University of Glasgow | School of Computing Science

Implementation of a Parallel Virtual Machine on a GPU using OpenCL

Gary Blackwood

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 22, 2013

Abstract

Existing methods for writing programs that execute on graphics processing units are complex and require code that does not express the intended computations of the programmer. We introduce a parallel virtual machine that abstracts the underlying GPU hardware to provide an environment in which programs can be developed using a high-level functional approach. Implemented on a GPU using OpenCL, the virtual machine uses a combination of native hardware parallelism and architectural design to execute programs in parallel. We present a configurable service-based method in which to write programs and show that the machine is a viable alternative to other GPU programming approaches by investigating system performance.

Education Use Consent

Contents

Chapter 1

Introduction

A virtual machine is a software implementation of a machine (abstract or real) that executes programs like a physical machine. In particular, a *process* virtual machine, unlike its *system* counterpart, is designed to run a single program in a portable and flexible manner. It provides a platform-independant programming environment by abstracting away the details of the underlying hardware. Software that runs on such a virtual machine is limited to the resources and abstractions provided and is constrained to the virtual machine's virtual environment. By targeting a graphics processing unit as an execution platform, the virtual machine can take advantage of the parallel nature of graphics processing. This parallel virtual machine enables large computational problems to be solved more efficiently while maintaining its portability characteristic and high-level abstractions.

1.1 Aims

The aim of this project is to design and implement a parallel process virtual machine that will run on a graphics processing unit. Typically, virtual machines will offload code to the GPU for acceleration. However, in this case the virtual machine will run directly on the GPU, receiving and executing bytecode from the host system. Implementation will be carried out using Open Computing Language (OpenCL) which will allow for the virtual machine to be executed across heterogeneous platforms. The virtual machine should enable task parallelism in what is traditionally a data parallel environment and allow users to write concise programs in a functional style to express their computations. In addition, performance is of critical importance and as such the virtual machine should cause minimal overhead.

1.2 Motivation

The major motivation behind this project is to provide users with a programming environment in which they can express their computations in an easy to understand manner while taking advantage of the parallelism offered by the utilisation of graphics processing units. Currently, development on such devices is often verbose and requires boilerplate code that does not directly express the intended computation. The abstractions provided by the virtual machine will focus user efforts on the problems that they wish to solve and not the tool(s) that they are using to do so.

1.3 Report Content

The following chapters will provide an in-depth look at the basis, design, implementation and evaluation of the virtual machine:

- Chapter 2 covers the necessary background knowledge required.
- Chapter 3 outlines the detailed requirements of the project.
- Chapter 4 explains how the project was designed and implemented.
- Chapter 5 demonstrates how testing was carried out.
- Chapter 6 details the project results and evaluation.
- Chapter 7 introduces future work and potential improvements.
- Chapter 8 provides a project summary and overall conclusion.

Chapter 2

Background

This section introduces the background material upon which the project was built. The reader will be provided with the knowledge necessary to understand the concepts discussed in later chapters. We begin by introducing General Purpose GPU programming, its applications and an in-depth look at one way in which it can be performed. Next, an overview of the Gannet System-on-Chip architecture is provided. Finally, we take a look at a number of existing virtual machine implementations.

2.1 GPGPU

The ever increasing popularity of video games and the insatiable demand for high-definition graphics has driven the development of highly parallel programmable GPUs which possess tremendous amounts of computational power and high memory bandwidth[6]. The performance of such GPUs is rapidly increasing and its rate of growth significantly outpaces Moore's Law as applied to traditional microprocessors[2]. Today, GPUs are one of the most cost-effective ways of accessing vast amounts of computational horsepower and as such developers have become increasingly interested in harnessing this power for general-purpose computing.

Programs traditionally handled by a central processing unit can be executed on a GPU using a technique called General Purpose GPU programming. GPUs are designed for highly parallel tasks such as rendering and the key to using them for other purposes is to view them as streaming, data-parallel computers[5].

Stream Processing

Stream processing is a programming paradigm in which many fragments of a stream are processed in parallel. A *stream* is a set of data in which each element requires similar computation. Formally, stream processing is defined as follows:

Given a data-set (stream) D, and a computation (kernel function) $f(d)$ which will be applied to each element $d \in D$. If the kernel function is independent then all computations $f(d_i)$ for $i = 1, ..., n$ can be evaluated in parallel.

This data parallelism is particuarly applicable to GPUs which only process independent data fragments.

Data Parallelism

Data parallelism is a form of parallelisation which emphasises the parallelised nature of data by distributing the set across a number of different computing nodes. Each processing node performs the same task on a different fragment of the distributed data. GPUs are naturally data parallel due to the memory restriction characteristics that they possess and their role as streaming processors.

Task Parallelism

Task parallelism is a form of parallelisation that emphasises the parallelised nature of processing whereby each processor executes a different thread of control on either the same or a different set of data. Unlike data parallelism, task parallelism does not scale with the size of a problem. Achieving task parallelism on a GPU requires much more effect than data parallelism as it is unnatural on the hardware. Task parallelism is the natural domain of CPUs.

2.1.1 Applications

Ideally, applications should have large, highly parallel data-sets which share minimal dependencies between elements. Other than graphical computations, GPGPU is commonly used for:

Sorting Parallel algorithms such as *bitonic mergesort* are particuarly applicable having been adapted and improved for GPUs[1].

Searching Various implementations of searching algorithms exist on GPUs such as *binary search*[5]. In addition, high-performance database queries can benefit from the additional performance on offer.

Bioinformatics DNA sequence analysis and alignment software may be GPU accelerated.

2.1.2 Platforms

In recent years a number of GPGPU platforms have been developed which have reduced the learning curve and development overhead by attempting to minimise the exposure to the low-level graphics rendering pipeline. Originally, GPGPU programs were developed using shader languages such as GLSL and HLSL[7]. These languages required in-depth knowledge of specific data types and algorithms in addition to problems being expressed in rendering primitives.

BrookGPU

BrookGPU is a stream programming language which facilitates non-graphical, general purpose computations. It is freely available and developed by the Stanford University graphics group.

CUDA

CUDA is nVidia's proprietary GPGPU platform that can only be used in conjunction with nVidia GPUs. One reason that it was not used for the project is that it has poor portability.

2.1.3 OpenCL

Open Computing Language is a standardised framework for writing programs that will execute across heterogeneous platforms i.e. systems that use different types of computational units. It provides a common language, application programming interface and hardware abstractions which allow developers to produce task and data-parallel applications in an environment consisting of a host CPU and any number of attached OpenCL devices.

An OpenCL application consists of a host program and one or more kernel programs. The host program is responsible for memory allocation, initialisation and deallocation. It sends commands to the devices to transfer data between the host and device memory buffers and to execute code on the device. Kernels are programs that are executed on devices and are written in a special OpenCL language.

OpenCL Language

The programming language used to develop OpenCL kernels is based on the ISO/IEC 9899:1999 version of the C programming language. It omits the use of features such as function pointers and variable-length arrays however it is extended to include OpenCL specific data types, memory region qualifers, functions and parallelism features.

Execution Model

Unlike the sequential code execution that occurs in a host program, kernel code is executed in parallel across a number of processing elements. The basic unit of work on an OpenCL device is a *work-item*. A kernel is executed across a global N-dimensional domain of work-items as shown in figure 2.1. The whole problem space, also known as the *global dimension* is the domain in which a kernel is executed across. The *local dimension* refers to how the work-items within the global dimension are grouped. A work-group executes on a single compute unit, which is composed of one or more processing elements. In figure 2.1 the global dimension is $1024*1024$. It is not a requirement that the target device have N hardware processing elements where N is the number of work-items. OpenCL will automatically schedule each work-item kernel execution across all available processing elements.

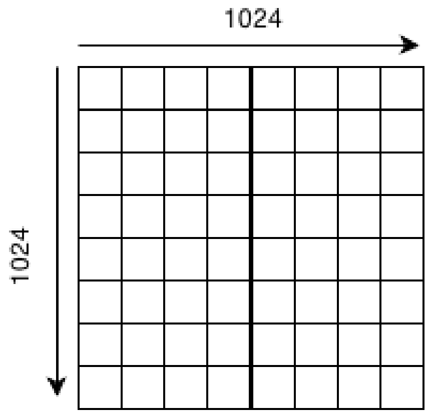

Figure 2.1: Two-dimensional domain of work-items.

Memory Model

There are four different types of memory on an OpenCL device, each of which has different levels of access.

	Global	Local	Constant	Private
Host	Dynamic allocation. Read/Write access.	Dynamic allocation. Read/Write access.	Dynamic allocation. No access.	No allocation. No access.
Kernel	No allocation. Read-/Write access.	Static allocation. Read-only access.	Static allocation. Read/Write access.	Static allocation. Read/Write access.

Table 2.1: Allocation and Memory Access Capabilities (adapted from [4]).

1. *Global* memory is the most abundant memory type on a device and is accessible by all work-item threads. However, it is the slowest memory subsystem and can be the cause of race conditions as it is shared memory.
2. *Local* memory is faster than global memory and in general, its physical location is much closer to processing elements. It is shared between work-items in a work group and is a precious resource due to its small capacity.
3. *Constant* memory is a read-only section of memory accessible to all work-items threads.
4. *Private* memory is an extremely fast access yet small capacity area of memory accessible to a single work-item. It is similar to registers in a CPU.

OpenCL memory management is explicit; the host program must transfer data to and from the device memory. Unfortunately this is slow and should therefore be avoided.

Data Programming Model

The dimensions associated with the execution model defines the work-items and how the data maps onto them. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in memory over which a kernel can be executed in parallel[4]. However, OpenCL implements a relaxed version that does not require this mapping. Using the explicit data programming model the programmer defines the total number of work-items to execute in parallel and specifies how the work-items are grouped. The implicit model only requires the total number of work-items as OpenCL handles work-item group division.

Task Programming Model

The task parallel programming model executes a single instance of a kernel regardless of the execution model dimensions. It is equivalent to executing a kernel on a compute unit with a work-group that contains a single work-item. Responsibility is left to the programmer to express such parallelism using methods such as enqueuing multiple tasks. This is no easy task and is often cumbersome.

```
__kernel void mmult(__global float *A,        // Input matrix 1.
                    __global float *B,        // Input matrix 2.
                    __global float *C,        // Memory for result.
                    uint width, uint height) // Dimensions of matrices.
{
  // Get the work-item row and column (2D space).
  uint row = get_global_id(0);
  uint col = get_global_id(1);

  // Compute the dot product.
  float dotProduct = 0;
  for (uint i = 0; i < width; i++) {
    dotProduct += A[row * width + i] * B[i * width + col];
  }

  // Write the result into memory.
  C[row * width + col] = dotProduct;
}
```

Listing 2.1: Matrix multiplication kernel.

Example

Listing 2.1 and figure 2.1 can be combined to illustrate the kernel execution process. Consider a host program which sets the global dimension to $1024*1024$ and randomly generates two input matrices A and B both of which have a width and height of 1024. The above kernel will be executed once for each cell in the input matrices as the number of work-items is the same as the number of cells.

This data parallel computation performed using OpenCL will be completed much faster than the equivalent single threaded sequential implementation shown in listing 2.2, especially if the input matrices are sufficiently large.

```
int main(void) {
  int n, width, height;
  n = width = height = 1024;
  float A[n * n];
  float B[n * n];
  float C[n * n];

  /* Initialise input matrices ... */

  for (int row = 0; row < width; row++) {
    for (int col = 0; col < height; col++) {
      int dotProduct = 0;
      for (int i = 0; i < width; i++) {
        dotProduct += A[row * n + i] * B[c * n + col];
      }

      C[row * n + col] = dotProduct;
    }
  }

  return 0;
}
```

Listing 2.2: Sequential matrix multiplication program.

2.2 Gannet

Gannet is a distributed service-based System-on-Chip architecture that consists of a network of services that are offered by software or hardware cores[9]. The architecture employs a functional programming approach to meet the growing demand for solutions that enable users to design very large Systems-on-Chip at high levels of abstraction. To create a fully concurrent distributed System-on-Chip, Gannet combines the flexible connectivity of a Network-on-Chip with a functional language paradigm that enables task-level reconfiguration.

2.2.1 Design

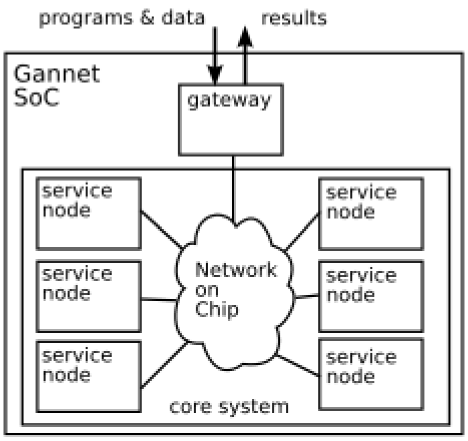

Figure 2.2: The Gannet architecture[9].

Gannet organises its system as a set of services that are called on-demand. Each service is typically handled by a self-contained processing unit known as an *Intellectual Property* core which offers a specific functionality and can be reconfigured at run-time. Each Intellectual Property core acts as a service provider and is incorporated into a *service node* which has the ability to communicate with all other nodes in the system using a packet-based communication mechanism. Due to the complexity of large Systems-on-Chip, traditional bus-style connectivity is not a viable option. Gannet solves this problem by using a packet-switched Network-on-Chip that avoids the communication bottleneck via flexible connectivity and an efficient mechanism for managing wires. Figure 2.2 illustrates how the Network-on-Chip is used as a communication mechanism between service nodes. We will see later how the inclusion of this Network-on-Chip will affect the design of the parallel virtual machine. Note that a special *gateway* node exists to act as an entry and exit point for all system data.

A service is defined as a unit that consumes data, produces data and can modify its internal state[8]. Due to its similarity to a function, a functional approach to task description can be taken. Adopting a functional paradigm allows Gannet to natively support parallelism and take advantage of the simplicity of functional composition. The resulting data produced by a service is determined by the results produced by its delegated service requests and constant data.

Gannet achieves its service-based behaviour by providing an interface between each Intellectual Property core and the rest of the system. In practice a service node has no knowledge of the overall task the system is performing; it simply performs a computation on an input data set and returns the result to its requester. It is the responsibility of the service manager, present in every service node and governed by a task description program, to determine where to request data and where to direct the computation results.

2.2.2 Task Description

The functionality of the system is not determined by the functionality of each individual service node. Instead, a task description program is written in a functional language to describe the overall task performed by the system. For example, the following task description program, written in the Gannet language, would specify a system that performed a matrix multiplication computation using two input matrices:

```
(mmult (matrix1) (matrix2))
```

Task description programs are transformed into lists of packets, each containing a module of the complete program. Packets are distributed to their destination service via the Network-on-Chip before code execution can begin. Each service node consumes and produces packets and as such the program has completed execution when all packets have been processed.

Configuration

In addition to the task description program, system configuration data is used to assign Intellectual Property cores to particular services and to provide a number of options for each service node. The above example would therefore require that one or more Intellectual Property cores be mapped to the *mmult*, *matrix1* and *matrix2* services. By assigning multiple cores to the same service parallel execution can be achieved even if two or more subtasks using that service are being evaluated.

The Gannet Language

The Gannet language is a functional assembly language that possesses an s-expression syntax and can be trivially transformed into machine code[9]. In Backus-Naur Form it is expressed as:

```
gannet-expression ::= (service-token argument-expression+)
argument-expression ::= (gannet-expression | literal)
```

Every Gannet service has a corresponding *service-token* which acts as an identifier. Due to the simplicity of the language it is the responsibility of these services and their implementers to provide additional language features such as control structures.

A literal is any quoted argument e.g. ’0 or ’255. All arguments that are quoted are treated as literal data values, even if they are calls to other services. A quoted service call will not be evaluated immediately.

High-Level Languages

The Gannet language is not intended to be used as a language with which to write task description programs. However, due to its functional nature, other higher-level languages can be used as suitable alternatives. For example, Scheme, as it is syntactically and semantically similar to Gannet, can be trivially transformed into the Gannet language syntax before being further processed by the Gannet system. The ability to use existing high-level languages allows developers to concentrate on the task at hand and avoid having to learn yet another language.

2.2.3 Packets

As Gannet uses a packet-switched Network-on-Chip for all service node communications, the fundamental unit of data transfer is a packet. Packets contain the bytecode to be executed by the Gannet system and are generated by the Gannet compiler as shown in figure 2.3. They are marshalled between service nodes by their service managers using a queue-based system which allows for parallel processing.

Essentially, the Gannet system works by decision making conditionally upon the contents of packets. If a *reference* packet is received a service will delegate a subtask and wait for its result to return in the form of a *data* packet.

2.2.4 Software Flow

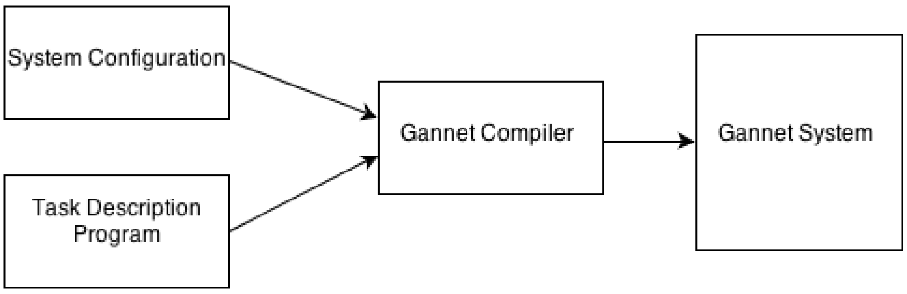

Figure 2.3: Gannet software flow.

The task description program is compiled into a set of instruction containing packets which enter the system via the dedicated gateway circuit. Memory is allocated for the program result before the packets are sent to their destination services via the Network-on-Chip. To begin code execution, a single reference packet is created (containing the code to be executed) and sent to the initial service. This packet causes a chain reaction of computational requests by activating the root task which, in turn, delegates its subtasks to the their corresponding services by creating and sending additional reference packets. This process continues for each requested task until no further subtasks need requesting. These lowest-level tasks may have constant literal valued arguments that do not need requesting or may not have any arguments whatsoever.

The entire process can be visualised as a tree reduction in which each node is a single task (service call) and each child node is its subtasks (arguments). To begin code execution the computation represented by the root node of the tree is requested. This root node creates a number of child nodes, one for each argument, before the process is repeated using each newly created child node as the root. This is possible due to the recursive definition of tasks which maps nicely onto the tree data structure. Upon encountering a lowest-level task (leaf node) the resulting value is passed to the task which requested it – its parent node. When the root node receives a result the computation is complete.

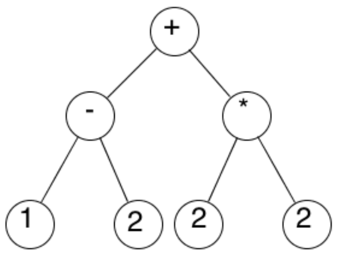

Figure 2.4: A complete tree reduction for the program (+ (- 1 2) (* 2 2)).

2.2.5 Deferred Evaluation

Before the computation represented by any particular task is performed, all of its subtasks must first be evaluated. However, it may not be the intention of the programmer to evaluate a function argument immediately, as is often the case when dealing with lambda functions. To handle this use case, a *quoting* mechanism was put in place that allows argument evaluation to be deferred. It works by marking the argument as a literal that is to be stored as data. Thus, upon receiving its packet, no subtask request will be made and instead the value will be stored as data.

2.2.6 Concurrency

Due to the way in which the Gannet system handles function calls, all function arguments are effectively evaluated concurrently. This parallel execution of hardware cores ensures that the available resources are being used optimally and that the fastest execution possible is being achieved. It is also one of the main reasons why the Gannet architecture will map naturally onto a graphics processing unit which is designed for parallel use of its many hardware cores.

2.2.7 Evolution

The Gannet system described in this section has since evolved into one which uses a shared memory architecture. It is this recent evolution of Gannet that the project work is based upon.

2.3 Virtual Machines

2.3.1 System Virtual Machine

A system virtual machine supports the execution of complete operating systems and is often built for the purpose of providing a platform with which to run programs in the event that the real hardware is not available. They allow for single systems to be abstracted into multiple systems using hardware virtualisation, the process of which leads to efficient use of computing resources. They are often utilised for operating system development as multiple operating system environments can co-exist of the same computer in isolation from one another. This prevents unfortunate bugs from causing problems in the host system. Virtual machines also possess better debugging facilities and faster reboot times. Popular implementations include Oracle Virtualbox and VMWare.

Unfortunately performance may be unreliable when multiple virtual machines are executing concurrently on the same host if interference between them is not appropriately limited. In addition, implementations are commonly stack or register based which makes them unsuitable for execution on graphics processing units. GPUs do not have an operating system layer and therefore cannot support a stack or register based virtual machine implementations.

2.3.2 Parallel Virtual Machine

Parallel Virtual Machine (PVM) is a software system that allows programmers to exploit the power of distributed computing across a wide variety of heterogeneous systems[3]. It abstracts away the interaction between a collection of computers until the entire system appears as only a single large virtual machine. The main advantage of such a machine is that large computational problems can be solved more cost effectively by using the collective power of many computer systems in parallel.

PVM was initially released in 1989 and has since grown into a complex software package that supports a number of languages and provides a complete library. Application programming involves writing one or more sequential programs using one of the supported languages. Each program is a task that makes up the complete application and is compiled for each computer architecture in the pool of machines to be used. Execution begins by starting one copy of a task by hand. This process quickly informs all other machines which subsequently starts the other PVM tasks.

There exists a number of similarities between PVM and the parallel virtual machine project objective. Both adopt a message passing system that is used as a communication mechanism between 'services'. An important difference is that PVM creates a true long distance network of computing hardware whereas this virtual machine implementation will simulate a Network-on-Chip where all communication occurs within the physical hardware of a single host system. Both virtual machines are built with parallelism in-mind and aim to be as portable as possible. Yet another similarity is that of user-configuration. Both machines allow users to select which service nodes, or in PVM's case which set of machines, performs a particular computation or service.

An important note to make for the purposes of this project is that even though both seem similar, PVM is *not* implemented on a graphical processing unit. It would not be suitable as it requires networking capabilities that would be too inefficient to deal with using a GPGPU platform and would require switching control between host and GPU frequently.

Chapter 3

Requirements

3.1 Problem Analysis

The objective was to implement a parallel process virtual machine on a GPU using OpenCL. In particular, the virtual machine should virtualise the Gannet System-on-Chip architecture by accepting as input the bytecode produced from compiling a task description program and service configuration using the Gannet compiler. The bytecode should then be parsed, processed and used to perform the computation that it represents.

The virtual machine should be fast, lightweight and cause minimal overhead. Only a small subset of Gannet functionality was required; the goal was to demonstrate correct execution of compiler generated bytecode. Compatibility with the existing Gannet system was important as to ensure maintainability and smooth feature integration in regards to future work.

Services need not be implemented or program input data handled for the user. These should be customisable in a way which allows users to express them in a clear and concise manner. We also wish to provide a way in which to perform task parallelism as the reduction tree evaluation may not be entirely data parallel. By virtualising the Gannet architecture we aim to provide a high-level functional way for users to program on GPU.

3.2 Proposed Solution

The virtual machine was to be developed as an OpenCL application that accepts as input a file containing the bytecode produced by Gannet compiler. The application would consist of a host program responsible for initialisation and a single kernel program that would implement the virtual machine. Implementation details would mimic those of the Gannet System-on-Chip architecture as closely as possible so that the project remained compatible.

It was clear that some structural changes would be necessary. For example, the Gannet architecture uses a packet-switched Network-on-Chip which will not be available. Instead, a new communication mechanism was to be developed between service nodes that uses the shared global memory of the GPU. A state machine design was to be used to avoid race conditions when accessing the shared memory.

Chapter 4

Design and Implementation

4.1 High-Level Overview

The design of the virtual machine is based heavily upon the current Gannet architecture. Like Gannet, a service-based approach is taken in which a network of services are offered by software or hardware cores. As the program is being developed as an OpenCL application, each core will be a compute unit, physical or virtual, on an OpenCL device.

Service nodes will communicate using a FIFO queue-based implementation of Gannet's Network-on-Chip. True network distribution is not a requirement therefore this new communication mechanism uses the shared memory of the GPU to facilitate message passing on a single physical device. A state machine computational model is used to prevent the occurrence of race conditions as a side-effect of using shared memory. The current state is changed as control alternates between the host system and device. Message passing consists of packet exchanges between service nodes, each of which acts as a trigger for certain actions such as creating subtasks and performing computations.

Subtasks are handled using two data structures; a subtask record and a subtask table. Subtask records contain detailed information about a single subtask and are stored in a subtask table. A single subtask table, containing all subtask records for a particular program execution, is used to keep track of pending subtasks and to create new ones using a stack allocation mechanism.

A bytecode file, representing the computation to be performed and produced by the Gannet compiler is passed to the virtual machine as input. Using this file, a data buffer known as the code store is initialised by reading and parsing individual bytecode words. It is this buffer, used by the virtual machine, that contains the actual computation to be performed. In addition, a dedicated memory buffer is allocated for input data, output data and partial results. This buffer acts as an entry and exit point for all system data, much like the gateway node in the Gannet architecture. Users are responsible for ensuring that the data stored within the buffer is correct and that their task description programs use the data as intended. All memory allocations and data structure initialisations carried out on the host system are copied into device memory before the virtual machine begins execution. Similarly, all results are copied from device memory to the host system after the computation has completed.

The virtual machine intentionally follows the Gannet design closely in key areas. It is only the implementation details, due to hardware and platform restrictions, that differ significantly. Convenient libraries such as POSIX threads and the C standard library cannot be used.

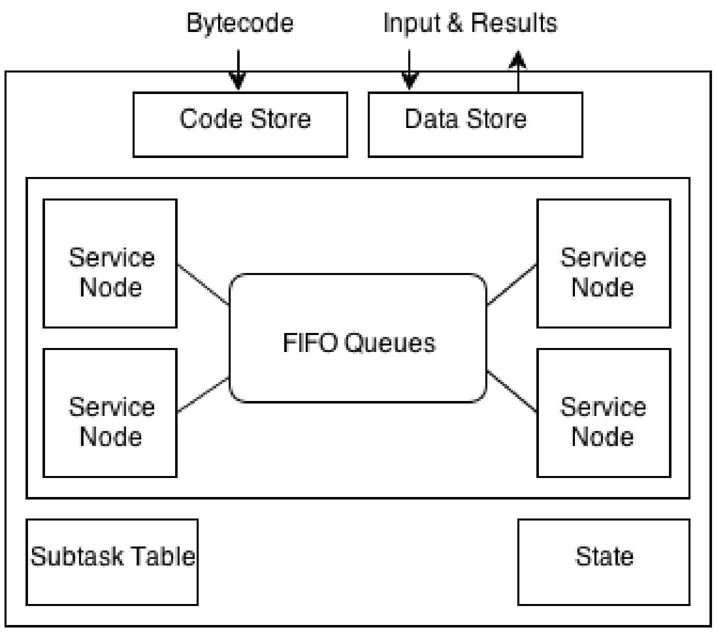

Figure 4.1: Virtual machine architecture.

4.2 Services

A service is a reconfigurable unit that consumes data, produces data and is offered by a hardware core. For the purposes of this project we can define a hardware core as a compute unit, with one or more processing elements, provided by an OpenCL device such as a GPU. Data exchanges between services should be programmable at high levels of abstraction and this is facilitated by the Network-on-Chip which allows every node to communicate with every other node in the system.

Each service node is similar to a function as it provides a specific user-defined functionality such as a mathematical computation or flow control service. It is this similarity that allows us to consider the task description program as a tree of nested functions and allows us to adopt a functional approach to composition[8]. By observing that all data, excluding constants, are the results of requests to other services, we can take advantage of the native parallelism support that a functional programming paradigm provides. This is particularly useful as it complements the natural parallelism of GPUs.

Service configuration is performed by a user before program execution. It involves implementing the functionality of every service used by the task description program and assigning each to one or more hardware cores.

4.3 States

To ensure fully parallel execution of all branches of the program tree a state machine computational model is used. At any given time the machine exists in one of three states; *read*, *write* and *complete*. Execution begins in a *read* state in which an entire reduction tree level is evaluated. It is during this state that service nodes process their received packets. Next, the machine enters a *write* state in which any packets sent during the *read* phase are forwarded onto their intended destination. The purpose of such a stateful separation is made clear in the following section.

The state continues to alternates between *read* and *write* until the entire tree reduction has been performed and the computation is complete, after which the state is set to *complete*. Upon completion of each *read* or *write* phase, control is passed back to the host program in which the state is checked and changed accordingly before once again executing the virtual machine kernel program which will now have new data to evaluate.

Executing a task description program can be visualised as a series of phases in each of which a level of a reduction tree is evaluated. This so-called reduction machine continues execution until the results propagates up to the root service request and the final result is obtained.

4.4 FIFO Queues

The role of the Network-on-Chip in the Gannet System-on-Chip architecture is essential as a medium for transferring packet-based data between services. However, the actual implementation of the Network-on-Chip will not impact the functionality of the system[8]. The role of the FIFO queues in the virtual machine is that of a Network-on-Chip. They facilitate packet-based data transfer between service nodes using the shared global memory of the GPU.

Each service node has N R_x and T_x queues where N is the total number of service nodes in the system. The R_x queues are used to receive packets from other service nodes and the T_x queues are used to transmit packets to other service nodes. By splitting the functionality of a single queue into two, race conditions are avoided. This is necessary as concurrent reads and writes by different service nodes into the same queue cannot be handled appropriately on a GPU as there are no means for mutual exclusion. It is essential that such race conditions are avoided as the system is designed from the ground-up with parallelism in mind.

While the virtual machine is in a *read* phase, a service node processes its received packets by reading from each of its R_x queues until they are empty. Processing may involve the creation of new packets which are destined for other service nodes. However, these packets are not written into the receivers R_x queues. Instead, they are written into the equivalent T_x queue for that node. Upon entering a *write* phase, all packets contained within T_x queues are moved into their R_x equivalents. This ensures that during the next *read* phase, each service node will be processing the packets created and sent during the previous *read* phase. It is necessary to not only have distinct read and write phases for race condition avoidance, but also to perform parallel execution of all reduction tree branches, albeit in an unspecified order.

To illustrate the operation of such a mechanism consider a system with three service nodes N_1, N_2 and N_3, each of which has three R_x queues R_1, R_2, R_3 and three T_x queues T_1, T_2, T_3 as $N = 3$. Assume all queues are empty and that we are currently in a *read* phase. To send a packet to N_2, N_1 would write the packet into N_2's T_1 queue. Similarly, to send a packet to N_1, N_3 would write the packet into N_1's T_3 queue. During the next *write* phase the packet in N_2's T_1 queue is moved into N_2's R_1 queue and the packet in N_1's T_3 queue is moved into N_1's R_3 queue. Now both sent packets will be in their destination node's R_x queue for processing during the next *read* phase.

4.5 Packets

Every service node in the virtual machine consumes and produces packets. The information that they contain is used to create subtasks, update existing subtasks and return results.

A packet, denoted *p(type, source, arg, subtask)* consists of a header and a payload. The header contains the packet type (*reference or data*), the packet source (identifier of the node who sent the packet), the argument position at which to return the result and the subtask identifier. The payload can either be a literal data value or the address of a service result in the data store.

- A *reference* packet contains a code address; the location of a subtask in the code store, compiled as a list of symbols. This compiled subtask is a single instruction to be executed by the virtual machine.
- A *data* packet contains the results of a service computation. The payload is either a constant value or the address of the result on the data store.

4.6 Symbols

A symbol is a multi-byte word which encodes operations, code references and constants. Depending on its kind, a symbol can have a variety of meanings and may encode additional fields. Symbols are produced by the Gannet compiler and group together to form a list of symbols that represent a single code instruction (subtask). The complete bytecode for a task description program consists of many subtasks that together make up the entire computational task.

- A *service* symbol encodes information about a single subtask including the service to request and the number of arguments N that it requires. The N symbols that directly follow this symbol in the code store represent the subtask arguments. Service symbols are used to create new subtasks.
- A *reference* symbol encodes information about a subtask argument, the value of which is the result of a service request. Encountering a reference symbol will result in the creation of a reference packet which will request a service be performed.
- A *constant* symbol encodes a constant value as a symbol. Like a reference symbol, it usually represents a subtask argument. However, it does not request that a service be performed – the constant value *is* the argument value.
- A *pointer* symbol encodes the address of a value in the data store. It is used to allow data packets to store an address into the data store within their payload.

4.7 Code Store

The code store is a dynamically allocated buffer that contains the code instructions to be executed by the virtual machine. It is populated using the individual symbols, grouped by subtask, that are parsed from the bytecode file that is supplied as input to the virtual machine. This file is the result produced from the compilation of a task description program and a system configuration by the Gannet compiler.

Within the virtual machine, the code store is used to create subtask records by parsing the information contained within the symbols that make up those subtasks.

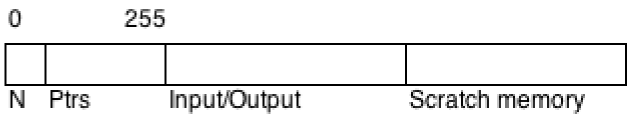

Figure 4.2: Data store layout.

4.8 Data Store

The data store is a user-populated dynamically allocated buffer that contains the program input data, reserves memory for partial results, reserves memory for complete results and allows remaining free space to be used for any purpose. Users are responsible for ensuring the correctness of their input data in addition to confirming that any references to such data within their task description programs are valid.

The size of the buffer depends upon the global memory available on the device. Generally, the data store will be the largest system buffer that will use all remaining global memory after other buffers and data structures have been initialised. This is an important design decision as users require access to sufficiently large continuous sections of memory so that they may perform computations on very large data sets.

Layout

As the data store is used for input, output and temporary results, it is important that each section of the buffer is easily accessible. Figure 4.2 illustrates the implemented layout. Each element of the buffer is a 32-bit unsigned integer.

- N is total number of memory sections that have been allocated for input or output data.
- *Ptrs* can contain the addresses of up to 255 allocated input or output data sections.
- *Input/Output* is the section of the buffer in which memory is actually allocated for input data or results.
- *Scratch* memory is the memory remaining after all input and output data memory has been allocated. It can be used for any purpose.

Users are required to set the value of N, allocate and initialise N areas of memory within the *Input/Output* section and then set the values of *Ptrs*$[0..N-1]$ to the address that is the start of each associated allocation. The address at location *Ptrs*$[N]$ will always point to the start of the scratch memory.

Consider a task description program that performs a matrix addition between a 2x2 input matrix and a 2x2 identity matrix. The user allocates two areas of memory, one for the input matrix and another for the result of the addition. Figure 4.3 shows the contents of data store in such a scenario.

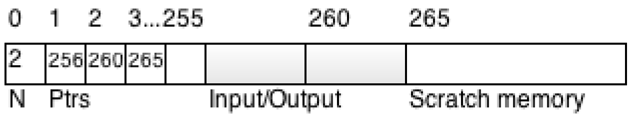

Figure 4.3: Example contents of the data store.

4.8.1 Data Services

We have covered data store implementation details however it is still unclear as to how a user would reference each allocated memory location from within a task description program. Fortunately, two services have been implemented that provide this functionality.

ptr

The *ptr* service takes a single argument, a constant I, that is an index into the *Ptrs* array. Using indirection, the service returns the address of the Ith allocated memory section by checking the value at $data[I + 1]$. The offset is necessary as the *Ptrs* array begins at data store index 1. This means that (ptr '0) returns the first address.

```
case M_OclGannet_MEM_ptr: {
  uint arg1 = get_arg_value(0, rec, data); // Get the first argument.
  return data[DATA_INFO_OFFSET + arg1]; // Return value at Ptrs[i+1]
}
```

Listing 4.1: Implemention of the ptr service.

const

The *const* service take a single argument, a constant I, that is an index into the *Ptrs* array. Unlike the *ptr* service, *const* returns the address of the 32-bit value located at $data[I + 1]$ and not the value contained within. Constant values that fit into a single 32-bit memory location need not be allocated in the input/output section of the data store.

```
case M_OclGannet_MEM_const: {
  uint arg1 = get_arg_value(0, rec, data);
  return arg1 + 1;
}
```

Listing 4.2: Implemention of the const service.

Usage

Although users should be encouraged to implement their own services, the inclusion of the data store along with both data services provides a convenient way to allocate and access memory. It is assumed that these services will be used frequently unless an ambitious user wishes to implement his or her own equivalents.

Listing 4.3 demonstrates how both services may be used in practice. Assuming the root task is a matrix multiplication, the first two arguments are pointers to the input matrices, the third is a pointer to the result location and the fourth is a constant value that represents one dimension of a square matrix.

```
(*
  (ptr '0)
  (ptr '1)
  (ptr '2)
  (const '3)
)
```

Listing 4.3: Example task description program that uses the ptr and const services.

4.9 Subtasks

The bytecode produces from the compilation of a task description program is partitioned into a list of packets, each of which contains a number of symbols. Each grouping of symbols represents a single subtask and together, these subtasks define the functionality of the complete program. It has been shown that for a task to complete a number of subtasks may have to be evaluated. It is therefore necessary to implement a mechanism that allows for the creation, updating and monitoring of subtasks.

4.9.1 Subtask Record

A subtask record is a data structure that contains all information regarding a single subtask. Each record is identified by a number of fields:

- The *service identifier* specifies which service to perform when the subtask is ready.
- A list of task *arguments* are stored which contain pointers to data or constants.
- For each argument, an *argument status* is maintained which determines whether the argument is present, pending or absent.
- The *subtask status* is used to determine whether the subtask has been recently created (new), is processing arguments (processing) or waiting on results from its own subtasks (pending).
- The *number of arguments absent* keeps track of how many argument values are missing from the record. The subtask is ready to perform its computation when all of its arguments are present.
- A *return to* address which is the service node to return the value to.
- A *return as* address which is the argument position within a specific subtask record.

4.9.2 Subtask Table

A subtask table is used to keep track of all pending subtasks that are currently active within the system. The table is simply a list of subtask records that are either in use or inactive. A stack of available subtask records is maintained which facilitates record recycling.

4.10 Performance Considerations

Local Memory Availability

All data structures and memory buffers used by the virtual machine are allocated on the global memory of the OpenCL device. For some, such as the data store, this is necessary as it is the only memory subsystem with enough capacity to meet the demand. However, random access to global memory is generally slow and system overhead could be reduced by using local memory, which is much faster. The overhead improvements would be insignificant in comparison to the performance increase that a developer could achieve by tailoring his programs to use such memory. It is for this reason that the entire pool of local memory is left available to users.

Memory Management

Usually, programs dynamically allocate memory from a large pool of memory known as a heap. However, while programming on a GPU you do not have access to such memory management facilities. In an OpenCL application all memory buffers are allocated and initialised in the host program before being transferred onto the GPU. It is impossible to increase the size of these buffers while executing code on the GPU therefore any dynamic memory allocations must be allocated from these statically allocated buffers. Implementing such a memory allocation system is not within the scope of this project and would would cause too much overhead. All memory must therefore be allocated in host program before the virtual machine begins executed. This why users are required to allocate and populate input/output data fragments of the data store.

Data Types

To save as much memory as possible many data types are implemented in a way which aims to waste minimal bits. For example, a packet is implemented using two 32-bit unsigned integers. The value of each packet attribute is stored within a range of these 64 bits and bitwise operations are used to access them. This saves a considerable amount of memory in comparison to an implementation that would use a 32-bit unsigned integer for each attribute.

Chapter 5

Testing

It is important that the virtual machine is implemented correctly otherwise task description programs will not produce their intended results. To provide a trustworthy implementation, an approach to testing was adopted that aimed to prove correctness both during and after development.

5.1 Test-Driven Development

The process of test-driven development was used throughout the development life cycle. It involved writing an initially failing test case for every desired function, feature or module, before producing code to pass that test. The importance of such testing cannot be understated as it was essential that each individual system component was implemented correctly before moving on to the next one. Used in conjunction with unit testing, test-driven development supplied the confidence to progress.

5.1.1 Unit Testing

The complete virtual machine kernel program is validated by a series of unit tests that act as the test-driven development test cases. Every function has one or more associated test cases and it is required that all tests pass before a program build is considered acceptable. Many programming languages provide convenient unit testing frameworks that accelerate this process, such as *JUnit* for *Java*. Unfortunately, no such framework exists for the OpenCL language. However, due to its similarity to the C programming language, unit testing frameworks that work with programs written C can be used for testing OpenCL kernels.

MinUnit

MinUnit is an extremely simple unit testing framework that can be used to test programs written in the C programming language. Unlike larger frameworks it does not provide a rich set of functionality. However, this characteristic is advantageous for the purposes of the project. Chosen for its simplicity, with a little ingenuity it can be used to test OpenCL kernels that are written using the OpenCL language.

```
// Tests that the function pkt_get_type correctly returns the type of a packet.
static char *test_pkt_get_type() {
  packet p = pkt_create(REFERENCE, 1, 2, 3, 4);
  mu_assert("FAIL: test_pkt_get_type [1]", pkt_get_type(p) == REFERENCE);
  return NULL;
}
```

Listing 5.1: A example test case.

5.2 Validation

Simply asserting that all unit tests pass is not enough to prove conclusively that machine functions correctly. Individual system components may act faultlessly however it is important to consider the interaction between each of them. To show that the virtual machine can successfully execute a task description program, the results of a number of test case programs are validated. Two such cases are outlined below, with the relevant service configurations given in listing 5.2.

```
...
NServiceNodes: 6
...
Aliases:
ptr:    c1.OclGannet.MEM.ptr
const:  c1.OclGannet.MEM.const
'+':    c2.OclGannet.MAT.add
'U':    c6.OclGannet.MAT.unit
'*':    c4.OclGannet.MAT.mult
...
```

Listing 5.2: Service configuration extract.

Test Case 1

```
(*
  (ptr '0)
  (ptr '1)
  (ptr '2)
  '2
)
```

Listing 5.3: A task description program that performs a single matrix multiplication.

The purpose of listing 5.3 is to validate the execution of a simple program. By using small input data sets we can confirm that the results produced are correct. The root task is a call to the * service which is offered by service node *c4* as defined in service configuration. This service performs a matrix multiplication and takes four arguments, for which the purpose of each is as follows:

- *Arguments 1 and 2* contain the locations within the data store at which the square matrix operands of the matrix multiplication are stored.
- *Argument 3* specifies the memory location at which to store the resulting matrix.
- *Argument 4* is the row or column dimension of each of the single square matrix operands.

Memory is allocated on the data store for the result and the following input matrices are used:

$$A = \begin{pmatrix} 1 & 2 \\ -3 & 11 \end{pmatrix} B = \begin{pmatrix} -2 & 4 \\ 7 & 1 \end{pmatrix}$$

After executing the task description program using the virtual machine, the resulting matrix *C* will be in the data store. By checking its contents we see that it is defined as:

$$C = \begin{pmatrix} 12 & 6 \\ 83 & -1 \end{pmatrix}$$

This program is simple enough that we can confirm the result by hand:

$$C = A * B = \begin{pmatrix} 1 & 2 \\ -3 & 11 \end{pmatrix} * \begin{pmatrix} -2 & 4 \\ 7 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} (1 * -2) + (2 * 7) & (1 * 4) + (2 * 1) \\ (-3 * -2) + (11 * 7) & (-3 * 4) + (11 * 1) \end{pmatrix}$$

$$= \begin{pmatrix} (-2 + 14) & (4 + 2) \\ (6 + 77) & (-12 + 11) \end{pmatrix}$$

$$= \begin{pmatrix} 12 & 6 \\ 83 & -1 \end{pmatrix}$$

By validating the results of listing 5.3 we have proven that:

- The way in which users specify input and output using the data store works.
- Subtask handling mechanisms work, allowing for the nested service calls to *ptr*.
- Data marshalling using the Network-on-Chip operates as intended.
- Task description programs can be correctly executed by the virtual machine.

As the size of the input matrices are so small, we have yet to prove that the virtual machine can deal with large data sets. This is important as GPU hardware is commonly utilised to solve very large computational problems, which is the likely use of the system. By using the same task description program (which we now know works correctly) and changing the scale of our input data we can prove that the virtual machine can work with large data sets.

In the previous example the dimension of each matrix was 2x2. This time, A and B will be two 1024x1024 matrices. Confirming the result by hand would take a very long time therefore a different approach is taken. A second matrix multiplication program was written using a general purpose programming language which computed a checksum by summing the contents of the resulting matrix. This checksum was then compared with the checksum that was computed after running the task description program. As both checksums were equal, it is proven that results match and that the virtual machine can handle large inputs correctly.

Test Case 2

```
(+
  (U (ptr '3) (const '4))
  (*
    (ptr '0)
    (ptr '1)
    (ptr '2)
    (const' 4)
  )
  (ptr '5)
  (const' 4)
)
```

Listing 5.4: A task description program that uses multi-nested service calls.

The purpose of listing 5.4 is to test a more complex computation that uses multiple nested service calls. The program performs a matrix addition on two operands; a unit matrix and a matrix that is the result of performing a matrix multiplication on two input matrices. If we use the same input matrices A and B that we used in the previous test case, we can again validate by result by hand.

$$U + (A * B) = U + C = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 12 & 6 \\ 83 & -1 \end{pmatrix} = \begin{pmatrix} 13 & 6 \\ 83 & 0 \end{pmatrix}$$

Similarly, we can increase the size of the input matrices and compute a checksum once more to confirm that the virtual machine operates correctly using large data sets. Again, we have proven that much of the core system works as intended. The key difference in this example is that both the *ptr* and *const* services are offered by the same service node *c1*. Also, it demonstrates how the arguments (subtasks) of the matrix addition service (root task) are evaluated in parallel. After calling the root task, service node *c6* will return a unit matrix, *c4* will multiply the two input matrices (creating further subtasks) and *c1* will handle both memory services. As the result is valid, we have proven that:

- Task delegation using nested service calls works.
- Arguments are evaluated in parallel using the subtask mechanism.
- A service node can provide multiple services.

To validate parallel evaluation, the machine should work if the task description program calls more services that the machine has compute units. The service configuration in listing 5.2 specifies that six service nodes (compute units) are required. By running both test cases on a device with N physical cores where $N < 6$, we confirm that the results are still correct. OpenCL solves the problem by creating and scheduling virtual compute units which are software cores.

Hardware

Although OpenCL allows the virtual machine to be run on any OpenCL device, the goal of this project was to run on a GPU. To ensure that this goal was met, all tests were performed on a dedicated graphics card.

Chapter 6

Evaluation

In chapter 5 we demonstrated how two task description programs successfully produced the correct results after being executed within the virtual machine. Both example programs, although small, are complex enough to conclusively prove that the implementation is correct. By validating parallel evaluation and delegation, we show how the Gannet architecture has been virtualised and that the virtual machine is indeed a parallel one. In addition, as these tests were carried out on a GPU, we can confirm that the goal of running the virtual machine on a GPU has been met.

Delegation

In the context of the virtual machine, delegation is the ability to perform nested service calls. The nested calls are treated as subtasks of the task (service call) that they are nested within. These subtasks are themselves tasks which may have associated subtasks. This recursivity means that we only need to prove that nested calls work one level 'deep'. Both programs used to demonstrate that the virtual machine is correct use nested service calls therefore we can conclude that the mechanisms used to handle delegation are correct.

Parallel Evaluation

To meet the objective of a *parallel* virtual machine, each level of the service call tree is evaluated in parallel. This means that each of the delegated subtasks of a single task are evaluated at the same time. In listing 5.4, service nodes *c1*, *c4* and *c6* are performing their associated services concurrently. Using this program, parallel evaluation is validated by confirming that each argument of call to service + is being evaluated at the same time.

Size

The implementation of the virtual machine kernel is extremely small, requiring only $\sim$1000 lines of source code. Its simplicity and modular design gives it an extensibility characteristic which may prove useful when implementing new features in the future. Due to its size, the virtual machine has the potential to be run within memory limited environments. When compiled using *gcc*, the size of the resulting binary is a mere 32kB. For reference, the existing Gannet virtual machine uses 300kB. It should be noted however that it has many more features and uses libraries that are unavailable for use on a GPU. Both the source code line count and compiled binary size indicate that we can consider the virtual machine implementation to be lightweight.

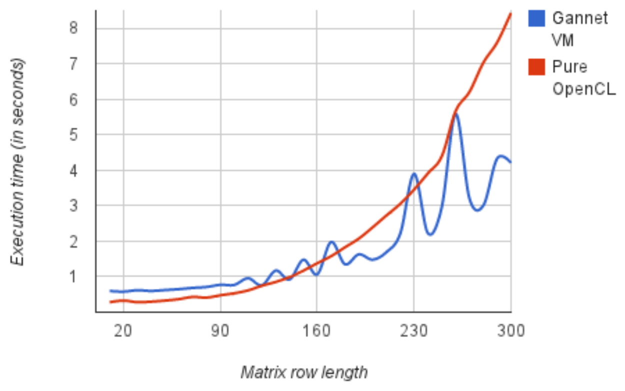

Figure 6.1: Execution time comparison between the virtual machine and a pure OpenCL equivalent.

Overhead

It was important that the virtual machine implementation cause minimal overhead as this would deter potential users who wish to perform performance critical computations. To determine if this criteria had been met, the difference in performance between a computation executed on the virtual machine and one executed in a pure OpenCL environment was analysed. The chosen computation was a single matrix multiplication, the service implementation of which is shown in listing 6.1. It was particularly suitable as in chapter 5 it was proven to execute correctly.

The task description program in listing 5.3 was run repeatedly and the execution time record (including the time taken to transfer input data.) Next, this process was repeated using the same program expressed as pure OpenCL application (listing 6.2). Table 6.1 contains a summary of the results.

The comparison of results (figure 6.1) show that there exists some overhead for multiplications which use small input matrices. Interestingly however, the performance difference increases in relation to the size of the matrices, with the virtual machine being the better performer of the two. We conclude the our objective has been met as what little overhead occurs is minimal.

```
case M_OclGannet_MAT_mult: {
  __global int *m1 = get_arg_value(0, rec, data);
  __global int *m2 = get_arg_value(1, rec, data);
  __global uint *result = get_arg_value(2, rec, data);
  __global int *sz = get_arg_value(3, rec, data);
  int n = *sz;

  for (int i = 0; i < n; i++) {
    for (int r = 0; r < n; r++) {
      int sum = 0;
      for (int c = 0; c < n; c++) {
        sum += m1[i * n + c] * m2[c * n + r];
      }

      *(result + (i * n + r)) = sum;
    }
  }

  return result - data;
}
```

Listing 6.1: A Gannet VM matrix multiplication service.

```
__kernel void mmult(__global uint *data, int n) {
  if (get_global_id(0) == 0) { // Emulate a single service.
    __global int *m1 = data + data[1];
    __global int *m2 = data + data[2];
    __global uint *result = data + data[3];

    for (int i = 0; i < n; i++) {
      for (int r = 0; r < n; r++) {
        int sum = 0;
        for (int c = 0; c < n; c++) {
          sum += m1[i * n + c] * m2[c * n + r];
        }

        *(result + (i * n + r)) = sum;
      }
    }
  }
}
```

Listing 6.2: An OpenCL matrix multiplication kernel.

Matrix Row Length	Matrix Size	Gannet VM Time	Pure OpenCL Time
10	100	0.598	0.281
20	400	0.580	0.328
30	900	0.619	0.281
40	1,600	0.599	0.297
50	2,500	0.624	0.328
60	3,600	0.650	0.368
70	4,900	0.685	0.431
80	6,400	0.711	0.411
90	8,100	0.773	0.477
100	10,000	0.769	0.533
110	12,100	0.957	0.614
120	14,400	0.759	0.747
130	16,900	1.173	0.853
140	19,600	0.924	0.989
150	22,500	1.479	1.172
160	25,600	1.067	1.382
170	28,900	1.979	1.579
180	32,400	1.353	1.829
190	36,100	1.631	2.073
200	40,000	1.481	2.392
210	44,100	1.693	2.728
220	48,400	2.218	3.055
230	52,900	3.900	3.459
240	57,600	2.224	3.918
250	62,500	2.974	4.397
260	67,600	5.585	5.673
270	72,900	3.201	6.213
280	78,400	3.008	7.034
290	84,100	4.335	7.608
300	90,000	4.209	8.439

Table 6.1: Execution times (in seconds) of a single matrix multiplication.

Gannet VM Variance: 1.829
Pure OpenCL Variance: 5.972

Applications

We have shown that the virtual machine can perform, with minimal overhead, parallel computations on a GPU. This is significant as if system performance was poor it is likely that potential users would be deterred from using the machine. Instead, its performance, in conjunction with the benefits that it provides, create an incentive for users.

The virtual machine allows programmers to write programs that will execute on a GPU. As discussed previously, this is already the purpose of GPGPU frameworks such as OpenCL and CUDA. A consequence of sharing the same purpose is that the virtual machine will share many of these framework's applications. That is, it will be used for the same reasons that developers choose to utilise GPU hardware. It therefore comes down to a choice between using an existing framework and using the virtual machine.

The virtual machine differentiates itself from frameworks by facilitating a high-level approach to GPU programming. It avoids the need for boilerplate code that does not express the problem to be solved. Implementation details are hidden within the services provided by the Gannet architecture which, although still user-implemented, allow task description programs to be concise and less complex than framework equivalents. In addition, the virtual machine provides an environment in which task parallel programs can be written and executed just as easily as data parallel ones can be using traditional kernel based GPGPU programming. The way in which pure OpenCL enables task parallelism is cumbersome in comparison.

Essentially, the virtual machine is a good choice for developers who would like to solve large computational problems on a GPU by expressing their computations in a high-level functional manner. In particular, parallel computations which can be abstracted into a number of services are especially appropriate.

Other than what GPGPU programming is commonly used for, potential virtual machine applications include:

- Functional languages, like the Gannet language, are commonly expressed in reverse polish notation. As normal algebraic notation can be trivially translated into reverse polish notation, those without a strong background in programming may use the virtual machine instead of an existing framework. It offers an abstraction that allows the user to focus on the computations.
- Due to its small memory footprint, the virtual machine may be used in environments that have extreme memory limitations.
- Developers may wish to run existing task description programs on the virtual machine if the ability to run on a GPU offers better performance.

Summary

Using specific examples we have shown the virtual machine can successfully execute task description programs on a GPU. In addition to meeting the main project objective, additional performance targets have been met. By running a computation on both the virtual machine and as a pure OpenCL application, it was demonstrated how the virtual machine has minimal overhead and in some cases outperforms the framework upon which it was built.

Chapter 7

Potential Improvements

Standard Services

Implementing a set of pre-defined standard services would increase the initial usability of the virtual machine and act as a 'standard library' for programmers to use. Currently, there are two memory services, *ptr* and *const*, which are used to access data within the data store. Offering additional 'built-in' services may help reduce user workload and act as a further incentive to use the virtual machine. The downside is that the size of the compiled binary will increase.

Turing Completeness

Currently, the overall system is not turing complete. Turing completeness can be achieved by implementing services that provide the missing features:

- There are no variables.
- There are no conditional constructs e.g *if then else*.
- There are no looping constructs.

Non-Tail Recursion

Due to time constraints non-tail recursive code is unable to be executed within the virtual machine's environment. Implementing this feature would allow developers to express their computations in an additional manner.

On-Demand Reconfiguration

The services offered by the Gannet system are re-configurable at run-time. Unfortunately, the virtual machine does not implement this feature as it was not within the scope of the project. The only service configuration available at the present time is that which occurs before the Gannet compiler uses it to create the system bytecode. It would be an interesting extension as services would have the ability to change function based upon some condition(s).

Stream Pipelining

GPUs are natural streaming processors which are designed to perform some operation on each element in an input stream before writing the results to an output stream. Gannet allows operations to be pipelined[8] on streams of data where the output of one operation is used as the input to the next. Implementing this feature would be particularly appropriate as pipelining require operations to be performed in parallel.

Premature Services

The result of a service call is currently always returned to some destination. For some services, such as one that does not modify data or produce new data, this may not be suitable. A future improvement would be to allow tasks to end before returning a result.

Sequencing

Subtasks will be evaluated in a random order due to parallel evaluation. Providing a way to sequence services in a specific order is useful in many ways. For example, consider two services that each print some messages to an output display (assuming the OpenCL device supports such a display). As there is no sequencing mechanism the messages will appear to interleave and be in the wrong order if they are evaluated concurrently.

Chapter 8

Conclusion

The aim of this project was to implement a Gannet virtual machine on a GPU that would execute task description programs in a parallel manner using the native parallelism of the GPU and the parallel design of the architecture. A virtual machine was produced using OpenCL that, using bytecode produced by the Gannet compiler, successfully executed the task description program that is represented.

Chapter 2 provided the necessary project background. First, we introduced GPGPU programming and explained how, by utilising OpenCL, the virtual machine could be implemented to run on a GPU. We then provided an overview of the Gannet System-on-Chip architecture that would be the target of our virtualisation. Next, existing virtual machine implementations were discussed in addition to other related works.

In Chapter 3 we described the project objectives in detail, emphasising the importance of performance and proposing a way in which to meet those objectives.

Chapter 4 outlined the design and implementation details of the project. The challenges faced by implementing on a GPU were discussed and the way in which they were handled described. It is the way in which these challenges were overcome that makes the project one of the first of its kind. In particular, using a state machine model and communication mechanism between service nodes using shared memory. The produced system could be accurately described as a parallel virtual reduction state machine.

Chapter 5 demonstrated the testing methodology with which the virtual machine implementation was tested and showed that the virtual machine was able to correctly execute task description programs.

Chapter 6 described the evaluation of the virtual machine. The results of the evaluation suggested that, overall, performance was good enough to consider the virtual machine a viable environment in which to execute programs on a GPU. Its service-based design and high-level development methods make it particularly unique in the domain of GPU programming.

Finally, further work was suggested that would improve the capabilities of the virtual machine.

Acknowledgements

The author acknowledges Wim Vanderbauwhede for his support, supervision and assistance throughout the duration of the project.

Appendices

.1 Obtaining the Source Code

The source code for the virtual machine is maintained using git and can be obtained by cloning the public Github repository located at:

```
https://github.com/Garee/vm
```

With git installed, enter the following command at a terminal to clone the repository in the current working directory:

```
git clone https://github.com/Garee/vm.git
```

.2 Compiling the program

A Makefile is supplied with the source code that can be used to build the program. An implementation of OpenCL must be installed for the program to compile. Please check your vendor for specific installation instructions.

- *make* - Build the program executable.
- *make clean* - Removes object files but leaves the executable.
- *make distclean* - Removes all files produced from compilation.
- *make test* - Runs the unit tests.

.3 Running the program

```
Usage: ./vm [bytecode-file] [n-services]
```

- *bytecode-file* - The file that contains the bytecode produced by the Gannet compiler.
- *n-services* - The number of service nodes that the virtual machine should use.

.4 Running the example

The source code has been submitted in a state which allows the task description program in listing 5.4 to be executed. It can be used to verify the answer to test case 2 in chapter 5.

```
./vm examples/4/test_ocl_gannet_4.tdc64 6
```

The service configuration file is examples/OclGannet.yml.

.5 User Guide

A user must complete the following steps to run a program within the virtual machine:

1. Write a task description program.
2. Implement the services used within this program.
3. Configure these services.
4. Compile the task description program and service configuration using the Gannet compiler to produce the input bytecode.
5. Provide any input data and allocate memory for results.

Service Implementations

All services are implemented within the service_compute function in the file kernels/VM.cl. To add a service, provide an additional case to the case statement using the associated service opcode and implement the service in the new block.

Input/Output Data

Inputs and outputs are handled by the data store. Read the relevant section in Chapter 4 for details on how to use the data store. Using this information, implement the populateData(...) function that is located within src/UserData.cpp. Example implementations have been provided. You will likely be using the provided *ptr* and *const* services within your task description program to access your data.

Important Files

- *src/VM.cpp* - The host program, access your results here.
- *src/UserData.cpp* - Contains the populateData(...) function which is used to populate the data store with input data.
- *kernels/VM.cl* - The kernel program that implements the virtual machine. Contains the service implementations in the service_compute function.
- *include/SharedMacros.h* - Contains configurable macros which may be used to tweak the system.
- *tests/vmtest.c* - Contains all unit tests.

Bibliography

[1] K. E. Batcher. Sorting networks and their applications, 1968.

[2] Nilsson J Ekman M, Warg F. An in-depth look at computer performance growth, 2005.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing, 2000.

[4] Khronos OpenCL Working Group. The OpenCL Specification, 2008.

[5] Randima Fernando Matt Pharr, Tim Sweeney. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.

[6] nVidia. nVidia CUDA Compute Unified Device Architecture Programming Guide, 2008.

[7] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing, 2008.

[8] Wim Vanderbauwhede. High-level Programming of Dynamically Reconfigurable NoC-based Heterogeneous Multicore SoCs.

[9] Wim Vanderbauwhede. Gannet: a Scheme for Task-level Reconfiguration of Service-based Systems-on-Chip, 2007.