ETH Library

Enabling GPU Acceleration for the Dandelion Serverless Executor

Bachelor Thesis

Author(s):
Smith, Jonathan

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000695089

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 504b

Systems Group, Department of Computer Science, ETH Zurich

Enabling GPU Acceleration for the Dandelion Serverless Executor

by

Jonathan P. L. Smith

Supervised by

Prof. Ana Klimovic, Tom Kuchler, Foteini Strati

July 9, 2024

Abstract

With the slowing of Moore's law, modern computing platforms are becoming increasingly heterogeneous. In particular, graphics processing units (GPUs) have found application in various general-purpose fields, for instance driving the machine learning revolution of recent years. Serverless cloud computing has, however, had trouble adjusting to this reality. Most systems in use today are still largely restricted to CPU-only execution.

At its core, this work is about extending Dandelion, a novel function-as-a-service (FaaS) platform, with the means to execute untrusted user code on GPUs. By making use of Dandelion's innovative programming model, we show that efficient and performant GPU-accelerated serverless computing is possible and can lead to significant benefits in key workloads.

Acknowledgements

I would like to start by extending my most sincere gratitude to my supervisors Prof. Ana Klimovic, Tom Kuchler, and Foteini Strati. Thank you Ana for welcoming me into the group with such remarkable friendliness. From the get-go, I felt included and heard. Thank you Tom for your committed question-answering and debugging help. Thank you Foteini for your expert insight and advice. Thank you all for being so kind and helpful. I couldn't have chosen a better team of supervisors.

Next, I would like to give my appreciation to my friends. I would like to thank everyone for their amazing support and companionship. Thank you for letting me rant to you when things weren't going well, and thank you for letting me rave when things were. I appreciate you all a lot.

Lastly, I would like to give my utmost thanks to my family for always supporting me and having my back. In particular, I would like to give special appreciation to my parents, for their never-ending love and encouragement.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Background 3
- 2.1 Graphics Processing Units 3
  - 2.1.1 GPU Vendors and Programming APIs 3
  - 2.1.2 GPU Architecture and Execution Model 4
  - 2.1.3 Isolation on GPUs 5
- 2.2 Cloud Computing 6
  - 2.2.1 Serverless 6
- 2.3 Dandelion 7
  - 2.3.1 Frontend 8
  - 2.3.2 Dispatcher 9
  - 2.3.3 Backends 9

3 Related Work 11
- 3.1 Service APIs 11
- 3.2 GPU attached VMs 11
- 3.3 Kernel-as-a-Service (KaaS) 12

4 Design and Implementation 14
- 4.1 User Workflow 14
  - 4.1.1 Limitations 16
- 4.2 Memory Management 16
  - 4.2.1 Buffer Pool 17
- 4.3 Sequential Engine 19
  - 4.3.1 Function Parsing 19

4.3.2 Runtime Life-Cycle 21
4.4 Concurrent Engine 23
4.4.1 Inter-Process Communication 24
4.4.2 Safety Discussion 25

5 Evaluation 26
5.1 Dedicated Server Design 27
5.2 Matrix/Matrix Multiplication 28
5.2.1 One Hundred Per Cent Hot 28
5.2.2 Zero Per Cent Hot 30
5.3 Inference Workload 30
5.3.1 Base Latency 31
5.3.2 Latency under Load 32
5.4 Effect of Worker Count on Concurrent Engine 33
5.5 Batched Inference 34
5.5.1 Base Latency 35
5.5.2 Latency under Load 36

6 Conclusion and Future Work 38

A Addendum on Evaluation Setup 40

Bibliography 41

Chapter 1

Introduction

Graphics processing units (GPUs) have become an integral component of modern computing systems. Through their ability to speed up certain highly parallel workloads, they have found use in a vast set of applications, especially as Moore's law has slowed down [Thompson and Spanuth, 2021]. In particular, the machine learning revolution of recent years has only been feasible due to the massive performance benefits GPUs bring.

Meanwhile, the serverless cloud computing paradigm has grown to become a central offering of all major cloud providers. With its principles of resource abstraction and transparent scaling, it promises end users the ability to develop applications more easily whilst only paying for what they use and offers cloud providers the opportunity to utilise their hardware more effectively [Schleier-Smith et al., 2021]. Despite this, many serverless systems in use today have trouble divorcing themselves from their VM-based roots, yielding high start-up latencies and largely CPU-restricted computation as a result.

Dandelion [Kuchler et al., 2023] is a novel function-as-a-service (FaaS) platform that aims to address the aforementioned shortcomings. Instead of expressing serverless functions via VMs, it uses a completely re-imagined data-flow-based programming model that cleanly separates computation from I/O. Thus, it enables the use of a different class of (lower overhead) isolation mechanisms. Additionally, it opens a gateway for easier heterogeneous serverless, which will be explored in this work in the context of a GPU backend for the Dandelion platform.

We aim to develop a system that combines the strengths of several current solutions, without being limited by their respective drawbacks. We strive to show that efficient and performant GPU-accelerated serverless is possible thanks to Dandelion's unique programming model.

This thesis is structured as follows: We begin by providing necessary background knowledge in Chapter 2, giving context on GPUs, as well as cloud computing, serverless, and a detailed account of Dandelion's principles and inner workings. In Chapter 3, we continue by describing the current state of GPU-accelerated serverless, highlighting the strengths and weaknesses of today's main approaches. Following this, in Chapter 4, we present our vision for GPU-accelerated serverless, explaining our intended user workflow and its resulting implications on our design. In particular, we expand on our approach to (GPU) memory management. Having provided all this context, we can finally present the architecture of our backend, which consists of two execution engines: a sequential and a more sophisticated concurrent engine. In Chapter 5, we examine their performance in terms of several metrics. We conclude in Chapter 6 by stating avenues for further research and giving our final thoughts.

Chapter 2

Background

2.1 Graphics Processing Units

With the advent of real-time 3D graphics in the 1990s, the need for dedicated graphics processing units (GPUs) emerged. These early accelerators were rather different from the GPUs of today, however, featuring mainly fixed-functionality circuits designed for the narrow use-case of graphics shading and rendering. In the early 2000s, initial cards with programmable shaders allowed a glimpse into the application of GPUs for general workloads, though it was not until the mid-2000s that truly general-purpose programmable GPUs entered the market, enabling their use in a plethora of fields [Owens et al., 2008].

2.1.1 GPU Vendors and Programming APIs

Today, the two major GPU vendors are NVIDIA and AMD, who enable general-purpose programming via their CUDA and ROCm/HIP APIs respectively. In practice, the programming models of both are very similar:[1] The *host* (the CPU) is responsible for managing execution on the *device* (the GPU). This mainly involves moving data between host and device and queuing *kernels* (GPU functions) for execution. The programmer may enqueue such data transfers and kernel executions on a number of independent *streams*, enabling a degree of asynchrony between non-dependent tasks [AMD, 2019b].

Under the hood, the hardware architectures of both vendors are similar as well. Nonetheless, somewhat differing nomenclature has emerged from

[1] AMD's HIPIFY tool can for instance easily translate CUDA into HIP [AMD, 2023].

each.[2] As we mainly examine AMD GPUs in this work, we will stick with AMD naming conventions unless specified otherwise.

2.1.2 GPU Architecture and Execution Model

At a conceptual level, kernels are executed on an (up-to) three-dimensional grid. Each point in the grid is called a *thread*, which can be executed independently from, and in parallel to, other threads. For example, in a matrix/matrix multiplication of the form $A \times B$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, a two-dimensional grid might be used. The thread at position (x, y) in the grid represents the entry $C_{x,y}$ in the resulting matrix $C \in \mathbb{R}^{m \times n}$, and thus needs to compute the inner product between row x of A and column y of B, which can be done in parallel to any other thread. Figure 2.1 gives a graphical illustration of this.

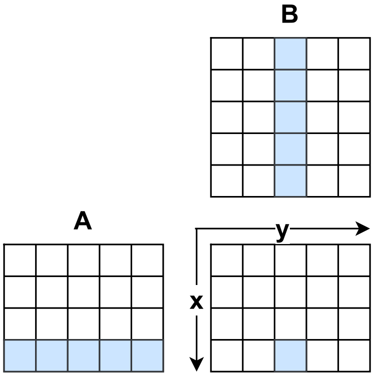


Figure 2.1: Example matrix/matrix multiplication: Each thread is one point in the resulting matrix and operates on the filled data respectively. Note the axis directions.

Threads are grouped together to form *blocks*, which are ultimately assigned to a *compute unit* (CU) (NVIDIA: *streaming multiprocessor*) for execution.[3] A CU is in effect one of the GPU's processors, featuring its own program counter, registers, ALUs, etc. Sets of threads running the same instruction (but operating on different data) are batched together by a CU to form a *wavefront* (NVIDIA: *warp*) for simultaneous execution. As such, CUs are in essence SIMD processors, except that the SIMD instructions are dynamically formed by the device [Luna and Mutlu, 2022]. As massively parallel

[2] Sometimes quite confusingly so: AMD calls its scratchpad memory *local memory* (NVIDIA calls it *shared memory*), whereas NVIDIA calls its per-thread private memory *local memory* (AMD calls it *private memory*).

[3] Note that CUs can typically handle more than one block at a time.

devices, modern GPUs consist of a large number (100+) of such compute units.

Along with their high degree of data parallelism, GPUs feature a multi-level memory hierarchy ranging from modest per-thread private memory, to per-block local memory, all the way to (cached) global memory along with massive memory bandwidths. In comparison, the transfer rate of the CPU/GPU interconnect is usually much lower [PCISIG, 2021]. As such, care must be taken by the programmer to ensure that these expensive transfers happen as little as possible.

Listing 2.1 shows the HIP code for our example matrix/matrix multiplication. Note that the programmer does not need to explicitly define how parallelism is achieved—it much rather results from the hardware taking advantage of the threads' inherent parallelism.

```
__global__ void matmul(float *C, float *A, float *B, int m, int n, int k) {
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    int col = blockDim.y * blockIdx.y + threadIdx.y;

    if (row < m && col < n) {
        float sum = 0.0f;
        for (int i = 0; i < k; i++) {
            sum += A[row * k + i] * B[i * n + col];
        }
        C[row * n + col] = sum;
    }
}
```

Listing 2.1: HIP code for matrix/matrix multiplication as seen in Figure 2.1.

2.1.3 Isolation on GPUs

Given that our end goal is to run untrusted GPU functions in our system, we must take steps to adequately isolate them—not only from each other but also from the system itself. In the following section, we will hence discuss the degree of isolation present in current AMD GPUs, mentioning similar mechanisms offered by NVIDIA when relevant.

Concerning device/host isolation: It holds that the device cannot access host memory which the host has not made available to it, nor can a kernel allocate host memory. Kernels may, however, call `malloc` to allocate global device memory, though the host can disable this by setting `hipSetDeviceLimit(hipLimitMallocHeapSize, 0)`.[4]

[4] Though calling `malloc` after having set this in current ROCm versions causes a rather unceremonious crash, see https://github.com/ROCm/HIP/issues/3429.

Kernels launched from the same process all access the same global memory. Since each process has its own virtual address space, processes can be used to isolate kernels from another.[5] Concurrently running processes on the GPU is supported by AMD's driver natively and can be enabled on NVIDIA devices by using their *multi-process service* (MPS) [NVIDIA, 2024]. In addition to this, recent NVIDIA GPUs can be spatially partitioned using their *multi-instance GPU* (MIG) technology [NVIDIA, 2020]. AMD and NVIDIA also offer virtualisation of their accelerators for use in VMs via their MxGPU and vGPU technologies, respectively [NVIDIA, 2020, AMD, 2017].

2.2 Cloud Computing

Cloud computing in its current sense has its beginnings in the early 2000s. Initial offerings were designed following the *Infrastructure-as-a-Service* (IaaS) paradigm, wherein the user is provided with a virtual machine by the cloud provider. While this freed customers from having to manage on-premises (local) hardware, they were ultimately still required to manage individual servers in the cloud. As such, this model is sometimes also referred to as *serverful cloud computing* [Schleier-Smith et al., 2021].

2.2.1 Serverless

Serverless cloud computing, sometimes succinctly referred to as just *serverless*, on the other hand, represents a rethinking of cloud computing that has gained significant traction in the past decade. Serverless is provided to the user via *backend-as-a-service* (BaaS), which provides object storage, queues, databases, etc., and *function-as-a-service* (FaaS), which provides the actual function execution. [Schleier-Smith et al., 2021] lays out three principles at its core:

Resource Abstraction. Servers are completely abstracted away from the customer. Instead, they are able to focus entirely on their application's relevant logic. Operating the hardware becomes the full responsibility of the service provider.

Pay-as-you-go. Customers are only billed for the resources they actively consume. In comparison, serverful is billed based on reserved resources and hence often charges for idle hardware.

Transparent scaling. Akin to resource abstraction, the cloud provider assumes the duty of scaling the services deployed on it fully automat-

[5] It's worth noting that a vulnerability [Sorensen and Khlaaf, 2024] that facilitates the leaking of local GPU memory across processes has recently been discovered. A large number of manufacturers, including AMD, are affected.

ically to match demand. As a result, the customer need not concern themselves with scaling at all.

In theory, both customer and provider benefit from serverless: The cloud provider can exercise more control over their resources, thus enabling them to potentially increase overall utilisation. The customer benefits from the abstracted infrastructure and simple deployment. Despite these clear advantages, several challenges remain unsolved in current platforms.

Typical FaaS functions are very short-lived, with execution times in the 100s of milliseconds (and increasingly less) being common [Shahrad et al., 2020]. This poses a significant hurdle for the virtual-machine-based systems used by the major providers today, as the overhead of starting up the virtual machine may in some cases dominate the actual computation time. Significant research has thus gone into decreasing start-up latencies. For instance, Amazon's Firecracker [Agache et al., 2020] uses a custom virtual-machine monitor (VMM) and features support for *snapshotting* (capturing the state of a VM for quicker start-up later). Nevertheless, the difference between *cold start* (the function has not been recently called) and *hot start* (the function has been recently called) remains large [Ustiugov et al., 2021].

2.3 Dandelion

Dandelion [Kuchler et al., 2023] is a novel FaaS executor which attempts to alleviate many of the issues current platforms are facing through a fundamental redesign of the programming model. Instead of using VMs, in Dandelion, programs are expressed as *compositions* of *functions* that perform either I/O (and are provided by the platform) or pure, i.e. side-effect free, computation (and are provided by the user). As compute functions are side-effect-free, they are unable to impact the overall system's state. To accomplish this in practice, compute functions are for example prevented from making system calls.

In other words: Compositions are data-flow graphs where all nodes (the functions) perform either compute or I/O. Consider the example in Figure 2.2—the green nodes represent compute functions whereas the blue node performs I/O. The bottom node could benefit greatly from GPU acceleration.

Since user code is not able to as easily escape its sandbox due to the pure computation, it may be run with less intrusive (and thus lower-overhead) isolation mechanisms, vastly improving startup latencies. Further, as the system is conscious of a composition's data flow, data-aware scheduling and caching are more easily enabled. Finally, Dandelion is highly amenable to hardware acceleration—each node may run on a different backend and

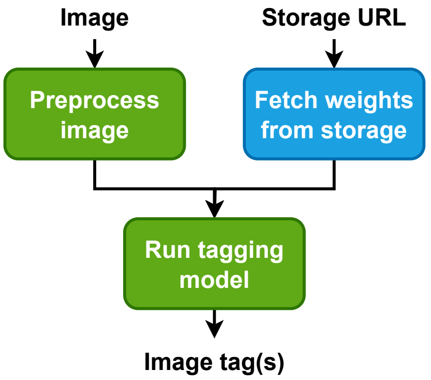


Figure 2.2: Example Dandelion composition for photo tagging using ML inference.

hence accelerator. Current backends based on MMU isolation, CHERI memory capabilities, and WASM [Thomm, 2024] already exist, with an FPGA backend in active development.

Under the hood, Dandelion is implemented in roughly 12k lines of Rust.[6] The system consists of three main components: the *frontend*, the *dispatcher*, and the *backends/engines*. Figure 2.3 contains an overview of the core Dandelion architecture.

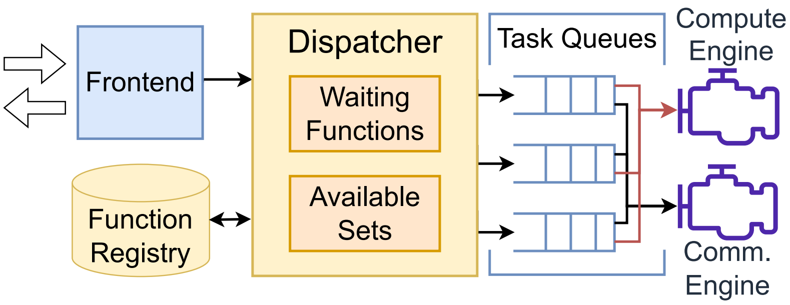


Figure 2.3: Dandelion system architecture.

2.3.1 Frontend

The Dandelion frontend is the smallest of the three major components. Its purpose is to handle communication with clients via HTTP. The frontend accepts requests, determines their type, enqueues them in the dispatcher for execution, and returns a response once fulfilled. Currently, requests either register functions/compositions, or invoke compositions (providing the input data necessary to do so).

[6] This includes our contribution.

2.3.2 Dispatcher

The dispatcher can be thought of as the brains of the Dandelion system—it exercises full control over all resources and executions in the entire system. Additionally, it maintains state over the currently available functions and compositions in the *registry*, storing metadata such as locations of binaries on disk, input/output requirements, and which functions run on which backends.

When a composition is invoked, the dispatcher keeps track of all data, ordering data transfers as necessary and invoking functions once their inputs become available.

2.3.3 Backends

Backends enable the actual execution of work by exposing *engines*, which process individual requests. Recall from Section 2.3 that backends may utilise a variety of hardware platforms and employ different isolation mechanisms. As a result, their implementations also vary substantially, only needing to communicate with the dispatcher via a narrow interface. Nonetheless, each backend type has an associated *driver* and *memory domain*, and spawns *engines*, each operating on many *contexts*.

Memory Domains and Contexts. A context is in essence the address space of a function invocation. As user functions are untrusted, a given function should never escape its context to ensure isolation in the system. Memory domains can be thought of as "context factories", providing fresh contexts when the dispatcher requests them. A context always belongs to, and is acquired via, its respective memory domain. Furthermore, memory domains implement data transfers between contexts (belonging to potentially different memory domains), utilising more efficient means wherever possible. For example, if both contexts reside in the same address space, a simple `memcpy()` can be used.

Drivers and Engines. Engines take on the principal load of executing function invocations. To this end, they retrieve invocation requests from a work queue provided by the dispatcher, obtaining a *debt* that is fulfilled once the desired computation has been completed. A particular backend might spawn many engines. On the other hand, each backend has only one associated driver. Much like the relationship between memory domains and contexts, drivers can be seen as a sort of "engine factory", starting up new engines as needed. In addition, drivers provide the means to parse newly registered functions, which involves preparing a static context and configuration that is used to instantiate the runtime state for every subsequent invocation.

Figure 2.4 gives an overview of the components of a backend.

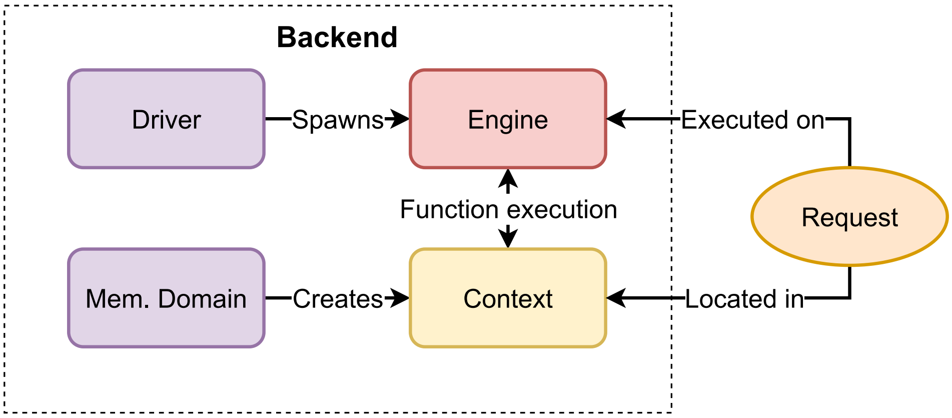


Figure 2.4: Components of a Dandelion backend and their relationships.

Chapter 3

Related Work

Since the inception of serverless, there has been a demand to extend it to enable GPU acceleration. Due to the challenges associated with pairing dedicated accelerators with short-lived functions that are often resident in VMs, several solutions and workarounds have been developed.

3.1 Service APIs

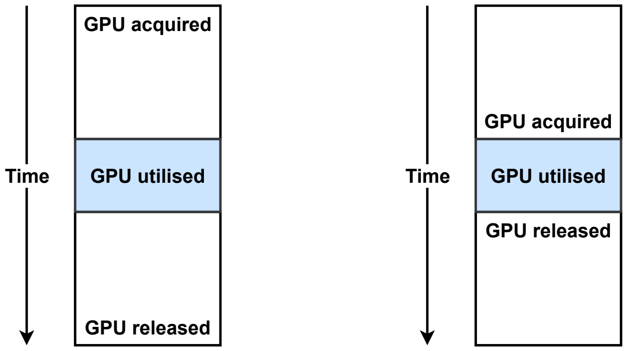
Rather than to directly provide functions with an accelerator, a popular workaround employed by many cloud providers is to instead wrap hardware acceleration in a service that can be called from serverless functions. Many such services have been developed, in particular for single-domain tasks, e.g. machine learning inference [Romero et al., 2021, Olston et al., 2017, BentoML, 2019]. At this point, it's also worth mentioning Azure's general-purpose Catapult service for FPGAs [Putnam et al., 2015].

At first sight, service APIs may appear a reasonable compromise. However, these systems come with a number of drawbacks. Firstly, they largely lack the ability to perform arbitrary computation. Even though many use cases may be captured, they thus lack the type of generality we are aiming for with this project. Secondly, they introduce non-trivial and unpredictable overheads by delegating computation to a remote service for execution. Furthermore, they massively limit the issuing system's control, as computation occurs opaquely to it. Thus, they are unsuitable for systems with a powerful scheduler, like Dandelion.

3.2 GPU attached VMs

Since most current FaaS platforms perform their computation inside virtual machines, it seems a natural step to simply attach a GPU to the VM/container running the function. [Kim et al., 2018] provides an initial approach based on NVIDIA-Docker, which was iterated upon by [Naranjo et al., 2020, Satzke et al., 2021] by employing different GPU virtualisation techniques.

Nevertheless, the key shortcoming of this design remains unsolved—a GPU (or virtualised version thereof) is tied to a container for its entire lifetime, even if it is only needed for a much shorter duration. This inherently limits the scalability of the platform, and severely underutilises GPU resources, even if virtualisation helps to some extent. See Figure 3.1 for a visualisation; it shows the lifetime of an example serverless function, as well as the points in time when the GPU is required, compared to when it is acquired and released.



(a) With GPU attached VMs. **(b)** Desired.

Figure 3.1: GPU lifetimes with different approaches.

In addition, acquiring a GPU on container startup adds significant latency, further compounding the issue of startup times VM-based systems are already plagued by.

3.3 Kernel-as-a-Service (KaaS)

[Pemberton et al., 2022] presents an entirely different approach. Here, GPU tasks are represented as directed graphs of GPU kernels. Instead of directly giving the platform host code to run, the user merely provides a library of GPU kernels and specifies an execution graph. Thus, all host code can be managed by the KaaS executor.

Rather than needing to keep GPUs around for an entire program's lifetime, they now only need to be reserved during a GPU task's execution[1], significantly improving performance. Further, as the executor maintains closer

[1] Programs can consist of many tasks.

control over the accelerator, novel caching techniques can be applied. This approach is implemented in a prototype for the Ray [Moritz et al., 2018] distributed computing framework.

As will become apparent, this work significantly influences our final design. Nonetheless, the paper omits discussion of key isolation questions. In Dandelion, we assume the presence of adversarial actors and thus must take steps to ensure honest users are not interfered with.

Chapter 4

Design and Implementation

In this chapter, we will present the design of our system, along with considerations made during the development process. We begin by describing our envisioned user workflow and its implications for our design, followed by our policy on memory management. Then, we give implementation details about our sequential and concurrent engine.

4.1 User Workflow

Similar to [Pemberton et al., 2022], we envision a design where a GPU function consists of a pre-compiled library of GPU kernels and a description of how they are to be invoked. More specifically, we accept kernels in the form of an `.hsaco` library, which is AMD's binary code format. The configuration is provided as a `.json` file which specifies the location of the library on disk, the relevant kernels to load from it, as well as the *blueprint*, which contains information about inputs/outputs, additional/temporary buffers, and the control flow. A single function can thus invoke many kernels. In addition, the user is given a lot of control, being able to set the exact compute grid dimensions and dynamic local memory for each kernel launch. See Listing 4.1 for an example configuration file.

Not only does this design give the user a tremendous amount of freedom, it also provides us two key benefits:

1. No user host code. As the user only supplies device kernels, there is no need for us to run any untrusted host code. This simplifies our setup greatly, as we hence require no isolation on the host side. Furthermore, the system can assume full responsibility for memory management and own all of device memory. Therefore, it can persist GPU memory allocations between different invocations, avoiding expensive calls to `hipMalloc()` on the

```
{
    "module_path": "<path_to_lib>.hsaco",
    "kernels": ["matmul"],
    "blueprint": {
        "inputs": ["cfg", "A"],
        "buffers": {"B": {"Sizeof": "A"}},
        "outputs": ["B"],
        "control_flow": [
            {"ExecKernel": ["matmul", [{"Ptr": "A"}, {"Sizeof": "A"}, {"Ptr": "B"}],
            {
                "grid_dim_x": {"FromInput": {"bufname": "cfg", "idx": 0}},
                "grid_dim_y": {"FromInput": {"bufname": "cfg", "idx": 0}},
                "grid_dim_z": {"Absolute": 1},
                "block_dim_x": {"Absolute": 32},
                "block_dim_y": {"Absolute": 32},
                "block_dim_z": {"Absolute": 1},
                "shared_mem_bytes": {"Absolute": 0}
            }]}
        ]
    }
}
```

Listing 4.1: Example configuration for a matrix/matrix multiplication

hot path.[1] Further, recall from Section 2.1.3 that kernels may be prevented from allocating additional memory by using `hipSetDeviceLimit()`,[2] thus preventing kernels from leaking memory.

2. More substantial functions. Many kernels are short-lived. In the workloads we are targeting (ML inference, image processing, video encoding, etc.), typical runtimes are in tens to hundreds of microseconds, see Figure 10 of [Han et al., 2022]. On top of this, useful programs tend to consist of a large number of kernels. For example, a standard CNN requires a large number of convolution, ReLU, max-pooling, etc. kernel invocations to compute a forward pass. Since a natural data dependency exists between such kernels, we would be foolish not to take advantage of it.

Recall from 2.1.1 that transferring data onto/off the GPU is a very costly operation. By allowing a control flow of many kernels, we must only move data onto and off the device once, significantly reducing the overhead compared to an approach where one function consists of only one kernel. We can similarly amortise other overheads, such as preparing the function context, transferring inputs, and returning outputs.

[1] Although this means effort must be put into maintaining isolation between invocations.

[2] Of course, this means kernels that dynamically allocate memory themselves cannot run, but this feature is rarely ever used in practice anyhow.

4.1.1 Limitations

While our design does yield a number of practical benefits, there are always trade-offs to be made when selecting an approach, and our design is no different in this regard.

Streams. Recalling Section 2.1.1, streams allow programmers to interleave non-dependent computation and transfers. For simplicity's sake, we have omitted integrating streams into our design, although they provide an excellent opportunity for future work. Instead, we launch all kernels and transfers on the implicit default stream. Our justification for this is that many of the workloads we are targeting feature a fairly linear data flow (e.g. ML inference), limiting the usefulness of streams. Furthermore, any program with many streams can be serialised to use only a single stream. We thus lose no expressiveness by opting for a single-stream approach.

Multiple devices. While our system is compatible with multiple GPUs, we restrict each invocation to using a singular GPU. Our motivation for this is simple: managing, and moving data between multiple GPUs involves significant overhead that we deem not worth it for the short-lived workloads we are targeting. We consider longer-lived workloads that require the parallelism offered by multiple accelerators to be more suited to traditional cloud computing offerings.

4.2 Memory Management

Each GPU features its own address space. Because current engines all run on the CPU and thus only use a single address space, managing multiple memory spaces is a novel challenge in the Dandelion ecosystem. One option could be to use unified memory (UM) which exposes one shared address space between CPU and GPUs, but the performance is typically not equivalent to other, more explicit, methods, see [Jin and Vetter, 2022]. We therefore opt not to use it.

The question becomes how to fit this reality into the abstraction of memory domains and contexts outlined in Section 2.3.3. We select the following approach: Each engine takes full ownership of one GPU and consequently owns and manages all of that GPU's memory. When a function is invoked, we use a Dandelion context to receive inputs and static data from the dispatcher in the CPU's address space. Before kernel execution begins, we copy relevant inputs into GPU memory. Similarly, we copy outputs from device memory into the context to make them available to the system. To some degree this breaks the notion that a context is a function's address space—we rather use it as a means of communication with the dispatcher. A function's true address space is GPU memory.

Our approach is motivated by three primary factors:

1. GPU memory cannot be oversubscribed (swapped out).[3]
2. Current ROCm versions do not support sharing GPU memory between processes as the allocation of virtual memory is tied to the allocation of physical memory.[4]
3. Allocating GPU memory is very expensive.

Because GPU memory is rather modestly sized (64GB vs. 500GB on our machine) and many functions may expect to be able to use most of device memory, we want to avoid keeping long-lived data in GPU memory unnecessarily.

Recall from Section 2.3.3 that function parsing produces a static context which is used to instantiate the (runtime) context of each subsequent request. We elaborate on our specific function parsing logic in Section 4.3.1. For the argument now, it's important to know a GPU function's static context contains the library of kernels. As these libraries can become quite large, it is thus not an option for us to keep (static) contexts directly in device memory. Even if we were able to keep static contexts directly in GPU memory without a penalty, we would need to make sure that malicious kernels couldn't corrupt them. An option could be to isolate them using processes, but sharing contexts with them would then become difficult due to point 2. Similarly, functions would then need to perform costly memory allocations on the hot path, which we want to avoid due to point 3. Furthermore, in a multi-GPU setting like ours, data would need to be replicated across accelerators.

4.2.1 Buffer Pool

Following the principles above, we next describe the architecture of our buffer pool, which is used to make the GPU buffers requested in a configuration file available to its kernels. We design the buffer pool with two goals in mind:

1. Acquiring buffers should be fast.
2. An honest function's data should never be leaked.

Recall that as our engine takes full ownership of device memory, we can persist allocations between function invocations, so long as no information is leaked. In practice, the buffer pool allocates (almost) the entirety

[3] This is possible with UM and AMD's XNACK technology for page migration, but it is disabled on our machine.

[4] NVIDIA offers an API that separates physical/virtual memory allocation.

of GPU memory as a contiguous chunk on engine startup and then makes buffers available as an offset from the base plus a length—see Listing 4.2 for the `struct` definitions. On function exit, all buffers are simply zeroed out. Thus, no data can be leaked for a function that accesses strictly the memory regions provided to it.

```
struct Buffer {
    offset: usize,
    length: usize,
}

pub struct BufferPool {
    allocation: DeviceAllocation,
    buffers: Vec<Buffer>,
}
```

Listing 4.2: Definition of the `Buffer` and `BufferPool` `struct`s

As kernels are prohibited from allocating global memory on the fly, all memory allocation occurs before the first kernel is launched. In a similar vein, deallocation occurs after the last kernel has finished execution. Our buffer pool will hence never have to deal with fragmentation. We can simply iterate over all requested buffers and allocate them in order, appending a new buffer behind the end of the last one. See Figure 4.1 for a visualisation. This simple design is highly efficient, offering a minuscule (and $\mathcal{O}(1)$) allocation time compared to an actual GPU memory allocation. In addition, deallocation is also very quick, as memory can be zeroed out in the contiguous region from the start of the first until the end of the last buffer using a single API call.

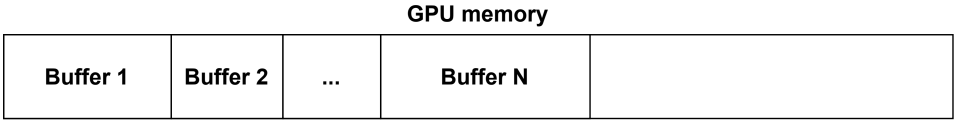


Figure 4.1: Layout of buffers in GPU memory.

Limitations. While this design shines through its simplicity, it currently ignores whether or not two buffers are live at once[5] and might be able to occupy the same memory. This may lead to situations where an invocation throws an out-of-memory error even if the memory capacity is theoretically sufficient. However, standard techniques from the field of compilers, like

[5] We say two buffers are *live at once* if there exists a point in time at which both hold values on which future computation is dependent.

register liveness analysis, could be applied here to alleviate this issue in future work (this problem can be modelled using graph colouring). Until then, the issue can also be circumvented if developers explicitly re-use temporary buffers themselves (such that all buffers are always live).

Implementation Approach

The next two sections, 4.3 and 4.4, discuss the implementation of our sequential and concurrent engines, respectively. As will become apparent, we in essence develop one backend that is able to spawn two different types of engines. As the engines belong to the same backend, other parts, such as function parsing, memory domain, and contexts, are entirely identical for both. Recall the purpose of these components from Section 2.3. We deliberately choose this structure—ultimately, we believe the dispatcher should decide which engine a function should run on, not the user.

Our entire implementation contribution, i.e. the backend consisting of both engines, helper files, tests, etc., consists of around 2.5k lines of Rust code. Recall that all of Dandelion is implemented in around 12k lines. In order to interface with the GPUs, we use the HIP runtime, which is written in C/C++. We therefore create custom (safe)[6] Rust bindings for use in Dandelion.

Apart from Dandelion itself, our trusted computing base (TCB) therefore contains the HIP runtime library, the Rust compiler, the Linux kernel, and the AMD GPU driver/firmware.

4.3 Sequential Engine

We continue by presenting the first of the two engines. Every sequential engine takes full ownership of one GPU, on which it executes requests one at a time in a thread. For this reason, we internally refer to it as `gpu_thread`.

4.3.1 Function Parsing

Although a GPU function consists of a library and a configuration file, we take the approach that they need not be registered together at once. Instead, uploading libraries occurs independently of function registration, in which only configuration files are uploaded. This enables the re-use of library code objects, reducing redundancy on disk and enabling the caching of commonly used libraries. A future iteration of the system could for ex-

[6] When we say safe here, we mean it in the Rust sense—calls to the library need not be wrapped in `unsafe` and memory leakage is prevented via the `Drop` trait.

ample keep commonly used libraries directly on the device to reduce load times.

Upon function registration, much like any other driver, we build a static configuration and context for the function. In our case, this involves two major steps:

1. Deserialising the provided configuration file into an internal Rust representation.
2. Acquiring a static context and loading the library file into it.

A code snippet of the function parsing code can be seen in Listing 4.3. Our context is implemented via `mmap()`-ed shared memory, which can thus be shared between processes. The necessity of this will become apparent when we discuss our concurrent engine in Section 4.4.

```rust
// Read config file from disk, deserialise, return Rust representation and extracted module (code object) path
let (mut gpu_config, module_path) = config_parsing::parse_config(&function_path)?;

// Read code object from disk
let code_object = load_u8_from_file(module_path)?;
let size = code_object.len() * size_of::<u8>();
gpu_config.code_object_offset = SYSDATA_OFFSET + std::mem::size_of::<DandelionSystemData<usize, usize>>();

// Write code object into static context
let mut context = static_domain.acquire_context(size)?;
context.write(0, &code_object)?;

// [... Instantiate returned Function object ...]
```

Listing 4.3: Snippet of function parsing code.

Note that this static configuration is not sufficient to execute kernels at runtime, as it performs no heap allocations and moves no code into device memory. Instead, a static configuration is loaded to create a runtime configuration upon invocation. Listing 4.4 contains a snippet of the respective code.

```
// Load module from context
let module = hip::module_load_data(base.wrapping_add(self.code_object_offset) as *
    const c_void)?;

// Get functions in loaded module
let kernels: HashMap<String, hip::FunctionT> = self
    .kernels
    .iter()
    .map(|kname| {
        hip::module_get_function(&module, kname).map(|module| (kname.clone(), module)
    )
    })
    .collect::<Result<HashMap<_, _>, _>>()?;

// [... Return RuntimeGpuConfig ...]
```

Listing 4.4: Snippet of runtime configuration loading. `self` is a static configuration, `base` points to the beginning of the context

4.3.2 Runtime Life-Cycle

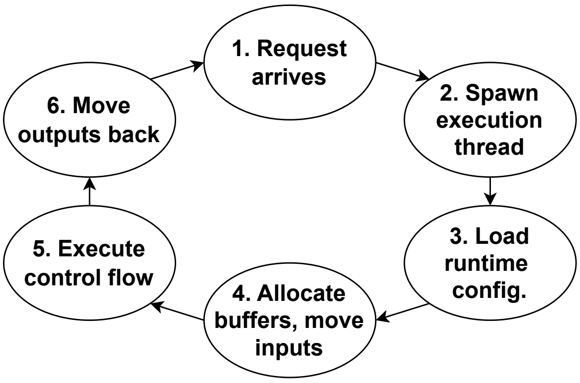


Figure 4.2: Life-cycle of the sequential backend.

As can also be seen in Figure 4.2, the sequential engine performs the following steps in order per request:

1. Busy loop on the work queue until a new request arrives.
2. Hand over control to a new thread for execution.
3. Load the runtime configuration.
4. Allocate buffers and move inputs into device memory.
5. Execute the control flow.
6. Write outputs back to the host, free allocated buffers.

Busy looping. During the development of Dandelion it was found that using other approaches for awaiting data (such as asynchronous runtimes) tended to bottleneck the system's maximum throughput. Thus, as the engine's CPU is idle anyway when no work is available, we opted for the busy looping approach.

Execution Thread. Functions are executed in a new thread in order to handle possible runtime faults (e.g. dereferencing `nullptr` in a kernel). A handler for such an event can be registered using the driver-level API's `hsa_amd_register_system_event_handler()`, which would then need to shoot down the main execution thread.

Control Flow. Executing the control flow is a fairly straightforward task. Currently, control flow consists of either a (finite) repetition of other control flow or a kernel execution. Hence, we can simply iterate over the control-flow graph, recursively executing repetitions. For now, we opt against adding more sophisticated control flow, e.g. branching or unbounded looping, for simplicity's sake and to prevent users from inadvertently causing an infinite loop, although this can easily be added in future work.

Right now, we support kernel arguments that are pointers (to GPU buffers), sizes of buffers in bytes (as an unsigned 64-bit integer), or signed 64-bit integer immediates. We deem this to capture the vast majority of all kernel arguments, though more can easily be added if the need arises. HIP expects arguments as an array of `void` pointers, so care must be taken that they point to valid memory. We achieve this by copying stack-located arguments into a heap-allocated `Vec` (dynamic array) that lives until the end of the kernel's invocation.

Listing 4.5 shows a snippet of the control-flow execution procedure.

```
// [... Function declaration ...]
for action in actions {
    match action {
        Action::ExecKernel(name, args, launch_config) => {
            // HIP expects arguments as an array of void pointers
            let mut params: Vec<*const c_void> = Vec::with_capacity(args.len());
            // [ ... Allocate storage for arguments here so pointers are valid ... ]
            for arg in args {
                match arg {
                    Argument::Ptr(bufname) => {
                        // [ ... ]
                    }
                    Argument::Sizeof(bufname) => {
                        // [ ... ]
                    }
                    Argument::Constant(constant) => {
                        // [ ... ]
                    }
                };
            }

            // Launch kernel using custom HIP bindings
            hip::module_launch_kernel(
                config.kernels.get(name).unwrap(),
                get_size(&launch_config.grid_dim_x, buffers, context)? as u32,
                // [... Get sizes of other dimensions ...]
                get_size(&launch_config.shared_mem_bytes, buffers, context)?,
                DEFAULT_STREAM,
                params.as_ptr(),
                null(),
            )?;
        }
        Action::Repeat(times, actions) => {
            let repetitions = get_size(times, buffers, context)?;
            for _ in 0..repetitions {
                execute(actions, buffers, buffer_pool, context, config)?;
            }
        }
    }
}
```

Listing 4.5: Snippet of control-flow execution procedure.

4.4 Concurrent Engine

While the sequential engine provides a good first step, room for improvement is still left. For instance, while a CPU/GPU data transfer is taking place or while host code is being executed, the GPU's compute units (CUs) are left unused. Similarly, many functions do not utilise all of the GPU's CUs. The concurrent engine intends to increase accelerator utilisation and therefore ultimately overall throughput.

An engine once again takes full ownership of one GPU, only this time it uses multiple worker processes to make progress on different requests simultaneously. For this reason, we internally refer to the concurrent engine as gpu_process. Recall from Section 2.1.3 that processes can be used to concurrently execute kernels on the GPU while maintaining isolation; something that is not possible by simply using threads, for example.

Figure 4.3 visualises the architecture of the concurrent engine. Each worker process has an associated engine thread in the main Dandelion process. Similar to gpu_thread, these engine threads all busy loop on the work queue until they obtain a request. While CPU-only requests (like transfers) are fulfilled directly in the engine thread, function invocations are serialised and dispatched to the respective worker process. In other words, an invocation is handled by exactly one worker. Upon receiving an invocation request, the worker deserialises it and executes it much like gpu_thread. Once a result

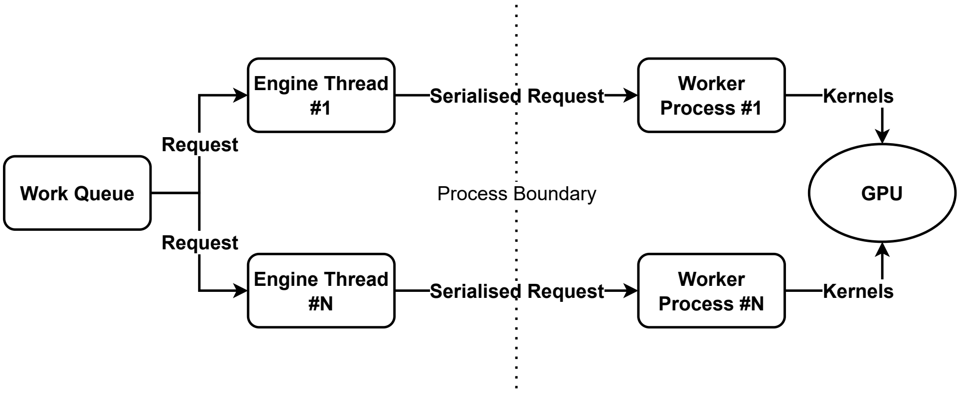


Figure 4.3: Architecture of the concurrent backend.

is produced, the respective engine thread is informed and can then fulfil its debt to the dispatcher.

Number of workers. When it comes to the number of worker processes, there is an inherent tradeoff. Recall from Section 4.2 that GPU memory cannot be overcommitted. Thus, when dividing memory up evenly among N workers, each may use only $\frac{1}{N}$ of the total memory capacity. Furthermore, while compute queues on the GPU may be oversubscribed, this can cause significant performance degradation [AMD, 2019a]. On the other hand, more worker processes allow for more concurrency.

While we initially tried four worker processes, we ultimately opted to use only two, though this is something that can easily and dynamically be adapted on engine startup. See Section 5.4 for a more detailed performance analysis.

4.4.1 Inter-Process Communication

An engine thread and its associated worker process exchange messages via piped standard I/O.

Fundamentally, all that is needed to execute a function is its context and its configuration.[7] While the configuration can be directly serialised, the context is a bit trickier. In addition to some metadata (e.g. which inputs/output exist, where data is spread, etc.) that can easily be serialised, a context holds a handle to process-local memory. We therefore obviously cannot directly share it with other processes. It is for this reason that we opted to choose a context backed by shared memory—we can simply share the filename with

[7] In practice the names of the outputs are also provided as an additional argument, but this could be part of the configuration as well.

the worker process which can then locally `mmap()` the memory. Figure 4.4 illustrates this further.

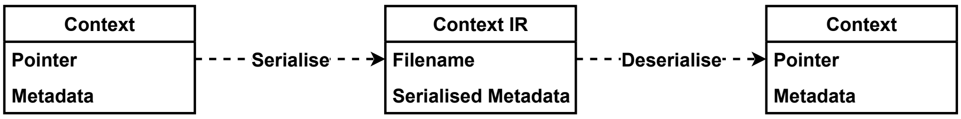


Figure 4.4: Sharing contexts with worker processes.

Returning results on the other hand is very simple, as the engine thread already has a handle on the shared memory. All it therefore needs to do is update its metadata, which it can simply do by scanning over the context.

4.4.2 Safety Discussion

Local Memory. Kernels are able to manage local memory[8] without intervention from the host. As a result, the kernel itself must take care to zero any local memory it allocates before exiting to prevent data leakage. The same goes for private memory (often mapped to registers), although both could feasibly be integrated into the (open-source) AMD driver.

LeftOverLocals. LeftOverLocals [Sorensen and Khlaaf, 2024] is a recent vulnerability affecting (among others) AMD GPUs, allowing the leakage of local memory between concurrent processes. A driver update that addresses the vulnerability is scheduled for release in July 2024 [AMD, 2024]. In order to prevent leakage, it will add a mode preventing processes from running in parallel on GPUs and clear local memory between processes. We expect this new mode to therefore decrease the performance of `gpu_process`. Our current system thus showcases the performance of (potentially future) hardware that is not affected by the exploit.[9]

[8] When we say *local memory* here, we mean it in the AMD sense—that is the per-block memory. NVIDIA would call this *shared memory*.

[9] NVIDIA accelerators are not affected, for example.

Chapter 5

Evaluation

In the following chapter, we will assess the performance of our engines relative to each other, as well as to a non-GPU-accelerated Dandelion engine and a GPU-accelerated bare-metal server.

Hardware setup. For our evaluation, we use a HACC box from the ETHZ-HACC cluster [Moya et al., 2023], featuring two AMD EPYC 7V13 64-core CPUs clocked at 2.45 GHz, 500GB RAM, and four AMD Instinct MI210 GPUs with 64GB VRAM each, running Ubuntu 20.04 LTS and ROCm 5.7.1. A separate 16-core machine generates load via a 100Gb/s link.

Server setup. We compare the following engines/servers with the specified setup:

1. `dedicated_gpu`: The GPU-accelerated bare-metal server, whose architecture is expanded upon in Section 5.1.
2. `dandelion_process`: A previously developed CPU-only Dandelion backend that isolates functions by executing them in a designated process. Each engine executes requests sequentially and requires one CPU core. We run the Dandelion server with two frontend cores and one dispatcher core.
3. `dandelion_gpu_thread`: The sequential GPU engine described in Section 4.3. We again run the server with two frontend cores and one dispatcher core. We use all four GPUs.
4. `dandelion_gpu_process`: The concurrent GPU engine described in Section 4.4. We choose the same server setup as previously.

Additional Information. Supplementary details about the evaluation setup, such as the exact git hashes used for each experiment can be found in Appendix A.

Technical Terms. We briefly define the terminology we will use in the following experiments' analyses.

Requests-per-second (RPS). The number of requests made to the server in a given second. We sometimes also refer to RPS as *load*.

Latency. The time elapsed between a given request being sent and its result being returned.

(Maximum) Throughput. The maximum number of requests the server can service (per second) without latency spiking or requests dropping. We sometimes call this the point until which the server *scales*.

Hot request. A request is hot if it is cached to some degree (e.g. its dependencies are present in main memory). This is typically the case if the respective function was accessed recently.

Cold request. A request is cold if it is not cached (e.g. its dependencies must be fetched from disk). This is typically the case if the respective function was not accessed recently.

Approach. In the following experiments, we focus mainly on the 99th percentile (P99) latency, as tail latency is a key concern in serverless. Additionally, as long as the load is below the maximum throughput, median (P50) and P99 latencies tend to be quite similar in our system. Whenever there is an interesting discrepancy between P50 and P99, we mention it explicitly.

5.1 Dedicated Server Design

As there exists no readily available competitor for what we are trying to achieve, we build a bare-metal, dedicated server as a baseline. The fundamental idea behind this server is to capture the performance of a dedicated, single-tenant server which is built specifically for each workload (instead of using a platform like Dandelion). In some sense, it is thus a primitive type of single-domain service API.

Philosophically speaking, we design the server in such a way that "similar" effort is put into its implementation as would be required to get the same functionality running on Dandelion. As a result, while it provides good performance, we do not claim that it extracts the maximum possible potential out of the provided hardware. Further, unlike in Dandelion, we place no requirements on security or isolation, as these would not be required in a dedicated server, due to it being single-tenant.

The dedicated server uses a similar architecture to `gpu_process`. Requests are handled by a frontend that dispatches work to a set of threads, each with an associated worker process. Once again, we choose two worker processes per GPU. We initially tried to instead use streams for concurrency but ran into scaling trouble when using more than three GPUs, which is why we finally opted for the design above. This unexpected behaviour has been reported to AMD. Unlike Dandelion, however, data is copied from pinned memory,[1] which can be transferred at a higher rate than regular (paged) memory, as it cannot be swapped out.

5.2 Matrix/Matrix Multiplication

In this experiment, we perform a singular matrix/matrix multiplication of the form $A \times A^T$ for a 128×128 matrix A of 64-bit signed integers, which is provided in the request. The primary purpose of this workload is to provide a comparison to other Dandelion backends. For instance, this benchmark is used in [Kuchler et al., 2023], and [Thomm, 2024]. Finally, as computing such an operation is very quick on GPUs, it provides insight into the overhead of the system (e.g. generating the context, scheduling, loading kernels, moving data, etc.).

We perform two sweeps over the load; in 5.2.1 every request is hot, whereas in 5.2.2 every request is cold. For each server, we increase the load so long as the error rate[2] remains below one per cent.

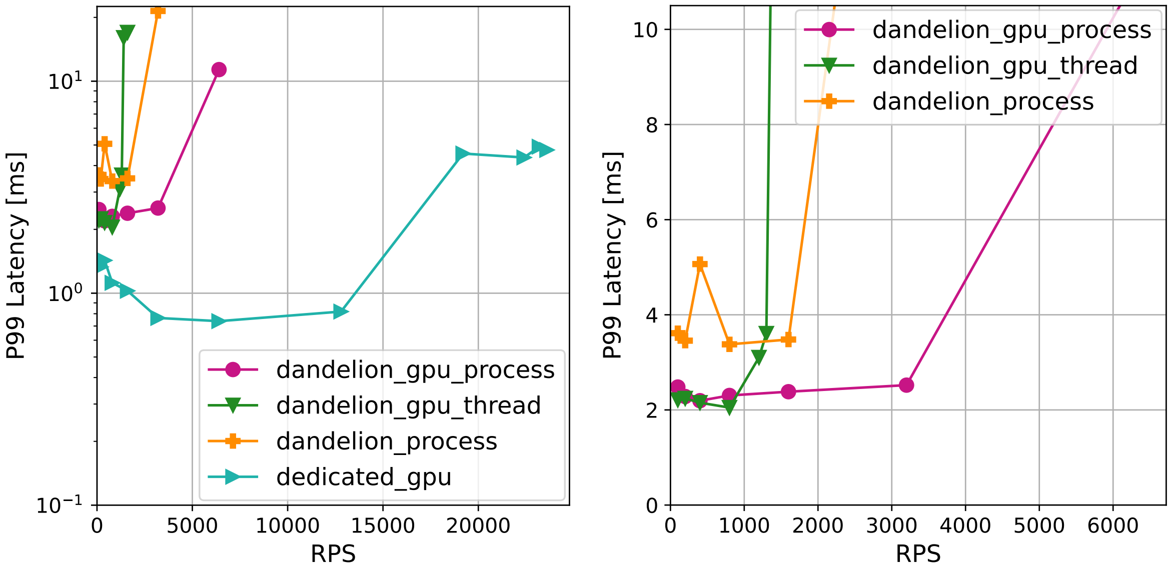
5.2.1 One Hundred Per Cent Hot

The results of the 100% hot load sweep are given in Figure 5.1. From 5.1a, one can clearly see that the dedicated server achieves a much lower base latency and hence higher throughput than its Dandelion counterparts. This is expected, given that the dedicated server

1. uses pinned memory for data transfers to the GPU, thus being able to make use of a higher bandwidth,
2. requires much less sophisticated scheduling logic than Dandelion,
3. does not need to load the kernels on the hot path, and
4. does not need to clear memory after use, as no isolation is required.

[1] To do this in practice, the HTTP library should write directly to shared pinned memory in order to avoid unnecessary data movement. We instead make use of the fact that each request is the same, and keep a copy of the data in pinned memory, allowing us to achieve a view of optimal performance without needing to extensively modify our HTTP library.

[2] Timeouts are considered errors as well.



(a) With `dedicated_gpu`. Note the log-scale.

(b) Without `dedicated_gpu`.

Figure 5.1: 100% hot 128 × 128 matrix/matrix multiplication load sweep.

The initial decrease in latency is likely due to frequency scaling of the CPU cores, which we unfortunately cannot disable.

Shifting to 5.1b, we can note that the base latency of the GPU engines is lower than the CPU-only `dandelion_process`. While GPU computation is of course much quicker, data still needs to be moved onto the accelerator. Indeed, looking at an execution trace reveals that the vast majority of the time is spent transferring data, whereas computation is almost instant by comparison. Considering this, the GPU engines beating `dandelion_process` speaks well for them. Also note that the spike at 400rps for `dandelion_process` is merely an outlier, likely caused by an OS job temporarily demanding compute resources or a brief change in frequency scaling. The median latency was not affected.

Comparing the GPU engines, we can see that their base latencies are very similar. Thanks to its concurrent architecture, `gpu_process` scales until about 2× the throughput of `gpu_thread`.

5.2.2 Zero Per Cent Hot

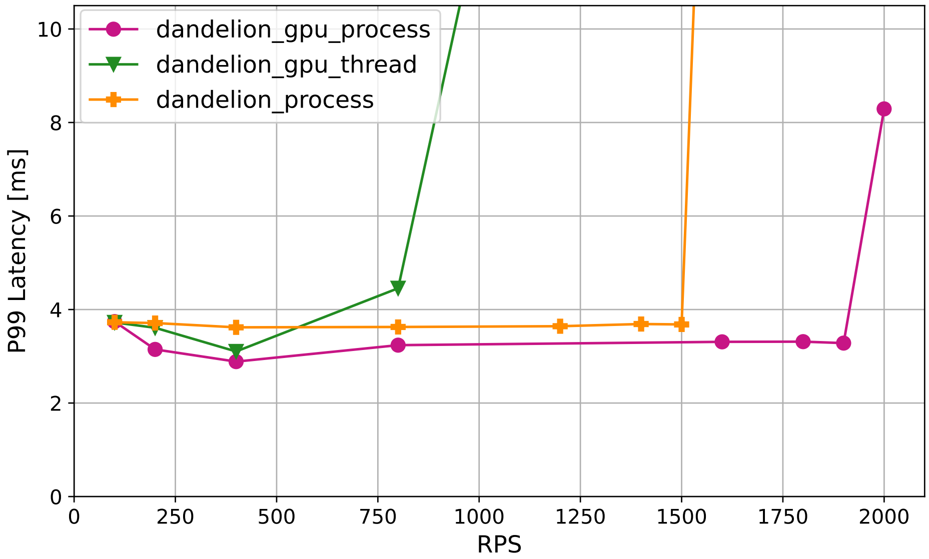


Figure 5.2: 0% hot 128 × 128 matrix/matrix multiplication load sweep.

Figure 5.2 shows the results of the 0% hot RPS sweep. Note the absence of `dedicated_gpu`, as the notion of a cold start does not exist for it, since its functionality is baked into the server binary and doesn't need to be loaded. Firstly, we observe that both GPU engines now produce a higher base latency and scale to a lower throughput as a result.

Curiously, `dandelion_process`'s base latency seems to only rise very slightly. We predict this is due to the different function parsing procedures—in the case of `dandelion_process`, a $\sim$ 8 KB binary needs to be loaded from disk and written to the static context, whereas `gpu_thread` and `gpu_process` need to load a $\sim$ 1KB `.json` file from disk, deserialise it, and then write a $\sim$ 45KB library (that should be in the OS's file buffer) to the static context. Therefore, it is not surprising that cold requests have a comparatively larger impact on the GPU backends' latency.

5.3 Inference Workload

Our inference workload measures the performance of our system in a typical machine learning inference / computer vision context. To this end, we perform a convolution, ReLU, and max-pooling by a factor of 2 on a provided 224 × 224 matrix of 32-bit floats.[3] The convolutional filter is of size 5 × 5. This process therefore effectively simulates one layer of a convolutional neural network (CNN). As CNNs are deep in practice, we repeat this process many times.

In Section 5.3.1, we sweep over the number of repetitions at fixed load. In Section 5.3.2 we fix the number of repetitions but vary the load. Once again, we increase load as long as the error rate remains below 1%. In both experiments, all requests are 100% hot, as we learned from the previous experiment that cold requests do not fundamentally affect the engines' behaviour.

5.3.1 Base Latency

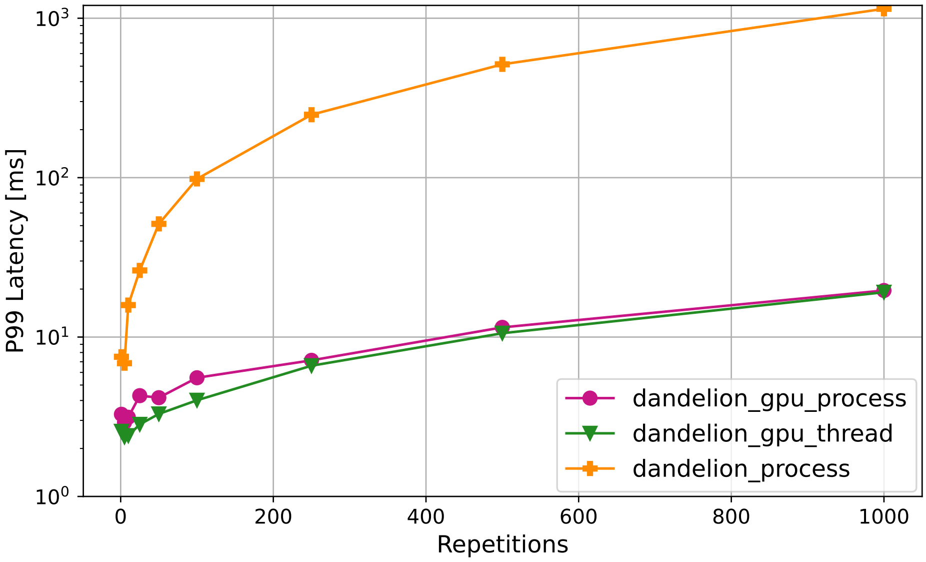


Figure 5.3: Inference workload repetition sweep at 25rps. Note the log scale.

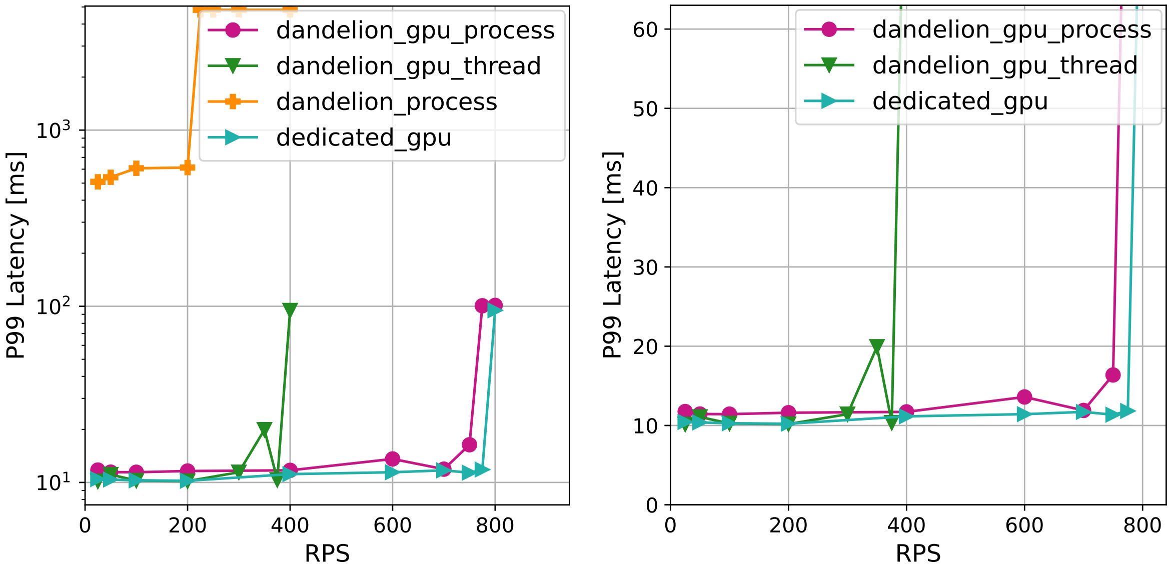
In Figure 5.3 we can see the results of increasing the number of repetitions of our convolution, ReLU, max-pooling pipeline. With this experiment, we aimed to investigate whether using the CPU-only engine would yield increased performance when repetitions were low compared the the GPU engines due to their associated data movement costs. If this was the case, we also wanted to find out where the break-even point was.

However, we can clearly see that even at just one repetition, the GPU engines achieve an around three to four times lower 99th percentile latency. This gap only widens as compute increases. Overall, we can note a linear relationship

[3] In other words, such a matrix is a greyscale image. The dimensions of 224 × 224 are the de facto standard for computer vision.

between repetitions and latency (the curve has a log-shape in the log-lin plot), which is what we expected. Furthermore, we can observe a slightly higher base latency for `gpu_process` than `gpu_thread`, which makes sense considering the engine has to share GPU resources and involves additional inter-process communication.

5.3.2 Latency under Load



(a) With `dandelion_process`. Note the log-scale.

(b) Without `dandelion_process`.

Figure 5.4: Inference workload load sweep at 500 repetitions.

Figure 5.4 showcases the results of the load sweep at 500 repetitions. From 5.4a we can discern that, as we would expect, `dandelion_process` cannot keep up with its GPU-accelerated counterparts. Nonetheless, it's worth pointing out that in this setting, the throughput of $\sim$ 200rps is indeed limited by the base latency, as

$$\frac{\text{Number of compute cores}}{\text{Base latency (seconds)}} \approx \frac{125}{0.6} \approx 200.$$

In other words—computation is the bottleneck here.

Discarding `dandelion_process` in 5.4b, we can take a closer look at the performance of the GPU-accelerated servers. Unlike in Figure 5.1a, we can ascertain that the GPU engines achieve similar base latencies as the dedicated server. We can explain this due to the much longer computation time amortising Dandelion's additional work on the critical path.

Next, we can see that the servers are all also latency-bound in their throughput, with gpu_process scaling twice as much as gpu_thread. The dedicated server behaves similarly to gpu_process, which makes sense given their similar architectures. Finally, it should be mentioned that the spike for gpu_thread at 350rps is again only an outlier that did not affect P50 latency.

5.4 Effect of Worker Count on Concurrent Engine

With this experiment, we aim to justify our choice of selecting two workers per gpu_process engine. We run the same experiment as in Section 5.3.2, except once using two workers per engine (i.e. the same as previously) and once using four workers per engine. For our hardware, we deem more than four workers to be impractical, as using any more would limit each worker to less than 16GB of GPU memory, which is required for many useful workloads.

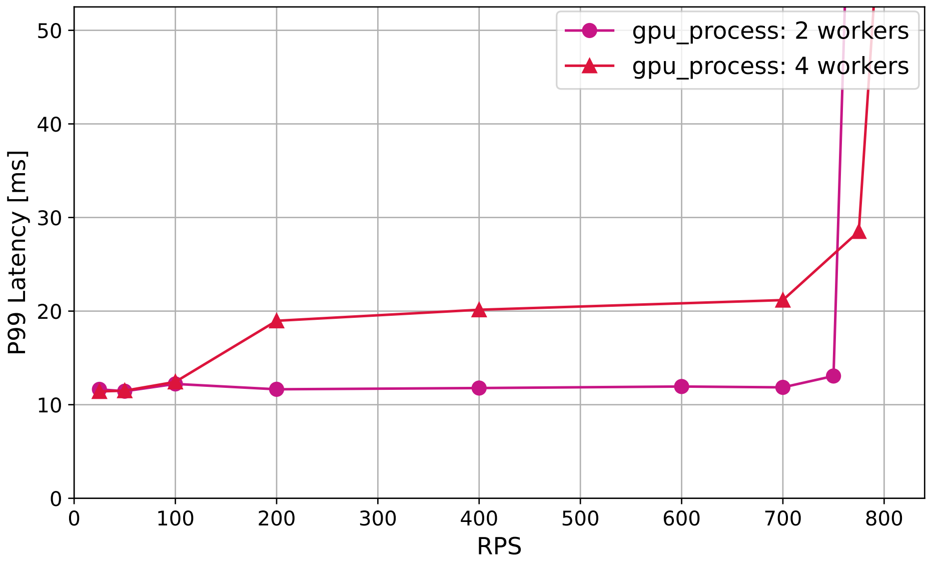


Figure 5.5: Inference workload load sweep varying the number of gpu_process workers.

Figure 5.5 shows the results of the measurement. While base latencies are initially similar, the four-worker variant quickly rises to around double that of the two-worker variant. As a result, the doubled parallelism is effectively cancelled out, with both scaling to a similar throughput of around 800rps. We will now explore some possible causes for this increase in latency.

Typically, when latency rises like this, it is due to resource contention. A first culprit could be that there are insufficient free compute units (CUs) around. Recall from Section 2.1.2 that CUs are the processors of GPUs, to which blocks of threads are assigned for execution. A quick calculation however shows that, at least in theory, four concurrent runs of our pipeline could fit on the GPU.

The largest kernel in our pipeline runs on a 224×224 grid. We choose the maximum block size our GPU supports, 32×32,[4] i.e. 1024 threads, thus resulting in a total of

$$\frac{224 \cdot 224}{32 \cdot 32} = 49$$

blocks. Our GPU has a limit of 2048 threads per CU, so each CU can handle up to 2 blocks at once. Hence, $\lceil \frac{49}{2} \rceil = 25$ CUs are sufficient to run our pipeline. With our GPU featuring 104 CUs, we therefore theoretically have the capacity for four concurrent pipeline runs. We thus expected the latency between two and four workers not to differ much.

If we do some simple profiling, we however observe that blocks are typically assigned to more CUs (we found the total number to be around 40). Such a policy would explain the scaling behaviour under the condition that CUs cannot host blocks from different processes simultaneously. Investigating this further is a challenging matter however, as it requires knowledge of micro-architectural scheduling policies that are often not publicly disclosed. See [Gilman et al., 2021] for a discussion of such a policy on NVIDIA devices.

Of course, many other resources can become contended as well, including execution queues, caches, and data interconnects to name a few. Finding an optimal setup is therefore dependent on a lot of factors and will certainly differ from one accelerator to the other.

We do not claim that always using two workers is a one-size-fits-all solution; in fact, it is conceivable that there exists a hardware setup and respective workload where more workers could yield a benefit. Thankfully, the number of workers can easily be varied upon engine startup, so this is feasibly something a future version of the dispatcher could have a bearing upon.

5.5 Batched Inference

Based on the previous experiments, one could be led to believe that using `gpu_process` offers doubled scaling “for free” in all cases since the penalty of halved memory capacity is only relevant in a select few applications. While this is certainly the case in the experiments presented until now (and therefore also applies to many real use cases), in this section we

[4] Choosing a larger block size is beneficial so long as enough resources (registers, local memory, etc.) are available, as local memory can be shared between threads in a block.

aim to explore conditions under which `gpu_process` scales less than double of `gpu_thread`.

Recall from our calculation in Section 5.4 that the inference workload does not issue enough blocks at once to fully saturate the GPU's compute units, which is precisely what enabled the concurrency of `gpu_process`. Thus, in this experiment, we add batching to our inference workload. Batching is a common technique in GPU programming, used to increase compute utilisation. Instead of operating on a single matrix at once, we extend our compute grid with a third dimension, operating on several matrices at once.

As before, we perform multiple iterations of our inference pipeline. In Section 5.5.1, we explore the impact different batch sizes have on the base latency. Section 5.5.2 compares the scaling behaviour for batch sizes two and eight.

5.5.1 Base Latency

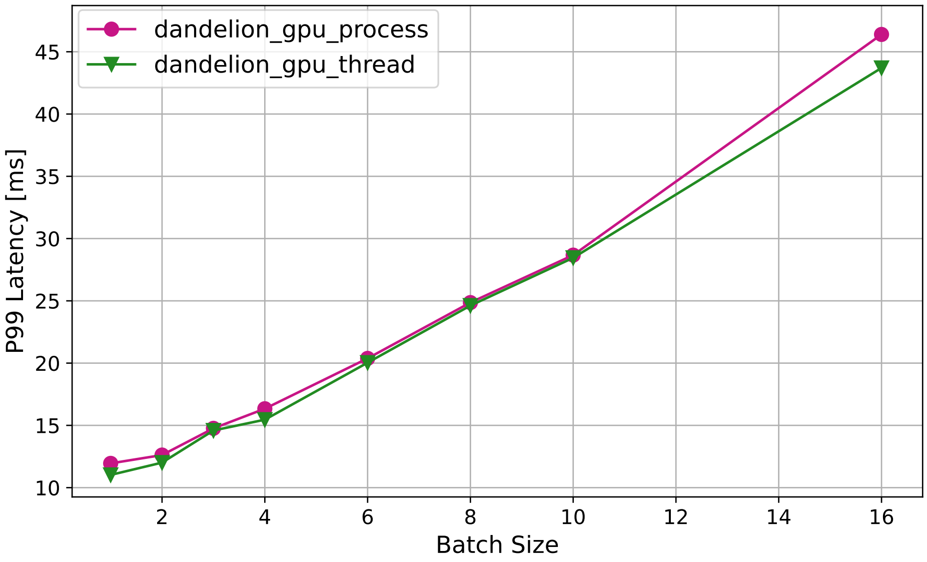
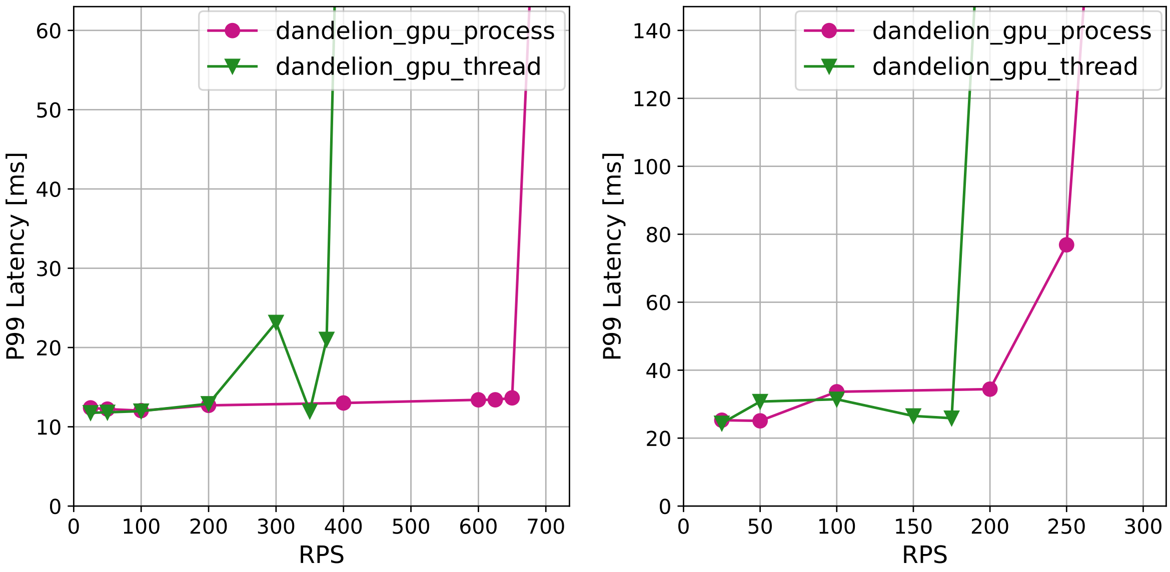


Figure 5.6: Sweep over the batch size for batched inference at 25rps.

Figure 5.6 shows the results of varying the batch size for our inference workload while keeping the load steady at 25 requests per second. With this experiment, we were particularly interested in seeing whether (and where) a saturation point exists. Before this point, the latency should not increase very much as additional compute units can still be utilised. Once the device is saturated, the latency should increase linearly as blocks start to queue for execution.

While there is only a slight increase in latency between batch sizes one and two, we can see that the GPU begins to become fully saturated starting at batch size three, with all subsequent points featuring a fairly linear relationship between batch size and latency. We can also be confident that the device is being fully utilised—recall from Section 5.4 that our pipeline requires at least 25 CUs without batching. Also, recall that each of our GPUs features 104 CUs. In other words, the pipeline only requires one-quarter of the total CUs in the optimal case. With batch size ten (i.e. a 10× increase in work) we can, for example, note an only $\sim 2.5\times$ increase in latency, suggesting that all CUs are being used, as the accelerator is capable of handling 4× the work without blocks needing to queue for execution. Compared to Section 5.4, we believe the accelerator is able to more effectively pack blocks onto CUs, as work is being issued in larger batches.

5.5.2 Latency under Load



(a) Batch size 2.

(b) Batch size 8.

Figure 5.7: Batched inference load sweep.

Figure 5.7 compares the scaling behaviour of `gpu_thread` and `gpu_process` at batch sizes two (5.7a) and eight (5.7b), i.e. before and after the saturation point. The main takeaway is that as long as the GPU is not fully utilised yet, `gpu_process` can aid scaling. For example, in 5.7a, we can observe an around 1.75× higher throughput. At batch size eight, the accelerator is fully loaded independently of which backend is used. It is worth noting that gpu_process still scales roughly 25rps further, likely due to it still gaining some parallelism by concurrently executing kernels and transferring data, though such a meagre increase in maximum throughput does not offset the increased base latency and reduced memory capacity in our eyes. Finally, gpu_thread's spike at 300rps in the left figure is once again an outlier likely caused by external factors.

As a result, we can now formulate circumstances under which dispatch to gpu_thread should be favoured:

1. The function requires more memory than gpu_process can offer.
2. The function issues enough blocks at once to fully saturate the GPU.

As both of these parameters are known at the time of invocation, a future version of the dispatcher could feasibly decide on which backend a function should run without any user intervention. Not only would this abstract away the hardware in the true spirit of serverless, but it would be a testament to Dandelion's potential for powerful dataflow-aware scheduling.

Chapter 6

Conclusion and Future Work

In this project, we developed two GPU-accelerated executors for the Dandelion platform. By taking advantage of Dandelion's innovative programming model and system design, we were able to show that efficient and general-purpose GPU-accelerated serverless is feasible.

We presented our vision for GPU functions in Dandelion and explored the details of our implementation. In particular, our policy on memory management prevents expensive memory allocations on the hot path, while our concurrent engine extracts the maximum performance out of our hardware, even if individual functions do not require the entire accelerator's capabilities. Further, we discussed how our engines enforce isolation between tenants.

In our evaluation, we showed that GPU acceleration can result in great performance benefits over CPU-only computation, even for computationally modest workloads. In addition, we showcased the differences between our sequential and concurrent engines and laid out clear cases in which one should be favoured over the other.

We continue by listing opportunities for future research:

Different Hardware. Our current system runs on AMD accelerators. However, in both industry and academia, NVIDIA hardware is the de facto standard. Extending the system to run on NVIDIA GPUs could therefore allow significantly more clients to make use of our platform. In addition, NVIDIA's multi-instance GPU (MIG) technology could be used to exercise more control over GPU sharing rather than relying on process scheduling like in our concurrent engine. Further, unified memory could be used to implement a genuine on-device memory domain.

Streams. Recall from Section 4.1.1 that we currently execute all kernels on a single stream. In the future, we would like to allow the programmer to use streams to more easily express complicated, independent control flow.

Better buffer pool. When we introduced our buffer pool in Section 4.2.1, we explained that our current design allocates all buffers sequentially in memory, even if they are not all live at the same time. This could be improved by a smarter buffer pool that allows temporary buffers that are not live at the same time to be re-used. In particular, this would be helpful for workloads that allocate many temporary buffers.

Automatic Configuration Generation. Today, a vast number of frameworks exist that allow users to take advantage of GPU acceleration without ever needing to write any kernels themselves. Our design on the other hand is a lot more low-level. However, we explicitly designed our configuration format with programmatic generation in mind. In future work, we envision whole GPU functions to be conveniently automatically generated. To achieve this, frameworks like Google's JAX [Bradbury et al., 2018] or Apache TVM [Apache, 2018] could be extended with custom backends that produce Dandelion GPU functions.

Finally, we hope that this thesis provided an interesting insight into the potential of GPU acceleration for serverless. We believe that the usage of dedicated hardware accelerators combined with innovative platforms like Dandelion can yield huge gains in both performance and efficiency.

Appendix A

Addendum on Evaluation Setup

Component	Type	Notes
CPU	2x AMD EPYC 7V13 64-core @ 2.45GHz	SMT disabled
RAM	500GB	—
GPU	4x AMD Instinct MI210 w/ 64GB VRAM	—
GPU Driver	ROCm 5.7.1	—
OS	Ubuntu 20.04 LTS	—
Kernel	Linux 5.4	—

Table A.1: Hardware setup of a HACC-box.

All experiments can be replicated with the `dandelionExperiments` repository at git hash `6dc5e0d1fa48320d856e556c81b7e34ffdd92a98`. Due to some values (e.g. number of inputs of a function or worker count) being hard-coded, the experiments were recorded with slightly different versions of the main `dandelion` repository. The core functionality however remains exactly the same.

Experiment	`dandelion` Repository Hash
Matrix Multiplication	a8fdd5f6beb1f3180f839d30d354102a26f5eb65
Inference	cbe10d2daacabef74204e2a8f0291b6d7cc6c2c3
Inference w/ 4 Workers	b9725f3be68886f3750bd9b5aeb6a3b1e16bf4d6
Batched inference	90db6d3a00f465cde8b5ccd83786821aca154b88

Table A.2: `dandelion` repository git hashes.

Bibliography

[Agache et al., 2020] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. (2020). Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA. USENIX Association.

[AMD, 2017] AMD (2017). Consistency and security: Amd's approach to gpu virtualization. https://www.amd.com/system/files/documents/gpu-consistency-security-whitepaper.pdf.

[AMD, 2019a] AMD (2019a). https://github.com/ROCm/ROCm-docker/issues/62#issuecomment-567691938.

[AMD, 2019b] AMD (2019b). Introduction to amd gpu programming with hip. https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf.

[AMD, 2023] AMD (2023). Hipify documentation. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/.

[AMD, 2024] AMD (2024). Leftoverlocals security bulletin. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-6010.html.

[Apache, 2018] Apache (2018). Apache tvm. https://tvm.apache.org/.

[BentoML, 2019] BentoML (2019). Bentoml. https://github.com/bentoml/BentoML.

[Bradbury et al., 2018] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

[Gilman et al., 2021] Gilman, G., Ogden, S. S., Guo, T., and Walls, R. J. (2021). Demystifying the placement policies of the nvidia gpu thread block scheduler for concurrent kernels. *SIGMETRICS Perform. Eval. Rev.*, 48(3):81–88.

[Han et al., 2022] Han, M., Zhang, H., Chen, R., and Chen, H. (2022). Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA. USENIX Association.

[Jin and Vetter, 2022] Jin, Z. and Vetter, J. S. (2022). Evaluating unified memory performance in hip. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 562–568.

[Kim et al., 2018] Kim, J., Jun, T. J., Kang, D., Kim, D., and Kim, D. (2018). Gpu enabled serverless computing framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 533–540.

[Kuchler et al., 2023] Kuchler, T., Giardino, M., Roscoe, T., and Klimovic, A. (2023). Function as a function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 81–92, New York, NY, USA. Association for Computing Machinery.

[Luna and Mutlu, 2022] Luna, J. G. and Mutlu, O. (2022). Digital design and computer architecture: Lecture 21: Graphics processing units. https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2022-lecture21-gpu-afterlecture.pdf.

[Moritz et al., 2018] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. (2018). Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA. USENIX Association.

[Moya et al., 2023] Moya, J., Gabathuler, M., Ruiz, M., and Alonso, G. (2023). fpgasystems/hacc: Ethz-hacc 2022.1. Zenodo. https://doi.org/10.5281/zenodo.8344513.

[Naranjo et al., 2020] Naranjo, D. M., Risco, S., de Alfonso, C., Pérez, A., Blanquer, I., and Moltó, G. (2020). Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42.

[NVIDIA, 2020] NVIDIA (2020). Nvidia multi-instance gpu and nvidia virtual compute server. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf.

[NVIDIA, 2024] NVIDIA (2024). Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[Olston et al., 2017] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. (2017). Tensorflow-serving: Flexible, high-performance ml serving.

[Owens et al., 2008] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879–899.

[PCISIG, 2021] PCISIG (2021). Pcie® 6.0 architecture: The future of pcie technology. https://pcisig.com/sites/default/files/files/PCI-SIG_PCIe%206.0%20Specification%20Infographic_Final_0.pdf.

[Pemberton et al., 2022] Pemberton, N., Zabreyko, A., Ding, Z., Katz, R., and Gonzalez, J. (2022). Kernel-as-a-service: A serverless interface to gpus.

[Putnam et al., 2015] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. (2015). A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22.

[Romero et al., 2021] Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. (2021). INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association.

[Satzke et al., 2021] Satzke, K., Akkus, I. E., Chen, R., Rimac, I., Stein, M., Beck, A., Aditya, P., Vanga, M., and Hilt, V. (2021). Efficient gpu sharing for serverless workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, page 17–24, New York, NY, USA. Association for Computing Machinery.

[Schleier-Smith et al., 2021] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. (2021). What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84.

[Shahrad et al., 2020] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association.

[Sorensen and Khlaaf, 2024] Sorensen, T. and Khlaaf, H. (2024). Leftoverlocals: Listening to llm responses through leaked gpu local memory.

[Thomm, 2024] Thomm, L. (2024). Exploring the use of webassembly for isolating functions in dandelion. Bachelor thesis, ETH Zurich, Zurich.

[Thompson and Spanuth, 2021] Thompson, N. C. and Spanuth, S. (2021). The decline of computers as a general purpose technology. *Commun. ACM*, 64(3):64–72.

[Ustiugov et al., 2021] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. (2021). Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA. Association for Computing Machinery.