where waiting for results has significant overhead.

4.3 Parallezation schemes

With the driver program and kernels designed, we now have the logic necessary to execute the IBFS algorithm. However, before launching any kernels, we must first determine an appropriate parallelization scheme.

Recall that the kernels we write are executed by a group of threads called a workgroup. The driver program can dispatch multiple of these workgroups depending on problem size and hardware capabilities. A parallelization scheme determines how to utilize the spawned threads. In compute kernels, we often see the usage of a parallel strided for loop to loop over

large arrays. In such a for loop, each thread starts at an offset determined by a linearized thread ID, and the offset is incremented by the total number of threads each iteration, which resembles the following `for (x = threadId; x < lim; x += totalThreads)`. The additional benefit of this template is that it gives coalesced access to an array that needs to be scanned.

For example, we could use the parallel strided for loop scheme to accelerate the parallel for in the expand algorithm. We start by determining the size of the workgroup. The optimal size of a workgroup depends on the GPU used. The default is using a workgroup of size 32. This default is set to 32 as NVIDIA GPUs (69) execute code in blocks of 32, while AMD executes code in blocks of 64 or 32 (7). As workgroup sizes are expressed using 3D vectors, we set our workgroup size to $(32, 1, 1)$. Spawning a total of 320 threads is done by dispatching $(10, 1, 1)$ workgroups.

The NVIDIA Nsight Compute profiler provides a list of issues for each kernels execution. The possible issues identified by the profiler are as follows.

- **Small grid:** A low amount of work was scheduled, so most of the GPU will remain underutilized. This can be solved by executing the kernel with more workgroups.
- **Achieved occupancy:** The difference between the calculated theoretical and measured achieved occupancy. This can be the result of workgroup scheduling or workload imbalance.
- **Theoretical occupancy:** The number of theoretical workgroups per scheduler is limited due to the number of workgroups executed or due to shared memory requirements.
- **Long scoreboard stalls:** a workgroup is stalled because it is waiting for a result to arrive (in our thesis, this is often due to a data dependency).
- **SMSP workload imbalance:** one or more SM has a much higher number of active cycles than the average number of active cycles.
- **Uncoalesced global accesses:** A kernel has uncoalesced global access to global memory.
- **L1Tex Global load access pattern:** The memory access pattern for global loads is non-optimal. This hints at a portion of the cache line being unused, typically due to reading a small number of values from global memory.

4.3.1 Naive solutions

Throughout this thesis, we have implemented multiple naive approaches in an attempt to accelerate performance. These naive solutions failed to meet performance expectations. In this section, we will discuss some of these naive solutions and explain why they failed to work, as well as how they led us to the three approaches that perform the best.

Parallelize over the JFQ: Our first attempt executed 32 searches in parallel. One workgroup was used for the identify step. For the expand step, we dispatched either the maximum supported number of workgroups or one thread for each vertex in the JFQ. This approach forces us to read the length of the JFQ each iteration. The iteration number was also written to the GPU each iteration. All this synchronization hindered performance. We also tried this approach with 128 searches instead of 32, which performed worse. We attempted to submit work to the GPU from multiple threads to avoid GPU starvation; however, this did not improve performance.

Uber shader: To reduce the synchronization costs associated with the **parallelize over the JFQ** approach, we implemented the entire algorithm in a single kernel. A single kernel saw significant improvement over the **parallelize over the JFQ approach**. This approach utilized one workgroup per IBFS instance. There is no synchronization across workgroups due to the limited synchronization primitives available in WebGPU. Resulting in us scaling up the number of IBFS instances. Before we are able to read back any results we have to wait until all instances finish executing. These execution times become so long that the GPU driver crashes on an integrated GPU. The long-running kernels triggered a device reset timer (DRT), causing the running kernel to crash. The driver uses this DRT mechanism to recover a GPU that has stopped responding.

Scale out: With this approach, we move to the pseudocode we have shown in Section 4.2. Parallelizing over the instances that we execute, thus one workgroup executes one instance of IBFS. With this approach, we ran 64 instances of IBFS concurrently. The Scale out saw significant improvements over previous attempts. However, it shows low GPU utilization and occupancy up to 25-30%.

4.3.2 Workgroups

In our workgroups[1] approach we continue with our findings from scale out. The next step we took was to reduce the number of concurrent instances of IBFS from 64 to either 1, 2, or 4, depending on a runtime environment variable. Dispatching just a few workgroups would leave the GPU underutilized. Thus, we aim to use multiple workgroups for each instance we run. To accomplish this, we utilize the three-dimensional nature of the dispatch call. Recall that when dispatching with size $(2, 2, 1)$ we spawn four workgroups with ids $\{(0, 0, 0), (1, 0, 0), (0, 1, 0), (1, 1, 0)\}$. So by dispatching with $(\text{Instances}, \text{Workers}, 1)$ we spawn workgroups with invocation ids $(0...\text{Instances}, 0...\text{Workers}, 1)$.

As we now have multiple workers per instance, we need to differentiate in the kernel which `SearchInfo`, `BSA`, `BSAK`, and `JFQ` to modify. As the `invocation_id` is determined by the dispatch call, the x component will be within range 0...Instances. To determine the instance to work on, we use `invocation_id.x`. This is used to index into the array of `SearchInfo` objects and to calculate offsets into the `BSA`, `BSAK`, and `JFQ`.

A parallel strided for loop is utilized to use multiple workgroups for each instance. We have *Workers* number of workgroups that share an instance. To linearize the thread ID, we utilize the built-in compute grid variables. The linearized thread ID is calculated using

[1] https://github.com/ThijsDreef/wgpu-msbfs/tree/workgroups

`local_id.x` + the `invocation_id.y` * `workgroup_size.x`. An example of this loop is given in Source Code 4.2 using WGSL.

```
for (
  var i : u32 = local_id.x + invocation_id.y * workgroup_size.x;
  i < length;
  i += workgroup_size.x * invocation_size.y
) {
```

Source Code 4.2: The parallelization scheme for the expand and identify step of the workgroups approach. Each thread handles the identification and expansion of one vertex.

To determine the workgroup size, we follow the default as discussed previously. So the x component of our workgroup size is set to 32. However, we found that by utilizing the y component of our workgroup, we can achieve a performance improvement in the expand step. These y number of extra threads are used in a parallel strided for looping over the ougoing edges. In Figure 4.2, we show the execution times for running the workgroups approach with one workgroup using varying y components for the workgroup sizes ranging 1 - 32 using powers of two. We conclude that the optimal size of the y component is eight.

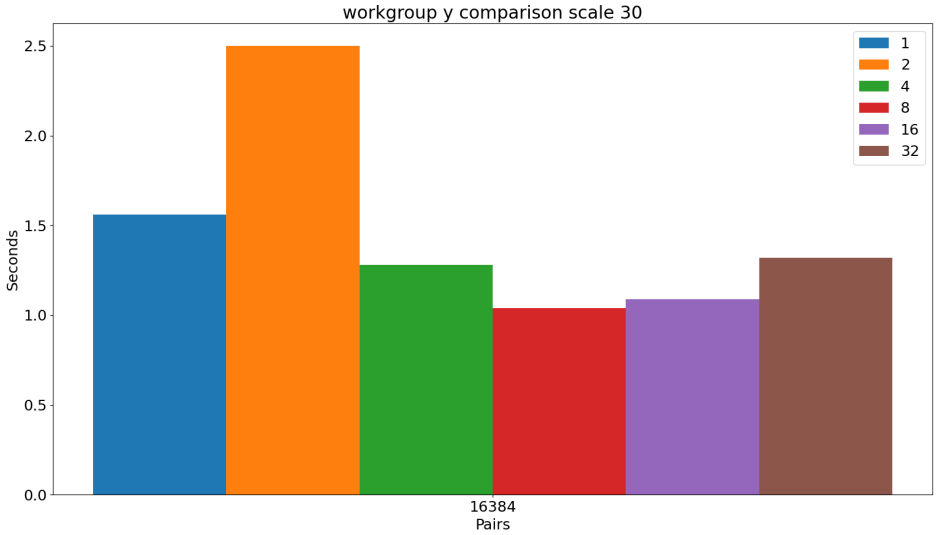


Figure 4.2: shows the execution time of the workgroups method using one workgroup with a varying workgroup size y component ranging from 1 - 32 using powers of two.

To keep the number of bindings that we use as low as possible, we store each instance's arrays in the same binding. The `JFQ` and `BSA` are laid out consecutively in memory for each IBFS instance. To find the start of either the JFQ or BSA, we compute the starting offset using $|V|$ multiplied by `invocation_id.x`. This data layout is depicted in Figure 4.3.

Bottlenecks:

Set first BSAK:

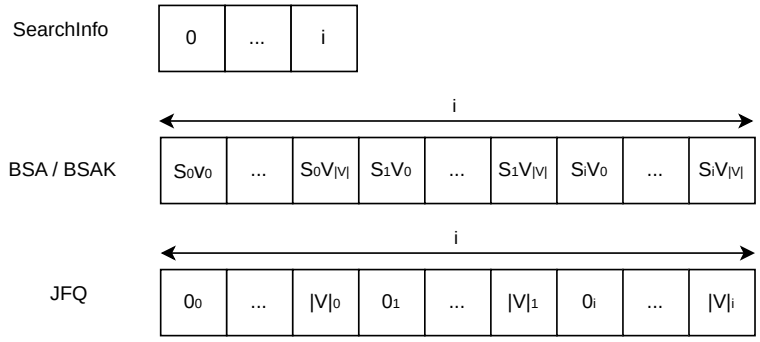


Figure 4.3: Depicts the data-layout of the workgroups approach where i is the number of instances. We have i number of search info objects, BSA and BSAK are laid out consecutively by the number of instances and there are i JFQ instances also laid out consecutively.

- **Small grid:** Due to the nature of the workload, we can only dispatch a small grid of work. For this kernel, we spawn 32 threads to do 32 atomic operations. There is no more work to schedule, thus we have a small grid issue.
- **Low achieved occupancy:** Low occupancy ties into the small grid issue, as we cannot utilize the whole GPU; therefore, we will see low occupancy.
- **Theoretical occupancy:** Due to the limited size of the workgroup the theoretical occupancy is limited as we can only fill half of a SM with the workload.

Identify step:

- **Achieved occupancy:** We achieve roughly 48% occupancy. Considering the branching required for the identity step, this is deemed adequate. As the identify step is a fraction of the total runtime (0.01 - 1 ms), compared to the expand step taking (1.2 - 16.8 ms).
- **Long scoreboard stalls:** According to the profiler, we stall due to the load operations on BSA and BSAK. Access to BSA/BSAK is required and cannot be expedited (apart from fetching earlier).
- **L1Tex Global load access pattern:** Access to the destination array only checks a few items and loads the whole global array into cache. As we do not access the entire array each time, the access pattern is flagged as a potential performance improvement.

Expand step:

- **Uncoalesced global access:** The expand step has uncoalesced access into the BSA and BSAK buffers, as these are consecutively laid out in memory. Access to the edge array to find the outgoing edges is also uncoalesced.
- **L1Tex global load access pattern:** Due to the nature of our CSR graph representation, we will not use all the bytes that are pulled into the cache. This issue is even more prevalent as the uncoalesced access into `BSA` and `BSAK` contributes to this metric.

- **Long scoreboard stalls:** Accessing items in `BSA`/`BSAK` and accessing the CSR are the cause of these long scoreboard stalls. The workgroup stalls are caused by waiting for data to arrive from the accesses to `BSAK`, `BSA`, `V`, and `E`

4.3.3 Coalesced access

The main issue with the workgroup approach is that we lack coalesced memory access and experience thread divergence during the expand step. To address these issues, we propose another parallelization scheme, which we call coalesced access[1]

Our coalesced access scheme avoids thread divergence and coalesced access by letting each thread on the x axis within a workgroup work on a different instance. These x instances of the algorithm share the same `JFQ`. To build this `JFQ`, we have to communicate between the threads in a workgroup during the identify step. If any thread encounters a difference between `BSA` and `BSAK`, then that vertex should be added to the `JFQ`. To do this efficiently, we use a "vote" instruction. in CUDA these take the form of `__ballot_sync(mask, condition)`. This intrinsic returns a mask indicating for which threads the condition is true. Using this mask, we can determine if any thread has found a difference between `BSA` and `BSAK`. The requirements for these vote instructions are that they execute within the same subgroup unit. Effectively limiting our workgroup size to **32**, which gives us the following workgroup size for the identify step $(32, 1, 1)$.

Due to the workgroup size being limited to 32, we are required to execute 1024 searches concurrently. Each thread executes one instance, which contains 32 searches, resulting in 32 * 32 = 1024 searches. Increasing the number of searches would necessitate the use of a different synchronization primitive, which would be less efficient. Lowering the number of searches would result in underutilized threads in the subgroup unit.

To make access to `BSA` and `BSAK` coalesced, these arrays need to be restructured. We change the data layout to a format where the seen state is grouped per instance. The first 32 entries store the seen state for vertex 1, the second set of 32 entries stores the state for vertex 2, and so on. As each thread on the x axis of the workgroup now accesses the same vertex but for different instances, we get a coalesced access pattern. To index into the BSA or BSAK arrays, we use the formula $\text{vertex} * 32 +$ `local_id.x`. This data layout is depicted in Figure 4.4.

We use a parallel strided for loop over `invocation_id.y` for the top-level loops in both the expand and identify steps. Giving us `invocation_id.y` workgroups, where each workgroup advances the state of 1 vertex for all 32 BFS instances concurrently. During the expand step, all threads within a workgroup access the same vertex, eliminating thread divergence in the loop over all outgoing edges and providing coalesced access for reads and writes to the status arrays. The dispatch vector from the driver program now looks like $(1, \textit{Workers}, 1)$, which results in 1024 total searches being executed using *Workers* workgroups.

Bottlenecks:

Set first BSAK:

[1] https://github.com/ThijsDreef/wgpu-msbfs/tree/coalesced-access

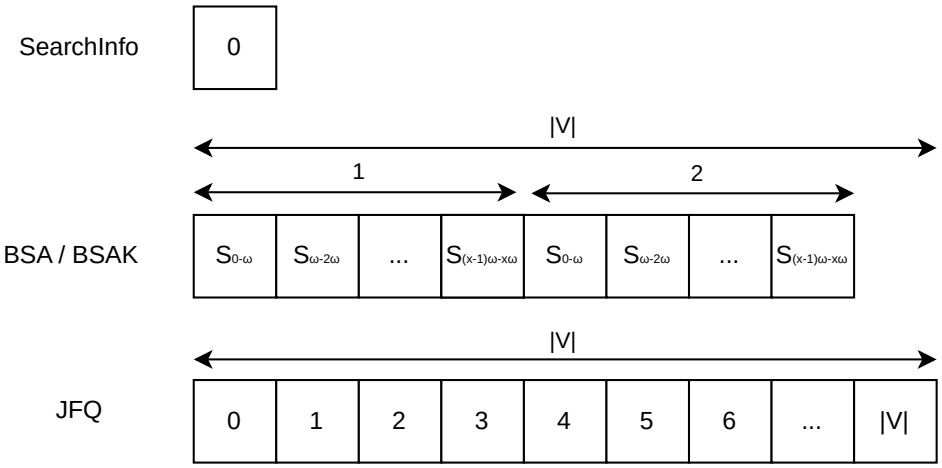


Figure 4.4: Depicts the data layout for the coalesced access method, where ω denotes the amount of concurrent searches and x denotes the amount of instances executed. Note that there is only one search info struct for all $\omega * x$ searches and all these searches share one JFQ.

- **Low achieved occupancy & Theoretical occupancy** Similar to the workgroups approach, we observe low occupancy, both theoretically and practically, due to a limited number of threads being utilized. It has improved over the workgroups approach, as we set BSAK for 1024 searches instead of 32, 64, or 128.

Identify step:

- **L1Tex Global load access pattern:** Access to the `destination` and `path_length` is unoptimal, as only parts of the cache line are read. We tried to optimize this using shared memory, but moving these arrays to shared memory harmed performance.
- **Uncoalesced global accesses:** This ties into the same issue as described for L1Tex Global load access patterns, as the uncoalesced global access to `destination` and `path_length` is the root cause.
- **Achieved occupancy:** The root cause of the achieved occupancy has not changed from the workgroups' approach, except for the expand step taking a maximum of 8.7ms.

Expand step:

- **Long scoreboard stalls:** 78% of all our stalls are due to two dependent reads. The reads are the `v[vertex + 1]` and `edges[outgoingedge]`. We are no longer bound by uncoalesced memory access. But, we are now limited by the memory latency, as we stall until the values of these variables arrive.
- **L1Tex Global load access pattern:** The access pattern for the edges is still not optimal. As we read parts of the edges array, we will always load some entries from the edge array into cache, which the profiler considers non-optimal.

4.3.3.1 Direction switching

With the coalesced access approach, we have achieved coalesced access and no thread divergence in the expand step. Another way to improve performance would be to apply some form of direction switching. With direction switching, the BFS is either executed top-down or bottom-up, depending on a specific heuristic. Direction switching has been shown to improve performance by Beamer et al. (14), Liu et al. (56), Then (86). We implement this optimization in another approach called coalesced access bottom-up[1]

The implementations provided so far are all top-down versions of the BFS algorithm. A top-down implementation of the IBFS algorithm checks the difference between `BSA` and `BSAK` to determine the frontier. To expand the frontier, it propagates the seen state to all outgoing edges. A bottom-up implementation determines the frontier by checking if a vertex has not yet been explored in any active search. The expand step of the bottom-up method is performed using the incoming edges instead of the outgoing edges. Thus, for bottom-up, we propagate the state of all incoming edges to the frontier vertex, and change our random writes into random reads instead.

The `identify` kernel is described in Algorithm 10. Note that this is identical to the default identify kernel, except that the condition to continue on line 4 is modified to only add to the JFQ if there is a search that has not yet encountered this vertex. In the `expand` step, as illustrated in Algorithm 11, we now perform the writes to a local variable (see lines 5 and 7), which we subsequently write out to `BSAK` after the loop on line 8.

Switching between top-down and bottom-up is done dynamically. Creating a sophisticated switching scheme is outside the scope of this thesis. Instead, we use the length of the `JFQ` as a heuristic to determine the direction of the following iterations. A bottom-up iteration in IBFS results in most vertices being in the JFQ, as **any** vertex that has not been seen in **any** search is appended. As our heuristic relies on the length of the JFQ, there is no moment when we can switch back to top-down; thus, we always continue with a top-down iteration after a bottom-up iteration.

Algorithm 10 Pseudocode of the bottom-up identify kernel

```
1:  function IDENTIFY(jfq, destinations, bsak, bsa, mask, lengths, iteration)
2:      for each vertex ∈ V in parallel do
3:          diff ← bsak[vertex] ⊕ bsa[vertex]
4:          if (b̃sak[vertex]) & mask > 0 then continue
5:          bsak[vertex] ← bsak[vertex] | bsa[vertex]
6:          jfq.enqueue(vertex)
7:          activeSearches ← countOneBits(diff)
8:          for x ∈ activeSearches do
9:              index ← countTrailingZeros(diff)
10:             diff ← diff ⊕ (1 << index)
11:             if destinations[index] != vertex then continue
12:             lengths[index] ← iteration
13:             mask ← mask ⊕ (1 << index)
```

[1] https://github.com/ThijsDreef/wgpu-msbfs/tree/coalesced-access-bottom-up

Algorithm 11 Pseudocode of the bottom-up expand kernel, note that V and E represent a reverse CSR thus looping over the incoming edges instead of outgoing.

```
1: function EXPAND(jfq, V, E, bsak, bsa)
2:     for each vertex ∈ jfq in parallel do
3:         first ← V[vertex]
4:         last ← V[vertex + 1]
5:         val = bsa[vertex]
6:         for id ∈ (first...last) do
7:             val ← val | bsa[e[id]]
8:         bsak[vertex] ← val
```

Bottlenecks:

Identify step bottom up: The profiler reports the same issues as for the coalesced access approach.

Expand step bottom up: With the bottom-up expansion, we observe that we wait 36 cycles on average for long scoreboard stalls, whereas this was 41 cycles for the top-down approach. The profiler now additionally reports **SM workload imbalance**, which indicates a roughly 13% load difference between the SM with the minimum workload and the SM with the maximum workload.

4.4 Implementation difference between CUDA and WebGPU

While developing both of our implementations (WebGPU and Cuda), we have run into backend-specific issues. The main issues will be discussed in this section.

Determining the optimal amount of worker threads: A tail effect occurs when one or more SMs are unutilized for a workload because there is nothing to schedule on that SM. In our implementation section, we have only mentioned that there should be an x number of workers. The optimal number of workers depends on the hardware. In our profiler reports using NVIDIA Nsight Compute, we have observed that not scheduling an exact multiple of SM units results in such a tail effect. The effect occurs because the number of workgroups cannot be distributed evenly among the SMs. We eventually reach a point where some SMs have no scheduled work, while others remain fully occupied. A tail effect can have a significant impact on performance. For CUDA, we can determine the number of SMs programatically on our device using `cudaDeviceProp`. However, there is currently no such functionality in WebGPU, which means that our implementation can suffer from a tail effect due to this missing information. We currently use 184 as the value for x, as this is a multiple of the amount of SM's (46) for the NVIDIA RTX 2080.

Voting instructions and synchronization: CUDA provides multiple functions to achieve synchronization between threads within a workgroup or synchronize all workgroups of an invocation, for example, using `__syncthreads`. WebGPU has less sophisticated synchronization primitives. Only providing the ability to synchronize memory access within a workgroup. To use these primitives in WebGPU, the control flow of your kernel is required to be uniform. The primitives available in WebGPU are the following:

- `storageBarrier` ensures that all in-flight storage operations on global GPU memory within a workgroup have been executed after that point.
- `WorkgroupBarrier` ensures that all in-flight storage operations on workgroup local memory have been executed after that point.

These limited primitives make it impossible to synchronize anything outside of a single workgroup in WebGPU efficiently. In our CUDA implementation, we use `__syncthreads()` to reset the JFQ length at the start of an iteration. WebGPU cannot let all threads in a workgroup wait until the length is reset; instead, a clear operation is encoded before the kernel executes instead. Incrementing the iteration is another example of such an issue. To solve this for WebGPU, we increment the iteration count in the expand step using a single thread by utilizing if statements. In CUDA, we can ensure that all threads read the same iteration using `__syncthreads()` and let all threads write the iteration + 1.

WebGPU also does not support voting instructions or other so-called subgroup operations (operations which are implemented as compiler intrinsics for fast communication within a workgroup). An extension is being developed for these subgroup operations (5), as most hardware supports these operations. However, this has not been adopted as of the time of writing this thesis. We observed that by naively implementing a vote instruction using shared memory, as shown in Source Code 4.3, similar performance to CUDA's vote instructions is achieved. We assume that these operations are optimized to vote instructions by the compiler.

```
ar<workgroup> shared: array<u32, workgroup_size>;

@compute
@workgroup_size(32)
fn main(@builtin(local_invocation_id) local_id: vec3<u32>) {
  shared[local_id.x] = variable;
  if (local_id.x == 0u) {
    var acc = 0u;
    for (var j : u32 = 0u; j < 32; j++) {
      acc |= prefix[j];
    }
    if (acc > 0) {
      // There is a thread for which variable was not 0
    }
  }
}
```

Source Code 4.3: Shows how vote instructions are naively implemented in WGSL. Note that all threads write to a shared memory variable and only one thread reads from it. So if any thread's variable was non-zero, the if clause will be executed, which is equivalent to the CUDA ballot function.

Portability of WebGPU using Dawn and WGPU: Throughout this thesis, we have seen that WebGPU delivers on the promise of being write-once, run-anywhere. We have made no changes to the WebGPU portion of the code for compatibility reasons. The

only issue encountered in terms of portability is crashes on Windows when the DirectX backend is used. DirectX does not allow context creation in `DLLMain`, which causes both Dawn and WGPU to crash on Windows. We solved this by creating the WebGPU context when the actual function is called, instead of when the shared library is loaded.

The primary differences between Dawn and WGPU consist mainly of the strictness of validation. Dawn enforces the WebGPU specification more strictly than WGPU. Uniform control flow in WGPU allows early exiting when branching on variables that do not change during execution (which technically breaks uniform control flow), while Dawn does not allow this. Another example is the use of atomics. Dawn forces you to use `atomicLoad` when fetching the value of a variable marked as atomic, while WGPU allows both `x = atomicLoad(&var)` and `x = var`. Both of the syntaxes for WGPU are loaded using atomics, as verified by transforming a WGSL snippet using NAGA (WGPU's shader transpiler).

Profiling and tools: The biggest hurdle in developing performant GPU-accelerated algorithms using WebGPU is the lack of profiling and tools. NVIDIA's Nsight systems (68) was the only tool that could collect any profiling information. The insights gathered by Nsight Systems are limited to unit throughput, GPU memory bandwidth, cache hit rates, and occupancy. Occupancy can be influenced by dispatching more workgroups, which does not impact performance but does result in higher occupancy numbers. Profiling our CUDA implementation using NVIDIA Nsight Compute (67) provides more detailed information, including performance insights, performance counters, and even suggestions on how to improve performance. The tooling for WebGPU is not at the level of existing GPU frameworks, which is to be expected as it is still under active development.

WebGPU, in both Dawn and WGPU, provides a validation layer that ensures the API is used correctly. The validation makes it trivial to track down bugs that coincide with incorrect API usage, such as incorrect pipeline layouts, bindings, or other constructs. The validation layer does not report bugs such as reading out of bounds or incorrect binding size. These issues can create hard-to-detect bugs, as no error is thrown.

Evaluation

We evaluate the performance of our three main methods workgroups, coalesced access and coalesced access bottom up through a series of experiments.

5.1 Experiment setup

We produced three "camera-ready" branches, one for each implementation (workgroups[1], coalesced access[2], coalesced access bottom up[3]). As building and testing on multiple machines can be time-consuming, we simplified this process using Python scripts. To reproduce the results of our experiments on any machine, first install the Python requirements using `pip install -r requirements.txt` after which the build script can be called using `python scripts/build.py`, which executes the following steps.

- Fetch the dataset
- Compute ground truth using DuckDB and DuckPGQ
- Build all supported backends
- Run correctness testing for built backends
- Run benchmarking for built backends

5.1.1 Dataset: LDBC Social Network Benchmark

The Linked Data Benchmark Council (LDBC) (8) provides datasets and standardized benchmarks to evaluate the performance of Graph Database Systems. It provides a set of graph data and queries that simulate real-world scenarios. We use their Social Network Benchmark (SNB) (9) as test data for our algorithms.

The SNB dataset is provided in multiple scale factors (SF), which scale the dataset's size. We run our experiments on SF 1, 3, 10, 30, 100, and 300. The original dataset is trimmed down to the graph represented by the `Person` and `Person_Knows_Person` columns. Where `Person` is used as the vertex and `Person_Knows_Person` is used as the edges. The number of vertices and edges for these graphs at all scale factors is shown in Table 5.1. Selecting source-destination pairs in this graph is done randomly, using the same seed as DuckPGQ's performance testing suite. Enabling us to compare performance results directly to DuckPGQ, as identical source-destination pairs are used for benchmarking. We

[1] https://github.com/ThijsDreef/wgpu-msbfs/tree/workgroups
[2] https://github.com/ThijsDreef/wgpu-msbfs/tree/coalesced-access
[3] https://github.com/ThijsDreef/wgpu-msbfs/tree/coalesced-access-bottom-up

Scale factor (SF)	Number of vertices	Number of edges
1	10620	219450
3	25870	668431
10	70800	2304951
30	175950	6880584
100	487700	23116805
300	1230500	68313982

Table 5.1: Number of vertices and edges for all scale factors used in the experiments.

test with the following number of pairs: 1, 10, 100, 1000, 2048, 4096, 8192, 16384, 32768, and 65536.

5.1.2 Environment

To build our project we require the following dependencies:

- CMAKE 3.15+
- Python 3.x.x
- A C++ compiler

Experiments were performed on multiple machines. The specifications of the machines used in the experiments are shown in Table 5.2. All required dependencies, such as GoogleTest, Google Benchmark, and WebGPU, are automatically fetched and compiled using cmake's `FetchContent`. Reducing the friction for building and testing on multiple machines. The versions of WebGPU backends used throughout this thesis have been kept as up-to-date as possible. Our repository will always attempt to fetch the most up-to-date versions as provided by WebGPU distribution[1] (29). The versions of all dependencies that were automatically fetched are frozen for the experiments to the following:

- Dawn 7187
- WGPU native v24.0.3.1
- Google Test 1.15.2
- Google Benchmark 1.9.1

ID	CPU	GPU	RAM	OS	Compiler	CUDA version
Desktop	I3-6100	RTX 2060	16GB	Windows	MSVC 14.44	12.8
				Linux Ubuntu 24.04	GCC 13.2	12.8
Laptop	I7-9750H	RTX 2080 Mobile	64GB	Arch Linux	GCC 15.1.1	12.8
MacBook	M1 pro	M1 Pro	16GB	Mac OS	Compiler	N/A

Table 5.2: Table depicting the specifications of the machines that ran the experiments

[1] https://github.com/eliemichel/WebGPU-distribution

5.2 Comparing approaches

We first evaluate the different approaches on the same system and use the insights gained to determine the best approach. To assess the three key methods (coalesced access, coalesced access bottom-up, and workgroups), we ran all methods on all test machines.

To compare the different approaches on the same system, we use a normalized execution time bar chart. The execution time is normalized to the fastest method for that scale factor and pair combination. Resulting in the lowest bar being the fastest and the other bars being a factor x slower than the fastest method. Figure 5.1 shows the results for the Linux laptop with the NVIDIA RTX 2080 mobile, Figure 5.2 shows the results for the Windows desktop with the NVIDIA RTX 2060, and Figure 5.3 shows the results for the M1 MacBook.

We observe that the workgroups approach outperforms the coalesced access methods for all numbers of pairs smaller than 1000. This can be attributed to the fact that the workgroups approach utilizes its threads more efficiently with a low number of pairs than the coalesced method. The coalesced methods always execute 1024 searches, whereas the workgroups method executes 32, 64, or 128 concurrently. When executing a search with fewer than 1024 searches, the workgroups approach will utilize its threads more effectively. As the coalesced access method is forced to evaluate 32 IBFS instances, most of which are empty. Executing 1000 or more searches using the coalesced methods outperforms the workgroup approach in all cases. The speedup gained by comparing coalesced access to workgroups is not proportional across all systems. The M1 integrated GPU only achieves a approximately 2.5x improvement, while the discrete cards (NVIDIA RTX 2060 and NVIDIA RTX 2080 Mobile) see a 4-5x improvement.

Determining which of the two coalesced methods is better is hard to decide based on normalized execution time. To better compare the two approaches, we plot the execution times of the coalesced methods for all machines, as can be seen in Figure 5.4, Figure 5.5, and Figure 5.6. These plots show that the runtime cost is low for small-scale factors, a few milliseconds at most, but the gain for larger scale factors can be as large as 50 seconds for a scale factor of 300. This trend is evident on all test machines. The exact scale factor at which bottom-up outperforms depends on the system. The RTX 2080 mobile and the M1 start outperforming top-down at a scale factor of 30, while the RTX 2060 outperforms at a scale factor of 10. From now on, we will compare performance for coalesced access bottom-up, as we claim this to be the most performant overall.

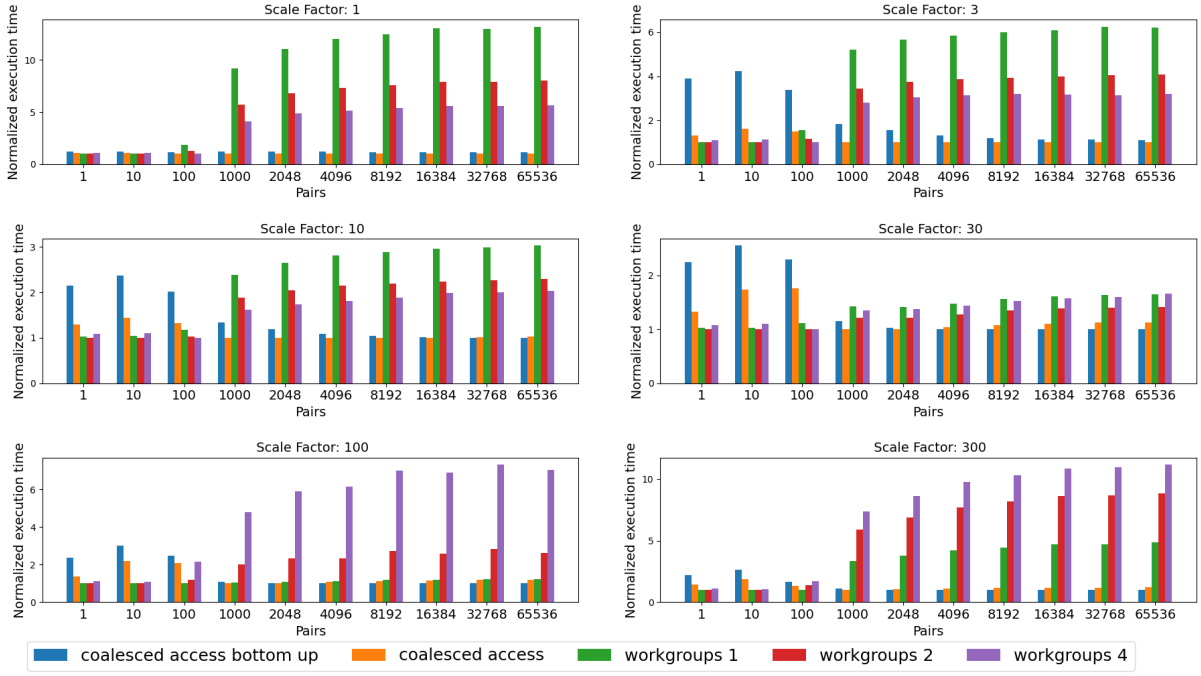


Figure 5.1: Shows the normalized execution times of all approaches ran on the Laptop with the NVIDIA 2080 RTX mobile. Execution times are normalized to the fastest algorithm.

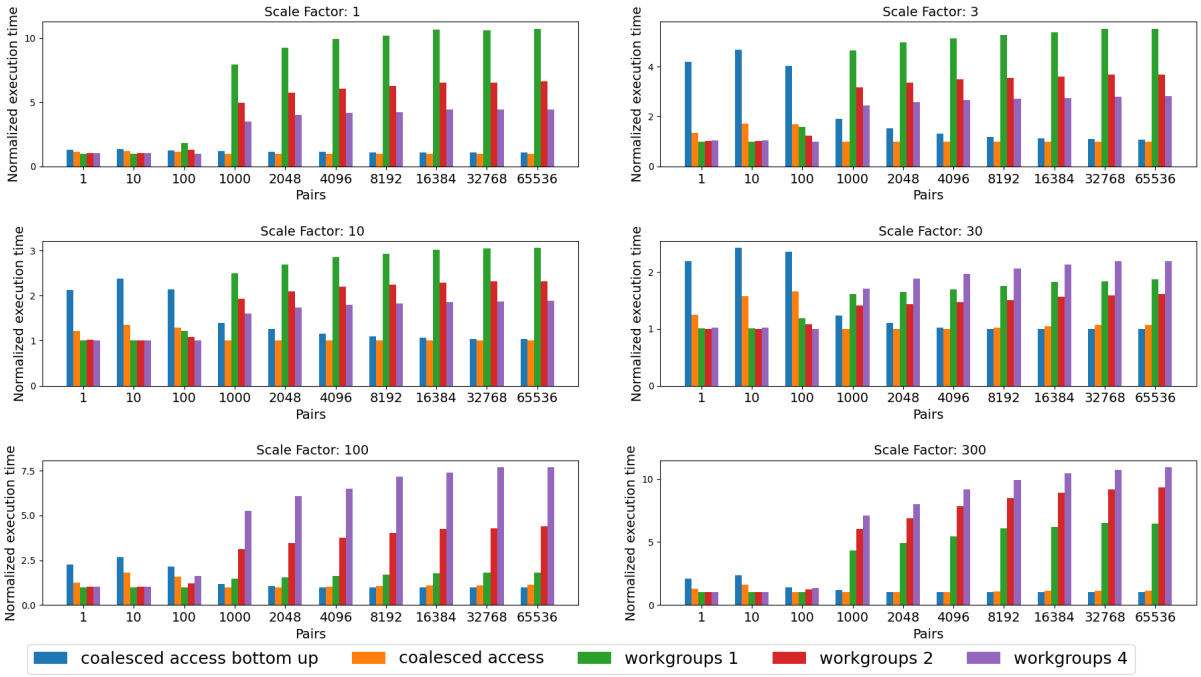


Figure 5.2: Shows the normalized execution times of all approaches ran on the Desktop with the NVIDIA 2060. Execution times are normalized to the fastest algorithm.

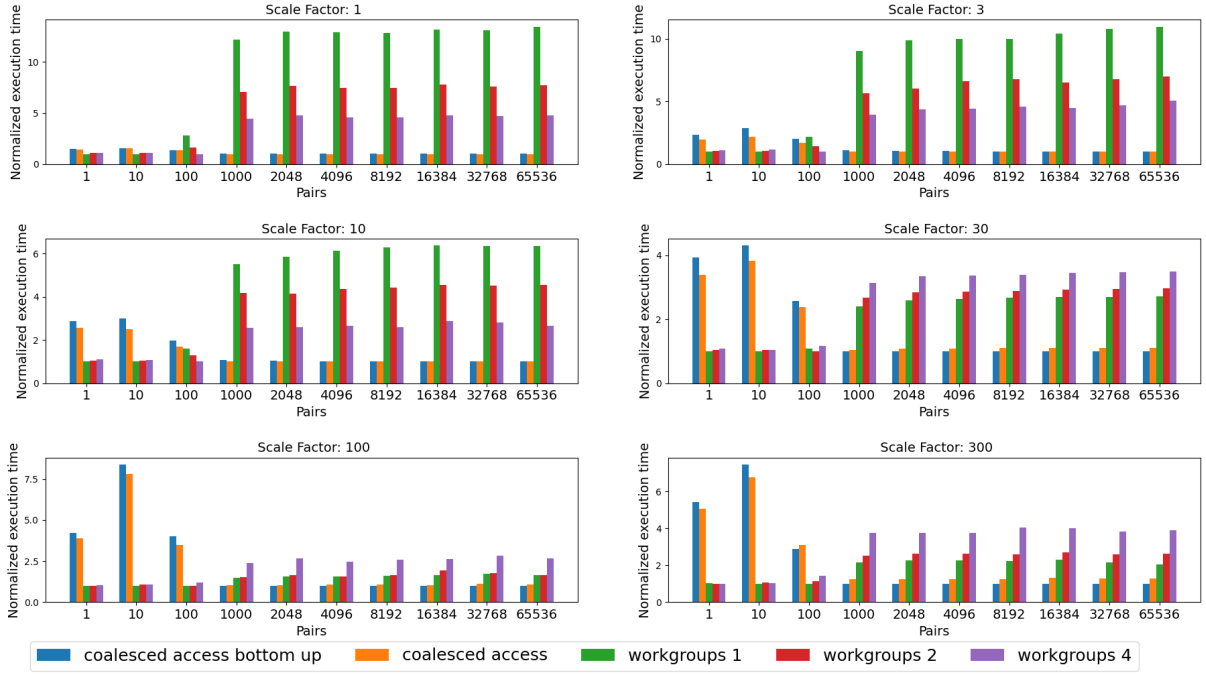


Figure 5.3: Shows the normalized execution times of all approaches ran on the M1 MacBook with the M1 GPU. Execution times are normalized to the fastest algorithm.

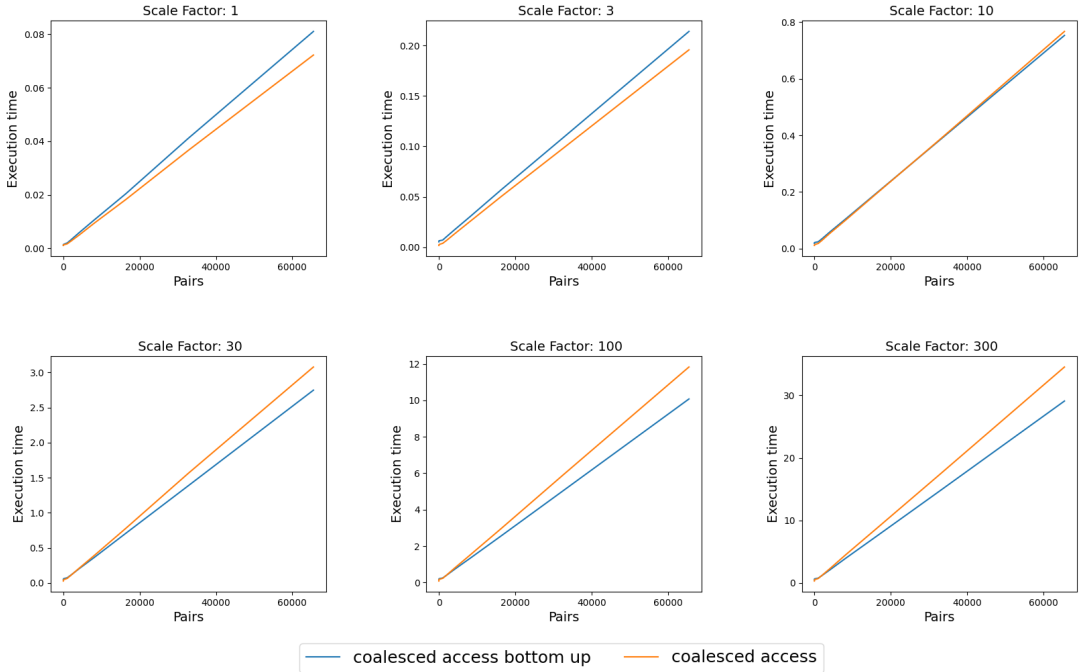


Figure 5.4: Shows the execution times of the coalesced approaches ran on the Laptop with the NVIDIA 2080 RTX mobile.

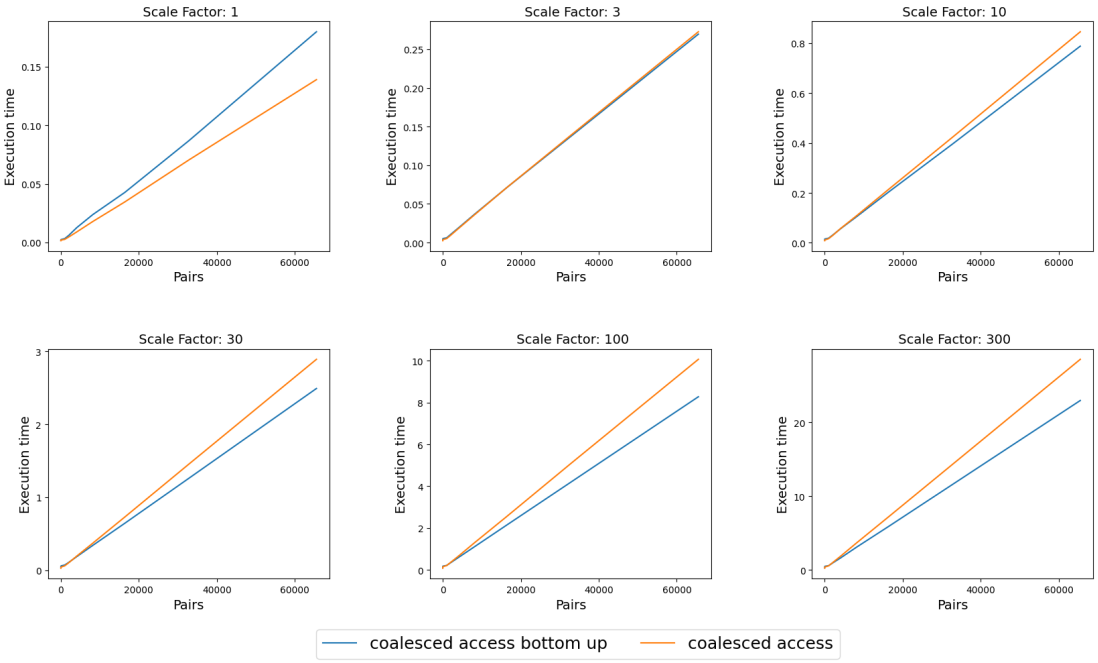


Figure 5.5: Shows the execution times of the coalesced approaches ran on the Desktop with the NVIDIA 2060 RTX mobile.

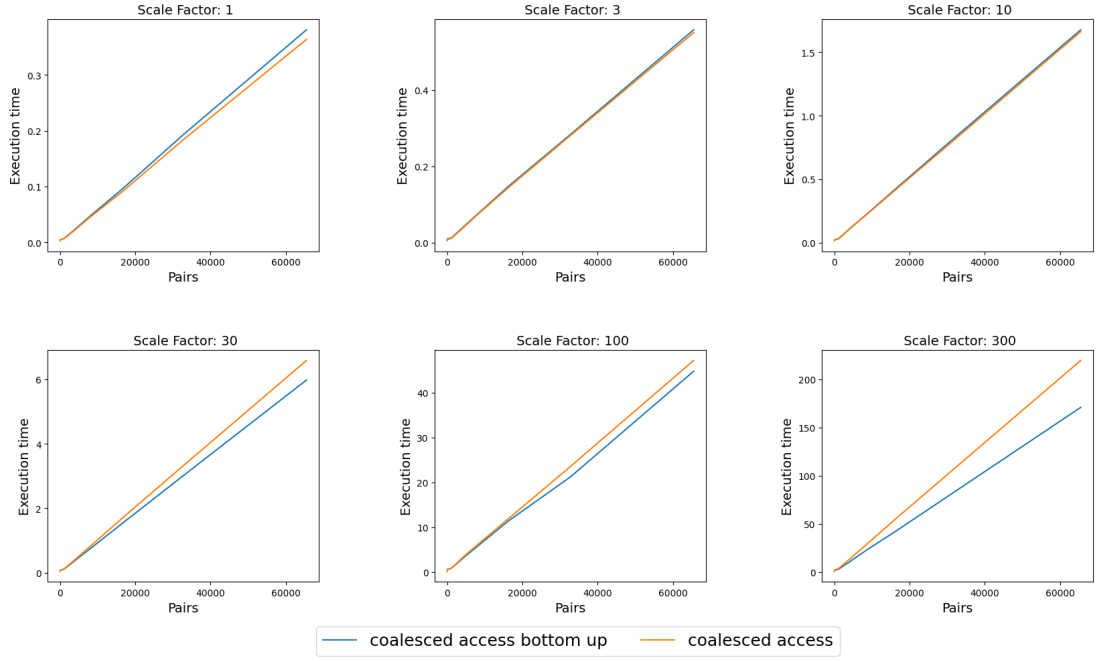


Figure 5.6: Shows the normalized execution times of all approaches ran on the M1 MacBook with the M1 GPU.

5.3 Comparison of different backends

As we have decided on the coalesced access bottom-up method, we want to gain insight into how it performs using the different backends on all machines. These benchmarks are plotted using normalized execution times normalized to the fastest approach.

In Figure 5.7, we see that CUDA outperforms both WebGPU implementations on the laptop with an NVIDIA RTX 2080 mobile. For the smaller scale factors, we see that Dawn outperforms WGPU. When the scale factors grow and the number of pairs increases, the execution times tend to equalize. From a scale factor of 3 and beyond, we observe that the disparity between WebGPU and CUDA trends downward as the number of pairs increases. This suggests that initialization costs for WebGPU are higher than those for CUDA.

Comparing backends for the desktop running Windows with the NVIDIA RTX 2060 can be seen in Figure 5.8. The differences between backends for this machine never exceed 2x. Dawn even outperforms CUDA on certain workloads. CUDA does have a performance benefit when evaluating a smaller number of pairs as compared to WebGPU.

The results for the M1 MacBook are shown in Figure 5.9. These results do not include a CUDA backend, as this is not supported on non-NVIDIA hardware. Dawn outperforms WGPU by 1.75 - 6x for all scale factors and pairs. Pinpointing the source of this performance discrepancy is challenging, as it stems from the differences in implementation between Dawn and WGPU. To investigate whether the issue stems from a shader generation issue, all shaders have been transpiled to their Metal equivalents using both Tint (Dawn) and Naga (WGPU). The key differences found in these Metal shaders are the following:.

- **Bitwise operations:** Dawn masks shift operations on variables using `1 << index & 31`. The mask ensures that we do not exceed the size of the variable, which might allow the compiler to perform optimizations that the WGPU version does not have.
- **Arguments in struct:** WGPU calls the kernel with all variables as arguments, providing a signature that matches our WGSL definition. Dawn creates a struct that holds all the parameters, which results in a smaller function signature.
- **Infinite loop protection:** Both transpilers protect against infinite loops by decrementing a large counter. When this counter hits zero, the loop exits. WGPU initializes these variables in the body of the while loop using a boolean to check if it's the first iteration. Dawn performs this initialization at the top of the loop and omits the check in the loop body.

We are unsure of the exact reason for the large disparity in performance. The Metal source code for all shaders can be found in Appendix A, as generating the Metal source code requires additional tools.

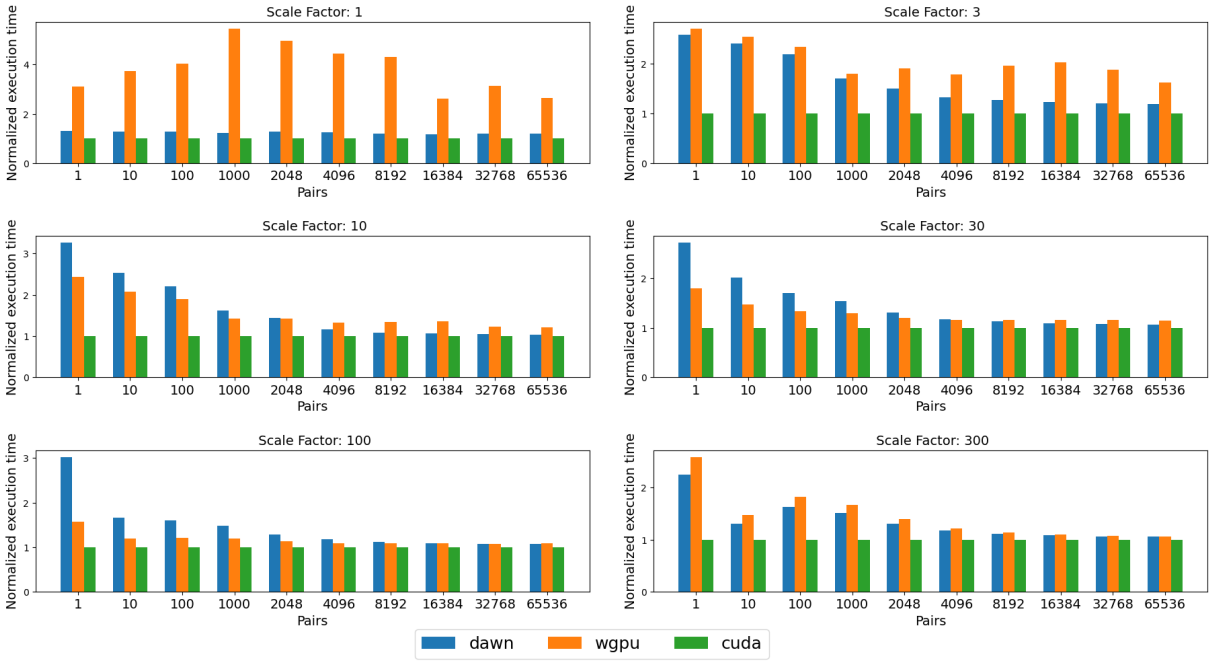


Figure 5.7: Shows the normalized execution times of all backends executing the coalesced bottom up method ran on the laptop with the NVIDIA RTX 2080 mobile.

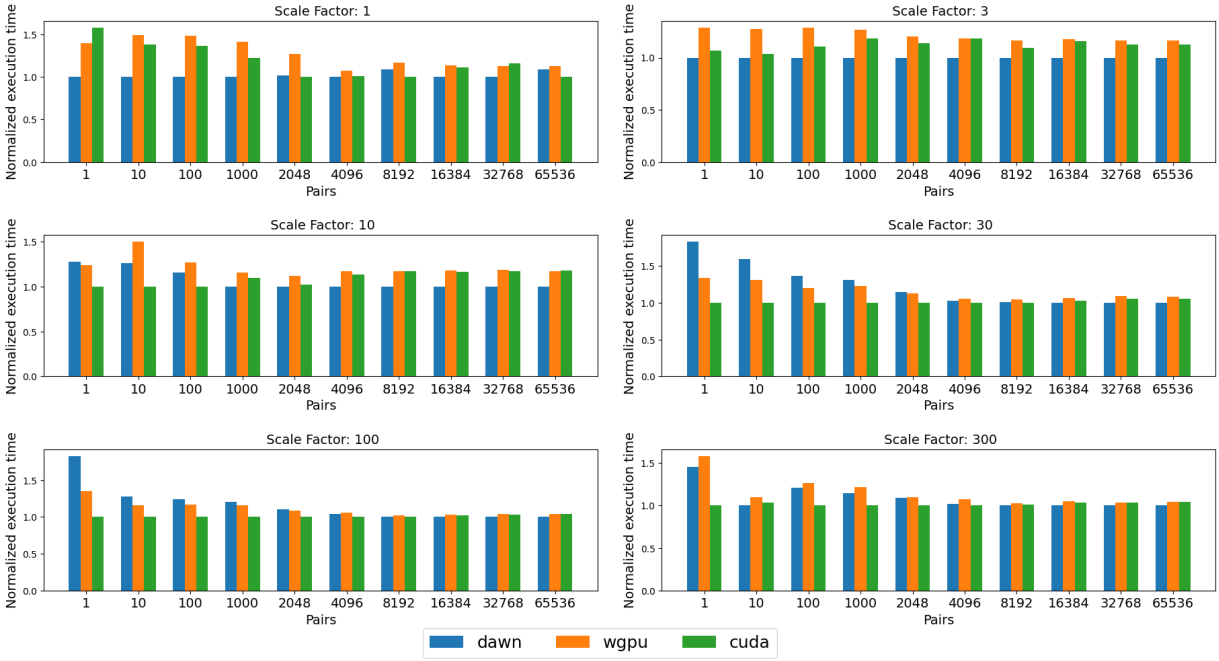


Figure 5.8: Shows the normalized execution times of all backends executing the coalesced bottom up method ran on the desktop running Windows with the NVIDIA RTX 2060.

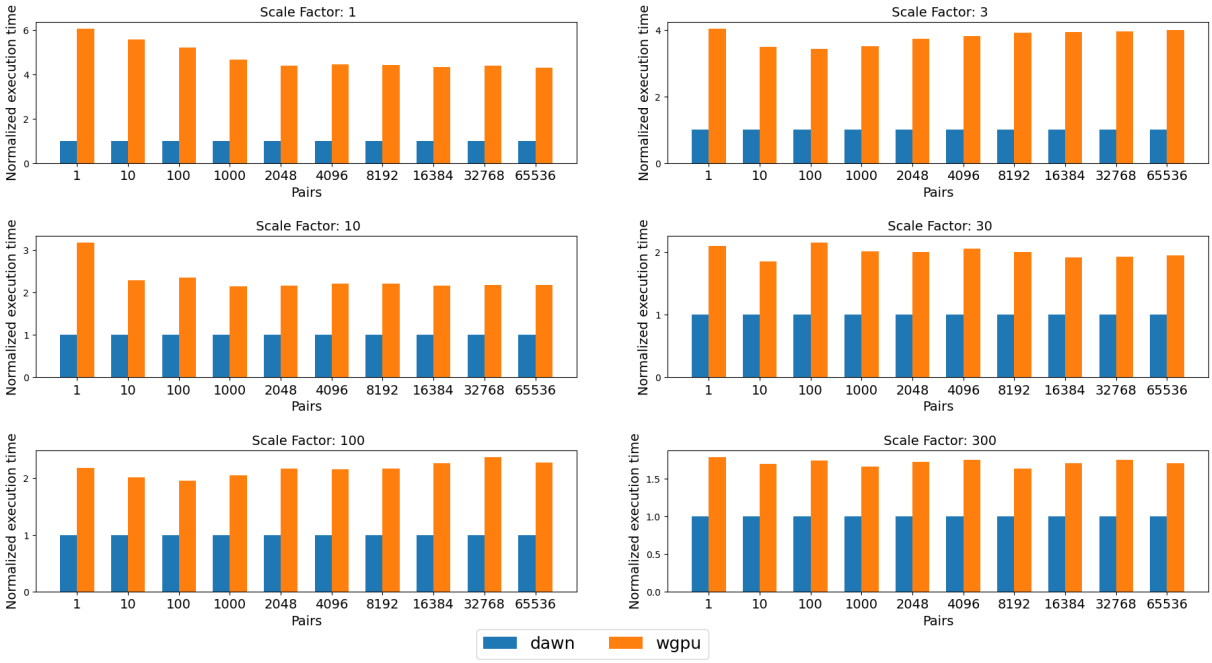


Figure 5.9: Shows the normalized execution times of all backends executing the coalesced bottom up method ran on the MacBook with the M1 GPU.

5.4 Operating systems impact

The experiment is run on the desktop machine using both Windows 10 and Linux (Ubuntu 24.04) to determine the impact that the operating system has on performance. The results for this experiment are shown using normalized execution times in Figure 5.10.

Linux using the CUDA backend performs the best overall. The most significant differences are observed when executing a search on just one pair. As execution times encompass not only algorithm execution but also resource initialization, we can attribute this to the initialization overhead, which differs between operating systems and backends. The same trend as before can be observed: when larger-scale factors and a higher number of pairs are executed, performance tends to equalize.

Overall, no significant performance difference is found between operating systems. The outliers shown in the graph have small scales and a low number of pairs. Since run times are low (smaller than 100 ms), we do not consider this significant.

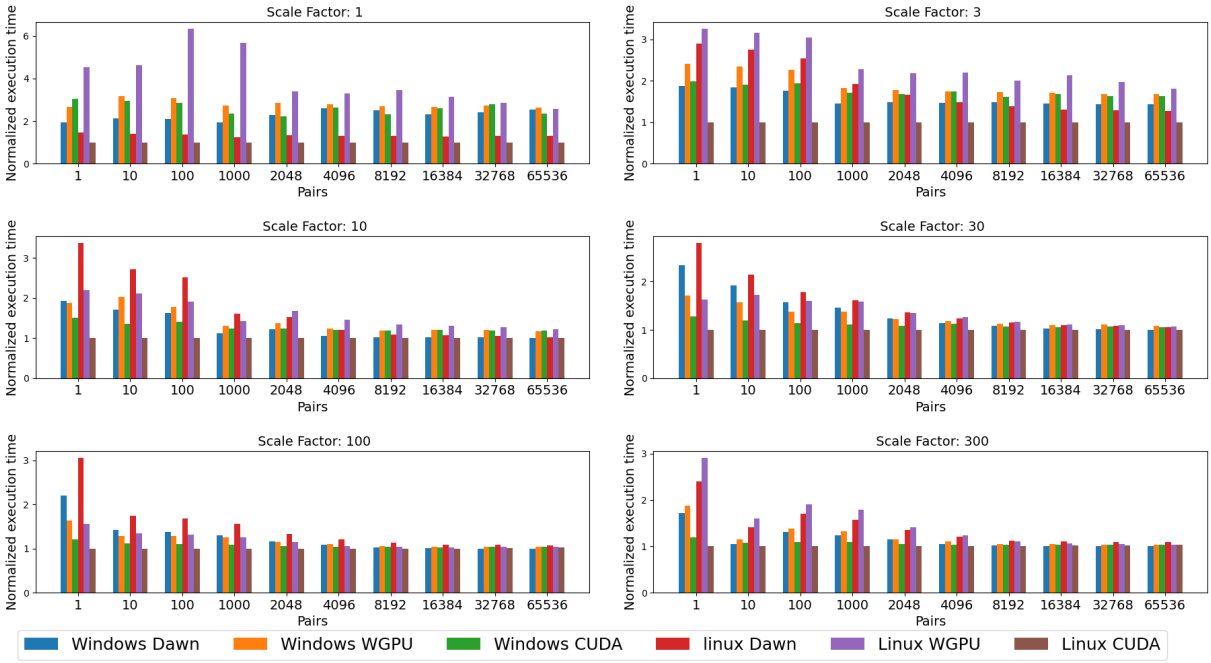


Figure 5.10: Shows the normalized execution times of all backends executing the coalesced bottom up method ran on the desktop with the NVIDIA RTX 2060 on both Windows 10 and Linux Ubuntu 24.04.

5.5 Executing in a web browser

We conducted both correctness testing and benchmarking using Firefox Nightly (142.0a1) and Chrome (137.0.7151.68) on the laptop with an NVIDIA RTX 2080 mobile. The interface for the testing is shown in Figure 5.11.

Executing the native version of the code in the browser resulted in significantly longer execution times. Our program waits for the results by polling the WebGPU implementation. In a native environment, this is required as WebGPU does not have a thread to signal our implementation that a result is ready. In the browser, the event loop runs each time we poll, as this is implemented using Emscripten's sleep function (as implemented by the EMDAWN port maintained by Google (34)). We only regain control when the next event loop runs. This bottleneck is demonstrated by both browser profilers, as shown in Figure 5.12 for Firefox and Figure 5.13 for Chrome. To reduce the number of read-backs, we increase the number of iterations executed before reading any results back to the CPU for the emscripten implementation.

After running the modified code using the test setup, we obtain the results, which we compare to those of the native version in Figure 5.14 using normalized execution time. The difference between web and native versions for Firefox is significant, with the web version being between 8 and 200 times slower. In contrast, Chrome lags behind the native version for the smaller scale factors, while performance tends to equalize for the larger scale factors

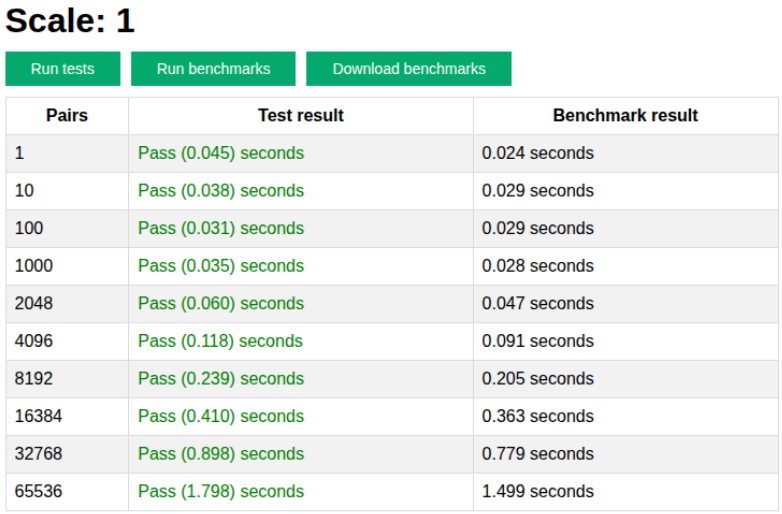


Figure 5.11: Illustrates the web based benchmarking and correctness testing setup. By pressing the buttons one can start correctness testing or benchmarking, after which the results can be downloaded in a JSON format.

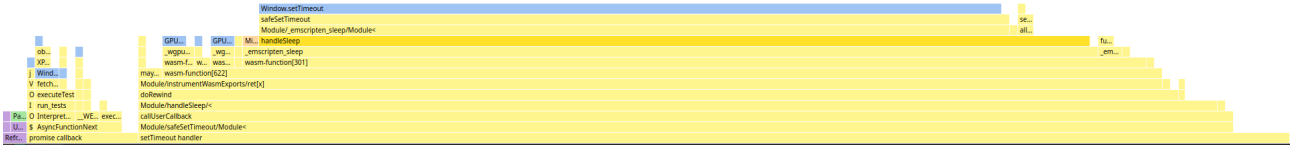


Figure 5.12: Gives a flame chart description of the execution of one run of the IBFS algorithm in the Firefox browser. Note barely any time is spent in WebGPU calls and most time is spent in timeouts waiting for results to be communicated back to the CPU.

Figure 5.13: Shows the profiler trace for one run of the IBFS algorithm. Note that there are long pauses between any GPU work being executed. These pauses are there due to the added overhead of communication between WebGPU and the CPU, resulting from the browser's event loop.

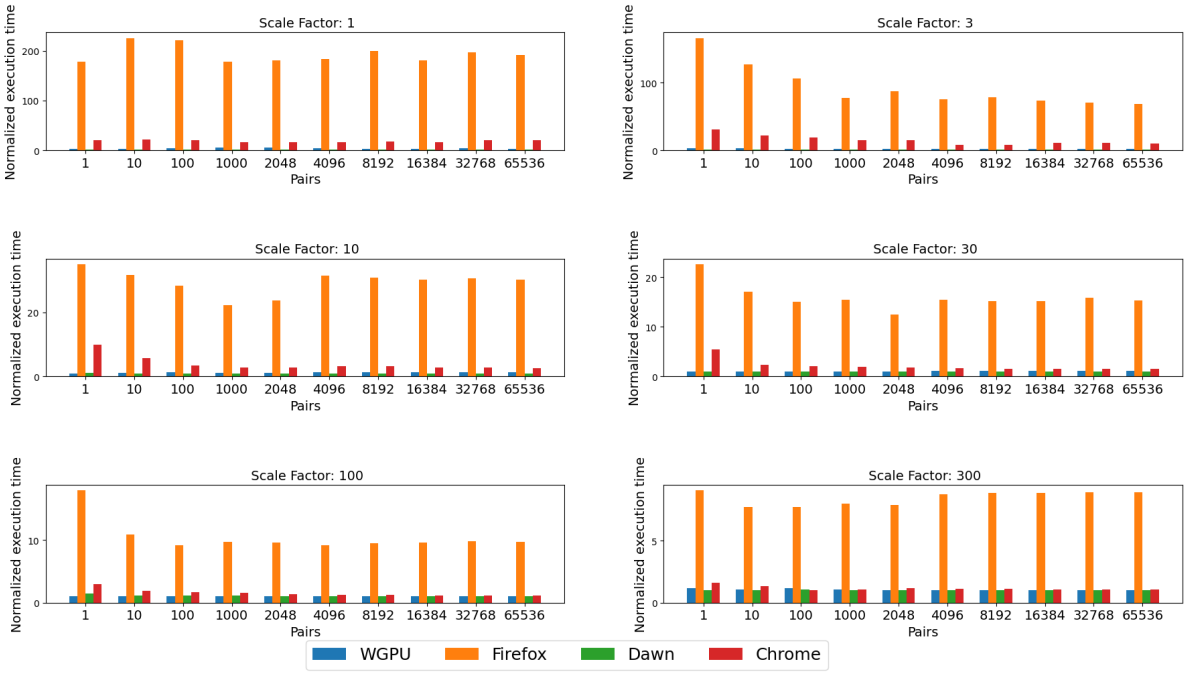


Figure 5.14: Shows the normalized execution times of laptop with the NVIDIA RTX 2080 GPU executing on both WebGPU backends natively and in browser.

5.6 Comparison with DuckPGQ multi-threaded CPU

DuckPGQ has an experimental operator based on the works of Ren (76) and Kaufmann et al. (51) [1]. This experimental operator outperforms the DuckPGQ's morsel-driven parallelism implementation. It achieves this by utilizing the MS-PBFS algorithm, combined with the operator design proposed by Ren (76), and thread local CSRs. With test results available for pairs 16384 - 65536. We first compare our GPU approach to their optimized CPU approach on the same hardware in Figure 5.15. Our approach begins to be outperformed by the CPU at higher scale factors. Starting from scale factor 100, the M1 CPU outperforms its integrated GPU. We are unsure of the exact cause for this observation.

Comparing the M1 CPU against all other discrete cards tested, we observe that the discrete cards outperform the M1 CPU in all cases, as illustrated in Figure 5.16. Note that this is not a fair comparison, as we are comparing the M1 MacBook's integrated GPU against discrete GPUs with much higher thermal design power (TDP), resulting in higher core speeds, higher core counts, and higher bandwidth memory.

[1] https://github.com/cwida/duckpgq-extension/tree/pathfindingoperator-two-phase-csr-lock-free

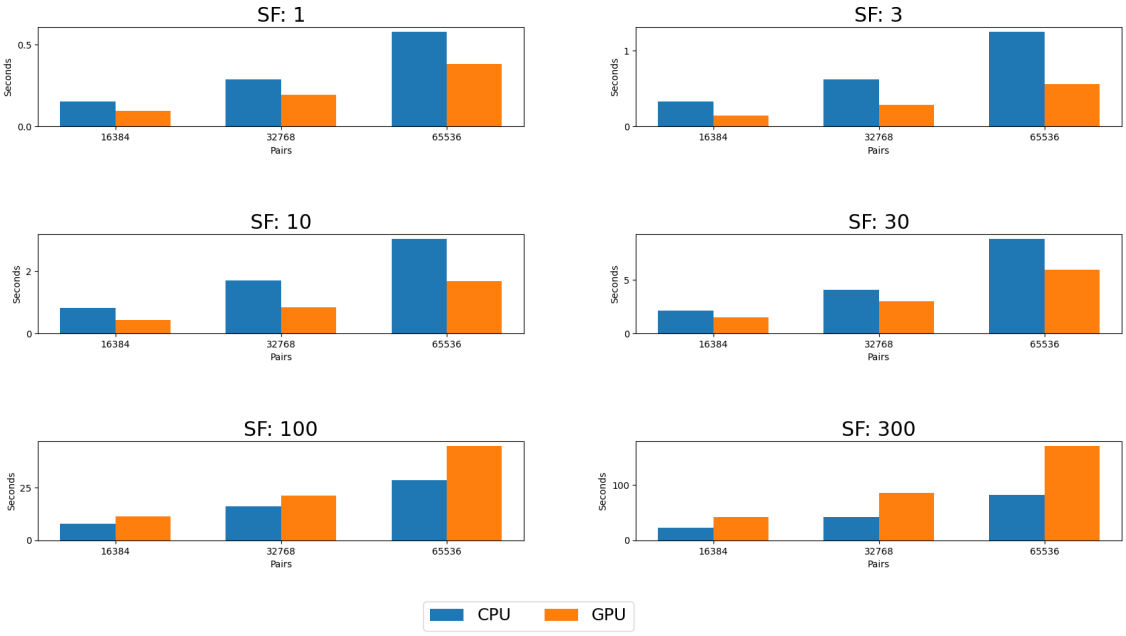


Figure 5.15: Shows the execution times of the M1 MacBook executing the experimental optimized CPU version used by DuckPGQ and our GPU approach ran on the M1 GPU.

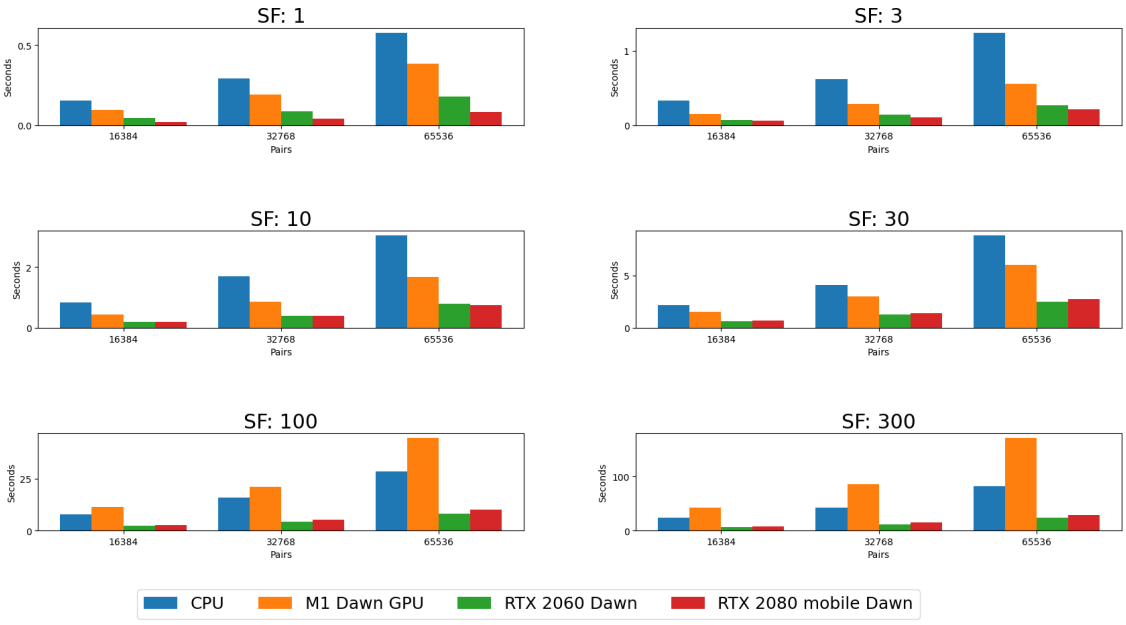


Figure 5.16: Shows the execution times of the M1 MacBook executing the experimental optimized CPU version used by DuckPGQ and our GPU approach on all systems.

5.7 Comparison with Mix and Match (Belewitte)

Comparing our approach with the implementation by Verstraaten et al. (87) (Belewitte) shows that we can perform on par with a small number of pairs using the workgroups approach. The coalesced bottom-up starts outperforming Belewitte starting from 100 pairs. Comparing against Belewitte is not a fair comparison, as this paper accelerates one BFS and outputs the full search tree. Our algorithm optimizes for 1024 BFS instances and only outputs the path length. These observations can be seen in Figure 5.17. We would have also liked to test our implementation against the original IBFS work (56). However, their repository did not compile with the listed dependencies.

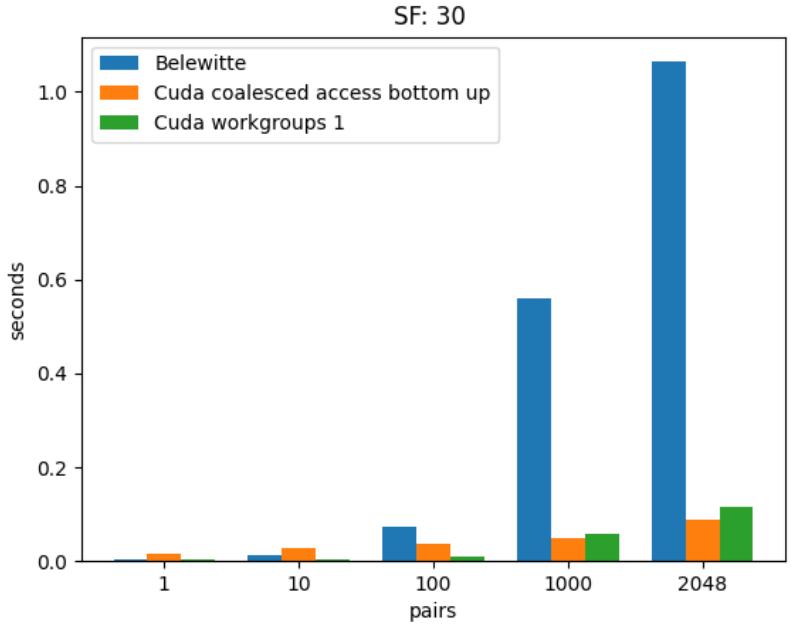


Figure 5.17: Shows the execution times of Mix and Match papers implementation (Belewitte) compared to our coalesced access bottom up and workgroups with one workgroup.

5.8 Underutilization

Using the NVIDIA Nsight Systems Profiler, we see one clear trend across all implementations: we never come close to 100% utilization of the GPU. Utilization of all tested methods, including the work by Verstraaten et al. (87), tends to utilize roughly 40% of bandwidth and compute resources.

Using NVIDIA NSight Compute, we investigate why this underutilization occurs with the coalesced access bottom-up approach. We inspect the performance of this approach, as it has the fewest performance issues, as indicated previously. The GPU spends most of the time executing the expand step. In this kernel, the profiler indicates that execution stalls because of **Long Scoreboard Stalls**. Two reads are responsible for these stalls, which together account for 78% of the kernel's runtime. These reads are `v[vertex + 1]`

and `e[outgoingedge]`. The GPU is forced to stall as these values are required in the computation directly after their request. We conclude that these stalls occur due to GPU memory latency as we are requesting multiple small blocks of data which are not cache coherent.

Future Work

6.1 Comparison between GPU APIs

In our thesis, we have demonstrated that executing a BFS on the GPU is latency-bound. Comparing the performance of different backends using this latency-bound workload provides a biased view of the performance. Before other work relies on WebGPU for portable GPU acceleration more research is required. Other workloads might run into other bottlenecks, they might encounter situations where CUDA outperforms WebGPU so significantly that the portability benefits do not outweight the performance costs.

Future work could focus on using WebGPU as a portable GPU accelerator for different workloads. Providing a clear overview of when using a portable GPU API outweighs the performance cost.

6.2 Extending current approach

6.2.1 Improving workload balance

We have seen that imbalance between workgroups could give up to a 20% performance increase. The workload imbalance has been shown to be improved by sorting the CSR based on the outdegree of vertices by various studies (51, 56, 63). As the scan over the vertices is done in a coalesced manner, the workgroups end up with a roughly equal workload.

Future work could incorporate a sorted CSR based on the outdegree to gain a slight performance improvement.

6.2.2 Computing full path

We have provided a method for computing the length of the path given a source-destination pair. Queries can also request the full path; to return the path, our method needs to be modified. To store the paths for all concurrent searches, $|V|*1024$ extra values are required. These additional values would store the search tree of all concurrent searches, requiring an atomic CAS to set the parent in the identify step. The search tree could then be walked using a new kernel to walk the tree and construct a path.

Future work could implement the creation of a search tree and the walking of the tree to compute the full length path.

6.3 Heterogeneous path length computation

As there are now optimized versions of MS-BFS for both CPU and GPU, heterogeneous computation could be leveraged for path finding on the same system. Utilizing both the CPU and the GPU of a system to compute path length information in DuckPGQ can enhance performance when a large number of source-destination pairs need to be processed. Both versions of the algorithm execute on the same graph and a subset of the pairs. Partitioning the source-destination pairs between the CPU and GPU enables a single system to utilize both processing units. Providing an increase in performance, as a higher number of source-destination pairs can be solved concurrently.

Future work can integrate our GPU method in DuckPGQ and combine it with heterogeneous computation for a performance improvement.

6.4 Optimizing query performance of DuckPGQ

Most queries in DuckPGQ search for a path between a single source and multiple destinations. These queries are broken down into a multitude of source destination pairs, which are then solved using some MS-BFS variant. Accelerating MS-BFS has yielded overall improvements, but the performance is still insufficient. Due to the nature of these queries, either the source values contain a small set of distinct values or the destinations do. Running a single BFS that computes the shortest path for all nodes from a single source would be less computationally intensive than generating and solving a large list of source-destination pairs.

```
-- Find mutual friends between two users
FROM GRAPH_TABLE (snb
  MATCH (p1:Person WHERE p1.id = 16)-[k:knows]->(p2:Person)<-[k2:knows]-(p3:Person WHERE p3.id = 32)
  COLUMNS (p2.firstName)
);
```

Source Code 6.1: Find mutual friend example from DuckPGQ's website `duckpgq.org`

For example, we examine one of DuckPGQ's example queries, as shown in Source Code 6.1. Currently, DuckPGQ solves this query by generating source-destination pairs from p1 and p3 to all other nodes. The same result can be achieved by executing two regular BFSs, which output the length iteration for all nodes, which is less computationally intensive than executing 2 * `nodes` searches using an MS-BFS variant.

It would not always be more performant to use a single-source BFS. When there are a large number of distinct source and destination nodes, an MS-BFS variant would be more performant. Determining when to use an MS-BFS variant or do a regular BFS depends on the query being executed. Picking either option based on an arbitrary query would require future work.

Conclusion

In this thesis, we have examined multiple implementations of the IBFS algorithm, a GPU-accelerated BFS traversal algorithm. These implementations have been done using both CUDA and WebGPU. During the design process, we have explored multiple parallelization schemes, demonstrating that those utilizing coalesced access and minimizing thread divergence perform the best. Using dynamic direction switching between bottom-up and top-down gives an additional performance benefit for higher scale factors. However, we have observed that global memory access latency limits the performance of BFS on the GPU. Furthermore, we demonstrate that our GPU method can outperform a highly optimized CPU version running on an M1 MacBook, depending on the scale factor. Comparing performance on discrete cards shows that GPUs can outperform the optimized CPU version of the algorithm running on a M1 MacBook pro in all cases.

Following are the answers to our research questions.

How to design a GPU based path-finding algorithm? Designing a path-finding algorithm to run on the GPU requires formulating your algorithm in the form of kernels. A driver program then coordinates the resource initialization and execution of these kernels. In this thesis, we designed three kernels, which are coordinated by a single driver function. The parallelization scheme has a significant impact on performance as it determines memory access patterns and control flow. By optimizing the parallelization scheme to access elements in a coalesced manner and minimizing thread divergence the best performance is achieved. The difficulty in designing a GPU-based path-finding algorithm lies in ensuring that memory accesses are coalesced and divergence is minimized. Access to the graph will remain a bottleneck as these lookups are inherently random.

What are the bottlenecks of a GPU path-finding algorithm? The bottlenecks encountered are synchronization costs, thread divergence, coalesced access, workload imbalance, and global access memory latency. We have seen that we can reduce synchronization costs by executing more work on the GPU before attempting to read results back to CPU memory. The chosen parallelization scheme and data layout can heavily influence thread divergence and coalesced access. We have not tried to improve workload imbalance. We conclude that global access memory latency is the biggest bottleneck in a GPU BFS algorithm. 78% of our execution time in the expand step is spent stalling due to accessing the graph structure. These reads cannot be hidden by the GPU as there is no computation between the reads of these values and their usage. This is further excarbated by the accesses to these arrays being random and just a few values are actually read.

When does the GPU provide a speedup compared to a CPU? We have seen that our method, based on IBFS, can outperform the optimized CPU version of MS-PBFS using thread local CSRs[1]. On the M1 MacBook, we observe that our GPU method outperforms the CPU method on scale factors (1, 3, 10, 30), where the CPU outperforms on the other scale factors (100, 300). Comparing the M1 CPU against discrete GPUs shows that GPUs outperform in all cases, but this is not a fair comparison, as the power draw of discrete cards is significantly higher than that of the M1 GPU or CPU.

How to optimize a GPU based path-finding algorithm? To utilize the GPU's resources effectively, it is crucial to adhere to a coalesced memory access pattern whenever possible. Changing the parallelization scheme to reduce thread divergence enables more threads to remain active, thereby improving performance. We observed that the bottleneck in executing path-finding algorithms on the GPU is access to the graph structure. 78% of our execution time is spend stalling due to **Long Scoreboard Stalls** based on our access to the graph structure.

How do performance characteristics differ between integrated and discrete cards? We observe that the performance disparity between methods is lower on integrated hardware than on discrete hardware. The difference between the performance gained by switching from the workgroups approach to coalesced access is disproportionate between discrete and integrated GPUs. For scale factor 300, discrete hardware becomes roughly 7 times faster, while the integrated GPU only sees a 3 times improvement. Overall, it is challenging to make a fair comparison between integrated and discrete GPUs, as core counts, clock speeds, memory bandwidth, and TDP vary.

How do performance characteristics differ between Operating systems? We have tested the same machine using both Windows 10 and Linux Ubuntu (24.04). The most significant differences are observed when low-scale factors and a low number of source-destination pairs are executed. When larger-scale factors and a greater number of source-destination pairs are executed, all performance figures tend to equalize. From this, we conclude that the observed overhead is primarily due to initialization costs and driver overhead.

How does a WebGPU implementation compare to a CUDA implementation? There can be a significant performance discrepancy between WebGPU implementations. For the M1 MacBook, we observe a performance disparity of 1.5 to 6 times between Dawn and WGPU, which we cannot explain. This disparity is not reproducible on other hardware or software setups. For our discrete hardware, we observe that all backends tend towards approximately equal execution times as the scale factors increase. Suggesting that the primary differentiating factors are the initialization costs and the library runtime.

[1] https://github.com/cwida/duckpgq-extension/tree/pathfindingoperator-two-phase-csr-lock-free

References

[1] Arangodb graph database, Jul 2024. URL https://arangodb.com/. Accessed on 27/01/2025. 1

[2] Neo4j graph database, Jul 2024. URL https://neo4j.com/. Accessed on 27/01/2025. 1

[3] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009. 1, 5

[4] Mlc AI. Webllm, 2025. URL https://webllm.mlc.ai/. 27

[5] teoxoy Alan baker, exrook. Webgpu subgroup operations proposal, 2025. URL https://github.com/gpuweb/gpuweb/blob/main/proposals/subgroups.md. 42

[6] AMD. Orochi, 2024. https://gpuopen.com/orochi/ [Accessed 13-02-2025]. 10

[7] AMD. Hip documentation about wavefronts / warps / workgroups, 2025. URL https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html. 34

[8] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC Social Network Benchmark. *CoRR*, abs/2001.02299, 2020. URL http://arxiv.org/abs/2001.02299. 21, 45

[9] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. *CoRR*, abs/2001.02299, 2020. URL http://arxiv.org/abs/2001.02299. 45

[10] Apple. metal, 2025. https://developer.apple.com/metal/ [Accessed 13-02-2025]. 11

[11] Apple. Opencl for macos developer, 2025. https://developer.apple.com/opencl/ [Accessed: (11 February 2025)]. 10

[12] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. Parallel breadth-first search ltl model-checking. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 106–115. IEEE, 2003. 20

[13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013. 20, 23, 25

[14] Scott Beamer, Aydin Buluç, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1618–1627, 2013. doi: 10.1109/IPDPSW.2013.159. 21, 40

[15] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. Darthshader: Fuzzing webgpu shader translators & compilers. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 690–704, 2024. 26

[16] Jim Blandy, Kai Ninomiya, and Brandon Jones. WebGPU. Candidate recommendation, W3C, January 2025. https://www.w3.org/TR/2025/CRD-webgpu-20250131/. 2, 11

[17] BlazingDB. Blazingsql, 2021. URL `https://github.com/BlazingDB/blazingsql?tab=readme-ov-file`. 18

[18] Avi Bleiweiss. Gpu accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, 2008. 2, 10

[19] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005. 5

[20] Angela Bonifati, M. Tamer Ozsu, Yuanyuan Tian, Hannes Voigt, Wenyuan Yu, and enjie Zhang. A roadmap to graph analytics. *SIGMOD Rec.*, 53(4):43–51, January 2025. ISSN 0163-5808. doi: 10.1145/3712311.3712323. URL `https://doi.org/10.1145/3712311.3712323`. 2

[21] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011. 20, 21

[22] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307710. doi: 10.1145/2063384.2063471. URL `https://doi.org/10.1145/2063384.2063471`. 21

[23] Antonio Jesús Chaves, Cristian Martín, and Manuel Díaz. The orchestration of machine learning frameworks with data streams and gpu acceleration in kafka-ml: A deep-learning performance comparative. *Expert Systems*, 41(2):e13287, 2024. 1

[24] Zhiyang Chen, Yun Ma, Haiyang Shen, and Mugeng Liu. Weinfer: Unleashing the power of webgpu on llm inference in web browsers. In *Proceedings of the ACM on Web Conference 2025*, pages 4264–4273, 2025. 27

[25] CHIP-SPV. chipstar, 2025. `https://github.com/CHIP-SPV/chipStar` [Accessed 13-02-2025]. 10

[26] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, January 2019. ISSN 2150-8097. doi: 10.14778/3303753.3303760. URL `https://doi.org/10.14778/3303753.3303760`. 19

[27] Periklis Chrysogelos, Aunn Raza, Manos Karpathiotakis, vsanca, Hamish Nicholson, Lionel Sambuc, panos-sioulas, tahirazim, ember-tomster, and Alex Huang. epfl-dias/proteus. https://github.com/epfl-dias/proteus, sep 29 2023. URL `https://github.com/epfl-dias/proteus`. 20

[28] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022. 1, 5

[29] elimichel. Webgpu distribution, 2025. URL `https://github.com/eliemichel/WebGPU-distribution`. 46

[30] Ethan Ferguson, Adam Wilson, and Hoda Naghibijouybari. Webgpu-spy: Finding fingerprints in the sandbox through gpu cache attacks. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 158–171, 2024. 25

[31] Zhisong Fu, Harish Kumar Dasari, Bradley Bebee, Martin Berzins, and Bryan Thompson. Parallel breadth first search on gpu clusters. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 110–118. IEEE, 2014. 20

[32] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for GPU computing. In *19th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2013, Seoul, Korea, December 15-18, 2013*, pages 275–282. IEEE Computer Society, 2013. doi: 10.1109/ICPADS.2013.47. URL `https://doi.org/10.1109/ICPADS.2013.47`. 14

[33] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*, pages 121–131, 2019. 33

[34] Google. emdawnwebgpu a emscripten port for webgpu, 2025. URL `https://dawn.googlesource.com/dawn/+/refs/heads/chromium/6814/src/emdawnwebgpu/`. 54

[35] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 325–336, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934340. doi: 10.1145/1142473.1142511. URL `https://doi.org/10.1145/1142473.1142511`. 15

[36] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*, pages 209–218. IEEE, 1993. 18

[37] Yudong Han, Weichen Bi, Ruibo An, Deyu Tian, Qi Yang, and Yun Ma. Gl2gpu: Accelerating webgl applications via dynamic api translation to webgpu. In *Proceedings of the ACM on Web Conference 2025*, pages 751–762, 2025. 26

[38] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008. 15

[39] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):1–39, 2009. 15

[40] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877*, 2022. 18

[41] HeavyAI. Omniscidb, 2019. URL `https://heavyai.github.io/heavydb/index.html`. 17

[42] HeavyAI. Heavdb, 2025. URL `https://www.heavy.ai/product/heavydb`. 18

[43] HeavyAI. Heavydb commercial info, 2025. URL `https://www.heavy.ai/product/overview`. 18

[44] HeteroDB. Pg-strom gpu direct, 2025. URL `https://heterodb.github.io/pg-strom/ssd2gpu/`. 17

[45] heterodb. pg-strom, 2025. URL `https://github.com/heterodb/pg-strom?tab=readme-ov-file`. 16

[46] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011. 25

[47] IEEE. IEEE Standard for Floating-Point Arithmetic. https://ieeexplore.ieee.org/document/8866810, 2019. IEEE Std 754-2019. 26

[48] intel. What Is Intel® AVX-512? - Intel — intel.com, 2025. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html [Accessed 13-02-2025]. 7

[49] Raghav G Jha and Abhishek Samlodia. Gpu-acceleration of tensor renormalization with pytorch using cuda. *Computer Physics Communications*, 294:108941, 2024. 2, 10

[50] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, page 55–62, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314459. doi: 10.1145/2236584.2236592. URL https://doi.org/10.1145/2236584.2236592. 15

[51] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. Parallel array-based single-and multi-source breadth first searches on large dense graphs. In *EDBT*, pages 1–12, 2017. 23, 56, 61

[52] Khronos. Opengl homepage, 2021. https://www.opengl.org/ [Accessed 13-02-2025]. 9

[53] Khronos. Vulkan, 2025. https://www.vulkan.org/ [Accessed 13-02-2025]. 11

[54] Richard E Korf and Peter Schultze. Large-scale parallel breadth-first search. In *AAAI*, volume 5, pages 1380–1385, 2005. 20, 21

[55] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2610507. URL https://doi.org/10.1145/2588555.2610507. 5

[56] Hang Liu, H Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416, 2016. 2, 7, 10, 24, 25, 29, 32, 40, 58, 61

[57] David Luebke. Gpu architecture: Implications & trends. *SIGGRAPH 2008: Beyond Programmable Shading Course Materials*, 2008. 9

[58] David Luebke and M Harris. General-purpose computation on graphics hardware. In *Workshop, SIGGRAPH*, volume 33, page 6, 2004. 9

[59] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55, 2010. 20

[60] Microsoft. Directx getting started page, 2025. https://learn.microsoft.com/en-us/windows/uwp/gaming/directx-programming [Accessed 13-02-2025]. 9, 11

[61] Marko J. Mišić, Đorđe M. Đurđević, and Milo V. Tomašević. Evolution and trends in gpu computing. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 289–294, 2012. 9

[62] Todd Mostak. An overview of mapd (massively parallel database). *White paper. Massachusetts Institute of Technology*, 2013. 17

[63] Reece Neff, Mostafa Eghbali Zarch, Marco Minutoli, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Michela Becchi. Fused breadth-first probabilistic traversals on distributed gpu systems. *arXiv preprint arXiv:2311.10201*, 2023. 24, 61

[NVIDIA] NVIDIA. V100 volta architecture. URL `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. 9

[64] nvidia. Introduction rapids by nvidia, 2018. URL `https://nvidianews.nvidia.com/news/nvidia-introduces-rapids-open-source-gpu-acceleration-platform-for-large-scale-da` 18

[65] NVIDIA. Gpu direct storage, 2019. URL `https://developer.nvidia.com/blog/gpudirect-storage/`. v, 17

[66] NVIDIA. About cuda, 2025. `https://developer.nvidia.com/about-cuda` [Accessed: (11 February 2025)]. 8, 10

[67] NVIDIA. Nvidia nsight compute, 2025. URL `https://developer.nvidia.com/nsight-compute`. 43

[68] NVIDIA. Nvidia nsight systems, 2025. URL `https://developer.nvidia.com/nsight-systems`. 43

[69] NVIDIA. Cuda documentation warp size, 2025. URL `https://docs.nvidia.com/cuda/parallel-thread-execution/`. 34

[70] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009. doi: 10.1007/978-3-642-10424-4_17. URL `https://doi.org/10.1007/978-3-642-10424-4_17`. 16

[71] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 8

[72] Sebastian Paarmann. A webgpu backend for futhark. 2024. URL `https://www.futhark-lang.org/student-projects/sebastian-msc-thesis.pdf`. 26

[73] Jaroslav Pokornỳ. Graph databases: their power and limitations. In *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*, pages 58–69. Springer, 2015. 1

[74] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019. 1, 5

[75] RAPIDS. Rapids homepage, 2025. URL `https://rapids.ai/`. 18

[76] Pingan Ren. Parallelized path-finding in duckpgq. 2024. 1, 56

[77] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352826. doi: 10.1145/3085504.3085521. URL `https://doi.org/10.1145/3085504.3085521`. 15

[78] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550. IEEE, 2015. 15

[79] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 1617–1632, 2020. v, 15, 16, 18

[80] Bin Shao, Yatao Li, Haixun Wang, and Huanhuan Xia. Trinity graph engine and its applications. *IEEE Data Eng. Bull.*, 40(3):18–29, 2017. URL `http://sites.computer.org/debull/A17sept/p18.pdf`. 1

[81] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, 2019. doi: 10.1109/ICDE.2019.00068. 15

[82] Evangelia A Sitaridi and Kenneth A Ross. Optimizing select conditions on gpus. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, pages 1–8, 2013. 15

[83] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40, 2011. 1

[84] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. *CoRR*, abs/1611.01137, 2016. URL `http://arxiv.org/abs/1611.01137`. 15

[85] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A. Boncz. Duckpgq: Efficient property graph queries in an analytical RDBMS. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. URL `https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf`. 1

[86] Manuel Then. *Efficient batched graph analytics through algorithmic transformation*. PhD thesis, Technical University Munchen, 2017. v, 1, 6, 21, 22, 23, 24, 25, 29, 40

[87] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. Mix-and-match: A model-driven runtime optimisation strategy for bfs on gpus. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 53–60, 2018. doi: 10.1109/IA3.2018.00014. 25, 58

[88] vosen. Zluda, 2025. `https://github.com/vosen/ZLUDA` [Accessed 13-02-2025]. 10

[89] W3C. wgsl, 2025. `https://www.w3.org/TR/WGSL/` [Accessed 13-02-2025]. 14

[90] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with gpus. *Proc. VLDB Endow.*, 7(11):1011–1022, July 2014. ISSN 2150-8097. doi: 10.14778/2732967.2732976. URL `https://doi.org/10.14778/2732967.2732976`. 15

[91] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. Duckpgq: Bringing sql/pgq´ to duckdb. *Proceedings of the VLDB Endowment*, 16(12):4034–4037, 2023. 5

[92] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118, 2012. doi: 10.1109/MICRO.2012.19. 15

[93] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. Relational joins on gpus: A closer look. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2663–2673, 2017. doi: 10.1109/TPDS.2017.2677451. 15

[94] Bobbi Yogatama, Brandon Miller, Yunsong Wang, Graham Markall, Jacob Hemstad, Gregory Kimball, and Xiangyao Yu. Accelerating user-defined aggregate functions (udaf) with block-wide execution and jit compilation on gpus. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, pages 19–26, 2023. 18

[95] Yanhong Zhuo, Tao Zhang, Feng Du, and Ruilin Liu. A parallel particle swarm optimization algorithm based on gpu/cuda. *Applied Soft Computing*, 144:110499, 2023. 2, 10

Metal shader sources

A.1 Set First BSAK Dawn

```
#include <metal_stdlib>
using namespace metal;

template<typename T, size_t N>
struct tint_array {
  const constant T& operator[](size_t i) const constant { return
  ↪  elements[i]; }
  device T& operator[](size_t i) device { return elements[i]; }
  const device T& operator[](size_t i) const device { return elements[i]; }
  thread T& operator[](size_t i) thread { return elements[i]; }
  const thread T& operator[](size_t i) const thread { return elements[i]; }
  threadgroup T& operator[](size_t i) threadgroup { return elements[i]; }
  const threadgroup T& operator[](size_t i) const threadgroup { return
  ↪  elements[i]; }
  T elements[N];
};

struct SearchInfo {
  /* 0x0000 */ uint offset;
  /* 0x0004 */ uint iteration;
  /* 0x0008 */ uint mask;
  /* 0x000c */ uint jfq_length;
  /* 0x0010 */ uint last_jfq;
};

struct tint_module_vars_struct {
  const device tint_array<SearchInfo, 1>* info;
  const device tint_array<uint, 1>* src;
  device tint_array<atomic_uint, 1>* bsak;
  const constant tint_array<uint4, 1>* tint_storage_buffer_sizes;
};

struct tint_array_lengths_struct {
  uint tint_array_length_0_1;
```

```
  uint tint_array_length_0_2;
  uint tint_array_length_0_0;
};

uint tint_div_u32(uint lhs, uint rhs) {
  return (lhs / select(rhs, 1u, (rhs == 0u)));
}

void main_inner(uint3 local_id, uint3 invocation, uint3 invocation_size,
↪  tint_module_vars_struct tint_module_vars) {
  tint_array_lengths_struct const v =
  ↪  tint_array_lengths_struct{.tint_array_length_0_1=((*tint_module_vars.tint_storage_buf
  ↪  / 4u),
  ↪  .tint_array_length_0_2=((*tint_module_vars.tint_storage_buffer_sizes)[0u].z
  ↪  / 4u),
  ↪  .tint_array_length_0_0=((*tint_module_vars.tint_storage_buffer_sizes)[0u].x
  ↪  / 20u)};
  uint index = (local_id.x + (*tint_module_vars.info)[min(invocation.x,
  ↪  (v.tint_array_length_0_0 - 1u))].offset);
  if ((index >= v.tint_array_length_0_1)) {
    return;
  }
  uint v_size = tint_div_u32(v.tint_array_length_0_2, invocation_size.x);
  uint temp = ((*tint_module_vars.src)[min(index, (v.tint_array_length_0_1
  ↪  - 1u))] + (v_size * invocation.x));
  atomic_fetch_or_explicit((&(*tint_module_vars.bsak)[min(temp,
  ↪  (v.tint_array_length_0_2 - 1u))]), (1u << (local_id.x & 31u)),
  ↪  memory_order_relaxed);
}

kernel void v_1(uint3 local_id [[thread_position_in_threadgroup]], uint3
↪  invocation [[threadgroup_position_in_grid]], uint3 invocation_size
↪  [[threadgroups_per_grid]], const device tint_array<SearchInfo, 1>* info
↪  [[buffer(0)]], const device tint_array<uint, 1>* src [[buffer(1)]],
↪  device tint_array<atomic_uint, 1>* bsak [[buffer(2)]], const constant
↪  tint_array<uint4, 1>* tint_storage_buffer_sizes [[buffer(30)]]) {
  tint_module_vars_struct const tint_module_vars =
  ↪  tint_module_vars_struct{.info=info, .src=src, .bsak=bsak,
  ↪  .tint_storage_buffer_sizes=tint_storage_buffer_sizes};
  main_inner(local_id, invocation, invocation_size, tint_module_vars);
}
```

A.2 Set First BSAK WGPU

```
// language: metal1.0
#include <metal_stdlib>
```

```
#include <simd/simd.h>

using metal::uint;

struct _mslBufferSizes {
    uint size0;
    uint size1;
    uint size2;
};

struct SearchInfo {
    uint offset;
    uint iteration;
    uint mask;
    uint jfq_length;
    uint last_jfq;
};
typedef SearchInfo type_1[1];
typedef uint type_2[1];
typedef metal::atomic_uint type_4[1];
uint naga_div(uint lhs, uint rhs) {
    return lhs / metal::select(rhs, 1u, rhs == 0u);
}


struct main_Input {
};
kernel void main_(
  metal::uint3 local_id [[thread_position_in_threadgroup]]
, metal::uint3 invocation [[threadgroup_position_in_grid]]
, metal::uint3 invocation_size [[threadgroups_per_grid]]
, device type_1 const& info [[user(fake0)]]
, device type_2 const& src [[user(fake0)]]
, device type_4& bsak [[user(fake0)]]
, constant _mslBufferSizes& _buffer_sizes [[user(fake0)]]
) {
    uint index = {};
    uint v_size = {};
    uint temp = {};
    uint _e8 = info[invocation.x].offset;
    index = local_id.x + _e8;
    uint _e11 = index;
    if (_e11 >= (1 + (_buffer_sizes.size1 - 0 - 4) / 4)) {
        return;
    }
    v_size = naga_div(1 + (_buffer_sizes.size2 - 0 - 4) / 4,
    ↪  invocation_size.x);
```

```
    uint _e21 = index;
    uint _e23 = src[_e21];
    uint _e24 = v_size;
    temp = _e23 + (_e24 * invocation.x);
    uint _e30 = temp;
    uint _e35 = metal::atomic_fetch_or_explicit(&bsak[_e30], 1u <<
    ↪  local_id.x, metal::memory_order_relaxed);
    return;
}
```

A.3 Frontier expansion Dawn

```
#include <metal_stdlib>
using namespace metal;

template<typename T, size_t N>
struct tint_array {
  const constant T& operator[](size_t i) const constant { return
  ↪  elements[i]; }
  device T& operator[](size_t i) device { return elements[i]; }
  const device T& operator[](size_t i) const device { return elements[i]; }
  thread T& operator[](size_t i) thread { return elements[i]; }
  const thread T& operator[](size_t i) const thread { return elements[i]; }
  threadgroup T& operator[](size_t i) threadgroup { return elements[i]; }
  const threadgroup T& operator[](size_t i) const threadgroup { return
  ↪  elements[i]; }
  T elements[N];
};

struct SearchInfo {
  /* 0x0000 */ uint offset;
  /* 0x0004 */ uint iteration;
  /* 0x0008 */ uint mask;
  /* 0x000c */ uint jfq_length;
  /* 0x0010 */ uint last_jfq;
};

struct tint_module_vars_struct {
  const device tint_array<uint, 1>* v;
  const device tint_array<uint, 1>* e;
  device tint_array<uint, 1>* jfq;
  device tint_array<SearchInfo, 1>* search_info;
  const device tint_array<uint, 1>* bsa;
  device tint_array<atomic_uint, 1>* bsak;
  const constant tint_array<uint4, 2>* tint_storage_buffer_sizes;
};
```

```
struct tint_array_lengths_struct {
  uint tint_array_length_0_0;
  uint tint_array_length_0_3;
  uint tint_array_length_0_2;
  uint tint_array_length_0_1;
  uint tint_array_length_0_5;
  uint tint_array_length_0_4;
};

void main_inner(uint3 local_id, uint3 invocation_size, uint3 invocation_id,
↪  tint_module_vars_struct tint_module_vars) {
  tint_array_lengths_struct const v_1 =
  ↪  tint_array_lengths_struct{.tint_array_length_0_0=((*tint_module_vars.tint_storage_b
  ↪  / 4u),
  ↪  .tint_array_length_0_3=((*tint_module_vars.tint_storage_buffer_sizes)[0u].w
  ↪  / 20u),
  ↪  .tint_array_length_0_2=((*tint_module_vars.tint_storage_buffer_sizes)[0u].z
  ↪  / 4u),
  ↪  .tint_array_length_0_1=((*tint_module_vars.tint_storage_buffer_sizes)[0u].y
  ↪  / 4u),
  ↪  .tint_array_length_0_5=((*tint_module_vars.tint_storage_buffer_sizes)[1u].y
  ↪  / 4u),
  ↪  .tint_array_length_0_4=((*tint_module_vars.tint_storage_buffer_sizes)[1u].x
  ↪  / 4u)};
  uint id = invocation_id.x;
  bool v_2 = false;
  if ((local_id.x == 0u)) {
    v_2 = (local_id.y == 0u);
  } else {
    v_2 = false;
  }
  bool v_3 = false;
  if (v_2) {
    v_3 = (invocation_id.y == 0u);
  } else {
    v_3 = false;
  }
  if (v_3) {
    device uint* const v_4 = (&(*tint_module_vars.search_info)[min(id,
    ↪  (v_1.tint_array_length_0_3 - 1u))].iteration);
    (*v_4) = ((*v_4) + 1u);
    (*tint_module_vars.search_info)[min(id, (v_1.tint_array_length_0_3 -
    ↪  1u))].last_jfq = (*tint_module_vars.search_info)[min(id,
    ↪  (v_1.tint_array_length_0_3 - 1u))].jfq_length;
  }
```

```
uint bsa_offset = (id * v_1.tint_array_length_0_0);
uint jfq_length = (*tint_module_vars.search_info)[min(id,
↪  (v_1.tint_array_length_0_3 - 1u))].jfq_length;
{
  uint2 tint_loop_idx = uint2(4294967295u);
  uint i = (local_id.x + (invocation_id.y * 128u));
  while(true) {
    if (all((tint_loop_idx == uint2(0u)))) {
      break;
    }
    if ((i < jfq_length)) {
    } else {
      break;
    }
    uint v_5 = (*tint_module_vars.jfq)[min((bsa_offset + i),
    ↪  (v_1.tint_array_length_0_2 - 1u))];
    uint start = ((*tint_module_vars.v)[min(v_5,
    ↪  (v_1.tint_array_length_0_0 - 1u))] + local_id.y);
    uint end = (*tint_module_vars.v)[min((v_5 + 1u),
    ↪  (v_1.tint_array_length_0_0 - 1u))];
    {
      uint2 tint_loop_idx_1 = uint2(4294967295u);
      while(true) {
        if (all((tint_loop_idx_1 == uint2(0u)))) {
          break;
        }
        if ((start < end)) {
        } else {
          break;
        }
        uint edge = (*tint_module_vars.e)[min(start,
        ↪  (v_1.tint_array_length_0_1 - 1u))];

        ↪  atomic_fetch_or_explicit((&(*tint_module_vars.bsak)[min((bsa_offset
        ↪  + edge), (v_1.tint_array_length_0_5 - 1u))]),
        ↪  (*tint_module_vars.bsa)[min((v_5 + bsa_offset),
        ↪  (v_1.tint_array_length_0_4 - 1u))], memory_order_relaxed);
        {
          uint const tint_low_inc_1 = (tint_loop_idx_1.x - 1u);
          tint_loop_idx_1.x = tint_low_inc_1;
          uint const tint_carry_1 = uint((tint_low_inc_1 ==
          ↪  4294967295u));
          tint_loop_idx_1.y = (tint_loop_idx_1.y - tint_carry_1);
          start = (start + 8u);
        }
        continue;
```

```
        }
      }
      {
        uint const tint_low_inc = (tint_loop_idx.x - 1u);
        tint_loop_idx.x = tint_low_inc;
        uint const tint_carry = uint((tint_low_inc == 4294967295u));
        tint_loop_idx.y = (tint_loop_idx.y - tint_carry);
        i = (i + (128u * invocation_size.y));
      }
      continue;
    }
  }
}

kernel void v_6(uint3 local_id [[thread_position_in_threadgroup]], uint3
↪  invocation_size [[threadgroups_per_grid]], uint3 invocation_id
↪  [[threadgroup_position_in_grid]], const device tint_array<uint, 1>* v
↪  [[buffer(0)]], const device tint_array<uint, 1>* e [[buffer(1)]],
↪  device tint_array<uint, 1>* jfq [[buffer(2)]], device
↪  tint_array<SearchInfo, 1>* search_info [[buffer(3)]], const device
↪  tint_array<uint, 1>* bsa [[buffer(4)]], device tint_array<atomic_uint,
↪  1>* bsak [[buffer(5)]], const constant tint_array<uint4, 2>*
↪  tint_storage_buffer_sizes [[buffer(30)]]) {
  tint_module_vars_struct const tint_module_vars =
  ↪  tint_module_vars_struct{.v=v, .e=e, .jfq=jfq,
  ↪  .search_info=search_info, .bsa=bsa, .bsak=bsak,
  ↪  .tint_storage_buffer_sizes=tint_storage_buffer_sizes};
  main_inner(local_id, invocation_size, invocation_id, tint_module_vars);
}
```

A.4 Frontier expansion WGPU

```
// language: metal1.0
#include <metal_stdlib>
#include <simd/simd.h>

using metal::uint;

struct _mslBufferSizes {
    uint size0;
    uint size1;
    uint size2;
    uint size3;
    uint size4;
    uint size5;
};
```

```
struct SearchInfo {
    uint offset;
    uint iteration;
    uint mask;
    uint jfq_length;
    uint last_jfq;
};
typedef uint type_1[1];
typedef SearchInfo type_2[1];
typedef metal::atomic_uint type_4[1];

struct main_Input {
};
kernel void main_(
  metal::uint3 local_id [[thread_position_in_threadgroup]]
, metal::uint3 invocation_size [[threadgroups_per_grid]]
, metal::uint3 invocation_id [[threadgroup_position_in_grid]]
, device type_1 const& v [[user(fake0)]]
, device type_1 const& e [[user(fake0)]]
, device type_1 const& jfq [[user(fake0)]]
, device type_2& search_info [[user(fake0)]]
, device type_1 const& bsa [[user(fake0)]]
, device type_4& bsak [[user(fake0)]]
, constant _mslBufferSizes& _buffer_sizes [[user(fake0)]]
) {
    uint id = {};
    uint bsa_offset = {};
    uint jfq_length = {};
    uint i = {};
    uint vertex_ = {};
    uint start = {};
    uint end = {};
    uint edge = {};
    id = invocation_id.x;
    if (((local_id.x == 0u) && (local_id.y == 0u)) && (invocation_id.y ==
    ↪  0u)) {
        uint _e17 = id;
        uint _e21 = search_info[_e17].iteration;
        search_info[_e17].iteration = _e21 + 1u;
        uint _e24 = id;
        uint _e28 = id;
        uint _e31 = search_info[_e28].jfq_length;
        search_info[_e24].last_jfq = _e31;
    }
    uint _e32 = id;
    bsa_offset = _e32 * (1 + (_buffer_sizes.size0 - 0 - 4) / 4);
```

```
uint _e38 = id;
uint _e41 = search_info[_e38].jfq_length;
jfq_length = _e41;
i = local_id.x + (invocation_id.y * 128u);
uint2 loop_bound = uint2(4294967295u);
bool loop_init = true;
while(true) {
    if (metal::all(loop_bound == uint2(0u))) { break; }
    loop_bound -= uint2(loop_bound.y == 0u, 1u);
    if (!loop_init) {
        uint _e99 = i;
        i = _e99 + (128u * invocation_size.y);
    }
    loop_init = false;
    uint _e49 = i;
    uint _e50 = jfq_length;
    if (_e49 < _e50) {
    } else {
        break;
    }
    {
        uint _e53 = bsa_offset;
        uint _e54 = i;
        uint _e57 = jfq[_e53 + _e54];
        vertex_ = _e57;
        uint _e60 = vertex_;
        uint _e62 = v[_e60];
        start = _e62 + local_id.y;
        uint _e67 = vertex_;
        uint _e71 = v[_e67 + 1u];
        end = _e71;
        uint2 loop_bound_1 = uint2(4294967295u);
        bool loop_init_1 = true;
        while(true) {
            if (metal::all(loop_bound_1 == uint2(0u))) { break; }
            loop_bound_1 -= uint2(loop_bound_1.y == 0u, 1u);
            if (!loop_init_1) {
                uint _e94 = start;
                start = _e94 + 8u;
            }
            loop_init_1 = false;
            uint _e73 = start;
            uint _e74 = end;
            if (_e73 < _e74) {
            } else {
                break;
            }
```

```
                {
                    uint _e77 = start;
                    uint _e79 = e[_e77];
                    edge = _e79;
                    uint _e82 = bsa_offset;
                    uint _e83 = edge;
                    uint _e87 = vertex_;
                    uint _e88 = bsa_offset;
                    uint _e91 = bsa[_e87 + _e88];
                    uint _e92 = metal::atomic_fetch_or_explicit(&bsak[_e82
                    ↪  + _e83], _e91, metal::memory_order_relaxed);
                }
            }
        }
    }
    return;
}
```

A.5 Frontier identification Dawn

```
#include <metal_stdlib>
using namespace metal;

template<typename T, size_t N>
struct tint_array {
  const constant T& operator[](size_t i) const constant { return
  ↪  elements[i]; }
  device T& operator[](size_t i) device { return elements[i]; }
  const device T& operator[](size_t i) const device { return elements[i]; }
  thread T& operator[](size_t i) thread { return elements[i]; }
  const thread T& operator[](size_t i) const thread { return elements[i]; }
  threadgroup T& operator[](size_t i) threadgroup { return elements[i]; }
  const threadgroup T& operator[](size_t i) const threadgroup { return
  ↪  elements[i]; }
  T elements[N];
};

struct SearchInfo {
  /* 0x0000 */ uint offset;
  /* 0x0004 */ uint iteration;
  /* 0x0008 */ atomic_uint mask;
  /* 0x000c */ atomic_uint jfq_length;
  /* 0x0010 */ uint last_jfq;
};

struct tint_module_vars_struct {
```

```
  device tint_array<uint, 1>* jfq;
  device tint_array<SearchInfo, 1>* search_info;
  const device tint_array<uint, 1>* dst;
  device tint_array<uint, 1>* path_length;
  const device tint_array<uint, 1>* bsa;
  device tint_array<uint, 1>* bsak;
  const constant tint_array<uint4, 2>* tint_storage_buffer_sizes;
};

struct tint_array_lengths_struct {
  uint tint_array_length_0_0;
  uint tint_array_length_0_1;
  uint tint_array_length_0_4;
  uint tint_array_length_0_5;
  uint tint_array_length_0_2;
  uint tint_array_length_0_3;
};

uint tint_div_u32(uint lhs, uint rhs) {
  return (lhs / select(rhs, 1u, (rhs == 0u)));
}

void main_inner(uint3 local_id, uint3 invocation, uint3 invocation_size,
↪  tint_module_vars_struct tint_module_vars) {
  tint_array_lengths_struct const v =
  ↪  tint_array_lengths_struct{.tint_array_length_0_0=((*tint_module_vars.tint_storage_b
  ↪  / 4u),
  ↪  .tint_array_length_0_1=((*tint_module_vars.tint_storage_buffer_sizes)[0u].y
  ↪  / 20u),
  ↪  .tint_array_length_0_4=((*tint_module_vars.tint_storage_buffer_sizes)[1u].x
  ↪  / 4u),
  ↪  .tint_array_length_0_5=((*tint_module_vars.tint_storage_buffer_sizes)[1u].y
  ↪  / 4u),
  ↪  .tint_array_length_0_2=((*tint_module_vars.tint_storage_buffer_sizes)[0u].z
  ↪  / 4u),
  ↪  .tint_array_length_0_3=((*tint_module_vars.tint_storage_buffer_sizes)[0u].w
  ↪  / 4u)};
  uint id = invocation.x;
  bool v_1 = false;
  if (((*tint_module_vars.search_info)[min(id, (v.tint_array_length_0_1 -
  ↪  1u))].iteration > 0u)) {
    v_1 = ((*tint_module_vars.search_info)[min(id, (v.tint_array_length_0_1
    ↪  - 1u))].last_jfq == 0u);
  } else {
    v_1 = false;
  }
```

```
if (v_1) {
  return;
}
uint const v_2 = id;
uint bsa_offset = (v_2 * tint_div_u32(v.tint_array_length_0_0,
↪  invocation_size.x));
uint dst_offset = (*tint_module_vars.search_info)[min(id,
↪  (v.tint_array_length_0_1 - 1u))].offset;
uint maskl =
↪  ~(atomic_load_explicit((&(*tint_module_vars.search_info)[min(id,
↪  (v.tint_array_length_0_1 - 1u))].mask), memory_order_relaxed));
uint iteration = (*tint_module_vars.search_info)[min(id,
↪  (v.tint_array_length_0_1 - 1u))].iteration;
{
  uint2 tint_loop_idx = uint2(4294967295u);
  uint i = (local_id.x + (invocation.y * 256u));
  while(true) {
    if (all((tint_loop_idx == uint2(0u)))) {
      break;
    }
    uint const v_3 = i;
    if ((v_3 < tint_div_u32(v.tint_array_length_0_0, invocation_size.x)))
    ↪  {
    } else {
      break;
    }
    uint diff = (((*tint_module_vars.bsa)[min((bsa_offset + i),
    ↪  (v.tint_array_length_0_4 - 1u))] ^
    ↪  (*tint_module_vars.bsak)[min((bsa_offset + i),
    ↪  (v.tint_array_length_0_5 - 1u))]) & maskl);
    if ((diff == 0u)) {
      {
        uint const tint_low_inc = (tint_loop_idx.x - 1u);
        tint_loop_idx.x = tint_low_inc;
        uint const tint_carry = uint((tint_low_inc == 4294967295u));
        tint_loop_idx.y = (tint_loop_idx.y - tint_carry);
        i = (i + (256u * invocation_size.y));
      }
      continue;
    }
    device uint* const v_4 = (&(*tint_module_vars.bsak)[min((bsa_offset +
    ↪  i), (v.tint_array_length_0_5 - 1u))]);
    (*v_4) = ((*v_4) | (*tint_module_vars.bsa)[min((bsa_offset + i),
    ↪  (v.tint_array_length_0_4 - 1u))]);
```

```
uint temp =
↪  atomic_fetch_add_explicit((&(*tint_module_vars.search_info)[min(id,
↪  (v.tint_array_length_0_1 - 1u))].jfq_length), 1u,
↪  memory_order_relaxed);
(*tint_module_vars.jfq)[min((bsa_offset + temp),
↪  (v.tint_array_length_0_0 - 1u))] = i;
uint length = popcount(diff);
{
  uint2 tint_loop_idx_1 = uint2(4294967295u);
  uint j = 0u;
  while(true) {
    if (all((tint_loop_idx_1 == uint2(0u)))) {
      break;
    }
    if ((j < length)) {
    } else {
      break;
    }
    uint index = ctz(diff);
    if (((*tint_module_vars.dst)[min((dst_offset + index),
    ↪  (v.tint_array_length_0_2 - 1u))] == i)) {
      (*tint_module_vars.path_length)[min((dst_offset + index),
      ↪  (v.tint_array_length_0_3 - 1u))] = iteration;

      ↪  atomic_fetch_or_explicit((&(*tint_module_vars.search_info)[min(id,
      ↪  (v.tint_array_length_0_1 - 1u))].mask), (1u << (index &
      ↪  31u)), memory_order_relaxed);
    }
    diff = (diff ^ (1u << (index & 31u)));
    {
      uint const tint_low_inc_1 = (tint_loop_idx_1.x - 1u);
      tint_loop_idx_1.x = tint_low_inc_1;
      uint const tint_carry_1 = uint((tint_low_inc_1 ==
      ↪  4294967295u));
      tint_loop_idx_1.y = (tint_loop_idx_1.y - tint_carry_1);
      j = (j + 1u);
    }
    continue;
  }
}
{
  uint const tint_low_inc = (tint_loop_idx.x - 1u);
  tint_loop_idx.x = tint_low_inc;
  uint const tint_carry = uint((tint_low_inc == 4294967295u));
  tint_loop_idx.y = (tint_loop_idx.y - tint_carry);
  i = (i + (256u * invocation_size.y));
```

```
      }
      continue;
    }
  }
}

kernel void v_5(uint3 local_id [[thread_position_in_threadgroup]], uint3
↪  invocation [[threadgroup_position_in_grid]], uint3 invocation_size
↪  [[threadgroups_per_grid]], device tint_array<uint, 1>* jfq
↪  [[buffer(0)]], device tint_array<SearchInfo, 1>* search_info
↪  [[buffer(1)]], const device tint_array<uint, 1>* dst [[buffer(2)]],
↪  device tint_array<uint, 1>* path_length [[buffer(3)]], const device
↪  tint_array<uint, 1>* bsa [[buffer(4)]], device tint_array<uint, 1>*
↪  bsak [[buffer(5)]], const constant tint_array<uint4, 2>*
↪  tint_storage_buffer_sizes [[buffer(30)]]) {
  tint_module_vars_struct const tint_module_vars =
  ↪  tint_module_vars_struct{.jfq=jfq, .search_info=search_info, .dst=dst,
  ↪  .path_length=path_length, .bsa=bsa, .bsak=bsak,
  ↪  .tint_storage_buffer_sizes=tint_storage_buffer_sizes};
  main_inner(local_id, invocation, invocation_size, tint_module_vars);
}
```

A.6 Frontier identification WGPU

```
// language: metal1.0
#include <metal_stdlib>
#include <simd/simd.h>

using metal::uint;

struct _mslBufferSizes {
    uint size0;
    uint size1;
    uint size2;
    uint size3;
    uint size4;
    uint size5;
};

struct SearchInfo {
    uint offset;
    uint iteration;
    metal::atomic_uint mask;
    metal::atomic_uint jfq_length;
    uint last_jfq;
};
```

```
typedef uint type_2[1];
typedef SearchInfo type_3[1];
uint naga_div(uint lhs, uint rhs) {
    return lhs / metal::select(rhs, 1u, rhs == 0u);
}


struct main_Input {
};
kernel void main_(
  metal::uint3 local_id [[thread_position_in_threadgroup]]
, metal::uint3 invocation [[threadgroup_position_in_grid]]
, metal::uint3 invocation_size [[threadgroups_per_grid]]
, device type_2& jfq [[user(fake0)]]
, device type_3& search_info [[user(fake0)]]
, device type_2 const& dst [[user(fake0)]]
, device type_2& path_length [[user(fake0)]]
, device type_2 const& bsa [[user(fake0)]]
, device type_2& bsak [[user(fake0)]]
, constant _mslBufferSizes& _buffer_sizes [[user(fake0)]]
) {
    uint id = {};
    uint bsa_offset = {};
    uint dst_offset = {};
    uint maskl = {};
    uint iteration = {};
    uint i = {};
    uint diff = {};
    uint temp = {};
    uint length = {};
    uint j = {};
    uint index = {};
    id = invocation.x;
    uint _e6 = id;
    uint _e9 = search_info[_e6].iteration;
    uint _e13 = id;
    uint _e16 = search_info[_e13].last_jfq;
    if ((_e9 > 0u) && (_e16 == 0u)) {
        return;
    }
    uint _e20 = id;
    bsa_offset = _e20 * naga_div(1 + (_buffer_sizes.size0 - 0 - 4) / 4,
    ↪  invocation_size.x);
    uint _e28 = id;
    uint _e31 = search_info[_e28].offset;
    dst_offset = _e31;
    uint _e34 = id;
```

```
uint _e37 = metal::atomic_load_explicit(&search_info[_e34].mask,
↪  metal::memory_order_relaxed);
maskl = ~(_e37);
uint _e41 = id;
uint _e44 = search_info[_e41].iteration;
iteration = _e44;
i = local_id.x + (invocation.y * 256u);
uint2 loop_bound = uint2(4294967295u);
bool loop_init = true;
while(true) {
    if (metal::all(loop_bound == uint2(0u))) { break; }
    loop_bound -= uint2(loop_bound.y == 0u, 1u);
    if (!loop_init) {
        uint _e147 = i;
        i = _e147 + (256u * invocation_size.y);
    }
    loop_init = false;
    uint _e52 = i;
    if (_e52 < naga_div(1 + (_buffer_sizes.size0 - 0 - 4) / 4,
    ↪  invocation_size.x)) {
    } else {
        break;
    }
    {
        uint _e59 = bsa_offset;
        uint _e60 = i;
        uint _e63 = bsa[_e59 + _e60];
        uint _e65 = bsa_offset;
        uint _e66 = i;
        uint _e69 = bsak[_e65 + _e66];
        uint _e71 = maskl;
        diff = (_e63 ^ _e69) & _e71;
        uint _e74 = diff;
        if (_e74 == 0u) {
            continue;
        }
        uint _e78 = bsa_offset;
        uint _e79 = i;
        uint _e83 = bsa_offset;
        uint _e84 = i;
        uint _e87 = bsa[_e83 + _e84];
        uint _e88 = bsak[_e78 + _e79];
        bsak[_e78 + _e79] = _e88 | _e87;
        uint _e91 = id;
        uint _e95 =
        ↪  metal::atomic_fetch_add_explicit(&search_info[_e91].jfq_length,
        ↪  1u, metal::memory_order_relaxed);
```

```
temp = _e95;
uint _e98 = bsa_offset;
uint _e99 = temp;
uint _e102 = i;
jfq[_e98 + _e99] = _e102;
uint _e103 = diff;
length = metal::popcount(_e103);
j = 0u;
uint2 loop_bound_1 = uint2(4294967295u);
bool loop_init_1 = true;
while(true) {
    if (metal::all(loop_bound_1 == uint2(0u))) { break; }
    loop_bound_1 -= uint2(loop_bound_1.y == 0u, 1u);
    if (!loop_init_1) {
        uint _e142 = j;
        j = _e142 + 1u;
    }
    loop_init_1 = false;
    uint _e108 = j;
    uint _e109 = length;
    if (_e108 < _e109) {
    } else {
        break;
    }
    {
        uint _e111 = diff;
        index = metal::ctz(_e111);
        uint _e115 = dst_offset;
        uint _e116 = index;
        uint _e119 = dst[_e115 + _e116];
        uint _e120 = i;
        if (_e119 == _e120) {
            uint _e123 = dst_offset;
            uint _e124 = index;
            uint _e127 = iteration;
            path_length[_e123 + _e124] = _e127;
            uint _e129 = id;
            uint _e133 = index;
            uint _e135 =
                ↪ metal::atomic_fetch_or_explicit(&search_info[_e129].mask,
                ↪ 1u << _e133, metal::memory_order_relaxed);
        }
        uint _e137 = index;
        uint _e139 = diff;
        diff = _e139 ^ (1u << _e137);
    }
}
```

```
        }
    }
    return;
}
```