RubiCL, an OpenCL Library Providing Easy-to-Use Parallelism on CPU and GPGU devices.

Aaron Cronin

Master of Informatics

School of Informatics
University of Edinburgh
2014

Abstract

I did some stuff that made the computer words happen faster.

Table of Contents

Todo list

Chapter 1

Motivation

1.1 Introduction

1.1.1 The need for parallelism

In the previous decade, a trend of ever-increasing hardware clock-speeds fuelled developer complacency. The often-cited “Moore’s Law” [1] suggested that our favourite algorithms will scale with demand, as executing systems increase in performance alongside complexity.

The stark truth is that this trend now seems to have lapsed (Figure 1.1). The latest generation of CPUs offer no significant clock-speed increases over the previous. Furthermore, improvements in per-clock performance are lacklustre. Physical hardware constraints are to blame for this disappointment. Namely, higher-than-anticipated levels of interference between subcomponents as a result of vastly increased circuit densities.

To combat this stall, hardware manufacturers have responded by increasing the number of independent execution units, or *cores*, present on produced system components. As a result, the total throughput available on a given device has continued to improve. Today’s data-driven economy generates computational problems of steadily increasing size. Therefore, software engineers must adapt to utilise this increased core-count.

Unfortunately, this tactic of improving performance by presenting a greater number of compute units is often incompatible with traditional programming approaches. The inapplicability of many tried-and-tested sequential architectural patterns forces engineers to consider new ones.

Constructing a parallel solution requires the study of new concepts, such as synchronisation and data dependencies. The next generation of software engineers are becoming familiar with these issues, but there is currently a significant knowledge-gap.

The necessary switch to parallel programming is not going as smoothly as desired. A short term solution for easing this transition is providing common developers with the capability to easily utilise all compute units within a system.

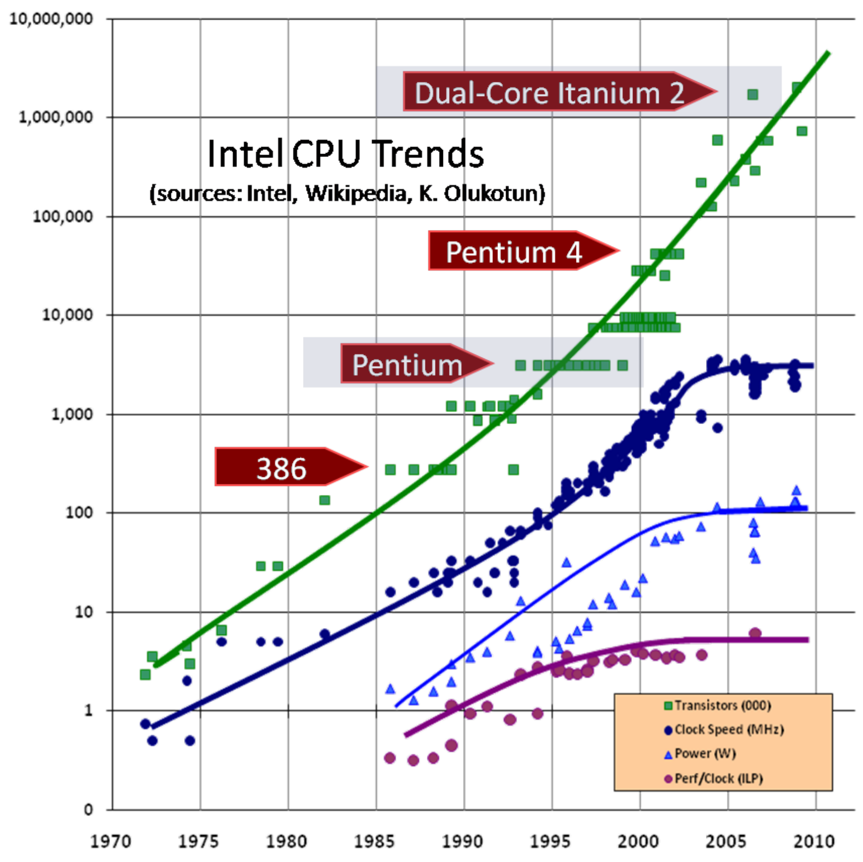


Figure 1.1: Graph demonstrating the recent plateau in clock-speed increase. Source: [2]

1.1.2 Prevalence of parallelism

As of 2014, many desktop machines contain 4-core CPUs, capable of scheduling 8 hardware threads simultaneously. Depending on whether they are aiming for performance or portability, typical laptop systems contain between 2 and 4 cores. Most commodity systems will attempt to improve performance by scheduling a user's tasks across underutilised cores, in order to avoid preemption. This still leaves sequential algorithms facing the bottleneck of a single core's rate of computation.

The other common source of potential parallelism within systems results from advances in computer graphics.

Graphics Processing Units (GPUs) are responsible for performing many computational stages of the graphics pipeline. They are highly parallel devices, tailored for high performance manipulation of pixel data. The popularity of playing games on home computers has led demand for increasingly powerful GPUs, producing a more responsive experience for consumers.

In recent generations, hardware manufacturers have explored combining specialised processing units, such as GPU, along with CPU on a single die. These hybrid devices, known as Accelerated Processing Unit (APU), often boast high transfer rates between components. They often allow modest graphical performance within a portable, low

System	Components	Discrete device count
Desktop (pre 2010)	CPU and GPU	2
Desktop	APU and GPU	3
Portable laptop	APU	2
Headless server	CPU	1

Figure 1.2: Components capable of parallel code execution, present in typical systems.

power device. As such, many laptops will contain an APU as instead of two discrete devices.

Several libraries have been developed to facilitate computation on hardware previously reserved for the graphics pipeline. As a result, GPUs capable of executing custom code in addition to their traditional roles are often referred to as GPGPUs. Lately, there has been a noticeable increase of interest in GPGPU computing and its suitability for common data-driven problems.

In short, most conventional computer systems purchased today will contain more than one available parallel processing device. A selection of common, parallel hardware configurations are detailed in Figure 1.2.

1.1.3 The holy grail of automatic parallelisation

Improvements in language design and compilation are areas of study hoping to increase the magnitude of parallel execution in the wild, without requiring user interaction.

Researchers are investigating the feasibility of discovering parallelism, inherent in user code, through analysis [3].

Whilst some breakthroughs have been made, progress is slow due to the massive complexity of the task. Languages with user-managed memory are hard to perform dependency analysis on correctly. In addition, the seemingly limitless variety of user programs greatly complicates any blanket solution.

It is likely that automatic parallelisation compilers will not become useful to a developer in the near future.

However, automatic parallelisation of code can be achieved if the scope of attempted transformation is reduced. In certain software paradigms, there is low-hanging fruit that can feasibly be parallelised automatically. This provides a stop-gap solution whilst research continues.

1.1.4 Embarrassing parallelism

Some problems are inherently parallel, containing no dependencies between subtasks. They can be dissected into a set of distinct work-units that can be executed concurrently. Such tasks are referred to as “embarrassingly parallel”.

Many *functional programming* primitives, such as `map` and `filter` are embarrassingly parallel. Any operation where the resultant state of each element in a transformed array only depends on a single input can be easily scheduled across many compute units.

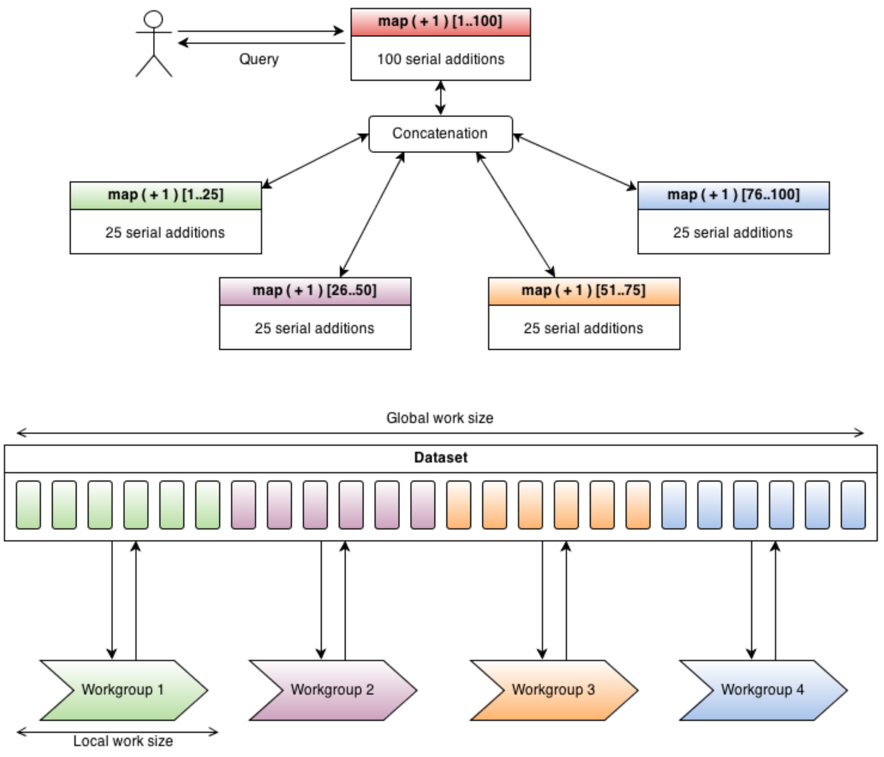


Figure 1.3: A partitioning of *map* computation over several compute devices.

Other tasks are more complicated to structure as the composition of concurrent subtasks. Computing the result to some tasks requires communication and synchronisation required parallel subproblems.

When manually parallelising code, programmers must be familiar with designing multithreaded algorithms. Subcomponents that cooperate must be produced in order to achieve processing speedup.

This effort is often unnecessary. Certain primitives are known to be suitable for parallel execution. These can be algorithmically scheduled across multiple compute units. By automating this translation and scheduling task, programmers can achieve increased throughput without the need for extensive studying and configuration.

1.2 Related Work

1.2.1 MARS

The MARS[4] project provides a MapReduce[5] runtime, executing on GPUs systems. It aims to take advantage of the significant computational resources available on GPGPU devices. To utilise available graphics hardware, it uses NVIDIA's Compute Unified Device Architecture (CUDA) library. It one of the first research papers presenting the idea of dispatching general-purpose tasks to GPGPU hardware.

MARS attempts to overcome key obstacles, faced when trying to produce a GPGPU computing platform. A GPGPU's high throughput, provided by its massively parallel structure, is only maintained if task idling is avoided. In addition, mapping of work units must avoid cores being under-utilised and producing artificial critical paths. Balancing tasks and scheduling them effectively is important. MARS demonstrates a procedure for load-balancing work-units across GPU devices in order to avoid such idling.

One shortcoming of MARS is its reliance on the MapReduce computation pattern for general purpose tasks. MapReduce is well suited to computation on large quantities of unstructured data. However, when execution is constrained to a single device host, the redundant infrastructure provided by the runtime is no longer beneficial. The communication pattern can produce unnecessary overhead.

Another disadvantage of MARS is the need to write the individual task code as CUDA source files. This is inconvenient for any programmer lacking prior knowledge of parallel programming. To utilise MARS effectively, you must first become familiar with CUDA programming.

Divergences Instead of taking a large-scale computation pattern and mapping it to GPGPU architectures, this project will start by providing interfaces to primitive operations that such devices are suited for. A suite of expressive operations, composed from efficient subcomponents will then be produced.

Following this work-flow should enable the finished library to achieve a significant performance benefits, as it centres around tasks that the target hardware is well suited to.

1.2.2 HadoopCL

HadoopCL[6] is an extension to the Hadoop[7] distributed-filesystem and computation framework. Again, it provides scheduling and execution of generic tasks on GPGPU hardware. Since it uses OpenCL, as opposed to CUDA, it also supports execution on CPU devices.

One benefit, for usability, of HadoopCL over MARS is the usage of the `aparapi` library[8] to generate required task kernels. Often, composing and scheduling custom OpenCL kernels requires significant amount of boilerplate. The purely-Java Application Programming Interface (API) allows programmers to skip a large portion of this boilerplate and focus instead on the task at hand.

The fact that the interface resembles threaded Java programming is another plus. However, it still requires writing functions with logic guided by the notion of kernel execution `ids`. This does not mitigate the need to become familiar with a new paradigm for data-parallel computation. Therefore, the system is still not suitable for novice users.

Divergences Instead of presenting an interface for programmers to write OpenCL code via shortcuts, the RubiCL project will boast the ability to automatically transform and parallelise simple computational primitives written in native code. This may suffer from reduced flexibility, but benefits from a significantly lower barrier-to-entry for inexperienced users.

Yet, constraining the user to the MapReduce computation pattern also reduces flexibility. The lack of arbitrary kernels for common tasks is not a significant drawback as long as any parallel task primitives are varied and composable.

1.2.3 CUDAfy.NET

The stated goal of the CUDAfy.NET[9] project is to allow ”easy development of high performance GPGPU applications completely from the Microsoft .NET framework”.

CUDAfy completely bypasses the need to write custom kernel code, either directly crafted or indirectly generated through an API. It performs code generation by examining the source code of dispatched methods at runtime, translating the Common Language Runtime (CLR) bytecode to generate equivalent OpenCL kernels.

CUDAfy benefits from significantly increased usability due to generating OpenCL kernels on behalf of the programmer. However, it does not have a high enough level of abstraction to avoid vastly altering the calling code’s structure. The programmer’s workflow is vastly altered when parallelising calculations. Anyone writing parallel CUDAfy code must still concern themselves with explicitly detecting onboard devices. In addition, the transfer of data to and from a CPU/GPGPU must be triggered manually.

Divergences Instead of requiring explicit device and memory management, this project aims to ensure that programmers do not have to concern themselves with such concepts in order to parallelise computation. It should be sufficient to solely provide the calculations that are to be executed, after stating that code should run on a particular device. Requiring any more interaction increases the mental taxation resultant from using the library.

1.2.4 Data.Array.Accelerate

Data.Array.Accelerate is a Haskell project[10], and accompanying library[11], providing massive parallelism to idiomatic Haskell code. It aims to approach the performance of ’raw’ CUDA implementations utilising custom kernels.

The library introduces new types for compute containers and built-in types are wrapped prior to inclusion in any computation. This allows the runtime to gather information about which datasets need to be transferred to compute devices.

It has received some significant optimisations[12] that target the inefficiencies present when an unnecessarily large number of kernels would be generated and executed, due to composition of functions.

Divergences A disadvantage of Data.Array.Accelerate for general-purpose computation is the relative difficulty often associated with becoming a competent Haskell programmer. The language diverges greatly from many mainstream languages. It requires programmers to state calculations in purely functional form.

The Accelerate library offers easy transition into GPGPU programming for existing Haskell programmers. However, people with little Haskell experience may struggle to construct valid code.

To counter this, a more forgiving non-purely-functional language will be chosen for this project. Using a language that is easy for beginners to pick up will allow more people to attempt parallelising execution of their calculations.

1.3 Synopsis

1.3.1 Recap of motivations for research

- Improvements in sequential execution performance are lacking, thus a switch to parallelism is necessary.
- There is a lack of software developers sufficiently experienced with parallel programming.
- Without parallel execution of code, much of the potential throughput of a modern system is wasted.
- Easing parallelisation of primitives will let novice developers achieve greater device utilisation.

1.3.2 Brief description of proposed solution

The proposed solution is a plug-in library that allows standard-library functions to be automatically executed in parallel, without complicating the calling code. This will allow investigation into the advantages of naively distributing computation over multiple compute-units.

By remaining as similar as possible to standard library code, and requiring no prior knowledge of parallel program construction, novice users will be able to benefit from any increased throughput.

The produced library should be assessed on ease-of-use and performance. Clearly demonstrating the benefits of the library will allow developers to recognise if and when its inclusion would be beneficial to a personal project.

Chapter 2

Overview

2.1 Project Aims

2.1.1 Improving the performance of dynamic languages when executing data-driven tasks

Dynamic, interpreted languages are commonly celebrated for their increased expressiveness over static, compiled languages. Often, they greatly reduce the amount of code that is necessary to perform common tasks. However, they continue to be overshadowed by optimised compilation of static languages, particularly for performance-intensive procedures. A typical performance divergence is shown in Figure 2.1.

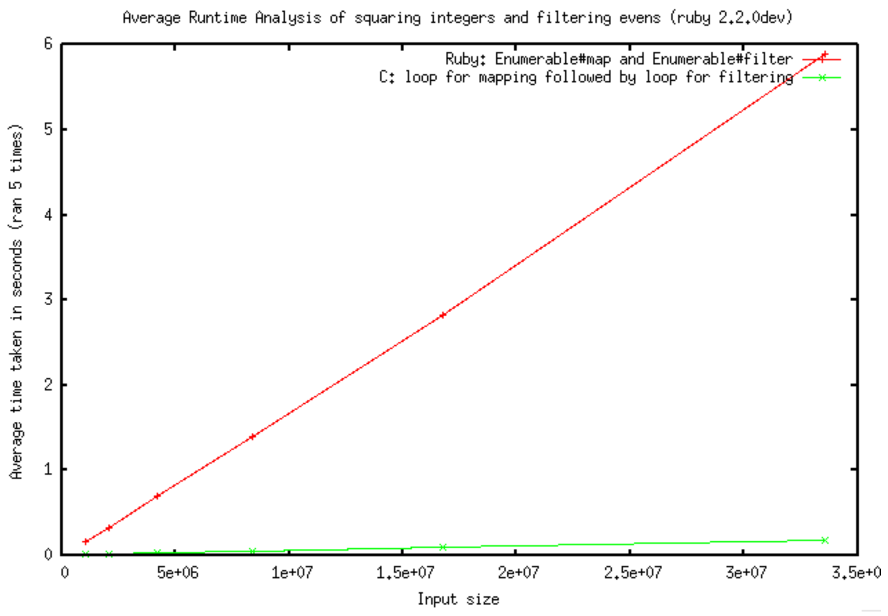


Figure 2.1: Graph demonstrating the significant difference in performance when operating on large datasets in C and Ruby.

RubiCL aims to investigate and mitigate any decreased performance when using the Ruby language for data processing. By producing a more efficient implementation of computational tasks, users will be able to tackle larger scale problems without needing to learn new toolchains.

Indicators of success Progress towards this goal can be evaluated by comparing the performance of Ruby implementations of tasks, before and after optimisation, to implementations in static languages. Further success can be measured by investigating whether increased magnitude computation terminates within reasonable time, due to the project's contributions.

2.1.2 Facilitating a larger scale of experimentation in a REPL environment by non-expert users

An interactive environment, such as a Read-Evaluate-Print Loop (REPL), is useful for rapid prototyping. It allows online processing of data without the need to produce all required code upfront, as shown in Listing 2.1. REPLs are absent from many languages. In supported languages, they allow the user to continuously query data and return intermediate results. Often, a quick turnaround between idea and response leads to more questions. This can enable an investigational attitude to computer programming.

Listing 2.1: A basic example of using a REPL environment for data analysis.

```ruby
dataset_1.mean
# =!> NoMethodError: undefined method 'mean' for Array

module Enumerable
  define_method(:mean) { map(&:to_f).inject(&:+) / size }
end
#=> :mean

dataset_1.mean
#=> 23.6

dataset_2.mean
#=> 23.2
```

By widening the scope of problems that can be evaluated within a REPL. RubiCL shall enable investigation into trends, unavailable available to novice users previously due to the scale of input data.

Indicators of success The completely library should be presented to novice analysts, users with mathematical insight but insignificant programming prowess. If they are able to easily answer queries about large datasets, the system's design will be judged as successful. As with the previous goal's evaluation, response time with in a REPL environment will be examined.

2.1.3 Exploring the extensibility of the Ruby programming language

Ruby has served as a suitable foundation for many Domain-Specific Languages (DSLs), including build tools[13] and web frameworks[14].

The language has open classes, whereby the structure of object classes can still be altered, even after definition ends. It also permits a variety of meta-programming techniques, allowing complicated code to appear misleadingly simple.

Listing 2.2: The Sinatra DSL for simple web programming hides complexity when writing basic web services.

```
require 'sinatra'

get '/hi' do
  "Hello␣World!"
end
```

Objects in Ruby are often regarded as *duck-typed*. This means that the system should care only about how an object behaves — “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.”[15]

Since function invocation uses a *method-sending* approach, the underlying implementation can be altered significantly as long as an expected dialog is presented to the runtime.

This project demonstrates integrating drastically different processing techniques into the language’s runtime. It achieves this without greatly affecting the code that a user must write in order to utilise them.

Indicators of success The integration will be successful if the interface for processing data remains consistent. Parallelism should be implicit and not requiring direction from the programmer. Again, user testing will evaluate this.

2.1.4 Effective code generation and reuse on the OpenCL platform

OpenCL can provide high throughput computation, often utilised by bespoke systems such as cryptographic hashers and video encoders. However, there is significant configuration and set-up code associated with each parallel tasks performed. Code reuse is difficult to achieve on the OpenCL platform due to the specificity of kernel execution.

Without techniques for reuse, advances made by one parallel project may not be applicable to others. Programmers writing parallel systems must implement all subtasks from the bottom up when constructing the full solution. As it is hard to incorporate the partial solutions of others, barriers to entry are further increased.

The project undertaken will attempt to recycle the partial solutions of primitives as much as possible. This allows investigate into how much reuse is possible, given an ideal system with a single author.

With code reuse, optimisations of a given subtask will improve all primitives utilising the component. This directs experimentation when searching for performance improvements.

Indicators of success Unfortunately, code reuse is often measured subjectively. Yet, the developer's opinion when reflecting on the development experience may provide useful insight. If code reuse techniques facilitate the development of this particular OpenCL project, it is likely that they may be beneficial to developers elsewhere.

2.1.5 Applicability to a variety of platforms, avoiding over-tailoring for a specific machine

The project should be package in a manner that facilitates installation onto a new supported system. In addition, it should achieve performance enhancement without having to be adjusted significantly by the user.

As a result, no assumptions about the specific hardware present can be made, apart from OpenCL support. This will allow the project to support a range of current and future compute devices.

Indicators of success Deployment of the system to new hardware will be attempted after the development phase has concluded. If the system remains performant and the deployment procedure does not require change, this is evidence of sufficient hardware agnosticism.

2.2 Leveraged Software Components

The project will provide functionality to users through utilisation of two previous bodies of software: The OpenCL library and the Ruby programming language.

2.2.1 OpenCL

The project requires interaction with heterogeneous processing devices present within a user's system. It achieves this via the hardware vendor's implementation of the OpenCL library.

OpenCL is an open framework for executing tasks, described by C99-syntax *kernels*, on a variety of devices. Suitable targets include a range of heterogeneous devices such as multi-core CPUs, APUs, and GPU from the majority of commodity hardware vendors.

The Khronos Group maintains and frequently updates the OpenCL standard[16]. Participating vendors include Advanced Micro Devices (AMD), Apple, Intel, and NVIDIA — although the quality and accessibility of implementations varies greatly.

A stated goal of the OpenCL project is to "allow cross-platform parallel programming". The underlying processing devices present on a system are abstracted, allowing code to be written without explicit knowledge of target architectures. This theoretically enables developers to write applications for a person system and then later scale execution to a massively parallel workstation, without significant code modification.

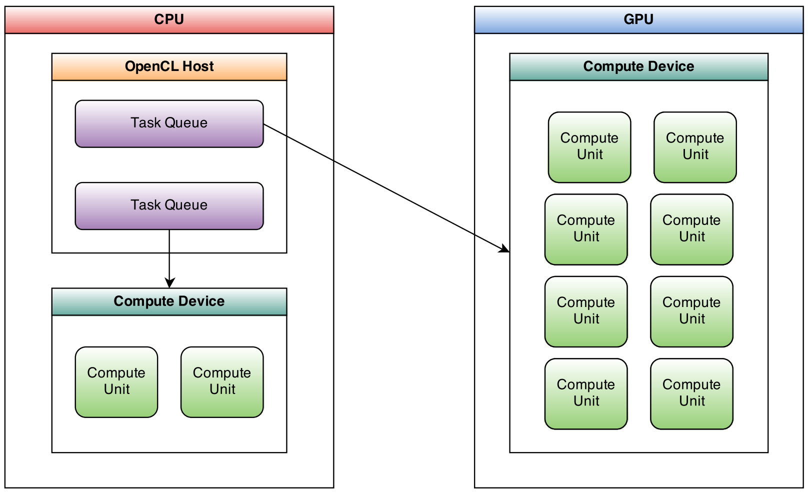


Figure 2.2: The OpenCL architecture model.

Architecture model As Figure 2.2 illustrates, the architecture model presented by OpenCL is as follows:

Host device The system's CPU. It interacts with an execution environment, responsible for discovering and selecting compute devices present on the system. The host device initialises one or more *contexts*, whereby any devices within a single context have access to shared task and memory buffers.

Compute devices The system's available processing devices, capable of scheduling OpenCL kernel work-groups. Before enumerating compute devices, the available *platforms* must be discovered by the runtime. Usually, there is a platform presented for each unique OpenCL supporting vendor with hardware installed. Devices are then retrieved on a per-platform bases, either filtered by type (CPU/GPU) or not.

Compute Units Discrete units of hardware present within a processing devices, capable of scheduling and executing OpenCL kernel instances. Kernel execution occurs across internal *processing units*, such as Arithmetic Logic Units (ALUs).

Execution Model OpenCL has a simple execution model, allowing both *coarse-grained* and *fine-grained* parallelism. Programmers write parallel code from the reference frame of a single kernel execution. Each instance orients itself only via access to its *local* and *global* `id`, expanded on shortly. Larger calculations are a direct result of cooperating kernel instances.

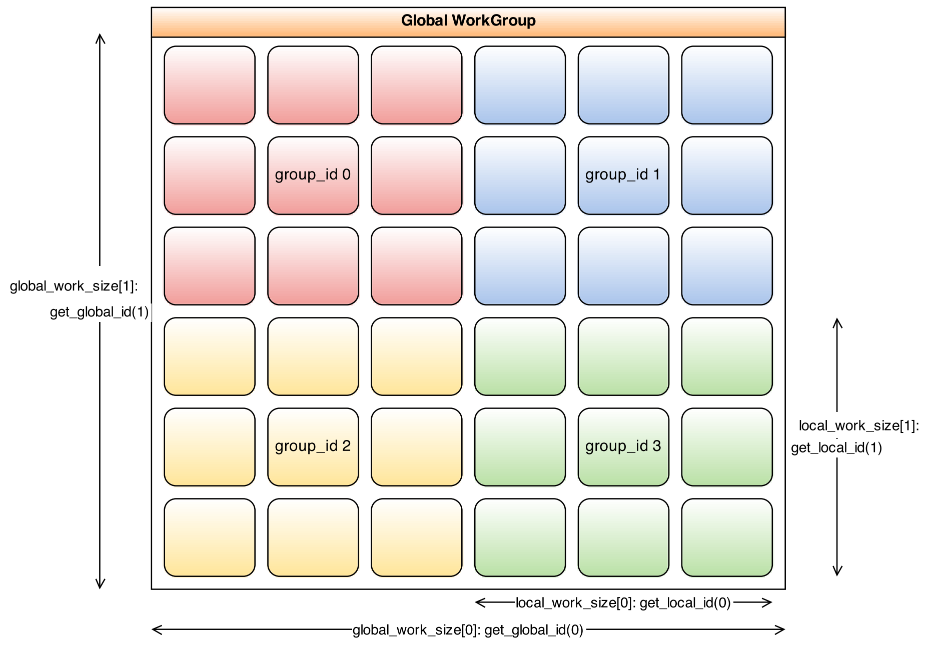

Global WorkGroup
group_id 0
group_id 1
group_id 2
group_id 3
global_work_size[1]: get_global_id(1)
local_work_size[1]: get_local_id(1)
local_work_size[0]: get_local_id(0)
global_work_size[0]: get_global_id(0)

Figure 2.3: The OpenCL execution model.

Kernel instances, scheduled for execution on compute devices, as referred to as *work-items*.

The OpenCL standard calls a collection of work-items a *work-group*. Work-groups are the unit of work dispatched to a device. The number of work-items within a group that a device should schedule is set by the programmer, providing the `global_work_size` parameter. Each kernel invocation is enumerated with a global `id`.

In addition to flat enumeration, the computation can benefit from the abstraction of work *dimensions*. For example, a calculation over 100 elements can represented as a 2-dimensional 10×10 calculation. When utilising dimensional abstraction, access to either unique global `id` or x, y offsets is available.

Higher dimensions are available for further structuring of tasks. It is clear that the spatial abstraction provided when there is a topological significant within the processed data. In these circumstances, the `local_work_size` parameter provides further benefits.

When `local_work_size` is specified, again possibly with dimensionality, the work-items are additionally divided into subgroups. Memory allocation can use the `__local` qualifier. In this case, data will reside in higher-bandwidth buffers that can only be synchronised between members of the same local work-group. With the addition of this second, local tier of memory, the OpenCL model hints at how efficient kernels should be constructed. Being aware of the positioning of data and its dependencies is key to developing efficient kernels, free of memory-synchronisation bottlenecks.

Much of the project's implementation effort concerning OpenCL will be mapping data required by common algorithms to efficient arrangements within device memory. This mirrors OpenCL development in general. The fact that this task is so laborious is a reason why heterogeneous parallel programming is still under-utilised in the wild.

Comparison with CUDA OpenCL is not the only computation framework choice when interacting with GPGPU devices. As mentioned earlier, a competing technology is NVIDIA's CUDA.

During the planning phase, RubiCL considered both choices. Ultimately, the decision was made to use OpenCL over CUDA for several major reasons:

Multiple vendor support CUDA is not an open standard. Its parallelism framework is only available on NVIDIA hardware. Using a library supported only by a single vendor to provide the project's hardware interfacing would lead to far fewer systems being able to benefit from accelerated processing.

CPU and GPGU execution By using OpenCL, RubiCL will be able to execute kernels on both CPU and GPGPU devices. This contrasts the GPGPU-only focus of CUDA. This greatly increases development convenience. Development can occur on a mobile laptop, functionally tested with its CPU. Afterwards, the library will be transferred to desktop workstations for performance testing on a variety of hardware.

In addition, this opens up the possibility of attempting code execution on both devices concurrently. This will investigate whether complete system-wide utilisation is beneficial for a single computation.

Disadvantages of choosing OpenCL There are several downsides to using OpenCL instead of CUDA. The programming model is generally agreed to be less developer friendly. This is perhaps due to the need to include far more abstraction over the range of target devices. In addition, due to the need for compatibility with several device families, OpenCL can be less performant out-of-the-box than CUDA. Advanced knowledge of how to tweak device-specific parameters to avoid execution bottlenecks can help avoid this.

Current state of OpenCL implementations A final major disadvantage of OpenCL at the moment, is the current quality of some vendor implementations. All vendors advertise themselves as being OpenCL-compatible when marketing their hardware. However, in reality it is harder than advertised to achieve a working system.

For example, NVIDIA, have made no attempt to hide the fact that they would much rather everybody used CUDA instead. They are so slow at releasing libraries that they are a full version of the OpenCL specification behind other vendors at the time of writing this report. Features that are supported also perform much worse than expected.

Intel are up-to-date with library implementations but only have non-Windows support if you have purchased non-consumer grade *Xeon* processors.

Hopefully these issues will be rectified if OpenCL continues to gain tracking in future. As a short term response, the RubyCL project has been implemented on well-supported hardware only: An Apple Macbook Air containing a Intel Haswell processor, and a desktop system containing an AMD CPU and GPU.

Apple and AMD both have high-quality OpenCL 1.2 libraries and seem to be the two companies most invested in increased OpenCL uptake. Apple have recently started encouraging desktop software developer to schedule suitable tasks on the GPU via OS X's Grand Central Dispatch (GCD).

2.2.2 Ruby

Language features Ruby[17] is a interpreted, dynamic language, supporting a variety of programming paradigms. It contains many features designed to increase its extensibility via meta-programming.

Execution centres around the creation of objects, every class inherits from at least `BasicObject`. Function invocation is triggered via sending the target object a *message*.

Upon receipt of a message, it is handled by the lowest level of an objects inheritance-hierarchy — the composed chain of class and module extensions that are mixed-into the object instance. A level will either respond to the message or, if unable to, propagate the request further up the chain.

A common meta-programming technique is to redefine how a module within the hierarchy responds when a method definition is missing. Instead of simply passing the message to the next level, it can inspect the name and arguments of the function, or its own state, and give a response. This cancels further progression of the request.

This flexibility is often used (or abused) to reduce the amount of boilerplate code written.

In addition to dynamically responding to received methods, objects are often substituted for objects of another class that have subtly different behavior. This will be successful as long as they respond the same method calls. Therefore, it is common practice to consider only the interface presented by interacting objects and not their individual types.

The benefit of this *duck-testing* is that the same series of of method requests, present in a line of code, can cause very different patterns of computation. If the response of a single link in the chain is altered, the programmer would be unaware, and unconcerned, as long as the expected pipeline result is returned.

This technique of redirecting computation will be explored by the RubiCL library. It allows a decoupling of the programmer's requests and the underlying, massively-parallel implementation.

Native extensions The latest versions of the Ruby language make it simple to produce native extensions. Extensions are provided via C shared objects that interact with the underlying Ruby Virtual Machine (RubyVM).

Listing 2.3: The C code defining a module with the ability to perform native actions.

```
#include "ruby.h"
```

```
#include "something_native.h"

/* All Ruby objects are of type VALUE and must be unboxed */
/* Every method takes self as an explicit argument */
VALUE
methodSomethingNative(VALUE self, VALUE int_param_object) {
    int param  = FIX2INT(int_param_object);
    int result = doSomethingNative(param);

    return INT2FIX(result);
}

void /* module_example is name defined in Makefile */
Init_module_example() {
    VALUE ModuleEx = rb_define_module("ModuleExample");
    /* Visibility, Module, Name, Method, Arg count. */
    rb_define_private_method(
        ModuleEx, "something_native",
        methodSomethingNative, 1);
}
```

Adding functionality is as simple as performing the heavy-lifting as one would in a pure C application. The `ruby.h` library allow the programmer to create a Ruby module or class with mappings between method names and underlying implementations.

Listing 2.4: A Ruby class utilising a native extension module.

```
require './module_example'

class NativeThing
  include ModuleExample

  def method_requiring_native
    something_native(1)
  end
end
```

The complication of converting between Ruby objects and basic C types is handled via macros, defined for all sensible conversions.

Once an extension has been compiled, the shared object file is `require`d and the object is available no differently to a pure Ruby implementation. In the case of RubiCL, modules for particular concerns are provided and then mixed-into classes that require native functionality.

Suitability as the project's target language The project decided to use Ruby as the target language as, alongside Python, it is often recommended to beginners for analytics and *data science*. This is perhaps due to the syntax being relatively straightforward and often self-documenting.

Unlike Python, Ruby's design is less opinionated about the *principle of least surprise*, and therefore makes it much easier to be drastically extended in capability while hiding complexity from any users.

In addition, the need for constant method-hierarchy lookup has been blamed for its poor performance. The potential for dynamic redefinition of methods complicates caching and can heavily impact compute-intensive tasks. This makes Ruby a suitable target for a project aiming to offer an optimised library for such tasks.

2.3 Project Progression

2.3.1 Timeline

Initial focus In the project's first year, most of the time was spent researching existing parallel frameworks. The project's initial goal was to present a *MapReduce* runtime. Therefore, systems like `Phoenix++`[18] and `StreamMR`[19] were evaluated.

Part-way through the year, during prototyping, it became clear that there are several disadvantages to producing (yet-another) *MapReduce* system:

- The runtime resource management is overkill on a singular, massively parallel machine. When isolated failure is unlikely, more lightweight processing paradigms can be used with greater efficiency.

- Previous projects have hit issues caused by GPGPU architecture. For example, tasks that emit tuples must be run twice: Once to count the number of tuples emitted, and again to actually produce the results. This is needed as OpenCL kernels do not allow dynamic allocation of memory.

- Since OpenCL compute devices execute only kernels, there are just two options for task specification:

  Firstly, users can specify tasks in OpenCL kernel form. This is terrible for system usability. Products such as *Hadoop* are successful due to features such as the *streaming API*, allowing code to be written in familiar languages when performing parallel tasks on tuple streams.

  Secondly, the system could translate an existing language into OpenCL kernels. This would allow users to stay within their comfort zone yet still utilise the parallel architecture of many modern systems. Unfortunately, this task is an enormous undertaking. Progress has made recently on generating CUDA executables from LLVM Intermediate Representation (IR), yet similar breakthroughs for the OpenCL platform are lacking.

Moving away from MapReduce There are clear benefits of utilising familiar languages for orchestrating parallel tasks, such as significant increases in usability. Yet, it is currently infeasible to translate the entirety of stated programs. With this in mind, the decision was made to lower the scope of direction, with the user stating only parallel subroutines and having OpenCL dispatch of said subroutines automated.

At the close of the first year, a prototype system was produced that allowed a user to perform `map` and `filter` tasks. Specification of tasks was provided by a string of C expressions that would be interpolated into stock kernels.

Listing 2.5: Example of prototype system workflow.

```
DataSet.create(
  name: :one_to_ten,
  type: :int,
  data: (1..10).to_a
)

FP::Map.create(
  name: :add_one,
  key: [:int, :i],
  function: 'i␣+=␣1;'
)

FP::Filter.create(
  name: :add_three_is_even,
  key: [:int, :i],
  function: 'i␣+=␣3;',
  test: 'i␣%␣2␣==␣0'
)

DEVICE = OCLDevice::CPU.get

DEVICE
  .load(:one_to_ten)
  .fp_map(:add_one).fp_filter(:add_three_is_even)
  .output
```

This proof-of-concept demonstrated that performance gains were achievable by performing computation outside the confines of the RubyVM.

However, evaluation of the prototype highlighted several flaws:

- Using the library was incredibly verbose. Creating named objects to represent each state of computation meant that a line of pure Ruby code could spawn tens of lines of library code when converted to parallel execution.
- Lots of redundant parameters were required by the system. There is no reason for a `map` task to specify it's input parameter type when the syntax can be checked by a compiler. The separate declaration and usage of element variables increased the potential for bugs, in addition to exposing implementation details to the user.
- A lack of task optimisation caused higher-than-necessary workloads. Two consecutive `map` tasks would require two passes over the data instead of just one, as the intermediate result was produced.
- Code quality was poor. This was mainly due to the combination of venturing into a previously unexperienced programming paradigm, and organic growth of functionality during rapid prototyping.

  A prototype with successful functionality is used to discover the requirements of a system. It is then much easier to redesign a new system, one that is much more elegant yet achieves the same results.

Learning from mistakes The system redesign at the start of the second year specifically targeted previously identified flaws:

- First-class function support allows the library's usage to mirror standard higher-order function usage. Since anonymous function are used, this removes the verbosity of many named objects requiring creation.
- Some parameters are no-longer required, or inferred, due to a change in internal design. For example, anonymous functions document the input parameter throughout the calculation, so specifying this separately is unnecessary.

  Another example is subroutine type information. The computation pipeline can now keep track of the buffer type at a given point in execution and use this to guide kernel creation, instead of requiring user-submitted type definition.
- By deferring and combining tasks, as documented in the *Design* chapter, unnecessary computation can be avoided.
- A redesigned architecture, alongside focusing on testing and ease-of-modification. This helps maintain the software quality of the replacement system, even as requirements shift.

2.3.2 Difficulties

Several unforeseen issues have slowed down progress in certain stages of the project's progression.

Initial development system The target GPGPU test-bed was originally a desktop system containing a NVIDIA *GTX 670* GPU. It also contained an Intel *Haswell* APU. Unfortunately, it became clear during the process of porting the codebase, initially developed solely on a MacBook, that the state of the required libraries was much worse than advertised.

Had the libraries for the system been available, the produced framework would allow task scheduling on either compute device within the APU, or the GPGPU.

The difficulties encountered were as follows:

- Intel's OpenCL implementation only allows access to the graphics-processing APU co-processor under two conditions: If the operating system is Windows 7/8, or if the device is a Xeon processor. Xeon devices are not present in affordable desktop systems. The owned test-system did not support the required socket-style.
- Development in Windows was not possible due to the need to build several system components. The latest Ruby language snapshot and the extension modules

necessary for the library to operate must be built, from source, during development. This process is still bug-ridden on non-*NIX systems.

Following these setbacks, the goal of utilising the APU's graphic subsystem in addition to the CPU was abandoned. *GNU/Linux* was installed on the desktop system. However, the situation was further hindered when it became clear that the capabilities of NVIDIA's OpenCL implementation were vastly overstated. With no support for OpenCL 1.2 and suboptimal performance on 1.1m the high-end GPU present would not provide as much of a performance boost as initially anticipated.

Replacement system In order to continue the project's goal of properly exploring the potential benefits of GPGPU programming, the decision was made to purchase an 'idea' hardware platform to continue development on.

The system, consisting of an AMD *FX-4130* CPU and an AMD *R7 260X* GPGPU, was ordered and assembled after the initial problems were encountered. This caused a slight stall in development. However, it was estimated that with hardware utilising a single, AMD OpenCL implementation, productivity would be vastly increased. In fact, this was the case, enough so to make the delay worth experiencing.'

2.4 Completed Work

2.4.1 Features

Complete overview o what was d brief but in sive.

Chapter 3

Design

3.1 System architecture

3.1.1 Interface

The produced system is able to interact seamlessly with existing Ruby code, via type annotation on collection objects.

Listing 3.1: Redirecting a computation through the RubiCL library via type annotation.

```
# Sequential stdlib code
(1..1_000_000)
  .map { |x| x + 15 }
  .select { |y| y % 15 == 0 }

# Parallel code using RubiCL
(1..1_000_000)[Int]
    .map { |x| x + 15 }
    .select { |y| y % 15 == 0 }[Fixnum]
```

When a user is sure that all objects within an `Enumerable` are of a single, basic type, they can append a type declaration to the container. This declaration lies within the method pipeline and states the equivalent C type, the object is then wrapped by the RubiCL execution environment.

Further method calls are captured by the `Device` instance handling the dataset, and pushed onto a work-queue.

Eventually, a result is requested, either by a user casting back to a Ruby object class, or by performing a terminal action such as summation. The work-queue is then optimised and mapped to OpenCL kernels, dispatched to the target compute device.

The produced wrapper solution for including additional functionality to the Ruby runtime is ideal. Programmers must only grasp the concept of annotating type-conversion at the beginning and end of any calculation pipelines. All other syntax of the library is identical to normally-written Ruby code.

Despite the simplicity of the library's presented interface, there is a lot of work going on behind the scenes. The technical details of which will be discussed in the *Implementation* chapter.

As an overview, the steps undertaken by the RubiCL library for the example given in Figure 3.1 include:

- Moving the dataset elements into continuous memory addressable by the compute device.
- Recording the loaded dataset type, to allow static type-system operations.
- Parsing the `block` argument of the `#map` task's bytecode and constructing an equivalent C99 expression.
- Parsing the `block` argument of the `#select` task's bytecode and constructing an equivalent C99 expression.
- Inserting a `Map` task at the beginning of the `TaskQueue` to convert from Ruby objects to C `int`s.
- Inserting a `Map` task at the end of the `TaskQueue` to convert from C `int`s back to Ruby objects.
- Simplifying the 4 tasks in the `TaskQueue` to a single, `MapFilter` task via *fusion*.
- Generating the OpenCL kernel required to perform the `MapFilter` task.
- Executing the produced kernel on the compute device, recording metrics.
- Releasing resources required by the OpenCL library during the task.
- Returning the resultant values as a Ruby array.

3.1.2 Software architecture

The library is constructed from the following set of modules and classes, alongside their responsibilities:

RubiCL Environment singleton and top level namespace. The library's functionality is included in an application by `require`ing this module. It handles the import of all sub-components of the runtime. Other responsibilities include storing versioning metadata and selecting which available device should be the default compute target.

Interface:

self.opencl_device Returns the current compute device. (Default: `RubiCL::CPU`)

self.opencl_device=(Device) Sets the current compute device.

CastAccess A module designed to extend container types with the ability to start a computation pipeline. For example, the `Array` built-in class is modified with `Array.class_eval { include RubiCL::CastAccess }`. This allows parallel primitives performed on `Arrays` to be executed by the compute device following an annotation, such as `[1, 2, 3][Int]`. Upon casting, the actual conversion operations performed are specified by the target class. This module is decoupled from implementation and provides purely syntactic enhancements.

Traditionally in Ruby, invoking the `[]` method of an `Enumerable` is only used for indexed access to members. The standard implementation supports receiving integer arguments and returns the element at the given offset. It also supports `Range` arguments and returns the corresponding continuous subset.

The decision was made that that massing a `Class` constant here is something that would never occur in common use. Therefore, the RubiCL library uses occurrences of this calling behaviour to indicate that a dataset should be wrapped.

Interface:

[](Type) Overridden on extended object to call the method provided by a `Class` argument's `rubicl_conversion` method on the current compute device, also providing the dataset it was called on. Behaviour when called with a non-`Class` argument is unchanged.

Target C-type classes An observant reader may notice that the constant `Int` is passed in Figure 3.1 when signalling that the container should be transformed into C type `int`s. This class is not defined within the standard library, instead `Fixnum` is the container for fixed-precision integers that can be encoded within a single machine word.

The `Int` class was constructed to represent the abstract type of C integers. In addition, the `Double` type has been defined for floating-point numbers.

Each C-type class defines how to transfer an input dataset to the compute device, via methods defined by the `BufferManager`.

Interface:

self.rubicl_conversion Provides the method and type arguments to call on the current compute device, alongside a dataset, in order to load it.

Native Ruby result classes At the end of the computation pipeline, results are retrieved either by casting back to a Ruby type, or by performing a reduction action such as summation.

The Ruby classes used to convert back to the calculation's result type are provided with the standard Ruby implementation: `Fixnum` and `Float`.

Mirroring the responsibilities of the C-type classes, additional static methods have been added to these classes to instruct the `BufferManager` how to return a result dataset for the given type.

Interface:

self.rubicl_conversion Provides the method to call on the current compute device, in order to retrieve the typed dataset.

BufferManager In order to prevent the `Device` class becoming a *god object*, manipulating the device buffer is performed by a service object. The `BufferManager` provides an interface to load objects, specifying their C-type, and later retrieve them. The type of the currently loaded buffer is then stored, to assist kernel generation for queued parallel tasks.

The manager also provides caching of the dataset to prevent unnecessary retrieval if no operations have been performed.

The ability to interact with an OpenCL buffer is provided by the `BufferBackend` native extension module.

Interface:

load(type: Type, object: Object) Makes the provided object addressable by the OpenCL compute device.

retrieve(type: Type) Retrieves the resultant object from the compute device address space.

access(type: Type) Returns a handle to the device address space, passed by `Device` when executing tasks.

Device An abstract superclass, providing all functionality of the execution context during method pipelines. Instantiated as a singleton, in either GPU or CPU flavour. The subclass overrides only the initialisation procedure, passing the correct device-type flags to the OpenCL API, and provides a means to later differentiate between device types. Knowing which kind of device a kernel will execute on allows specific optimisations, such as avoiding *bank conflicts* for `Scan` tasks occurring on a `GPU`.

Interface: Where possible, all methods return the device context to allow method chaining.

[]**(Type)** Used to signal the end of a computation pipeline. Sends the method provided by `Type.rubicl_conversion` to itself.

load_object(Type, Object) Delegates to the buffer manager.

sort Enqueues a task to sort the buffer.

zip(Enumerable) Flushes the current pipeline and then creates a tuple buffer from the result and the inputted `Enumerable`.

fsts Bifurcates a loaded tuple buffer, keeping only the first elements.

snds Like the previous method, but keeps only the second elements.

braid(&Block) Collapses a buffer containing a list of tuples into a list of single values, using the provided combination function.

map(&Block) Mutates all elements within the buffer using the provided function.

filter(&Block) Rejects elements from the buffer that do not pass the provided predicate function. Aliased also as `select` to be consistent with the Ruby standard library.

scan(Style, Pperator) Produces an array of intermediate results, equivalent to traversing the array and applying the reduction operator up until each point. Inclusive or exclusive option set via parameter, inclusive by default.

sum Returns the summation of all values in the buffer.

count(Value) Returns the number of times that the given value appears in the buffer.

LambdaBytecodeParser Receives an anonymous Ruby function during instantiation and returns a set of equivalent C expressions on demand. The details of this procedure will be explained in the *Implementation* chapter. This translation stage enables the library to operate when the user states a problem in standard Ruby syntax only.

Interface:

to_infix Returns the function supplied to the constructor in infix form, using C syntax.

Logger A singleton used to log key actions to the terminal or disk, facilitating debugging. The current log level set determines whether output will be produced. Enables debug mode to be toggled in a single location.

Interface:

loud_mode Causes any logged actions to be displayed in the terminal.

quiet_mode Ensures logged actions do not appear in the terminal.

show_timing_info= Toggles whether segmented timing analysis appears before produced computation results.

TaskKernelGenerator Instantiated with a `Task` object, the `TaskKernelGenerator` assembles an OpenCL kernel performing the task. It handles the majority of OpenCL kernel boilerplate, with the task only providing specific computational operations.

Interface:

create_kernel Returns the kernel source for the given task.

Task An abstract superclass representing a stage in the computation pipeline. Subclassed with the specific type of operation. Provides tracking of variables required, computation statements and each task's unique title.

Interface:

descriptor A pretty-printed description of the task. Provides its name alongside a summary of actions performed.

to_kernel Returns the full OpenCL source of the task, obtained through the `TaskKernelGenerator`.

fuse!(Map) Present on `Map` tasks, allows a following `Map` task to be combined with the current task.

fuse!(Filter) Present on `Filter` tasks, allows a following `Filter` task to be combined with the current task.

pre_fuse!(Map) Present on `MapFilter` tasks, prepends the previous `Map` task's statements to the current task.

post_fuse!(Map) Present on `MapFilter` tasks, appends the following `Map` task's statements to the current task.

TaskQueue Stores the entire computation pipeline of the current execution chain. Enqueued tasks are appended to the queue. When a result is requested, the entire queue is optimised and then dispatched in as few tasks as possible. The rules for queue optimisation are discussed in the *Implementation* chapter.

Interface:

push(Task) Adds a `Task` onto the end of the queue.

shift Removes the first `Task` from the queue and returns it.

simplify! Compresses the `TaskQueue` by performing *fusion* optimisations.

Example interaction Figure 3.1 shows the interactions between classes during a typical parallelised computation.

3.1.3 Interacting with hardware devices

Interaction with hardware devices present on the system occurs via native extensions. These extension modules are mixed-into device singletons, created when the library is first launched. Figure 3.2 shows the functionality of these singletons and their subcomponents.

Both `CPU` and `GPU` objects, tasked with managing device state, inherit from a common `Device` superclass. The main difference in their implementation is differing initialisation procedure. Having two device types allows target-specific optimisation by the code generator, shown later.

The `Device` subclasses delegate maintaining the list of tasks to a `TaskQueue` object. In addition, they lack the ability to call memory management functions on devices and instead trigger functionality via an instance of `DeviceService::BufferManager`.

Implementing all device logic that does not require hardware interoperability in Ruby made the system much easier to test. The time taken for the device control flow to execute is insignificant compared to the time taken for data processing. Writing this section in C would have been misguided as the performance benefits would not be worth the impaired rate of development.

3.2 Design choices

Include mo
design choi

3.2.1 Type annotation

When parallelising computation using the RubiCL library, a dataset is initially ‘cast’ to the C-type equivalent. To signify the end of a computation it is finally ‘cast’ back to the Ruby type.

This method of redirecting a method chain using a wrapped object is intentionally similar to `Enumerable#lazy` in Ruby’s standard library.

`Enumerable#lazy` allows computation to be deferred until is known how many results are needed. In some cases, such as the example presented in Figure 3.2, computation can be avoided when the results would be discarded.

Listing 3.2: Redirecting a computation through Enumerable#lazy.

```ruby
def side_effect_increment(x, str)
  puts str
  x + 1
end

(1..5).map { |x| side_effect_increment x, "Non-lazy" }
    .take_while { |x| x < 4 }
# => [2, 3]

(1..5).lazy
    .map { |x| side_effect_increment x, "Lazy" }
    .take_while { |x| x < 4 }
    .to_a
# => [2, 3]

# Non-lazy function evaluates function 5 times:
# >> Non-lazy
# >> Non-lazy
# >> Non-lazy
# >> Non-lazy
# >> Non-lazy
# Lazy function evaluates function 3 times:
# >> Lazy
# >> Lazy
# >> Lazy
```

Keeping the usage akin to a conceptually similar feature should make the library easier to get to grips with.

3.2.2 Eager or deferred task dispatching

During system design, the decision had to be made whether to eagerly evaluate parallel primitives or to buffer all requests and then dispatch when a result is requested. This choice is not straightforward as there are benefits to either option.

Advantages of eager dispatch

- The kernel build and execution stages can be pipelined. For example, this allows the code generation and compilation stages to execute on the host CPU while the previous task is executing on the GPU.
- Compute device can easily be changed mid-chain. Although it will suffer a performance penalty due to the need to transfer the data buffer, having the buffer always in a consistent state allows a device well-suited in a particular primitive to pick up where another left off.
- Simplicity. No need to study equivalence rules.

Advantages of deferred pipeline

- Fewer resultant `Tasks` to schedule. Since adjacent combinable tasks are fused, there is less work done by the OpenCL compiler and work-group scheduler.
- Fewer accesses to global device memory. In the setup phase of each kernel, the elements to be transformed are loaded from the global device buffer into local storage. When multiple tasks are combined into one kernel, the intermediate result remains in unsynchronised memory until the task finalizes. This causes much less stress on the compute-device's memory subsystem.

- Fewer work-units scheduled. It is a waste of iterations to have 3 `for-loops` each modify a collection when all operations could occur in a single loop. It is similarly wasteful to execute N work-units 3 times when only N are needed.

Justify choi
to defer
pipeline

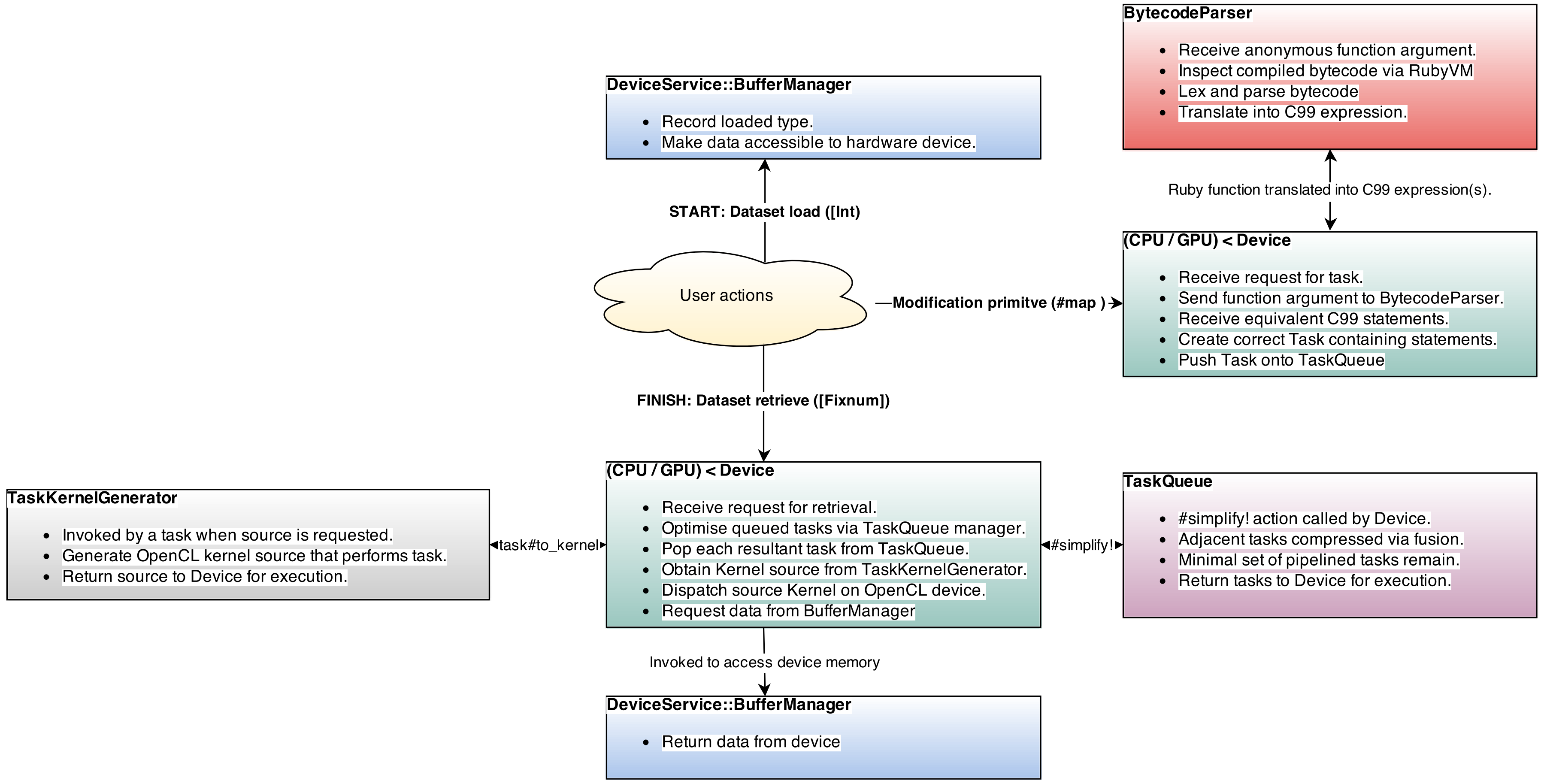

DeviceService::BufferManager
Record loaded type.
Make data accessible to hardware device.
START: Dataset load ([Int)
User actions
—Modification primitve (#map)
FINISH: Dataset retrieve ([Fixnum])
BytecodeParser
Receive anonymous function argument.
Inspect compiled bytecode via RubyVM
Lex and parse bytecode
Translate into C99 expression.
Ruby function translated into C99 expression(s).
(CPU / GPU) < Device
Receive request for task.
Send function argument to BytecodeParser.
Receive equivalent C99 statements.
Create correct Task containing statements.
Push Task onto TaskQueue
TaskKernelGenerator
Invoked by a task when source is requested.
Generate OpenCL kernel source that performs task.
Return source to Device for execution.
task#to_kernel
(CPU / GPU) < Device
Receive request for retrieval.
Optimise queued tasks via TaskQueue manager.
Pop each resultant task from TaskQueue.
Obtain Kernel source from TaskKernelGenerator.
Dispatch source Kernel on OpenCL device.
Request data from BufferManager
#simplify!
TaskQueue
#simplify! action called by Device.
Adjacent tasks compressed via fusion.
Minimal set of pipelined tasks remain.
Return tasks to Device for execution.
Invoked to access device memory
DeviceService::BufferManager
Return data from device

Figure 3.1: An overview of the interacting software components during the lifetime of a typical computation

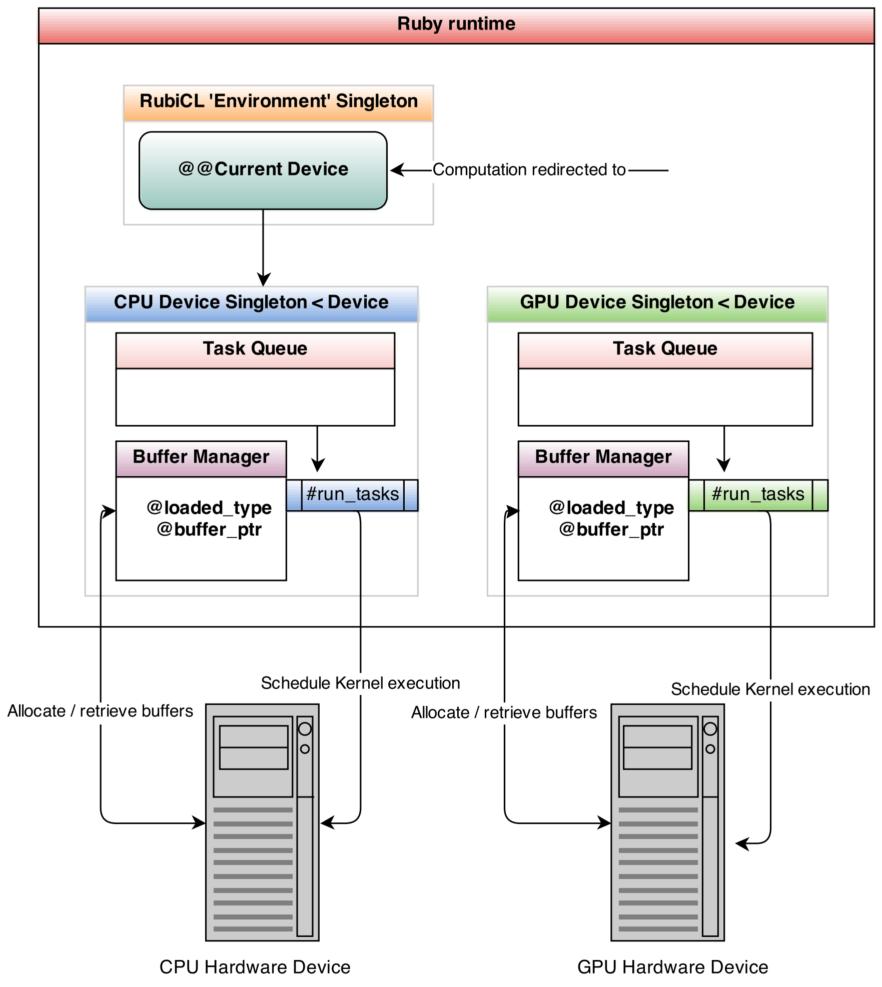

Ruby runtime
RubiCL 'Environment' Singleton
@@Current Device
Computation redirected to
CPU Device Singleton < Device
Task Queue
Buffer Manager
@loaded_type
@buffer_ptr
#run_tasks
GPU Device Singleton < Device
Task Queue
Buffer Manager
@loaded_type
@buffer_ptr
#run_tasks
Allocate / retrieve buffers
Schedule Kernel execution
Allocate / retrieve buffers
Schedule Kernel execution
CPU Hardware Device
GPU Hardware Device

Figure 3.2: The RubiCL runtime maintains singletons for each device, used to trigger management functions and execute kernels.

Chapter 4

Implementation

4.1 Differences between CPU and GPGPU hardware

Before detailing the specific algorithms and configurations used by the RubiCL project, it is necessary to contrast two of the heterogeneous target architectures.

CPU and GPGPU architectures have diverged significantly due to differing traditional applications.

CPU devices have had a long history of optimisation for sequential processing. As such, they have high clock-speeds and integrated hardware designed to increase instruction throughput; Modern processors rely heavily on *branch predictors* and identifying opportunities for speedup via *out-of-order execution*.

Conversely, the graphics pipeline tends to mainly utilise Single Instruction, Multiple Data (SIMD) operations. These consist of periods where the same few calculations are applied to vast quantities of data. With this execution pattern, complicated optimising hardware and higher clock-speeds are less favourable. Instead, hardware designers achieve incredible throughput by placing many massively parallel, yet simple, execution units on a single chipset.

In short, common GPGPU hardware lacks many optimisations targeting single-threaded performance. More significantly, devices lack the ability to branch by jumping during execution. A flat sequence of instructions is processed by the hardware scheduler. Conditional logic is provided by condition-variable flags set on individual instructions. These masking flags state whether execution of each statement within a branch segment should occur.

The inability to jump causes code that branches to be necessarily inefficient. Any branching logic within a kernel will leave some execution units idling until the code path converges again.

Luckily, GPGPU devices compensate by being exceptional at tasks resembling those that they were designed for: SIMD computation patterns. A high-end GPU, such as the *Radeon R9 290X*, can contain as many as 44 compute-units. Each compute unit is capable of scheduling 64 concurrent SIMD operations. At full utilisation, this vastly outperforms the raw instruction-rate of any CPU device. The amount of parallelism possible in a latest-generation desktop GPGPU is simply several orders of magnitude higher.

The goal of this project's implementation phase is to produce an easy-to-use library that presents significant throughput gains to an end-user performing common tasks.

4.2 Parallel primitives

4.2.1 Map

`Map` is a higher-order function that mutates all elements in a provided input vector by applying a function parameter. It can be used to concisely describe a uniform alteration. `Map` is simple to parallelise since no sharing of each individual thread's state is required.

Algorithm 1 *Map* higher-order function with sequential execution.

```
function SeqMap($f$, $A$)
    for all $a_i \in A$ do
        $a_i \Leftarrow f(a_i)$
    end for
end function
```

Algorithm design Upon examining the sequential implementation of the `map` primitive shown in Algorithm 1, it is clear that iteration i only reads and writes value a_i.

The dependency graph for a `map` of $\|A\| = 6$ is shown in Figure 4.1.

Figure 4.1: *Map* dependency graph

$$\begin{tikzcd}
a_1 \arrow[loop above] & a_2 \arrow[loop above] & a_3 \arrow[loop above] & a_4 \arrow[loop above] & a_5 \arrow[loop above] & a_6 \arrow[loop above]
\end{tikzcd}$$

When analysing data-dependency graphs, such as the one above, any partitioning that doesn't sever edges denotes a valid parallel strategy. Since Figure 4.1 contains no inter-node dependencies, it is trivial to schedule the task concurrently on many compute units. The `map` task is *embarrassingly parallel.*

Equivalent OpenCL kernel design The OpenCL execution model suggests performing tasks over a dataset by scheduling many distinct work-units. As a result, the

Algorithm 2 *Map* higher-order function with parallel execution.

```
function ParMap($f,A$)
    in parallel, for $a_i \in A$
        $a_i \Leftarrow f(a_i)$
    end parallel for
end function
```

side-effects of Algorithm 2's loop body are now provided by the result of many individual kernel-function invocations. Algorithm 3 describes an OpenCL kernel that performs `map` computation with a size $||A||$ work-group.

Algorithm 3 *Map* higher-order function in OpenCL kernel form.

```
$f \Leftarrow$ MutationFunction
function MapKernel($A$)
    DeclareVariables($f$)
    $i \Leftarrow$ GetGlobalID
    $a_i \Leftarrow f(a_i)$
end function
```

Alternative kernel investigation

Motivation After producing a system that performs `map` parallelisation akin to Algorithm 3, suspicion arose over whether it was excessive to schedule one work-unit per element. With traditional threaded programming, there is a significant performance cost when creating each parallel subroutine. In addition, with many kernel invocations all writing to offsets in the globally-available A, it was theorised that large numbers of competing memory access requests would hamper throughput.

Kernel adaption In order to ensure that any anticipated scaling issues were avoided, a new kernel design was constructed. The alternate design avoids scheduling a number of work-units greater than the number of compute-units present.

The adapted kernel, now performing `map` computation using a size $||CU||$ work-group, is presented in Algorithm 4.

Results After benchmarking the execution time of the kernels presented in Algorithms 3 and 4, no significant difference in performance was found. This suggests that the overhead for work-unit scheduling within the OpenCL framework is very low. It also suggests that simultaneous access to neighbouring global-buffer elements does not affect latency worse than strided simultaneous access.

Influenced by these findings, the decision was made to use Algorithm 3 for `map` tasks. This is due to the design being conceptually simpler, and therefore choosing the most basic solution that works well.

Verify this conclusion desktop

Algorithm 4 *Map* higher-order function in reduced-work-unit OpenCL kernel form.

```
$f \Leftarrow$ MUTATIONFUNCTION
$width \Leftarrow \lceil \frac{\|A\|}{compute\_units} \rceil$
function MAPKERNEL($A, width, \|A\|$)
    DECLAREVARIABLES($f$)
    $i \Leftarrow$ GETGLOBALID
    $i_{initial} \Leftarrow i \times width$
    $i_{next} \Leftarrow (i+1) \times width$
    for $i \in ((i_{initial} \ldots (i_{next} - 1) \cap (i_{initial} \ldots (\|A\| - 1))$ do
        $a_i \Leftarrow f(a_i)$
    end for
end function
```

4.2.2 Scan

`Reduce` is a higher-order function that takes an array and an initial 'result' value (usually an identity value) and then repeatedly applies a combining function to produce an output.

The final result is equivalent to repeatedly updating the initial value with the output of itself and the next set member using the combiner. Using this technique, the input array is consumed once while the result is cumulatively generated. Any associative reduction function can be parallelised to increase throughput.

A well-known example of reduction is when the initial value is 0 and the combining function is `+(x, y)`. This results in *summation* of an input dataset.

`Scan` is similar to `Reduce` in that it takes an input vector and a combining function.

Instead of returning the final result, Scan returns a vector that is equal to the intermediate values if the combining function was incrementally applied from one end of the dataset to the other. `Scan` can also exploit a highly-parallel architecture when supplied with suitable operators.

Algorithm 5 *Inclusive Scan* higher-order function with sequential execution.

```
function SEQSCAN($f, a_{-1}, A$)
    for all $a_i \in A$ do
        $a_i \Leftarrow f(a_{i-1}, a_i)$
    end for
end function
```

Algorithm design Unlike sequential `map`, iteration i now reads from both a_{i-1} and a_i in addition to writing a_i. This produces a data-dependency graph with greater connectedness, shown in Figure 4.2

It is clear that no partitioning of this graph exists that does not sever edges. Therefore, this task is not embarrassingly parallel.

Figure 4.2: *Inclusive Scan* dependency graph

$$\begin{tikzcd}
initial & a_1 \arrow[l] \arrow[loop above] & a_2 \arrow[l] \arrow[loop above] & a_3 \arrow[l] \arrow[loop above] & a_4 \arrow[l] \arrow[loop above] & a_5 \arrow[l] \arrow[loop above] & a_6 \arrow[l] \arrow[loop above]
\end{tikzcd}$$

However, this does not mean that all hope is lost. It is possible to efficiently parallelise a `scan` task, but it requires performing computation split over multiple *stages*. One method for achieving this is demonstrated in Figure 4.3.

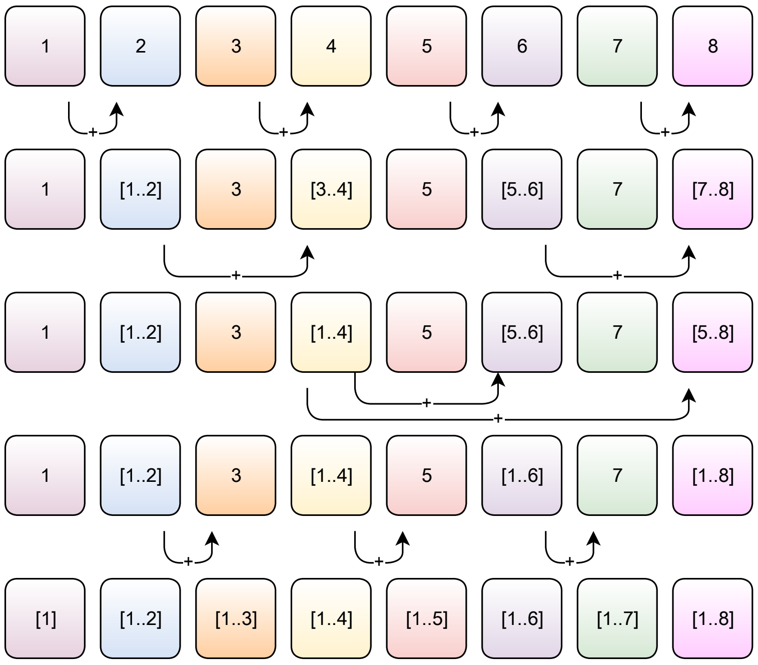


Figure 4.3: An example of parallelised *inclusive scan* using the *odd-even* algorithm, detailed in Algorithm 6

A parallel algorithm's *cost* is defined as its asymptotic runtime multiplied by the required number of compute-units.

The *odd-even* prefix sum algorithm can process a dataset of size n in $O(\log n) + \frac{O(n)}{\|CU\|}$ stages of execution. This gives a cost of $O(\|CU\| \log n) + O(n)$. Importantly, it is *cost-optimal*, meaning that its cost is equal to that of the best-known sequential algorithm, when $\|CU\| = O(\frac{n}{\log n})$. This is an entirely reasonable assumption given large datasets and the number of compute-units $(4-48)$ present on commodity OpenCL devices.

This discovery suggests that it is possible to increase the throughput of `scan` tasks significantly, by scheduling them across massively parallel OpenCL devices.

OpenCL kernel design

Mention di
fering algo-
rithm, code
provided by
Apple SDK

Algorithm 6 Odd-even style *Scan* higher-order function with parallel execution.

```
function ParScan($f$,$A$)
    $level \Leftarrow 2$
    while $level <= \|A\|$ do
        in parallel, for $l \in (level \ldots 2 \times level \ldots \|A\|)$
            $A_l \Leftarrow f(A_l, A_{l-\frac{level}{2}})$
        end parallel for
        $level \Leftarrow 2 \times level$
    end while
    if $level = \|A\|$ then
        $level \Leftarrow \frac{level}{2}$
    end if
    while $level > 1$ do
        in parallel, for $l \in (level + \frac{level}{2} \ldots 2 \times level + \frac{level}{2} \ldots \|A\|)$
            $A_l \Leftarrow f(A_l, A_{l-\frac{level}{2}})$
        end parallel for
        $level \Leftarrow \frac{level}{2}$
    end while
end function
```

4.2.3 Scatter

The `Scatter` primitives receives an input array A, an array of indices I, and an output array B. It updates B such that $B_{I_i} \Leftarrow A_i$. Put otherwise, it inserts the value given at offset i of A into B, at the position given by the value at offset i of I.

`Scatter` is useful for reordering a collection or projecting a subset of an input dataset into an output dataset.

Algorithm 7 *Scatter* primitive with sequential execution.

```
function SeqScatter($A$,$I$,$B$)
    for all $a_i \in A$ do
        $B_{I_i} \Leftarrow A_i$
    end for
end function
```

Permutation scatter It is important to draw attention to an important distinction in types of `scatter` operation. *Permutation Scatter* is defined as a scatter operation where all $i \in I$ are unique. Therefore, there are no two writes to the same destination in B. Other forms of `scatter` increase complexity, as rules that state how to handle write collisions within the transaction must be introduced.

Luckily, for this project's needs we only need to analyse the simpler *permutation scatter*. We can assume that no two writes to the same destination offset will occur.

Figure 4.4: *Permutation Scatter* dependency graph

$$\begin{tikzcd}
a_1 & a_2 & a_3 & a_4 & a_5 \\
b_1 \arrow[urr] \arrow[drr] & b_2 \arrow[urr] \arrow[drr] & b_3 \arrow[ul] \arrow[dl] & b_4 \arrow[ur] \arrow[dr] & b_5 \arrow[ullll] \arrow[dllll] \\
i_1 & i_2 & i_3 & i_4 & i_5
\end{tikzcd}$$

A data-dependency graph for a typical scatter operation is shown in Figure 4.4. At first, it may appear complicated. However, when nodes $b_i \in B$ are reordered by their data-source, a valid partitioning becomes clear. The result of this simplification is shown in Figure 4.5.

Figure 4.5: *Permutation Scatter* dependency graph, simplified.

$$\begin{tikzcd}
a_1 & a_2 & a_3 & a_4 & a_5 \\
b_5 \arrow[u] \arrow[d] & b_3 \arrow[u] \arrow[d] & b_1 \arrow[u] \arrow[d] & b_2 \arrow[u] \arrow[d] & b_4 \arrow[u] \arrow[d] \\
i_1 & i_2 & i_3 & i_4 & i_5
\end{tikzcd}$$

This suggests that `scatter`, when using unique indices, is *embarrassingly parallel*. Like `map`, this produces an easy-to-understand parallel conversion, shown in Algorithm 8.

Algorithm 8 *Permutation Scatter* primitive with parallel execution.

```
function ParScatter($A, I, B$)
    in parallel, for $a_i \in A$
        $B_{I_i} \Leftarrow A_i$
    end parallel for
end function
```

Equivalent OpenCL kernel design The kernel design is simpler than that of `map`, since function side-effects do not need to be included. It is presented in Algorithm 9.

Algorithm 9 *Permutation Scatter* primitive in OpenCL kernel form.

```
function ScatterKernel($A, I, B$)
    $i \Leftarrow$ GetGlobalID
    $B_{I_i} \Leftarrow A_i$
end function
```

4.2.4 Filter

`filter` is a higher-order function that applies a predicate function on elements of a dataset. It returns the subset of the input vector for which the predicate evaluates true.

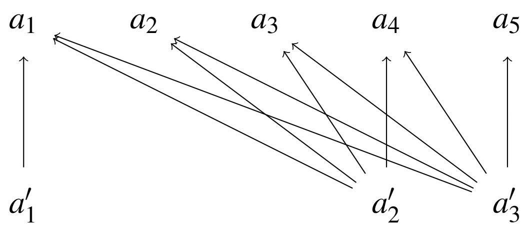
A sequential implementation of the primitive is shown in Algorithm 10.

Algorithm 10 *Filter* higher-order function with sequential execution.

```
function SeqFilter($A, predicate$)
    $Result \Leftarrow$ [ ]
    for all $a_i \in A$ do
        if $predicate(a_i)$ then
            Push($Result, a_i$)
        end if
    end for
    $A \Leftarrow Result$
end function
```

Checking the predicate is simple to do in parallel, since whether to keep each element depends only on the value of that element. However, producing and returning the subset is slightly more involved. The complication stems from the position of each kept item in the output array depending on the state of previous elements in the input vector.

Figure 4.6: *Filter* dependency graph.



The `filter` operation is clearly not *embarrassingly parallel*. However, there is no need to search for an involved parallel algorithm. We can construct an efficient `filter` operation by reusing the previously defined parallel primitives.

Composing a parallel solution The first stage of producing a parallel `filter` primitive is recognising the distinct data dependencies:

1. Whether an element is kept.

2. Where any kept element appears in the result.
3. The total number of elements kept, since we cannot dynamically allocate memory.

Identifying kept elements The information required by dependency 1 can be obtained by performing a `map` task on the dataset using the predicate function. The sole difference is that the result should be stored in a new buffer instead of overwriting the previous value.

Assuming we have an input vector A and a newly created predicate buffer P, we now know that any A_i should be kept if, and only if, P_i.

Knowing where to place kept elements Once we have produced a predicate buffer, via 1, we can easily derive the destination of kept elements (2). If the predicate buffer is stored as a vector of bit-flags, the number of kept elements at point P_i is equal to element i of the prefix-summation of P. This connection is illustrated in Figure 4.7

Figure 4.7: Using prefix-sum to determine insertion points.

$predicate =$ `keep_if_even`

Input dataset	0	1	2	3	4
Presence buffer	1	0	1	0	1
Prefix sum	1	1	2	2	3
Insertion point	0	-	1	-	2

The translation from prefix-summed buffer element to insertion point is just an off-by-one adjustment. Furthermore, since `map` followed by `scan` is cost $O(n) + O(n) = O(n)$, we can obtain these insertion points *cost-optimally*.

Counting the number of kept elements Following the calculation of dependencies 1 and 2, obtaining 3 is trivial. It is simply the final element of the prefix-sum buffer. This can be retrieved by a single lookup after the other sub-problems have been solved.

Complete solution By utilising `map`, `scan`, and a conditional-modified `scatter`, `filter` can be performed with cost $O(n) + O(n) + O(1) + O(n) = O(n)$. This is identical to the sequential algorithm presented earlier and is therefore *cost-optimal*. Again, this suggests that `filter` tasks can benefit from increased throughput when scheduled across multiple compute-units.

The combined process is demonstrated in Algorithm 11.

The parallel loop body is a modified version of the `scatter` task. The divergence is that it only performs scattering if the predicate element is set.

Algorithm 11 *Filter* higher-order function with parallel execution, composed from other primitives.

```
function ParFilter($A, p$)
    $P \Leftarrow$ ParMap($A, p$)
    $I \Leftarrow$ ParScan($P, +$)
    $B \Leftarrow$ Zeros($I_{\|I\|}$)
    in parallel, for $a_i \in A$
        if $P_i$ then
            $B_{I_i - 1} \Leftarrow A_i$
        end if
    end parallel for
    $A \Leftarrow B$
end function
```

4.2.5 Sort

lain bitonic

4.3 Management System

4.3.1 Function parser

The system's function parser is responsible for converting a supplied anonymous function into C syntax. The functionality of the parser is demonstrated in Listing 4.1

Listing 4.1: The *LambdaBytecodeParser* converts an anonymous function Ruby object into an array of C expressions.

```
foo = 3
a_function = ->(x){ foo * (2 + x) }
#=> #<Proc:0x007f976207ff48@(pry):12 (lambda)>

parser = RubiCL::LambdaBytecodeParser.new(a_function)
#=> #<struct RubiCL::LambdaBytecodeParser
#   function=#<Proc:0x007f9761c362c0@(pry):15 (lambda)>>

parser.bytecode
#=> " == disasm: <RubyVM::InstructionSequence:block in __pry__
# == catch table
# | catch type: redo   st: 0000 ed: 0016 sp: 0000 cont: 0000
# | catch type: next   st: 0000 ed: 0016 sp: 0000 cont: 0016
# |-----------------------------------------------------------
# local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0,
#                                  block: -1, keyword: 0@3] s3)
#   [ 2] x<Arg>
#   0000 trace            256                            (   22)
#   0002 trace            1
#   0004 getlocal         foo, 2
#    0007 trace            1
#    0009 putobject        2
#    0011 getlocal_OP__WC__0 2
#    0013 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
#    0015 opt_mult         <callinfo!mid:*, argc:1, ARGS_SKIP>
#    0017 trace            512
#    0019 leave"
parser.parsed_operations
#=> [3, 2, "x", "+", "*"]

parser.to_infix
#=> ["3 * (2 + x)"]
```

The conversion process occurs over three stages: dumping bytecode, lexing, and reconstruction.

Obtaining function bytecode The bytecode instructions produced by a compiled anonymous function object is provided by the `RubyVM::InstructionSequence` module's `disassemble` method. It returns a human readable string that includes all stack-machine instructions.

Lexing bytecode string Instructions of interest are extracted from the human-readable string. This is achieved via a regular expression containing a whitelist of keywords:

`/(?:\d*\s*(?:(getlocal.*|putobject.*|opt_.*).?))/`

The instructions are then tokenised, by the process detailed in Listing 4.2. The end result is a list of tokens representing stack-machine instructions, in Reverse Polish Notation (RPN).

The heavy reliance on regular expressions to parse bytecode is inelegant and fragile. However, with access only to a human-readable string, and a lack of any formal grammar, it was the best tool at hand to get the job done.

Listing 4.2: Tokenisation rules for lexing human-readable bytecode.

```
def translate(operation)
  case operation
  # First function argument
  when /getlocal_OP__WC__0 #{function.arity + 1}/
    'x'

  # Second function argument
  when /getlocal_OP__WC__0 #{function.arity}/
    'y'

  # Indexed bound variable
  when /getlocal_OP__WC__1 \d+/
    id = /WC__1 (?<i>\d+)/.match(operation)[:i].to_i
    index = locals_table.length - (id - 1)
    beta_reduction locals_table[index]

  # Named bound variable
  when /getlocal\s+\w+,\s\d+/
    name = /getlocal\s+(?<name>\w+),/.match(operation)[:name].to_sym
    beta_reduction name

  # Literal Zero
  when /putobject_OP_INT2FIX_O_0_C_/
    0

  # Literal One
  when /putobject_OP_INT2FIX_O_1_C_/
    1

  # Floating-Point Literal
  when /putobject\s+-?\d+\.\d+/
    operation.split('␣').last.to_f

  # Integer Literal
  when /putobject\s+-?\d+/
    operation.split('␣').last.to_i

  # Method Sending
  when /opt_send_simple/
    /mid:(?<method>.*?),/.match(operation)[:method].to_sym

  # Built-in Operator
  when /opt_/
    LOOKUP_TABLE.fetch operation[/opt_\w+/].to_sym
  else
    raise "Could␣not␣parse:␣#{operation}␣in␣#{bytecode}"
  end
end

def beta_reduction variable_name
  function.binding.local_variable_get variable_name
end
```

Expression reconstruction The final stage of the conversion pipeline. It requires converting RPN to infix form. Luckily, this part is less crazy. There is a well-defined algorithm for doing so, provided in Algorithm 12.

The conversion algorithm makes the assumption that all non-literals are functions with arity 2. This is justified since it covers all the mathematical operators required. Outliers include unary negation and method sending operations. These are detected and handled by an additional level of logic, omitted from the basic algorithm for brevity.

Algorithm 12 RPN to infix expression conversion.

```
function RPNTOINFIX($tokens$)
    $Stack \Leftarrow$ [ ]
    while LENGTH($tokens$) $> 0$ do
        $token \Leftarrow$SHIFT($tokens$)
        if ISLITERAL($token$) then
            PUSH($Stack, token$)
        else
            $right \Leftarrow$POP($Stack$)
            $left \Leftarrow$POP($Stack$)
            $combined \Leftarrow$COMBINE($token, left, right$)
            PUSH($Stack, combined$)
        end if
    end while
end function
```

4.3.2 Task queue

4.4 Functionality Testing

4.5 Performance Testing

Chapter 5

Analysis

5.1 Benchmarking

5.2 User Evaluation

5.3 Portability

Chapter 6

Results

Chapter 7

Conclusions

Bibliography

[1] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[2] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[3] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM Sigplan Notices*, volume 44, pages 177–187. ACM, 2009.

[4] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1918–1927. IEEE Computer Society, 2013.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[8] aparapi: Api for data parallel java. `https://code.google.com/p/aparapi/`.

[9] Cudafy.net. `https://cudafy.codeplex.com/`.

[10] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[11] Hackage: Data.array.accelerate. `http://hackage.haskell.org/package/accelerate`.

[12] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 49–60. ACM, 2013.

[13] Rake ruby make. `http://rake.rubyforge.org/`.

[14] Sinatra. `http://www.sinatrarb.com/`.

[15] Wikipedia: Duck test. `http://en.wikipedia.org/wiki/Duck_test`.

[16] Khronos group: The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`.

[17] The ruby programming language. `https://www.ruby-lang.org/en/`.

[18] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[19] Marwa Elteir, Heshan Lin, Wu-chun Feng, and Tom Scogland. Streammr: an optimized mapreduce framework for amd gpus. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 364–371. IEEE, 2011.