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

Insert something awesome here.

Table of Contents

Todo list

Chapter 1

Motivation

1.1 Introduction

1.1.1 The need for parallelism

Over the previous few decades, 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 Central Processing Units (CPUs) offer no significant clock-speed improvements over the previous. Furthermore, increases 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 relentlessly 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

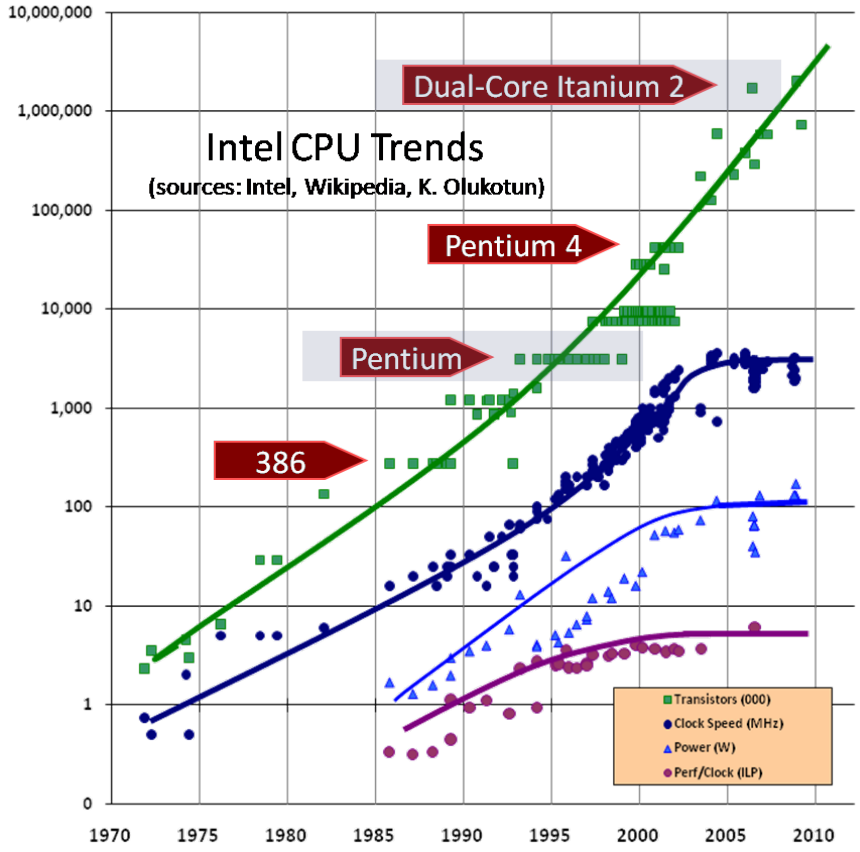


Figure 1.1: Graph demonstrating the recent plateau in clock-speed increase. Source: [2]

short term solution for easing this transition is to provide common developers with the capability to easily utilise all compute-units within a system.

1.1.2 Prevalence of parallelism

As of 2014, many desktop machines contain 4-core CPUs, capable of scheduling 8 hardware threads simultaneously through a technology termed *Hyper-Threading*[3]. 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 a system results from advances in computer graphics. Graphics Processing Units (GPUs) are usually responsible for performing various 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.

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.

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 power device. As such, many laptops will contain an APU 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 General Purpose Graphics Processing Units (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 [4]. 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`, are embarrassingly parallel, or parallelisable at some level of granularity. When transforming a dataset, any operation where the resultant state of each element in the output only depends on a single input can be easily scheduled across many compute units.

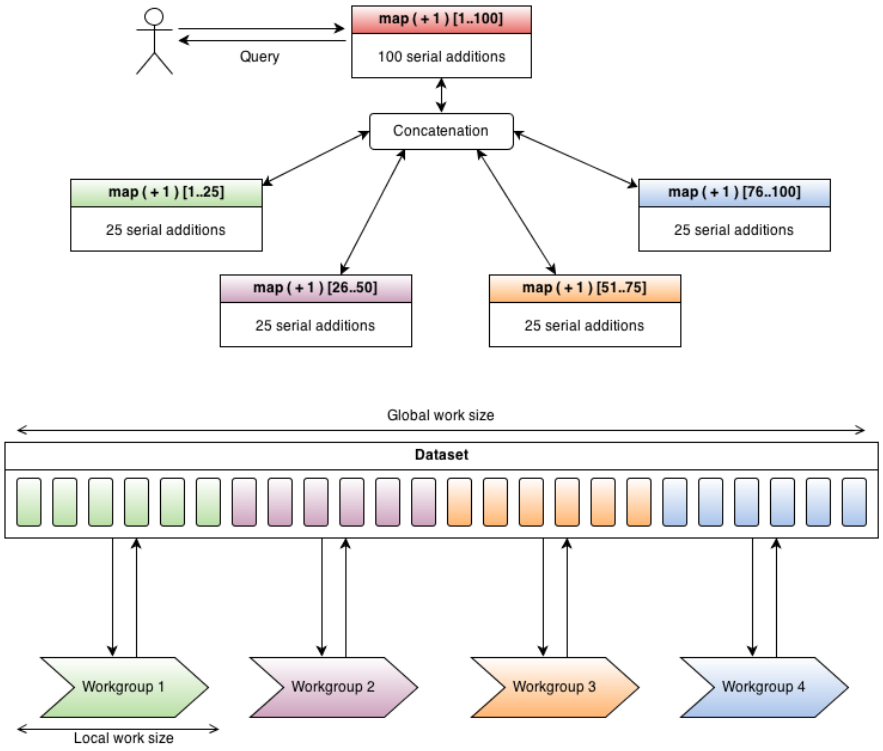


Figure 1.3: A partitioning of *map* computation over several compute devices.

Other tasks are more complicated, and require structuring as the composition of concurrent subtasks. In the worst case, computing a result requires communication and synchronisation between 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

State more influencing features of related work.

1.2.1 MARS

The MARS[5] project provides a MapReduce[6] 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 is one of the earliest 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[7] is an extension to the Hadoop[8] 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[9] 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[10] project is to allow “easy development of high performance GPGPU applications completely from the Microsoft .NET framework”.

Despite the name, CUDAfy.NET supports the OpenCL platform as a target back-end, in addition to CUDA.

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 CUDA or OpenCL kernels.

CUDAfy benefits from significantly increased usability, as it generates 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 still vastly altered when parallelising calculations. Anyone writing parallel CUDAfy code must 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 automate these tasks. This ensures 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[11], and accompanying library[12], 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. 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, and how to structure them in device memory.

It has received some significant optimisations[13] 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 certain 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 succinctness 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 the optimised compilation of static languages, particularly for performance-intensive procedures. A typical performance divergence is shown in Figure 2.1.

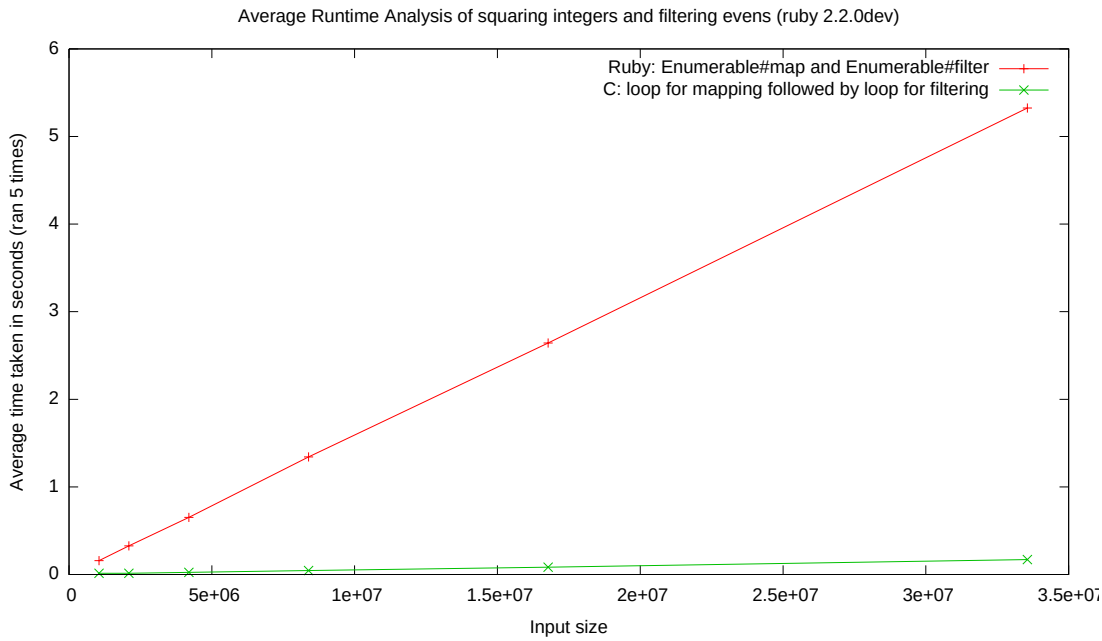


Figure 2.1: Graph demonstrating the significant difference in performance when operating on large datasets in C and Ruby.

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.

```
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 a larger scale of investigation. Analysis of particularly large datasets is currently unavailable to novice users, due to the amount of computation required.

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[14] and web frameworks[15].

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."[16]

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 investigation 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 the 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 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[17]. Participating vendors include AMD, Apple, Intel, and NVIDIA. However, 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 personal system and then later scale execution to a massively parallel workstation, without significant code modification.

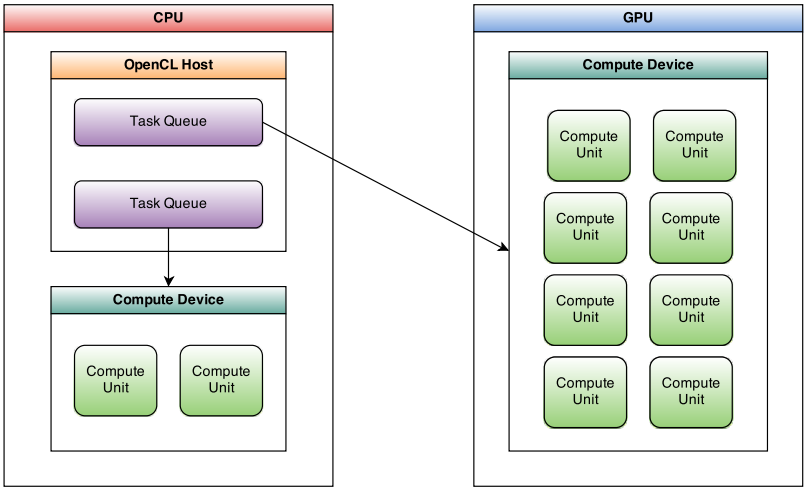


Figure 2.2: The OpenCL architecture model.

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 basis, 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

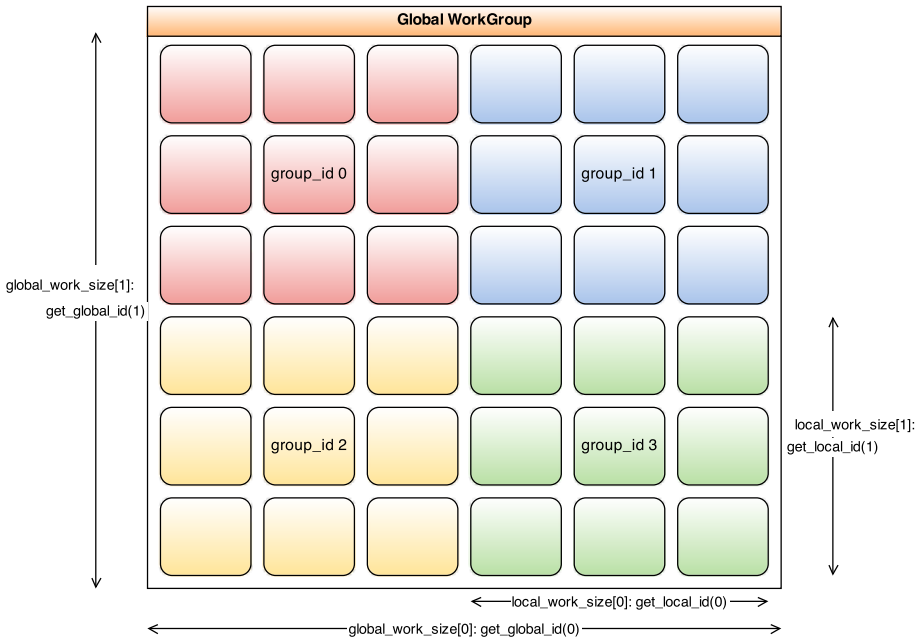


Figure 2.3: The OpenCL execution model.

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.

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 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 be 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. The spatial abstraction provided is particularly useful when there is topological significance to 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 on the OpenCL platform is still under-utilised in the wild.

Comparison with CUDA OpenCL is not the only available computation framework for 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 often 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 sufficiently slow enough at releasing libraries as to be 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 full non-Windows support if you have purchased non-consumer grade *Xeon* processors.

Hopefully these issues will be rectified if OpenCL continues to gain traction in future. As a short term response, the RubiCL 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[18] is an interpreted, dynamic language, embracing 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 by 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. Functionality is provided by 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.

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

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

```ruby
require './module_example'

class NativeThing
  include ModuleExample

  def method_requiring_native
    something_native(1)
  end
end
```

The required 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 `required` and the constructed 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 drastically extend its capability while hiding complexity from unaware 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 certain 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++`[19] and `StreamMR`[20] 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. Recently, progress has been made on generating CUDA executables from LLVM Intermediate Representation (IR). Similar breakthroughs for the OpenCL platform are lacking.

Moving away from MapReduce There are clear benefits of utilising familiar languages when 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
#=> [3, 5, 7, 9, 11]
```

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 its 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 causing higher-than-necessary workloads. Two con-

secutive `map` tasks would require two passes over the data instead of just one, as the intermediate result was produced despite remaining unused.

- 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.

Producing a prototype with full functionality is useful to discover the requirements of a system. It is then much easier to redesign a new system, one that is much more elegant yet achieving 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 creating many named objects.
- 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 any given point in execution. This can be used 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.
- The system architecture was redesigned, with a focus 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 using 1.1, 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 'ideal' 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. Luckily, this replacement system was successful enough to make the delay worth experiencing.

2.4 Completed Work

2.4.1 Features

Complete overview of what was done, brief but inclusive.

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 swallowed by the `Device` instance handling the dataset, and pushed onto a work-queue.

Eventually, a result is requested. This occurs 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 for maintaining usability. Programmers must grasp only the concept of annotating

type-conversion at the beginning and end of any calculation pipeline. 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 over the following 2 chapters.

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 `requir`eing 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, providing the ability to initiate 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 collection 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 assumption was made that that providing a `Class` constant as an argument 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 conversion method, provided by a `Class` argument's `rubicl_conversion`, on the current compute device, referencing 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 `ints`. 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 double precision floating-point numbers.

Each C-type class defines how to transfer a similarly typed 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 terminal 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 through 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 hardware 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 a method pipeline. 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 what type of hardware 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.

retrieve_integers Delegates to the buffer manager.

retrieve_doubles Delegates to the buffer manager.

sort Enqueues a task to sort the buffer.

zip(Enumerable) Flushes the current pipeline, 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?, &Block?) If provided with a value, returns the number of times that the given value appears in the buffer. If provided with an anonymous function, returns the number of values in the buffer that satisfy the predicate. If no arguments are provided, returns the length of 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 providing only 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 name.

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.

filter_fuse!(Filter) Present on `MapFilter` tasks, updates the filtering action to also require the following `Filter` task's predicate to be satisfied.

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 3.2 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 3.1 shows the functionality of these singletons and their subcomponents.

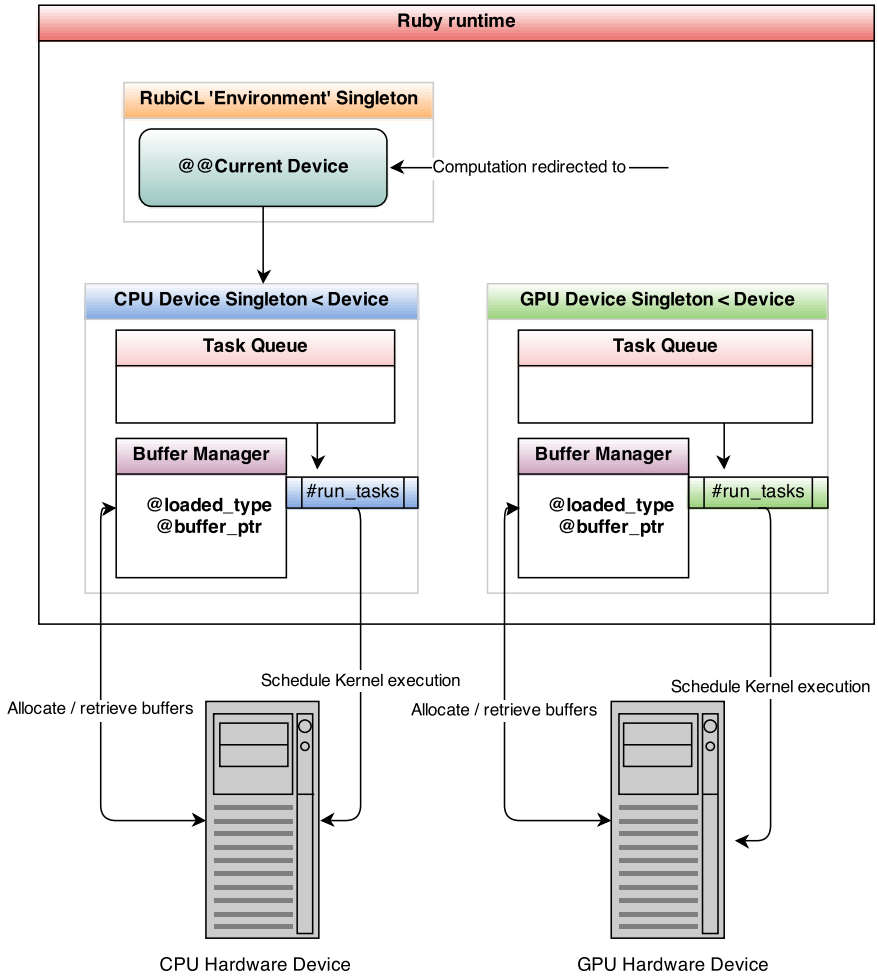


Figure 3.1: The RubiCL runtime maintains singletons for each device, used to trigger management functions and execute kernels.

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

Figure 3.2: An overview of the interacting software components during the lifetime of a typical computation

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 more design choices

Tuples design

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.

```
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 invocation evaluates 'side_effect_increment' 5 times:
# >> Non-lazy
# >> Non-lazy
```

```
# >> Non-lazy
# >> Non-lazy
# >> Non-lazy
# Lazy invocation evaluates 'side_effect_increment' 3 times:
# >> Lazy
# >> Lazy
# >> Lazy
```

Keeping the usage akin to a conceptually similar feature should make the library easier for inexperienced programmers 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 `Task`s 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-loop`s 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.

The chosen execution style was to defer task requests and then execute optimised tasks when a result is required.

This mirrors the approach taken by many *Object-Relational Mappers*, such as `ActiveRecord`, combining a pipeline of queries into optimal SQL.

The assumption was made that requiring multiple passes over the input dataset was significantly more expensive than the one-off code generation tasks. This was verified by timing information collected during empirical testing.

Chapter 4

Implementation

4.1 Differences between CPU and GPGPU hardware

Before detailing the specific algorithms 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.

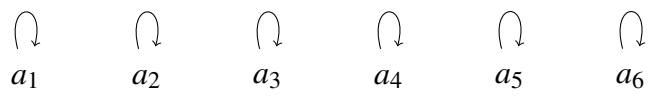
Algorithm 1 *Map* higher-order function with sequential execution.

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



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 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 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

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.

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

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.

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.

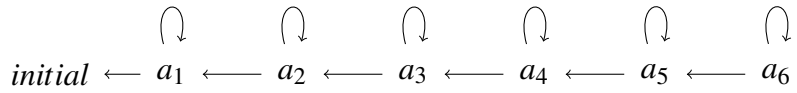
Instead of returning the final result, Scan returns a vector that is equal to the intermediate values if the combining function was incrementally applied from one end of the dataset to the other. `Scan` can also exploit a highly-parallel architecture when supplied with suitable operators.

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

```
function SEQSCAN(f, a_{-1}, A)
    for all a_i ∈ A do
        a_i ⇐ 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

Figure 4.2: *Inclusive Scan* dependency graph



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

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.

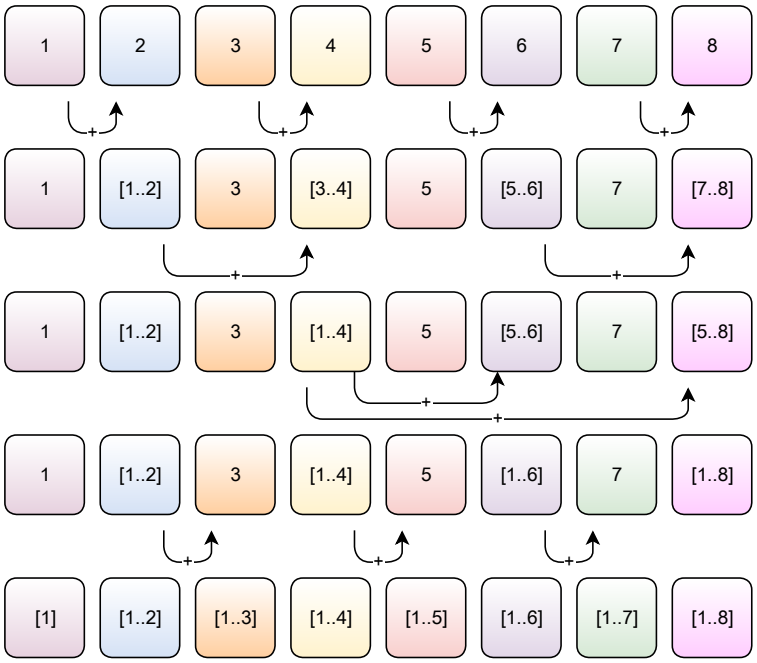


Figure 4.3: An example of parallelised *inclusive scan* using the *odd-even* algorithm, detailed in Algorithm 6

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 In fact, the kernel and calling algorithm utilised differs from *odd-even* scan in that it is designed to perform better on typical OpenCL compute devices.

It was adapted from an example provided within the Apple OpenCL SDK, modified to allow arbitrary element types and operate on buffers within the RubiCL environment.

Explain bank conflicts and mitigating their effects?

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 $l \in (level \dots 2 \times level \dots \|A\|)$
      $A_l \Leftarrow f(A_l, A_{l\_\frac{level}{2}})$
    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 $l \in (level + \frac{level}{2} \dots 2 \times level + \frac{level}{2} \dots \|A\|)$
      $A_l \Leftarrow f(A_l, A_{l\_\frac{level}{2}})$
    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.

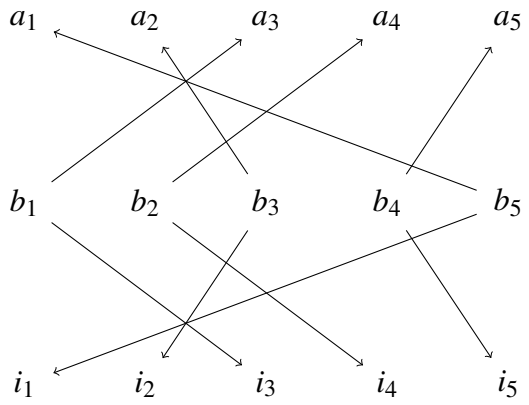
Algorithm 7 *Scatter* primitive with sequential execution.

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.

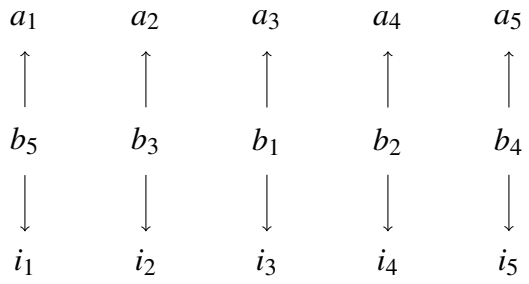
Luckily, for this project's needs we only need to analyse the simpler *permutation scatter*. We can assume that no two writes to the same destination offset will occur.

Figure 4.4: *Permutation Scatter* dependency graph



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.



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 ⇐ GETGLOBALID
    B_{I_i} ⇐ 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.

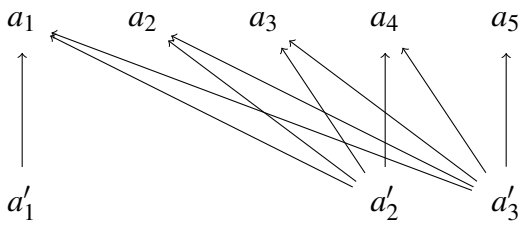
A sequential implementation of the primitive is shown in Algorithm 10.

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

```
function SEQFILTER(A, predicate)
    Result ⇐ [ ]
    for all a_i ∈ A do
        if predicate(a_i) then
            PUSH(Result, a_i)
        end if
    end for
    A ⇐ 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 significantly more involved. 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, `map`, `scan`, and `scatter`. This insight transforms a hard problem into one much easier to solve.

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 Count

The `count` function is very similar to the aforementioned `filter` primitive. Instead of returning the subset of a dataset that passes a predicate, it returns how many times the predicate was passed.

To avoid writing extra code providing counting functionality, we can re-use many components of the `filter` implementation. Namely, the `map` task that sets predicate bits for each element, followed by the `scan` task to yield the number of kept elements. Reduction can be performed in less work than a `scan` primitive, as the intermediate values are not required. However it is asymptotically identical, therefore the `scan` primitive can be reused to reduce developer workload. Further work can include replacing this suboptimal lack of specialisation with a faster reduction task.

4.2.6 Sort

Explain bitonic sort

4.3 Management System

4.3.1 Converting between Ruby and C objects

Explain this. Macros, bitshifting tagged pointer etc.

4.3.2 Transferring data to and from device

Explain this. Pinned memory vs write-buffer etc.

4.3.3 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, are 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_.*|branch.*).?))/
```

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
  # Conditionals
  when /branch/
    LOOKUP_TABLE.fetch operation[/branch\w+/].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
```

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

Expression reconstruction The final stage of the translation process. It requires converting RPN to infix form. 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 mathematical operators required by the library. 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.

Handling conditionals Conditional operators, such as &&, complicate the reconstruction process. Instead of modifying the value of a stack-machine, they may perform a branch. Whether or not this occurs depends on the `truthyness` of the current value. This is caused by the optimisation of short-circuiting boolean calculations.

Luckily, at the point that the branch is possibly triggered, no mathematical operators will mutate the previous value directly. This means that the difficultly of handling branching can be sidestepped by simply abandoning the current expression, ending in a conditional, and starting a new stack. All expressions produced are then just combined in order after the token stream has been exhausted.

4.3.4 Task queue

The `TaskQueue` management system buffers all deferred tasks, scheduled during the computation pipeline. It is responsible for detecting potential optimisations and applying them prior to dispatch. By fusing compatible tasks, the number of passes over the data

required can be reduced. The rules utilised to select and process tasks eligible for fusion are detailed in Listing 4.3.

In the order presented, the types of fusion supported are as follows:

Map-map fusion Adjacent `map` tasks can be replaced by a single task that performs the side-effects of both tasks combined.

Filter-filter fusion Adjacent `filter` tasks can be replaced by a single task that only retains elements that pass both predicates.

Map-filter fusion A `filter` task following a `map` task can replace it, performing its mutation before generating presence flags. Filter tasks that have gained the additional responsibility to mutate are hereafter referred to as `mapfilter` tasks.

Filter-map fusion Similarly, a `map` task following a `filter` task should not necessarily be scheduled. The side-effects of the `map` can be performed after filtering by a fused `mapfilter` kernel. This has the disadvantage that branching in the following map task, to avoid unnecessary calculation on items that won't be kept, will cause inefficient stalling in execution. However, if enough work-units are scheduled, the OpenCL runtime can identify non-stalled units to swap-in. Nonetheless, time wasted by stalls in a fused kernel is insignificant compared to the time to schedule a new kernel and pass over the data again in a separate `map` task.

Map-mapfilter fusion No different to `map-filter` fusion. The side-effects of the replaced `map` task are prepended to the `mapfilter`'s preprocessing actions.

Mapfilter-map fusion Again, advantageous as it avoids scheduling another pass over the data. The side-effects of the unnecessary `map` are appended to the `mapfilter`'s post-processing actions.

Mapfilter-filter fusion In `mapfilter` tasks that have no post-processing actions, the `filter` segment can be updated in the same manner as `filter-filter` fusion.

Listing 4.3: Fusion rules for combining tasks within the *TaskQueue*.

```
@tasks = @tasks.reduce [] do |queue, task|
  if (*fixed_queue, previous = queue).empty? then [task]
  else
    case [previous.class, task.class]
    when ([RubiCL::Map] * 2), ([RubiCL::Filter] * 2)
      fixed_queue << previous.fuse!(task)

    when [RubiCL::Map, RubiCL::Filter]
      fixed_queue << RubiCL::MappingFilter.new(
          pre_map: previous, filter: task)

    when [RubiCL::Filter, RubiCL::Map]
      fixed_queue << RubiCL::MappingFilter.new(
          filter: previous, post_map: task)

    when [RubiCL::Map, RubiCL::MappingFilter]
```

```
      fixed_queue << task.pre_fuse!(previous)

    when [RubiCL::MappingFilter, RubiCL::Map]
      fixed_queue << previous.post_fuse!(task)

    when [RubiCL::MappingFilter, RubiCL::Filter]
      if previous.has_post_map?
        fixed_queue << previous << task
      else
        fixed_queue << previous.filter_fuse!(task)
      end
    else
      fixed_queue << previous << task
    end
  end
end
```

Options to turn-off `TaskQueue` optimisation were introduced so that the magnitude of benefits can be studied. This will be revisited in the *Evaluation* chapter.

4.4 Functionality Testing

A performant system for calculation parallelisation isn't much use if its behaviour is incorrect. In order to increase confidence that the system performs as expected, the library was developed alongside a comprehensive test suite.

Having significant tests around behaviour enables more significant alterations to occur smoothly. This allowed the pace of experimentation to increase. New ideas can be verified as enhancing performance without introducing behaviour regressions.

The `RSpec`[21] testing library was used to produce test-cases. It presents a DSL for defining the intended behaviour of objects.

By describing a context corresponding to each feature that an subcomponent is designed to present, and testing boundary cases within that context, a rigid specification of correct behaviour was defined.

The advantages of testing were significant in terms of effort-economy. With a full test-suite execution taking less than 100ms on the development laptop, it was responsive enough to be triggered by each updated file within the development directory. This immediately highlighted interface clashes and regressions introduced during development. In addition, it reduced the amount of time wasted, manually checking that the system performed as advertised.

Compared to the stressful development practices of other projects witnessed, this development style is subjectively judged to be a significant success of the project.

RSpec –format documentation in appendix

4.5 Performance testing

A stated goal of the project refers to “improving dynamic language performance”. Therefore, it is important that the project provides a method for producing meaningful metrics. In order to facilitate measurement, a benchmarking suite for easy graph creation was developed. In addition, an execution mode that displays timing information alongside results was added to the `Logger`.

4.5.1 Custom benchmarking environment

During the lifetime of the project, a benchmarking library was created. It was originally designed as personal project, but was utilised heavily during development of the library. The benchmark library eases the production of graphs that plot function runtime over a range of input sizes.

A user provides several parameters to the library:

- A name for the graph.
- The number of iterations to average benchmark results over.
- Several function descriptions to test.

Each function description also contains parameters:

- A description of what is being benchmarked.
- An `Enumerable` providing seeds to the benchmark environment.
- A function that turns each seed into an *input*. This could be any value, given to the benchmarked function, that responds to `size`.
- The function to benchmark.

The benchmarking environment was used to produce Figure 2.1, shown earlier in the *Overview* chapter. The code used to generate the graph is shown in Listing 4.4.

Listing 4.4: The *asymptotic* library used to generate quick benchmark graphs.

```
require 'asymptotic'
require 'ostruct'

seeds = (20..25)

ruby_input = {
  input_seeds: seeds,
  input_function: ->(seed){ (1..2**seed).to_a }
}

command_line_input = {
  input_seeds: seeds,
  input_function: ->(seed){
    OpenStruct.new.tap { |s| s.size = 2**seed }
```

```
  }
}

Asymptotic::Graph.plot(5, "squaring integers and filtering evens",
  "Ruby: Enumerable#map and Enumerable#filter" => {
    function: ->(array){
      array.map { |x| x * x }.select { |x| x % 2 == 0 }
    },
  }.merge(ruby_input),

  "C: loop for mapping followed by loop for filtering" => {
    function: ->(struct){ `./just_c.o #{struct.size}` },
  }.merge(command_line_input),
)
```

The library handles generating an average runtime, using the specified number of test iterations, for each $(function, ||input||)$ pair. The garbage collector is turned off for the duration of each test and manual sweeping is triggered after each measurement is taken. A graph is then produced, using the `gnuplot` library, that compares the performance of all provided functions.

The ability to effortlessly create runtime graphs, for arbitrary given functions, proved useful during experimental development. When changes were introduced into the codebase, corresponding feature flags were added to the configuration module. Then, the benchmark environment was be used to plot the performance of the feature turned off against the performance with the feature enabled. This made it easy to highlight changes in design that altered performance for a given task over a variety of input sizes.

4.5.2 Segmented timing information from execution environment

Overall execution time is an important metric. However, it is helpful to be able to tell what proportion of time is spend doing various tasks during runtime.

In order to achieve this, code that gathers timing information was introduced to the library's native extensions that interact with hardware devices.

With each action triggered by the system, the resultant transaction time was measured. The low-level code, handling device management, obtains measurements via observing the start and stop time of OpenCL library functions. This data could then be retrieved by the library and inspected to determine the duration of subtasks, transferring or processing data.

Execution duration measurements are taken by native interaction modules, and presented to the management system. To make these readings available, configuration flags were added to the `Logger` stating that this data should be displayed during runtime.

An example of the finer granularity of timing information presented is given in Listing 4.5.

Listing 4.5: Segment times presented during command execution.

```
RubiCL::Logger.show_timing_info = true
```

```
#=> true

(1..10)[Int].map { |x| x + 100 }.filter { |y| y.even? }[Fixnum]
#> Pipeline Started
#> Pinned Integer Range in 0.039 ms
#> Enqueued
#>  (rubiclmappingfilter5 =>
#>    [
#>      "x = x >> 1", "x = x + 100",
#>      "?{(x % 2 == 0)}?", "x = (x << 1) | 0x01"
#>     ]
#>  )
#> in 3.175 ms
#> Waiting for in-progress tasks took 0.004 ms
#> Retrieved 5 Integers in 0.017 ms
#> Pipeline Complete in 5.109 ms
#=> [102, 104, 106, 108, 110]
```

Chapter 5

Evaluation

This chapter describes the means by which the project's success will be evaluated. Since both subjective and objective goals are stated in the *Overview* chapter, this will be taken into account when constructing evaluation criteria.

The results of individual evaluation procedures will be presented in the *Results* chapter, and interpreted in the corresponding *Analysis* section.

5.1 Recap of project aims

5.1.1 Objective goals

Improving the performance of the Ruby language when executing tasks on datasets In order to investigate whether exploitable performance improvements are given by the completed project, a series of scenario-driven benchmarks were constructed. The library was then utilised to perform the scenarios and compared against competing solutions: Standard Ruby code, and bespoke native extensions developed solely to achieve the evaluated task. Further discussion of all benchmarking will be presented in the *Benchmarking* subsection.

Increasing the scale of REPL experimentation possible This goal depends on favourable results in the previous evaluation criteria, as improved performance should lead to better REPL response time. However, unlike the previous benchmark's contenders, there is no longer a need to consider bespoke low-level solutions. This is due to the assumption that not all code is written upfront when investigating data using a REPL. As such, someone employed to analyse and draw conclusions from a dataset may not have the skill-set required to produce low-level code on demand.

5.1.2 Subjective goals

Gracefully extending the Ruby language runtime In order to judge how well the library is designed from a usability perspective, a series of user trials were conducted. Each trial presented a participant with a task description to solve, with the use of the RubiCL library. Notes were taken on applicant performance, specifically whether they had difficulties using the project's deliverable. Experimental design will be discussed in further detail in the *User Evaluation* section.

Effective reuse of OpenCL code Progress towards this goal will be summarised later by a discussion on code reuse within the project, highlighting any successes. Since it is hard to evaluate objectively, it will be presented as a list of what was learned over the course of the project's lifetime.

Suitability for deployment on unseen systems Portability is hard to measure. There is a near-infinite number of system configurations that could benefit from parallelism libraries. Instead of focusing on trying to install the library on as many systems as possible, installation from scratch was attempted on a typical desktop system. The results of this installation will be presented. In addition, a list of the assumptions made and requirements for the project will be produced. These requirements will then be evaluated, in the context of what was intended during the project's conception.

5.2 Benchmarking

5.2.1 Range of tests

The benchmarking procedure is responsible for investigating how successful the project was with regards to it's performance-oriented goals. As such, it must be designed in a way that demonstrates the potential performance of the system, as well as being representative of realistic usage.

The decision was made to test the variety of system primitives in isolation, in addition to a combined task that produces a fused kernel. The range of primitives shortlisted for investigation were as follows:

- A basic `map` task, incrementing the dataset.
- A 'dense' `filter` task returning 50% of the input data.
- A 'sparse' `filter` task returning 5% of the input data.
- A fused `mapfilter` task, consisting of the previous `map` task followed by the dense `filter` task.
- A `sort` task.

5.2.2 Test systems

5.2.2.1 Hardware

Two systems were used for running tests, the *MacBook* laptop used primarily for development and the ordered AMD desktop system. With library development and regular testing occurring primarily on the laptop, the benchmarking procedure provided an opportunity to evaluate how portable the performance characteristics of the OpenCL framework are.

Specs etc

5.2.2.2 Software

For both hardware systems, several methods of performing each specified task were measured:

- The standard implementation provided by unmodified Ruby 2.2.
- A handwritten native extension that performs the task using the optimal sequential method.
- The RubiCL library performing the task, executing on the system CPU.
- The RubiCL library performing the task, executing on the system GPU.

The standard implementation was included as it is important that a comparison between any new solutions and existing functionality is made.

The reason to additionally measure the performance of a bespoke native extension performing the task is as follows:

By investigating sequential, best-case performance, we can deduce whether performance gains stem purely from parallelism or more-optimal execution. If the library exceeds performance of the best possible sequential implementation, throughput of compute devices has been harnessed most effectively. Otherwise, assuming that the standard implementation performance is exceeded, one of two things is true: Either the library provides throughput gains but the overhead of scheduling parallel tasks through OpenCL mitigates any potential improvements. Or, the throughput provided by the increased number of execution units is not enough to compensate for the extra work encountered by the parallel algorithms employed.

The project's performance on both hardware devices present in the test system was measured so that the relationship between the type of task and the optimal device architecture can be investigated.

Custom native extension The optimal sequential implementation of benchmarked tasks was provided by a handwritten C native extension, shown in Listing 5.1.

Listing 5.1: Custom native extension performing tasks in the optimal sequential manner.

```c
#include "ruby.h"

VALUE mMapAddOne(VALUE self, VALUE input) {
    int elements = RARRAY_LEN(input);
    VALUE output = rb_ary_new2(elements);

    for (int i = 0; i < elements; ++i) {
        VALUE transformed = INT2FIX(FIX2INT(rb_ary_entry(input, i)) + 1);
        rb_ary_store(output, i, transformed);
    }

    return output;
}

VALUE mFilterEven(VALUE self, VALUE input) {
    int elements = RARRAY_LEN(input);

    int kept = 0;
    int size = 2;
    VALUE* out = malloc(size * sizeof(VALUE));

    for (int i = 0; i < elements; ++i) {
        VALUE entry = rb_ary_entry(input, i);
        if (FIX2INT(entry) % 2 == 0) {
            if (++kept > size) {
                out = realloc(out, (size *= 2) * sizeof(VALUE));
            }
            out[kept - 1] = entry;
        }
    }

    VALUE output = rb_ary_new2(kept);
    for (int i = 0; i < kept; ++i) rb_ary_store(output, i, out[i]);
    free(out);

    return output;
}

VALUE mFilterModTwen(VALUE self, VALUE input) {
    int elements = RARRAY_LEN(input);

    int kept = 0;
    int size = 2;
    VALUE* out = malloc(size * sizeof(VALUE));

    for (int i = 0; i < elements; ++i) {
        VALUE entry = rb_ary_entry(input, i);
        if (FIX2INT(entry) % 20 == 0) {
            if (++kept > size) {
                out = realloc(out, (size *= 2) * sizeof(VALUE));
            }
            out[kept - 1] = entry;
        }
    }

    VALUE output = rb_ary_new2(kept);
```

```
    for (int i = 0; i < kept; ++i) rb_ary_store(output, i, out[i]);
    free(out);

    return output;
}

VALUE mMapAddHalfFilterEven(VALUE self, VALUE input) {
    int elements = RARRAY_LEN(input);

    int kept = 0;
    int size = 2;
    int* out = malloc(size * sizeof(int));

    for (int i = 0; i < elements; ++i) {
        int entry = FIX2INT(rb_ary_entry(input, i));
        entry += entry / 2;
        if (entry % 2 == 0) {
            if (++kept > size) {
                out = realloc(out, (size *= 2) * sizeof(int));
            }
            out[kept - 1] = entry;
        }
    }

    VALUE output = rb_ary_new2(kept);
    for (int i = 0; i < kept; ++i) rb_ary_store(output, i, INT2FIX(out[i]));
    free(out);

    return output;
}
void Init_bespoke_backend() {
    VALUE BespokeBackend = rb_define_module("BespokeBackend");
    rb_define_singleton_method(BespokeBackend, "map_add_one",
            mMapAddOne, 1);
    rb_define_singleton_method(BespokeBackend, "filter_even",
            mFilterEven, 1);
    rb_define_singleton_method(BespokeBackend, "filter_modtwen",
            mFilterModTwen, 1);
    rb_define_singleton_method(BespokeBackend, "map_add_half_filter_even",
            mMapAddHalfFilterEven, 1);
}
```

5.2.3 Variety of data-types

The RubiCL library supports accelerating computation on homogeneous collections of both `Fixnum` and `Float` objects. However, presenting benchmarks of the full range of parallel primitives on both types and both systems, then commenting in depth on both graphs, would provide a lot of redundant data.

Instead, the performance of the library on integer datasets, across both hardware systems, will be explored in depth. This will be followed by a brief analysis of how performance differs when presented with floating-point datasets.

The major difference when operating on `Float` objects, is that they lack tagged-pointers. Therefore, the value of each object present must be determined by dereferencing the object pointer and examining the resultant `RFloat` struct. This is impossible to achieve in parallel on an external hardware device, unlike when bit-shifting the tagged-pointers of `Fixnum` objects. In addition, new `RFloat` objects must be created to wrap the computed results. These two stages of extra computation must also be performed by the RubyVM implementation, therefore the RubiCL library should not be significantly disadvantaged.

Also worth mentioning is the difference in hardware performance concerning integer and floating-point operations. While tasks scheduled on the CPU should utilise the same execution units as the RubyVM, tasks scheduled on the GPGPU may be affected by how well the hardware is suited to floating-point calculations.

The two factors mentioned result in a linear increase in pre-processing and post-processing time, and a possible rate-of-computation shift. Therefore, they can be summarised by merely presenting the lower-bound at which computation becomes worth outsourcing for both data-types, alongside measurement of the total speed-up rate achievable.

Lack of 'double' support on Intel HD5000 Unfortunately, the GPGPU present on the testing laptop does not support calculation on double precision floating-point numbers. As a result, the floating-point benchmarks presented only document CPU performance. This means that merely the linear processing addition, and not the rate-of-computation change, can be commented on.

5.2.4 Method

5.2.4.1 Gathering results

In order to speed up the process of gathering many benchmark results, a simple utility was produced. Listing 5.2 shows the internals of the *benchmark helper* utility.

Since Ruby is a language that provides automatic memory management and garbage collection of objects, this must be disabled when benchmarking. Otherwise, the large number of repeated tests may trigger a sudden stop-the-world sweep during the latter timing rounds and skew the results.

To ensure that the results gathered represented average use effectively, each benchmark, at a particular dataset size, was ran 20 times. The mean of timing information recorded over all repetitions was then returned. This lessens the weighting of anomalous readings, when the system's external utilisation may have reduced performance slightly.

Listing 5.2: Helper function defined for benchmarking a block of code.

```
require 'benchmark'

def benchmark(times: 1, input:[], &block)
  GC.start
```

```
  (1..times).map do
    GC.disable
    time_taken = Benchmark.realtime { block.call(input) }
    GC.enable
    GC.start

    time_taken
  end.inject(&:+).to_f / times
end
```

Gathering readings using the helper function involved substituting the block argument with the implementation to test, and calling the utility over a range of sizes. Once obtained, the readings for each set of competing software elements were written to disk, available for the graph generation process to consume. Listing 5.3 shows a typical method of gathering the set of results for a standard Ruby *MapFilter* task.

Listing 5.3: Using the benchmark helper to gather readings over a range of input datasets.

```
require './bench_helper'

results = (1..19).step(2).map do |millions|
  dataset = Array(1..millions * 1_000_000)
  benchmark(times: 20, input: dataset) do |d|
    d.map { |x| x + 1 }.select { |x| x.even? }
  end
end
```

5.2.5 Issues

One issue experienced whilst collecting results was the fluctuation in APU performance on the test laptop.

Likely due to system heat management protocols, if too many benchmarks were ran in quick succession, the performance of the on-board GPGPU would decrease.

This was initially unrealised. Initial benchmarking methods, relied on throughout library construction, ran all tests at once, with the CPU system and then the GPGPU system being tested. As a result, laptop GPGPU performance was consistently under-estimated. Luckily, this outside factor was realised as final benchmarks were being designed. As a result, care was taken to reduce the length of benchmark runs and instead schedule more subsets of the desired test range, allowing device temperature to stabilise between measurements.

5.3 User Evaluation

5.3.1 Subjects

The user evaluation procedure was designed to both showcase the potential of the project, when applied to a realistic scenario, and highlight any usability issues of the library. 7 applicants were recruited for a 5 question challenge, using RubiCL to answer questions about a dataset.

The fictitious scenario presented was that of an online banking service with two, separately stored datasets, corresponding to the triggering `user_id` and transfer `amount` of the past $10,000,000$ transactions. Subjects had to utilise higher-order primitives to summarise activity presented in the data, under the guise of investigating a suspicious user.

When searching for potential test-subjects, care was taken to explore a variety of backgrounds and programming proficiencies. As such, the range of applicable skills present in users evaluated ranged from high levels of parallel programming experience to barely any programming experience whatsoever.

5.3.2 Method

Before testing began, each test-subject was shown a quick demonstration of the project's capabilities, alongside a discussion about the research goal. The aim of this quick demonstration was to hint at how the library may be applied to the later provided problems, in addition to an introduction or refresher to Ruby syntax for higher-order functions. After the demonstration concluded, a link[22] containing brief API documentation was provided. The majority of documentation was lifted directly from the Ruby documentation source, as the library mirrors the `Enumerable` API closely. The few additional methods documented were RubiCL specific functionality such as the bifurcation of tuple buffers. The hint document was provided so that stuck or novice users could remind themselves of basic language functionality. Care was taken as to not reveal how to solve problems directly.

Users were provided with a skeleton file, containing a DSL for answering several given questions. Each question accepted an anonymous function response, with bound variables signifying the resources that could be used to answer the query. To respond, each applicant would replace the body of the function, originally a placeholder method, with the required query pipeline.

Subjects were unaware that the provided dataset variables were actually file handles, pointing to large collections of integer data on disk. In addition, the test system GPGPU was assigned as the default RubiCL compute device.

Users were encouraged to save the test file each time a function body had been specified. An analysis program, listening on file-system events in the working directory, would then call the provided functions, providing test data, and report whether answers given were

correct. This gave immediate feedback to subjects as to when they should move on as particular question was solved. In addition to verifying calculated answers, reports of the time taken for each task were collated.

Notes were taken during the study, followed by brief discussion with each finished applicant. Further discussion of user evaluation results and analysis will occur in the *Results* chapter.

5.3.2.1 Issues

User evaluation took place late in the project's life-cycle. As a result, many potential candidates were reluctant to participate as they were busy working on their own projects. Scheduling this session earlier would have helped avoid this.

Another reason to schedule the session earlier is the benefit of using feedback to guide design. Luckily, most constructive criticism resultant from the user evaluation trials was simple enough to fix quickly. However, any significant issues would be realised too late before the project's deadline to justify an intrusive overhaul.

Listing 5.4: The test file full of questions given to each subject.

```
require_relative './evaluation/user_test.rb'

# Imagine that you run a hugely successful bitcoin exchange that hasn't gone bust yet.
#
# You have access to logged details of the last 10 million transactions, semi-structured form.
# The following variables represent the data you have at hand:
#
#   transaction_user_ids: a file that contains the user_id associated with each transaction 1..10_000_000
#   transaction_amounts:  a file that contains the amount transferred for each transaction 1..10_000_000
#   target_user_id:       a variable containing the user_id of a person-of-interest, under investigation.

# In each solution, the variables target_user_id, transaction_user_ids, and
#                   transaction_amounts will be available.
#
# Use these variables in addition to the methods shown in the handout in order
#                   to solve each question.

here_we_go!

# QUESTION ONE
#   Task:
#         Return the total number of transactions triggered by the target user.
answer_for question: 1, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  your_solution_here!
}

# QUESTION TWO
#   Task:
#         Return the total number of deposits triggered by the target user.
#         Deposits are transactions that have a positive value.
answer_for question: 2, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  your_solution_here!
}
```

```
# QUESTION THREE
#   Task:
#         Return the target user's total change in balance over the recorded period.
answer_for question: 3, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  your_solution_here!
}

# QUESTION FOUR
#   Task:
#         Return the target user's total magnitude of even-valued withdrawals over the recorded period,
#         WITHDRAWALS WHERE THE AMOUNT WITHDRAWN IS EVEN.
#         Withdrawals are transactions with value below 0.
answer_for question: 4, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  your_solution_here!
}

# QUESTION FIVE
#   Task:
#         Return the number of times that someone with a multiple-of-222 user-id (0 inclusive) has deposited
#         a multiple-of-222 (0 exclusive since deposit).
#         Deposits are transactions with value above 0.
answer_for question: 5, is: ->(target_user_id, transaction_user_ids, transaction_amounts) {
  your_solution_here!
}

tasks_complete!
```

Listing 5.5: Sample answers to the questions given.

```
require_relative './evaluation/user_test.rb'

here_we_go!
answer_for question: 1, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  transaction_user_ids[Int].count(target_user_id)
}
answer_for question: 2, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  transaction_user_ids[Int].zip(transaction_amounts)
                           .count { |id, amount| id == target_user_id && amount > 0 }
}
answer_for question: 3, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  transaction_user_ids[Int].zip(transaction_amounts)
                           .select { |id, amount| id == target_user_id }
                           .snds
                           .sum
}
answer_for question: 4, is: ->(target_user_id, transaction_user_ids, transaction_amounts){
  transaction_user_ids[Int].zip(transaction_amounts)
                           .select { |id, amount| id == target_user_id && amount < 0 && amount.even? }
                           .snds
                           .sum
}
answer_for question: 5, is: ->(target_user_id, transaction_user_ids, transaction_amounts) {
  transaction_user_ids[Int].zip(transaction_amounts)
                           .select { |id, amount| id % 222 == 0 }
                           .select { |id, amount| amount % 222 == 0 && amount > 0 }
                           .count
}
tasks_complete!
```

Chapter 6

Results

6.1 Benchmarks

6.1.1 Map tasks

6.1.1.1 Integer performance

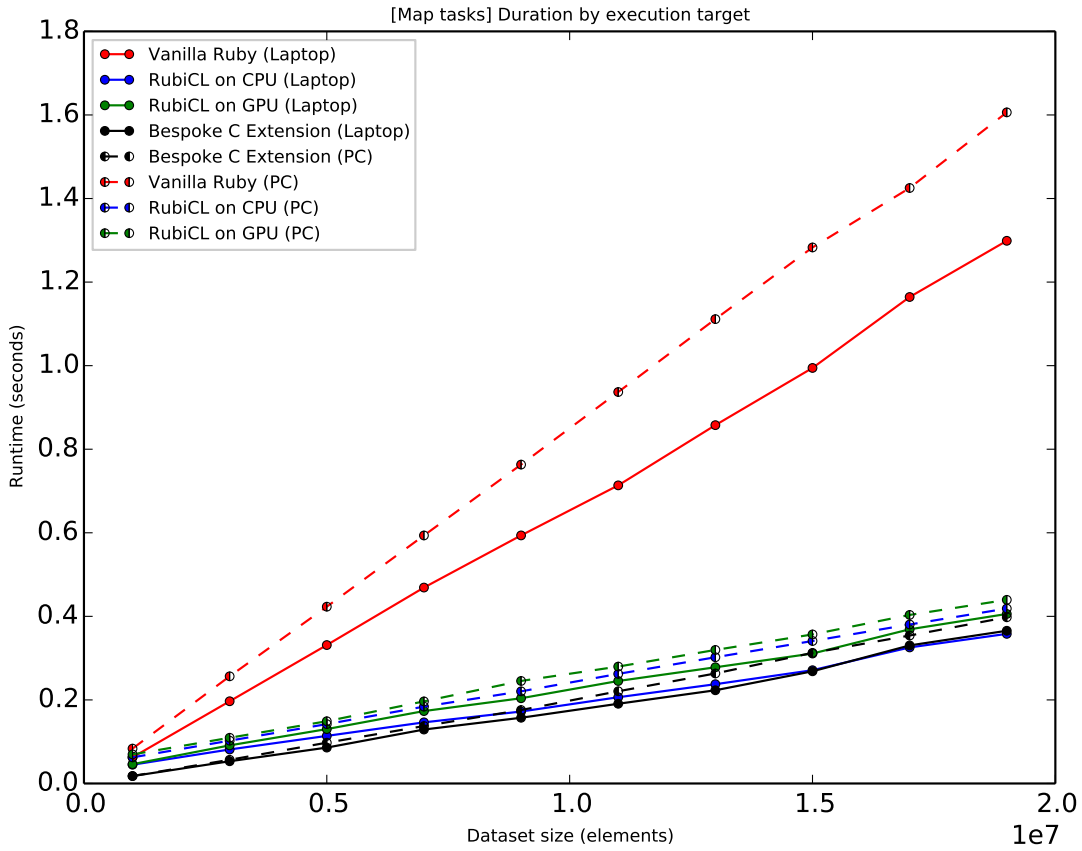


Figure 6.1: Task duration by execution target for *Map*.

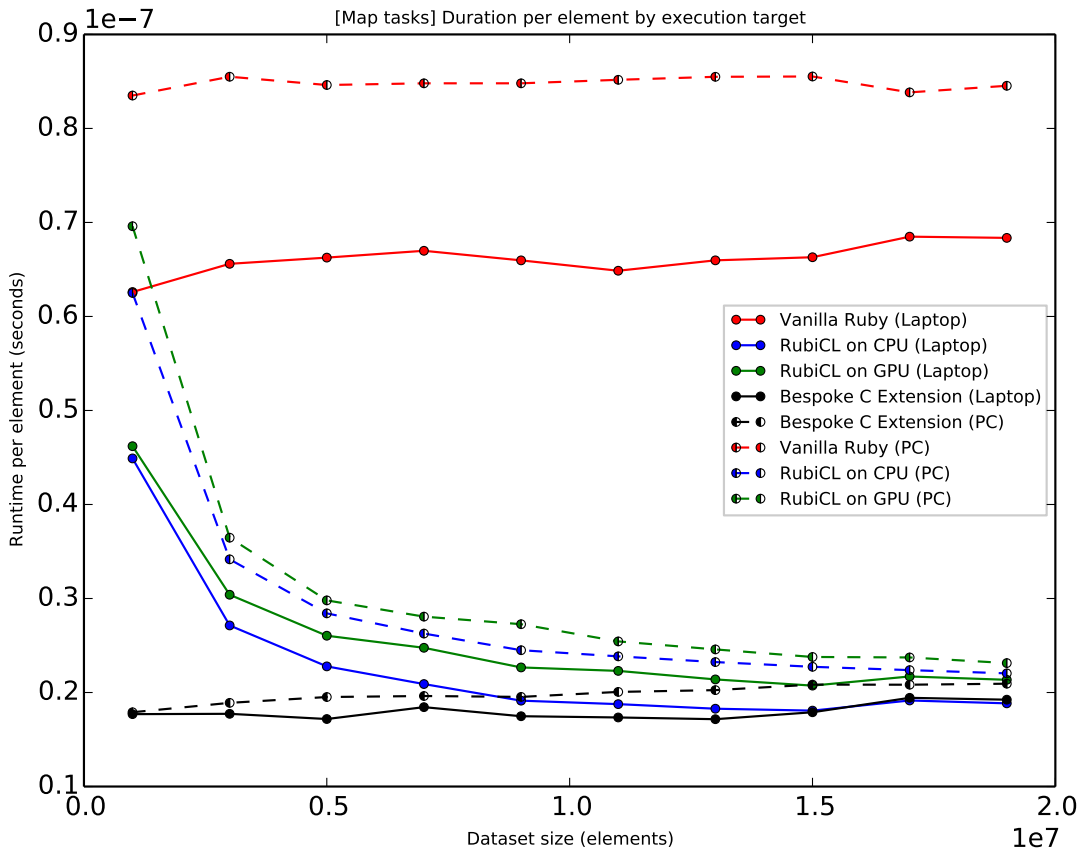


Figure 6.2: Task duration per processed element for *Map*.

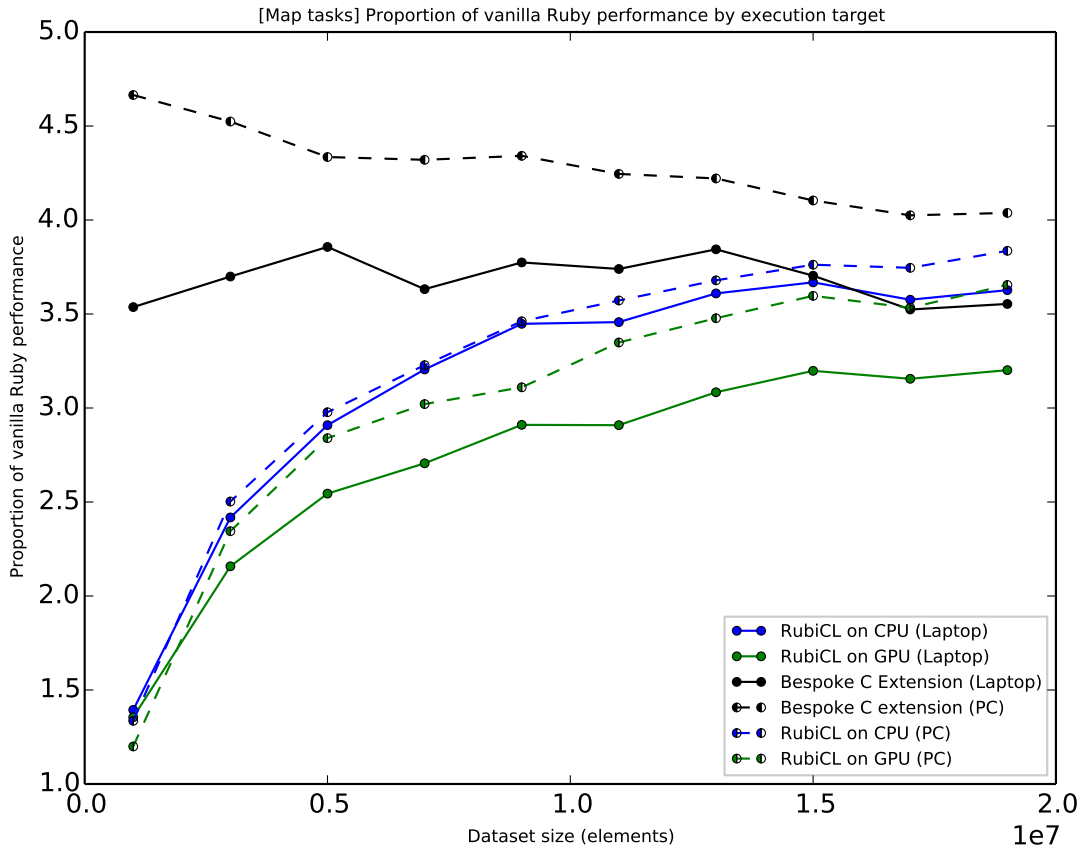


Figure 6.3: Proportion of vanilla Ruby performance achieved for *Map*.

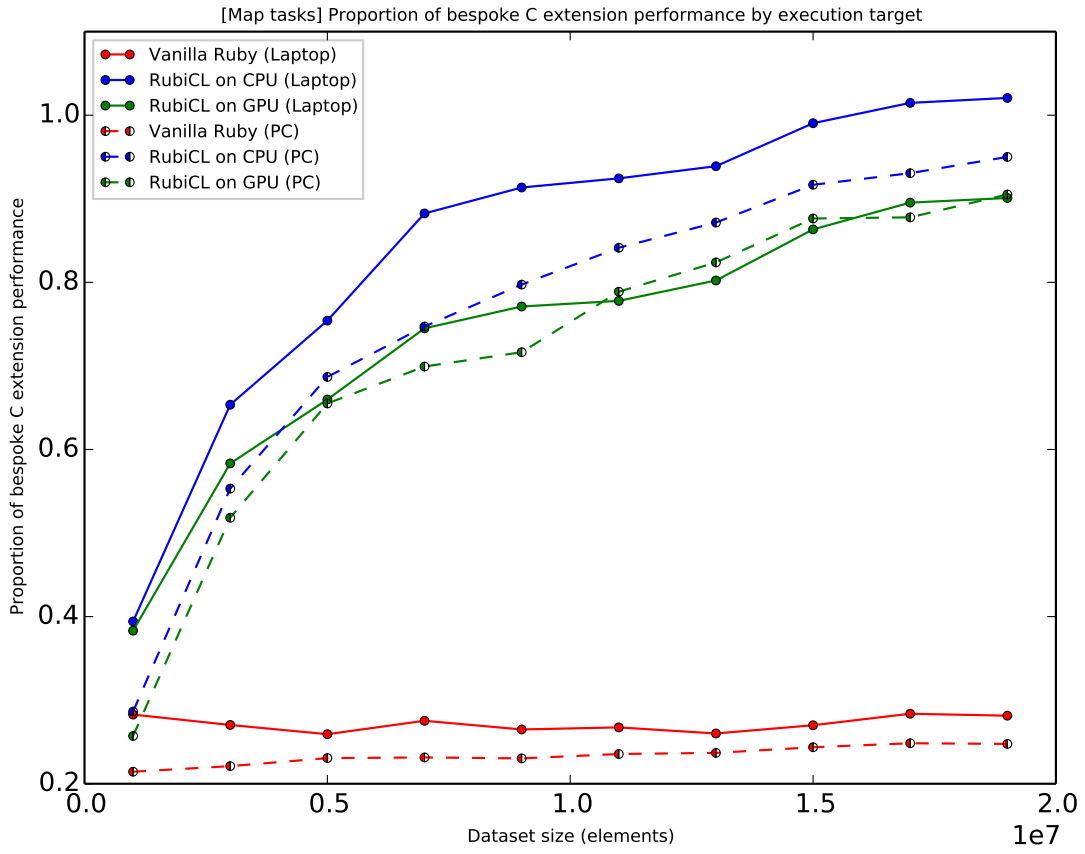


Figure 6.4: Proportion of bespoke C extension performance achieved for *Map*.

Recap of operation performed The *Map* task performed was the equivalent of `#map { |x| x + 1 }`. This has the effect of incrementing every member of the input dataset. Performing little work in the supplied function body produces a worst-case evaluation of library performance. This is because performing more work will allow increased throughput to mask any increased latency that the system has. Many tasks will be more involved than incrementation, and very few less so. A result showing that the library is beneficial here should apply even more to tasks that are of greater intensity.

Observations and analysis Figure 6.1 shows that all trialled alternatives exhibit similar performance when executing *Map* tasks. This is likely due to the computational simplicity of the task. The probable cause for the bottleneck is the need to move data in and out of the RubyVM's internal array structures. Although the gap is slight, the CPU compute-devices installed in both machines outperform the GPGPUs over the range of tested datasets. This is easiest to observe in Figure 6.2.

When parallelising purely-map tasks, the project library performs favourably compared to the standard `Enumerable#map` implementation of Ruby 2.2. Figure 6.3 demonstrates that, at best, a factor 3.5–4 speed-up over typical execution can be achieved on both systems. The figure also shows that for all tested datasets, the smallest of which contained 1 million elements, outsourcing computation to the library was beneficial.

In Figure 6.4, it can be seen that on both systems RubiCL throughput is not far from bespoke sequential code. On the laptop, the parallel CPU implementation exceeds the non-parallel native extension. It achieves 1.02 times the rate of processing on 19 million elements. However, while the laptop CPU presents 4 hardware threads to OpenCL, the native extension utilises only a single thread of execution. As both implementations are performing roughly the same number of operations, it appears that here OpenCL's raw throughput increase is insignificant after the processing-model overhead is accounted for.

It is clear that the throughput with which Ruby is capable of performing *Map* tasks has been significantly increased. On the other hand, the project's parallel library does not significantly outperform a custom, tailored, sequential solution. Yet, for all systems choosing the optimal device, no less than 80% of the best-presented throughput is achieved at 10 million elements, improving to no less than 95% at 19 million. With the library providing automatic translation of all functions stated into parallel execution patterns, this deficit is insignificant. Much more significant is the mitigation of the need to write and compile native extensions for every required calculation.

Smaller scale Map tasks Figure 6.5 shows that the RubiCL library, when executing on the laptop system, is beneficial for *Map* tasks containing greater than $20,000$–$25,000$ elements. It also demonstrates that the laptop system experiences task latency of around 20ms.

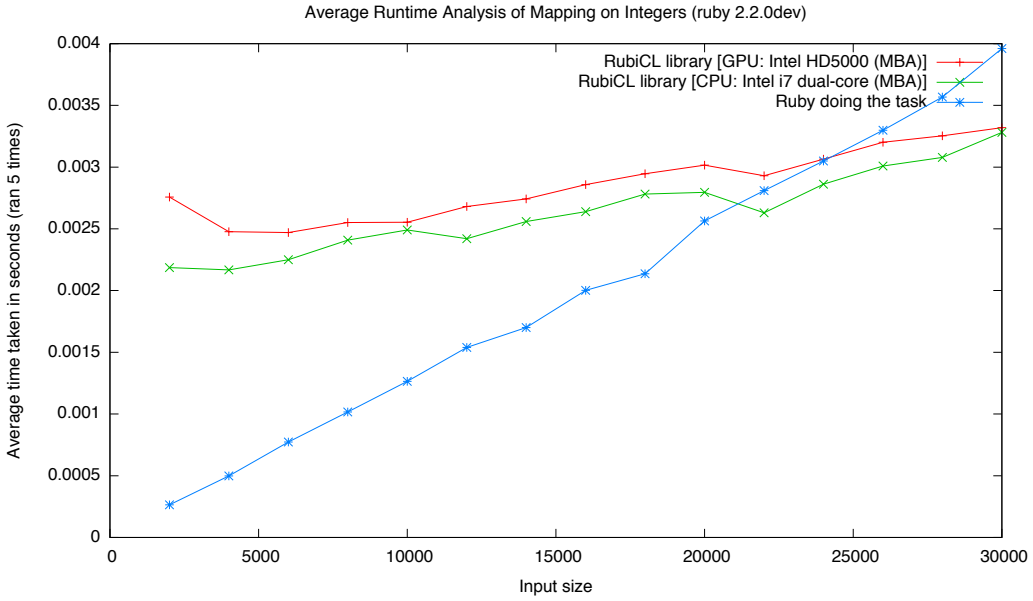


Figure 6.5: Duration for smaller-scale *Map* tasks.

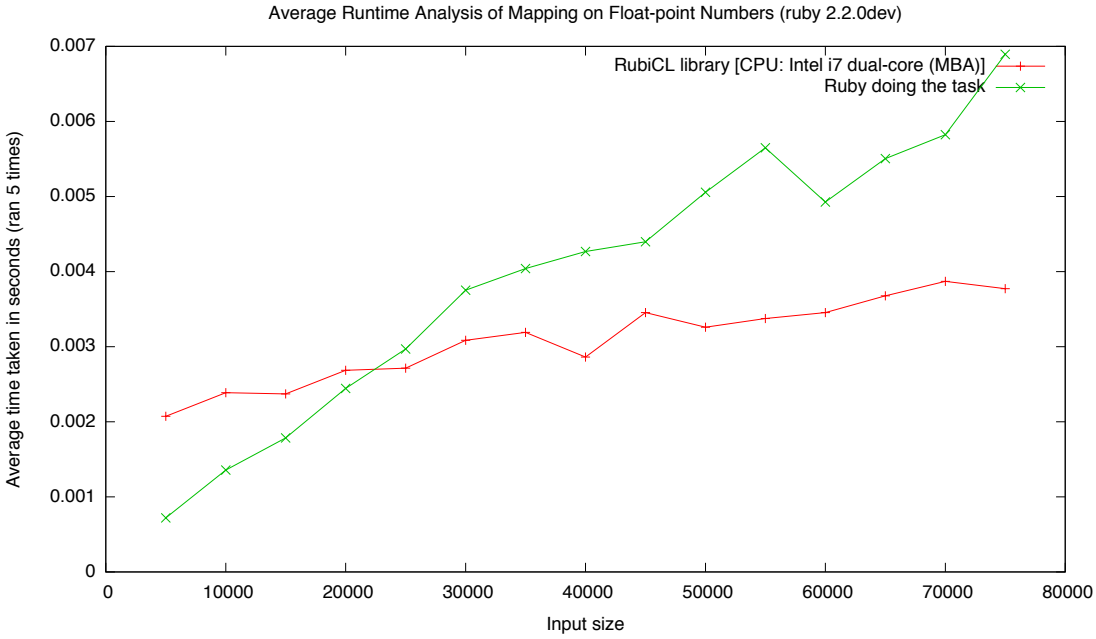


Figure 6.6: Duration for smaller-scale *Map* tasks on floating-point numbers

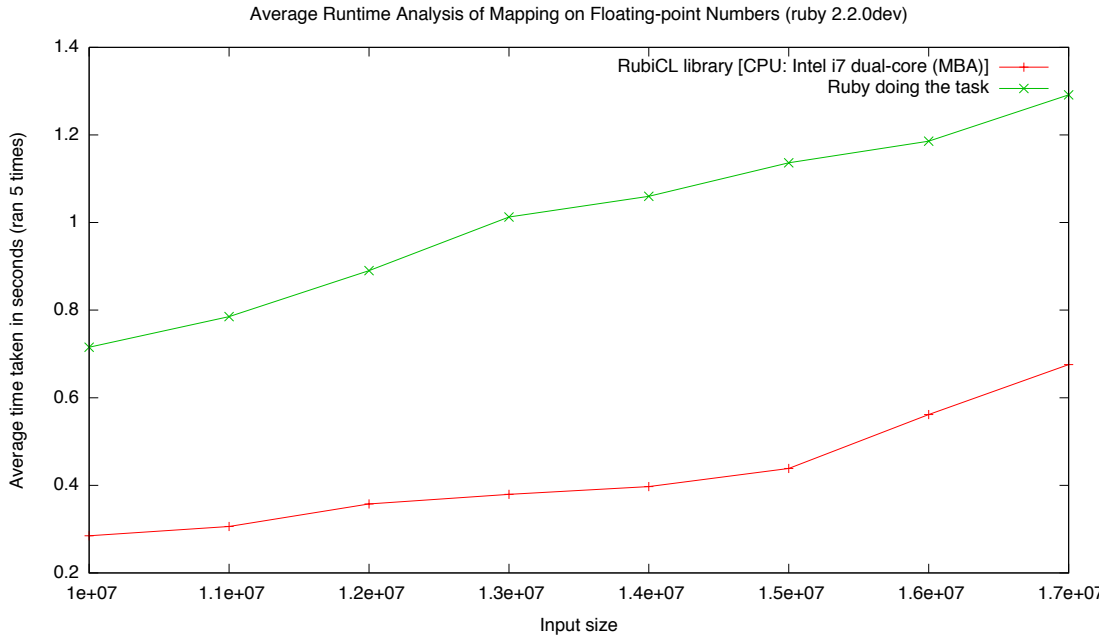


Figure 6.7: Duration for larger-scale *Map* tasks on floating-point numbers

6.1.1.2 Floating-point performance

Figure 6.7 shows that the lower-bound for beneficial *Map* acceleration by the RubiCL library remains roughly the same for floating-point datasets as what was observed for integer datasets. Again, comparing OpenCL execution on the CPU to standard Ruby, it is worth outsourcing computation above $20,000$ elements.

The RubiCL library offers a speedup of around 2 times the traditional implementation. The reduction over integer speedup is likely due to object conversion no longer occurring in parallel, and the linear dereference cost added to both implementations.

6.1.2 Dense Filter tasks

6.1.2.1 Integer performance

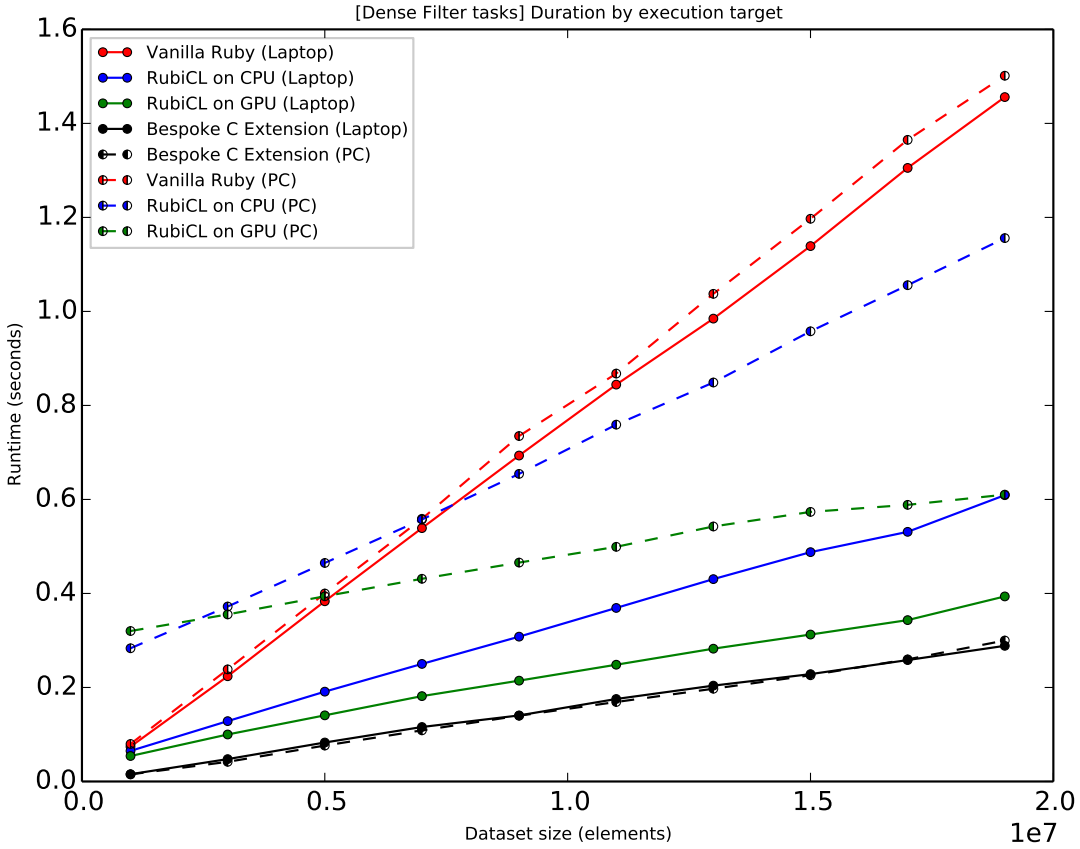


Figure 6.8: Task duration by execution target for dense *Filter*.

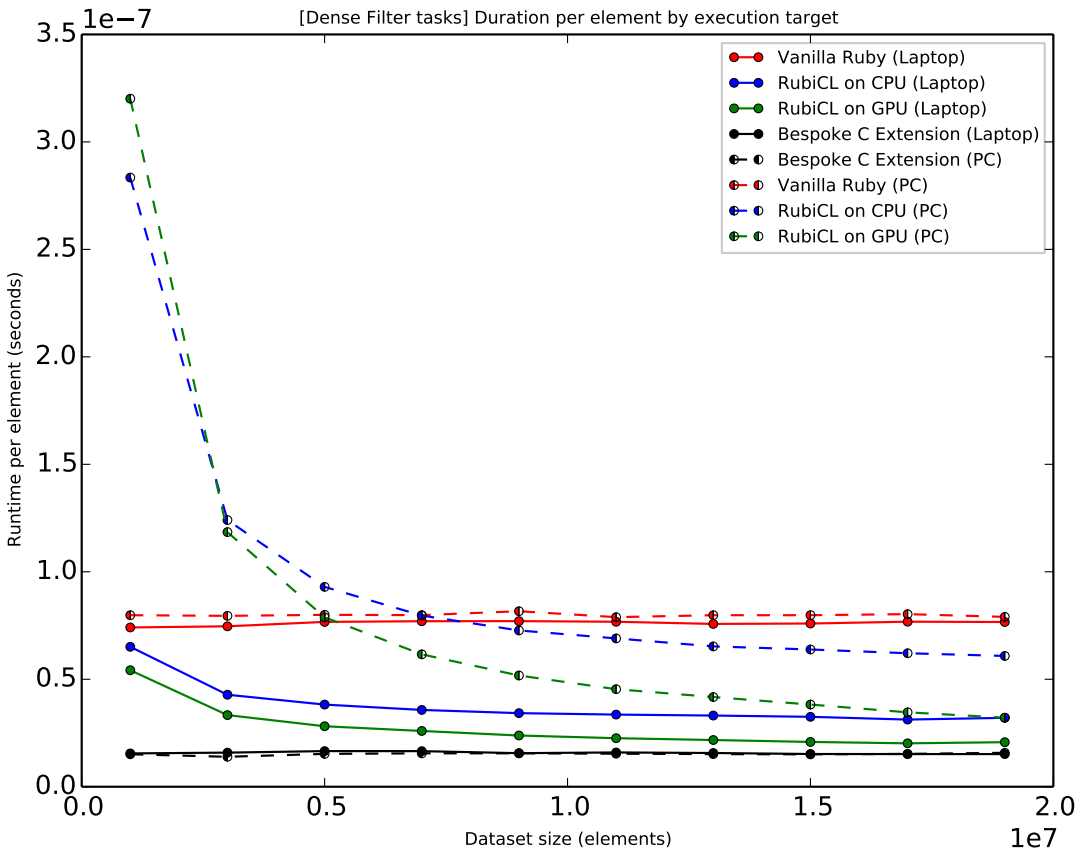


Figure 6.9: Task duration per processed element for dense *Filter*.

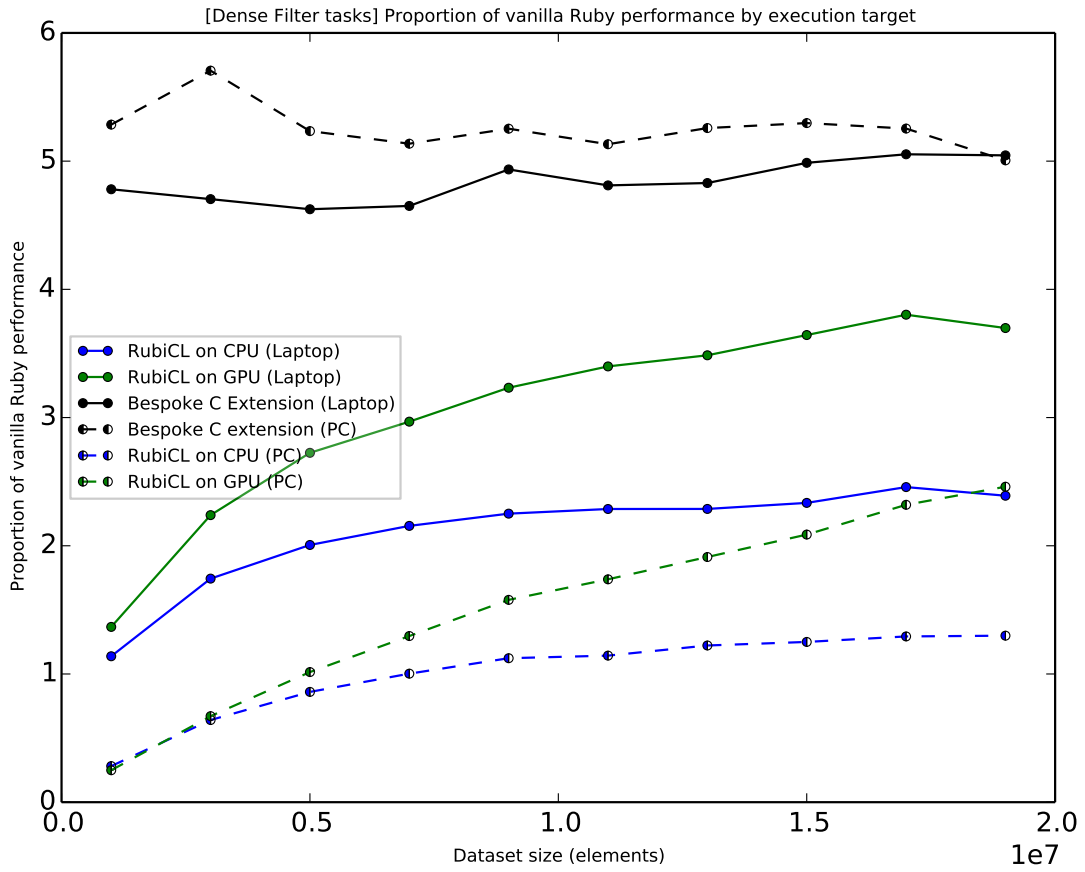


Figure 6.10: Proportion of vanilla Ruby performance achieved for dense *Filter*.

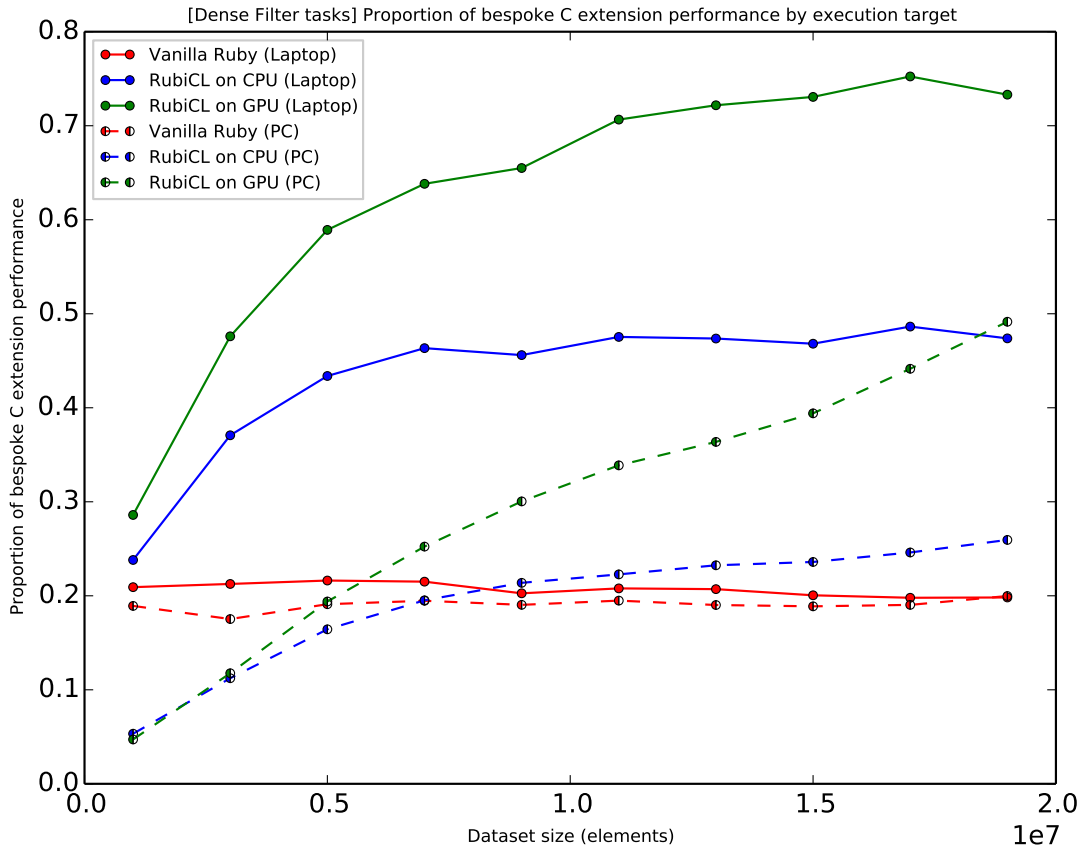


Figure 6.11: Proportion of bespoke C extension performance achieved for dense *Filter*.

Recap of operation performed The dense *Filter* task performed was the equivalent of `#select { |x| x.even? }`. With the ascending range of data supplied, this will return a subset of the input dataset with half the number of elements.

Observations and analysis Figure 6.8 is less cluttered than the corresponding graph for *Map* tasks, as there is more variation between the performance of dense *Filter* implementations. The figure suggests that *Filter* tasks scheduled on the desktop system suffer from a higher latency than their counterparts on the laptop. This is signified by the comparatively elevated runtime for smaller datasets.

Further unlike *Map* tasks, on both systems the GPGPU compute-devices outperform the CPU when filtering large datasets. For the laptop system, the domination is present on every tested dataset size. On the contrary, the desktop CPU is initially a shade faster but is quickly dwarfed by the significantly higher throughput of the GPGPU device.

On the laptop system, Figure 6.10 demonstrates that the library is beneficial when processing all datasets within the tested range. The desktop is not quite as successful at performing dense purely-*Filter* tasks, hampered when producing subsets of smaller datasets by its increased latency. At least 5 million elements are required before it is worthwhile to offload computation onto the GPGPU via RubiCL. The CPU trails not long after, at 8 million elements, but never achieves more than 25% speedup over the standard Ruby implementation.

On both test systems, when using the GPGPU, noticeable speedup of *Filter* operations can be achieved. With a 2.5 times speedup on the desktop and over 3.5 times on the laptop, large filtering operations are significantly accelerated by the RubiCL library.

Figure 6.11 shows that the laptop GPGPU achieves a high proportion, peaking at 75%, of bespoke sequential code performance. The desktop system boasts a lesser proportion, at 50%, but the lack of plateau in the figure suggests that this gap would close further as datasets increase in size.

The lacklustre performance when executing on CPU devices can be partly explained by the fact that the parallel implementation of filtering, although asymptotically identical in cost, requires much more work than a sequential filter. In this case, the hidden constants involved for distributing computation have a large effect on the task duration, larger than that of parallelising the predicate scan and index calculations along the data.

This reasoning can also explain why the GPGPU devices fail to outperform the custom extension over the given range of datasets. Nonetheless, the need to write and compile native extensions for each distinct query performed is again removed when using the RubiCL library. A system-dependent performance hit of between 25% and 50% behind a handwritten extension is far more justifiable when it facilitates rapid-prototyping, especially when compared to the 80% penalty that Figure 6.11 highlights for unoptimised Ruby 2.2.

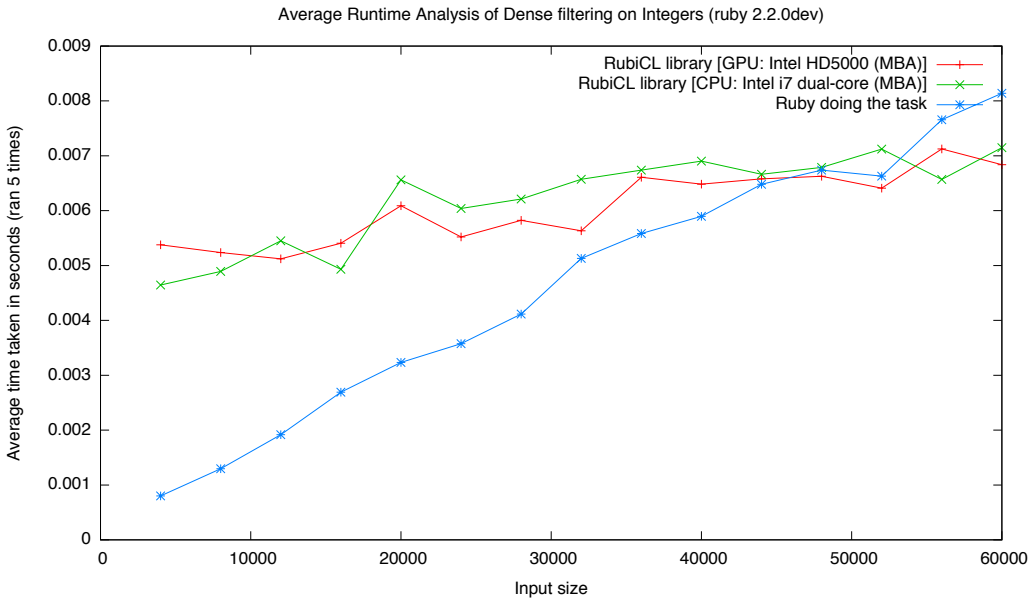


Figure 6.12: Duration for smaller-scale dense *Filter* tasks.

Smaller scale dense Filter tasks Figure 6.12 shows that the RubiCL library, when executing on the laptop system, is beneficial for dense *Filter* tasks containing greater than $50,000$ elements. It also demonstrates that the laptop system experiences task latency of around 50ms.

6.1.2.2 Floating-point performance

Figure 6.13 shows that the lower-bound for beneficial dense *Filter* acceleration by the RubiCL library remains roughly the same for floating-point datasets as what was observed for integer datasets. Again, comparing OpenCL execution on the CPU to standard Ruby, it is worth outsourcing computation above $50,000$ elements.

The RubiCL library offers a speedup of around 2 times the traditional implementation. This is shown in Figure 6.14 and is nearly identical to the CPU speedup achieved for integer datasets. This suggests that the added cost of object dereferencing and re-creation, necessary for all implementations, is insignificant compared to the work involved when performing a filtering operation.

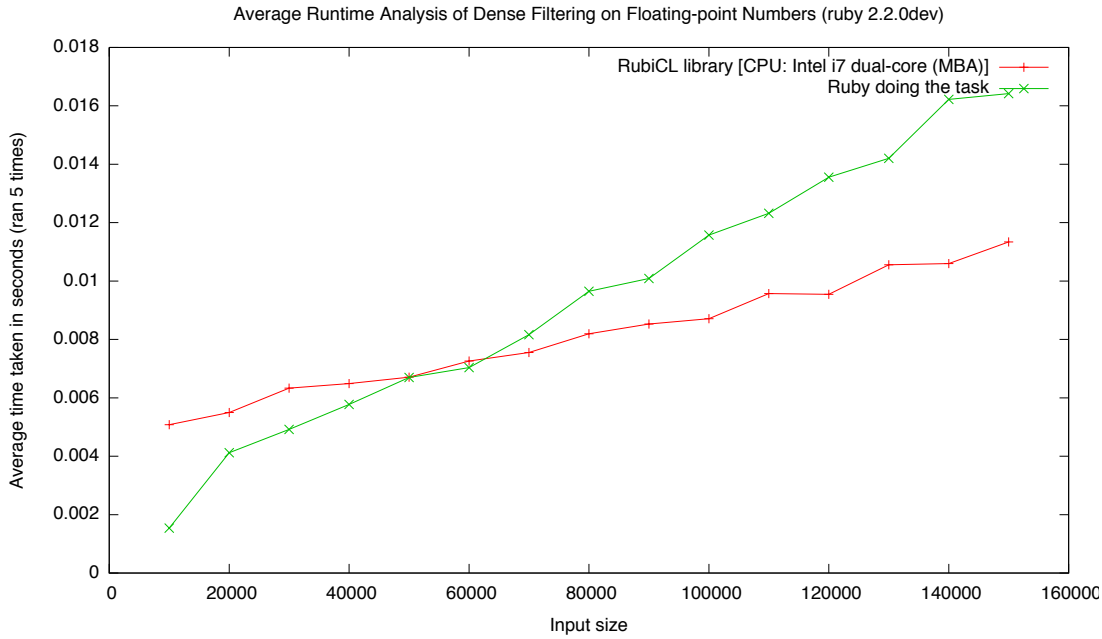


Figure 6.13: Duration for smaller-scale dense *Filter* tasks on floating-point numbers

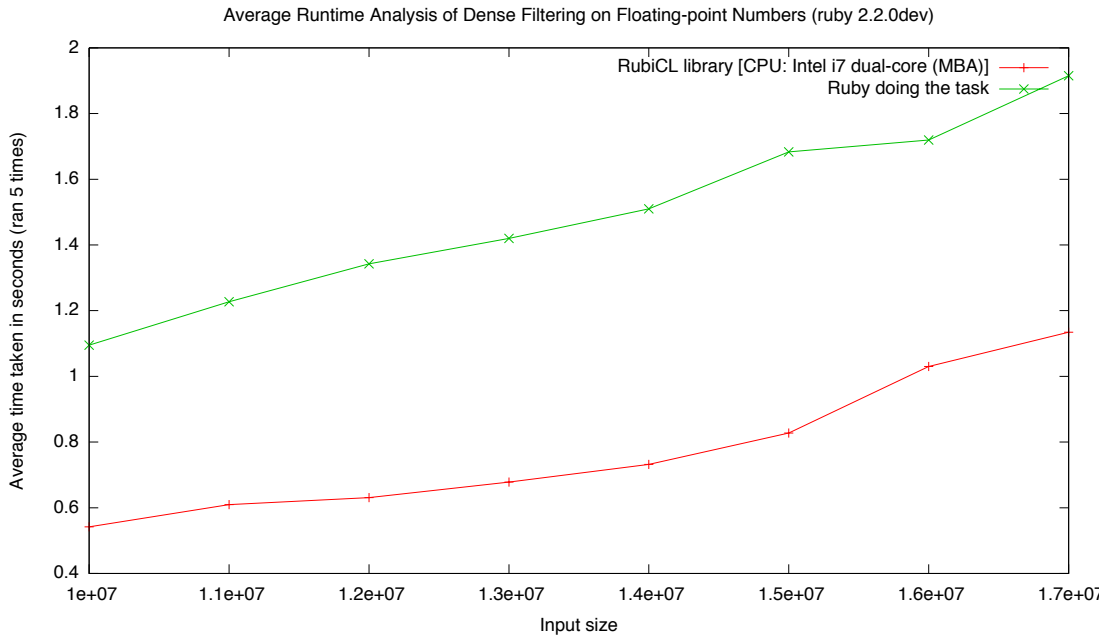


Figure 6.14: Duration for larger-scale dense *Filter* tasks on floating-point numbers

6.1.3 Sparse Filter tasks

6.1.3.1 Integer performance

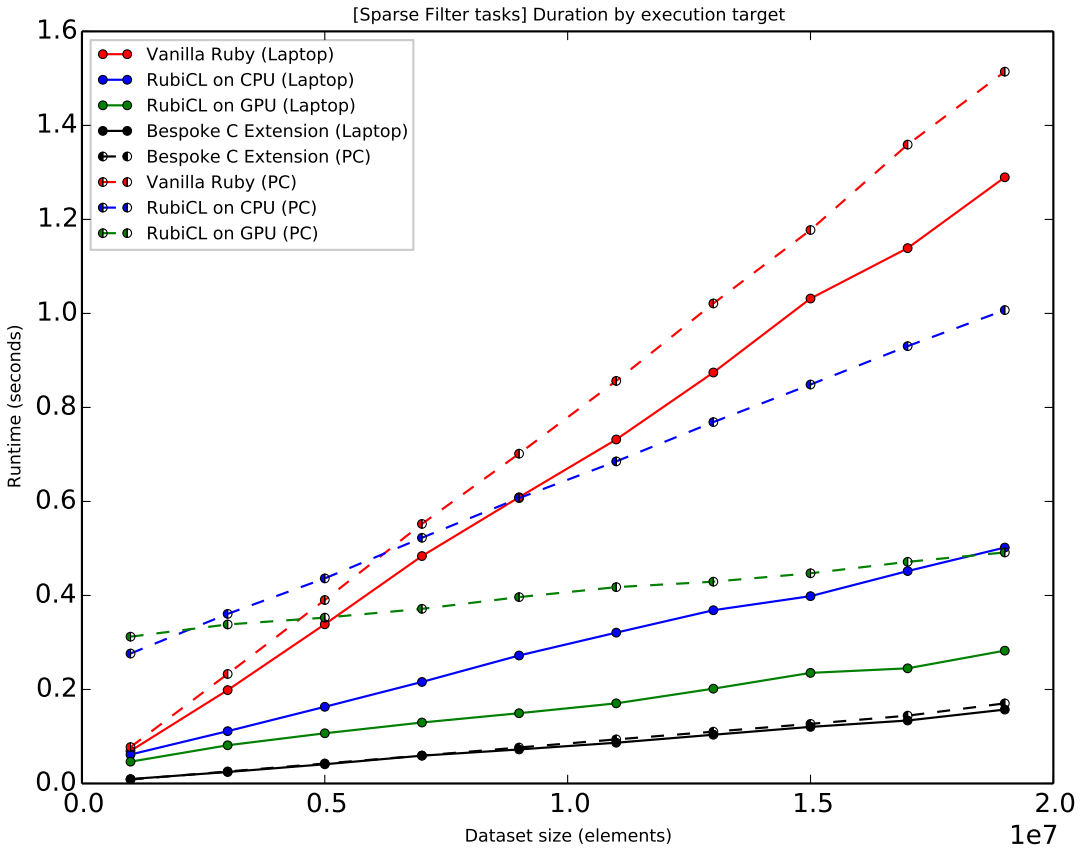


Figure 6.15: Task duration by execution target for sparse *Filter*.

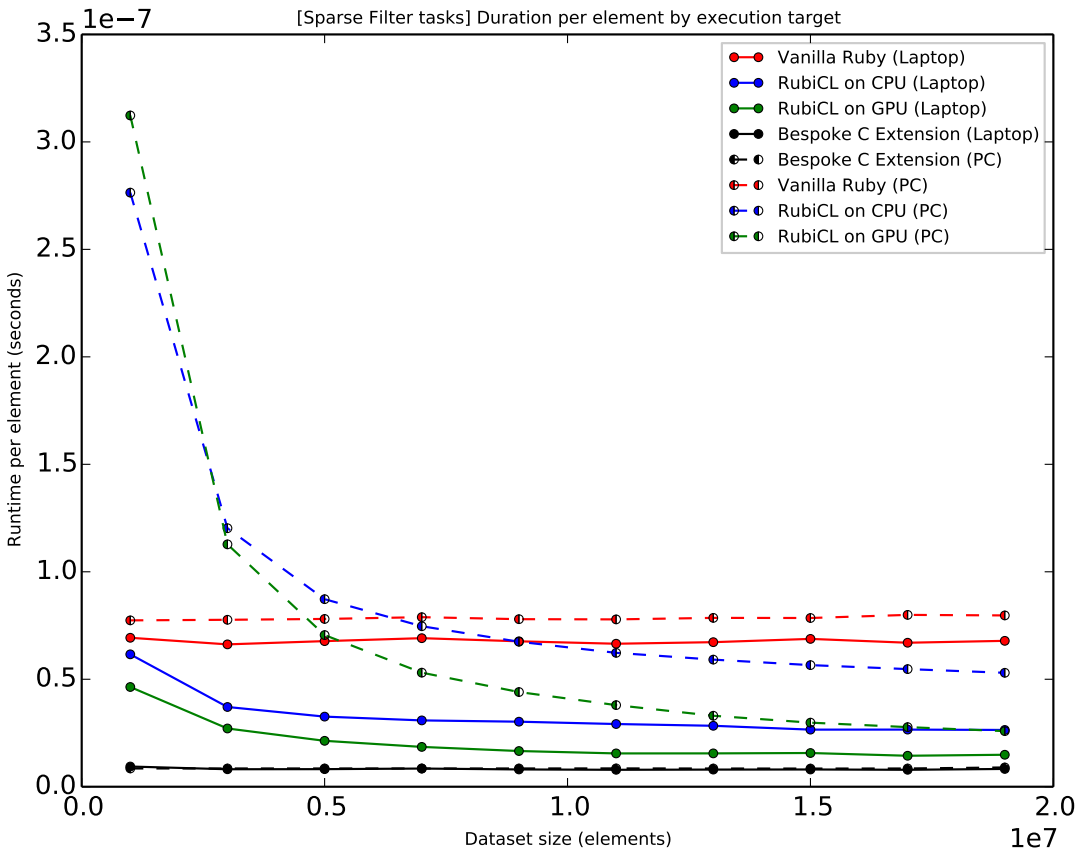


Figure 6.16: Task duration per processed element for sparse *Filter*.

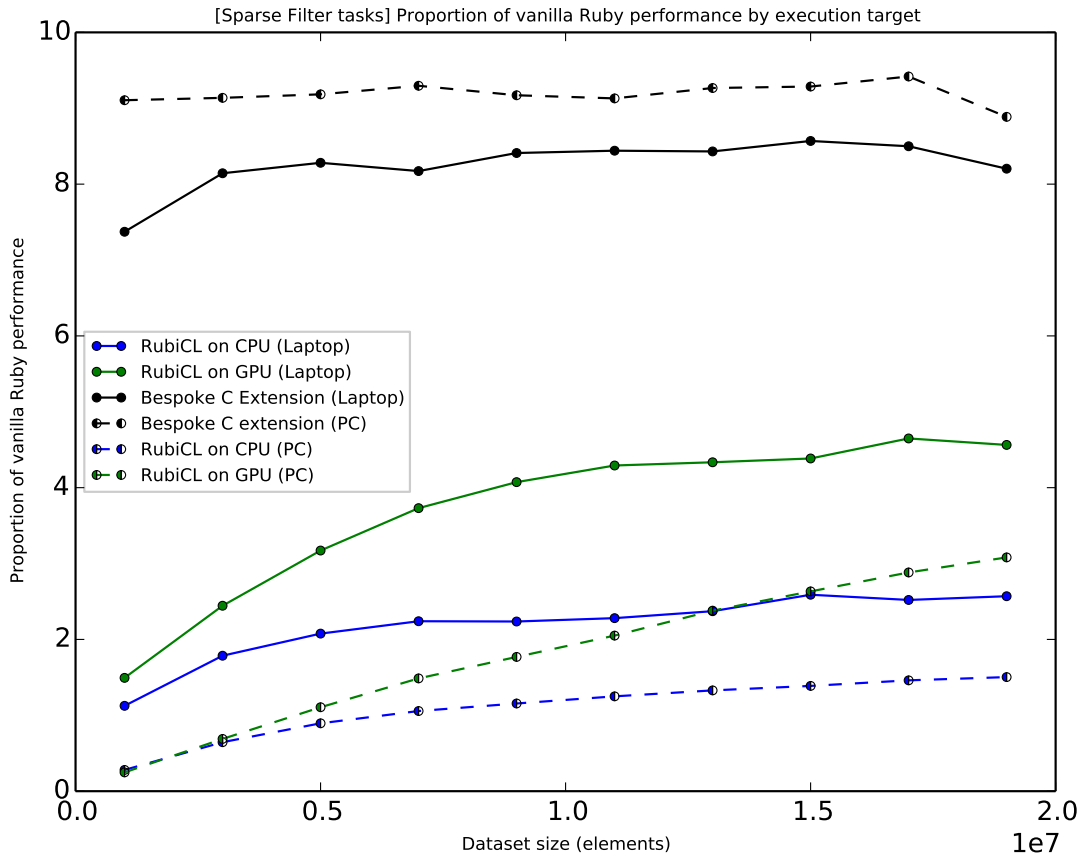


Figure 6.17: Proportion of vanilla Ruby performance achieved for sparse *Filter*.

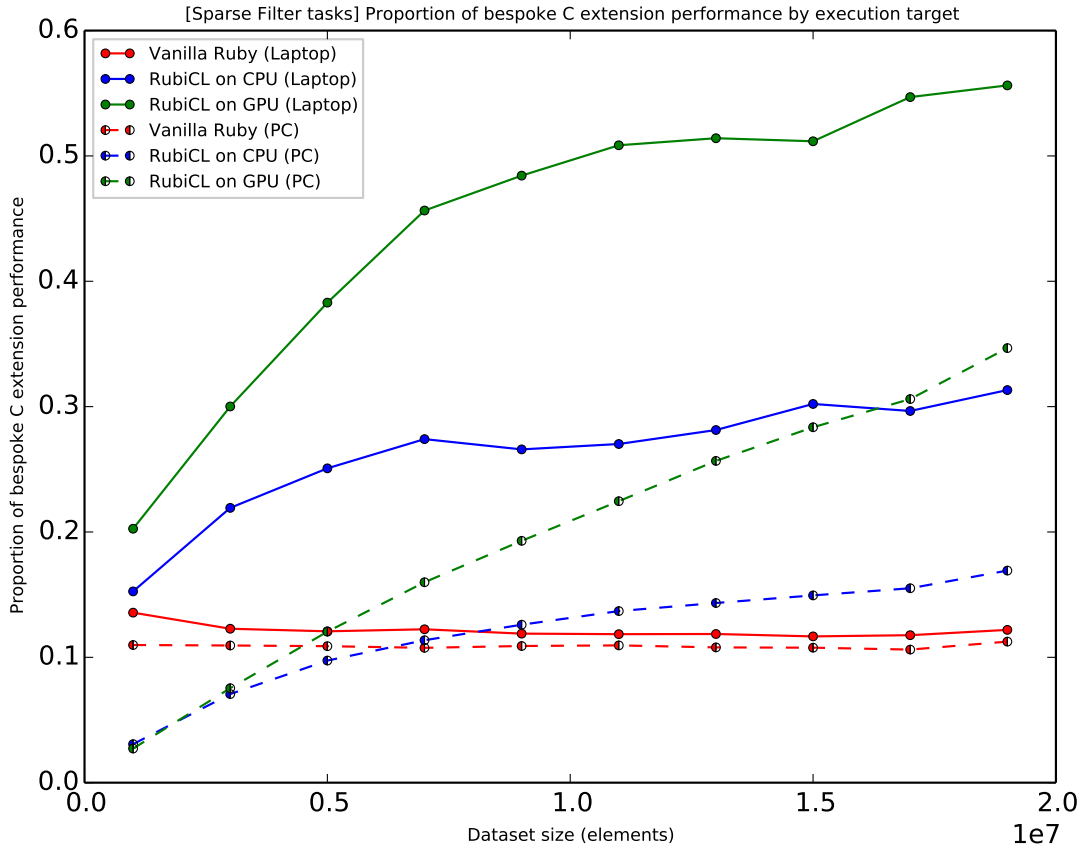


Figure 6.18: Proportion of bespoke C extension performance achieved for sparse *Filter*.

Recap of operation performed The sparse *Filter* task, returning fewer elements than the dense task, performed was the equivalent of `#select { |x| x % 20 == 0 }`. This selects all elements that are evenly divisible by 20. With the ascending range of data supplied, this will return a subset of the input dataset with 5% of its elements remaining.

Observations and analysis Figure 6.15 looks very similar to that of dense filtering. Indeed, many of the observations are the same. One reason for this is that none of the code benchmarked, including `Enumerable#select`, changes behaviour when datasets are sparse. Instead, the slight change in performance can be explained by differing proportions of time spent inserting elements or transferring device datasets when the set of returned results is smaller in size.

As before, the difference in latency between the laptop and desktop systems causes a significant proportional gap for smaller datasets. It leaves sparse *Filter* needing the same minimum dataset sizes for beneficial inclusion as dense *Filter*. This identical result is useful as it suggests that the threshold for parallelisation can be estimated without prior knowledge of the proportion of data retained.

Yet again, GPGPU devices are dominant among the OpenCL filtering implementations, for all but the smallest dataset on the desktop. Much like the previous graph for filtering, the desktop GPGPU does not appear to have plateaued in time-per-element. Therefore, further study should be performed to see at what size dataset this occurs.

The bespoke extension performs much better comparatively at sparse filtering than dense filtering. Figure 6.17 shows a 9 times performance speedup over unoptimised code, nearly twice the speedup of dense filtering. With this in mind, Figure 6.18 shows a decrease in performance of RubiCL relative to bespoke filtering, compared to the denser predicate examined earlier.

The inverse is true in Figure 6.17, with RubiCL demonstrating an improved 3–5 times speedup when executing on GPGPU devices over Ruby 2.2 for dense filtering. The differences in relative performance between dense and sparse *Filter* task implementations suggest that the cause for diverging ratios may result from there being fewer conditional insertions to the result vector. Alternatively, when less data is returned from a compute-device, the one-off latency penalty is more significant and may offset the benefits of relatively-high transfer bandwidth.

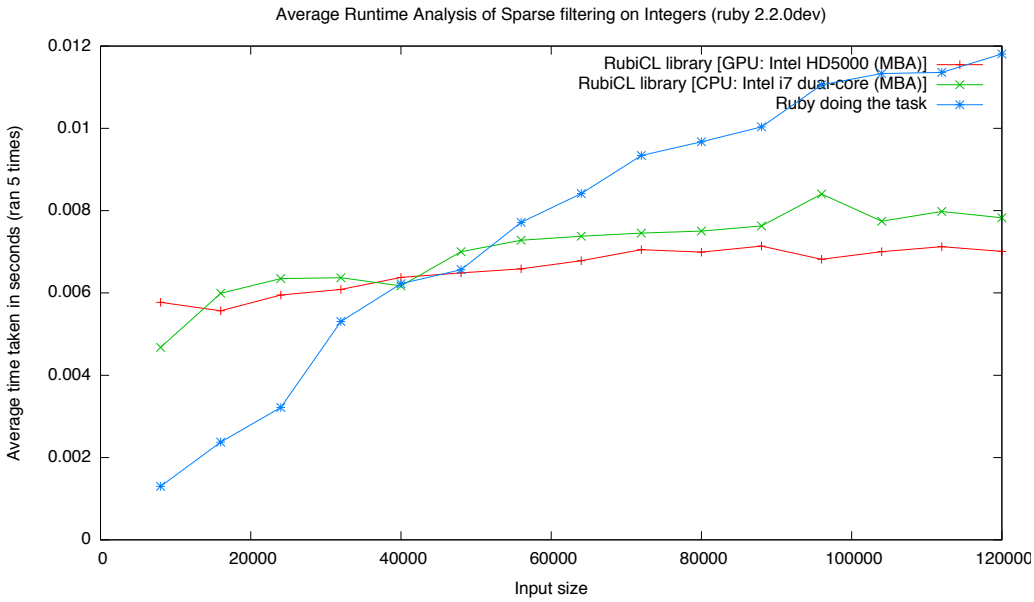


Figure 6.19: Duration for smaller-scale sparse *Filter* tasks.

Smaller scale sparse Filter tasks Figure 6.19 shows that the RubiCL library, when executing on the laptop system, is beneficial for sparse *Filter* tasks containing greater than $50,000$ elements. This is identical to the dense *Filter* cutoff. It also demonstrates that the laptop system experiences task latency of around 50ms.

6.1.3.2 Floating-point performance

Figure 6.20 shows that the lower-bound for beneficial sparse *Filter* acceleration by the RubiCL library remains roughly the same for floating-point datasets as what was observed for integer datasets. Again, comparing OpenCL execution on the CPU to standard Ruby, it is worth outsourcing computation above $50,000$ elements.

Figure 6.21 demonstrates that RubiCL library offers a speedup of around 2 times the traditional implementation. Again, this is nearly identical to the CPU speedup achieved for integer datasets and suggests that the added cost of object dereferencing and re-creation, necessary for all implementations, is insignificant compared to the work involved when performing a filtering operation.

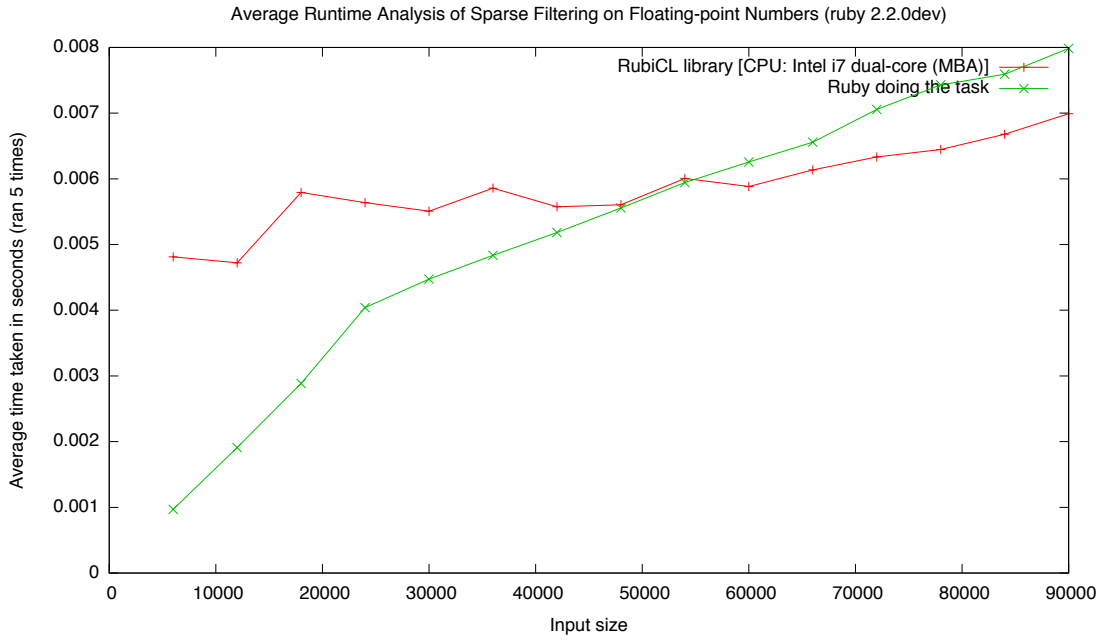


Figure 6.20: Duration for smaller-scale sparse *Filter* tasks on floating-point numbers

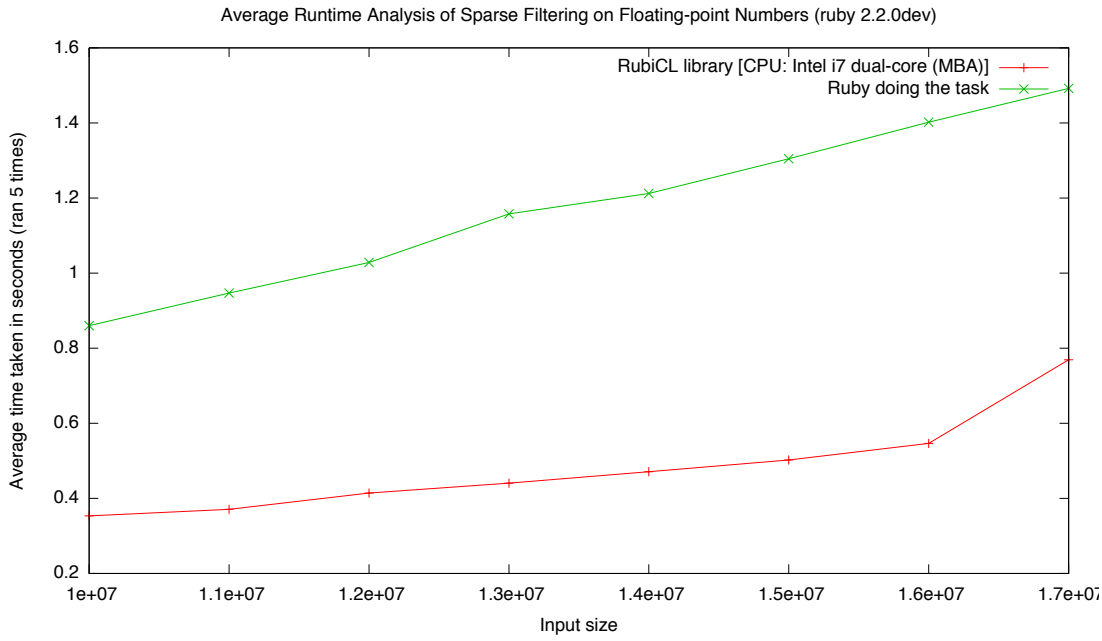


Figure 6.21: Duration for larger-scale sparse *Filter* tasks on floating-point numbers

6.1.4 MapFilter tasks

6.1.4.1 Integer performance

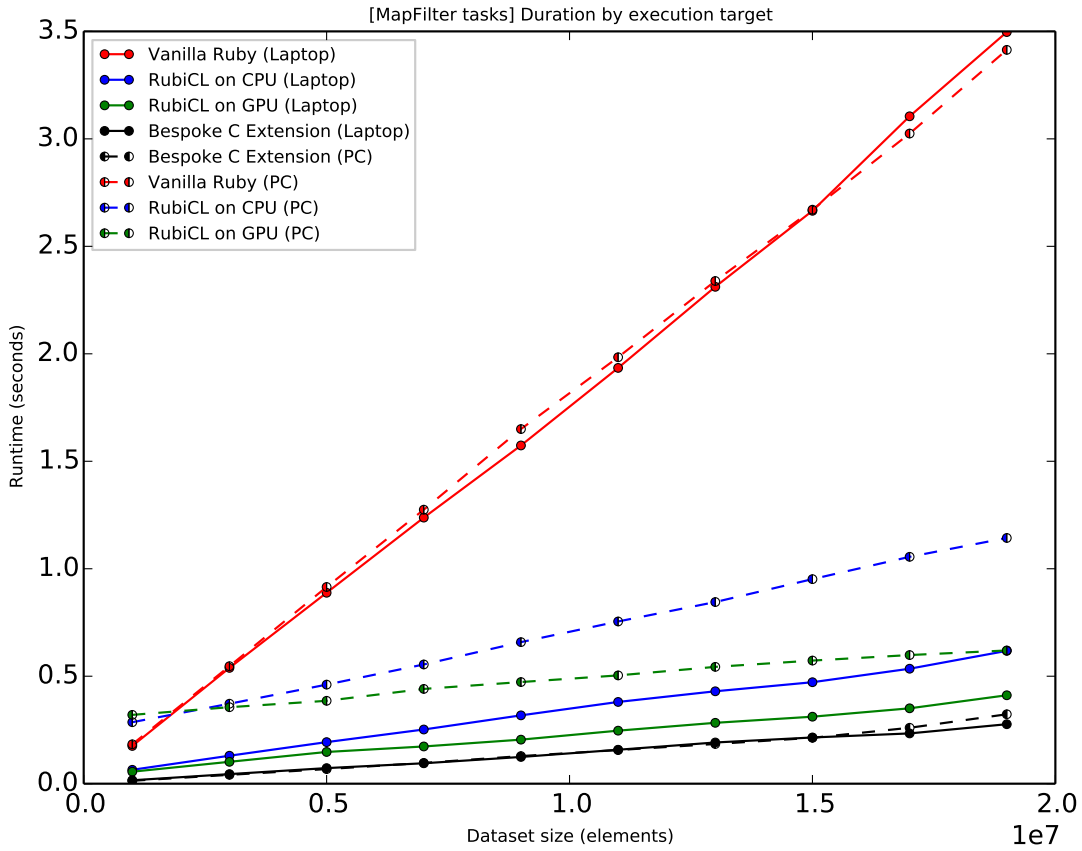


Figure 6.22: Task duration by execution target for *MapFilter*.

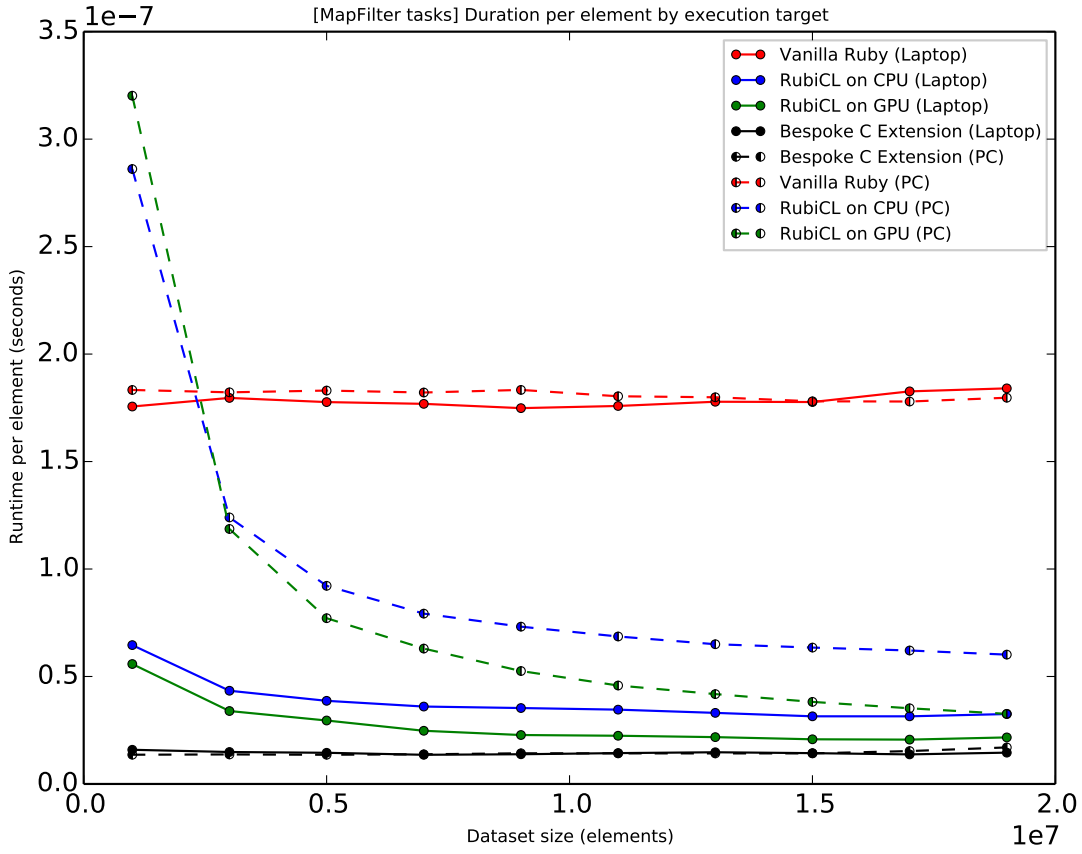


Figure 6.23: Task duration per processed element for *MapFilter*.

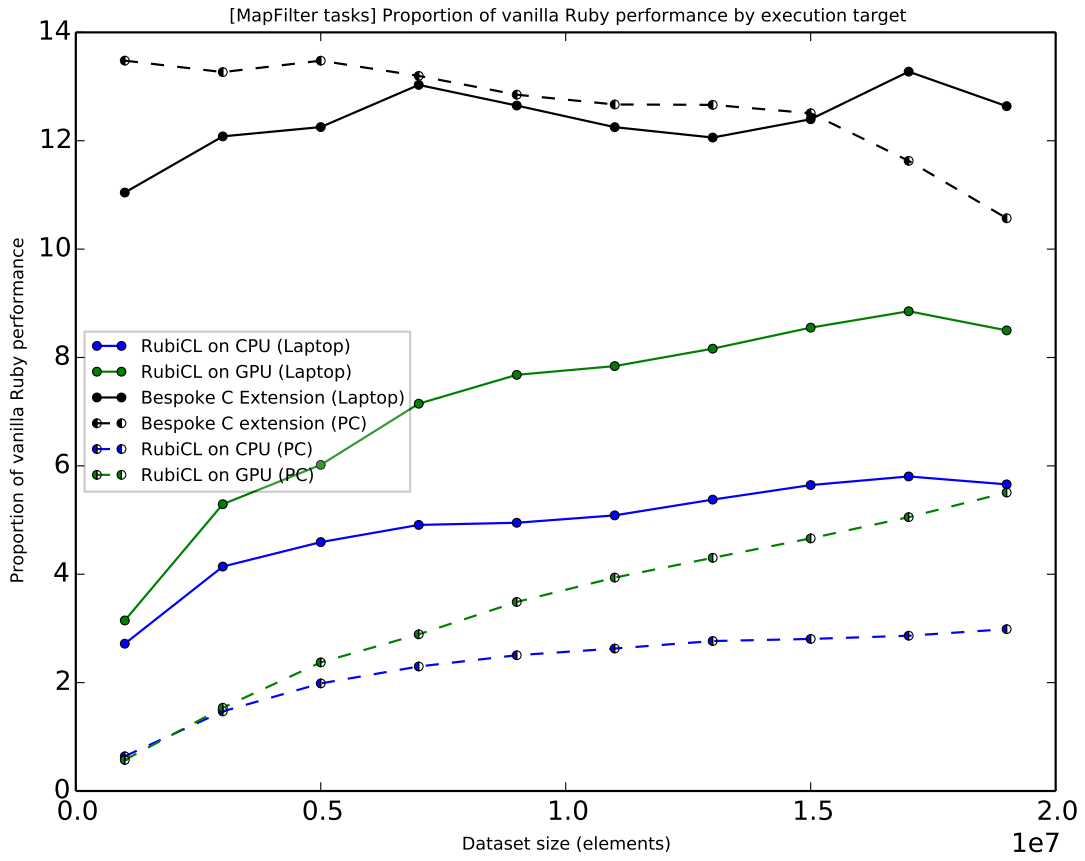


Figure 6.24: Proportion of vanilla Ruby performance achieved for *MapFilter*.

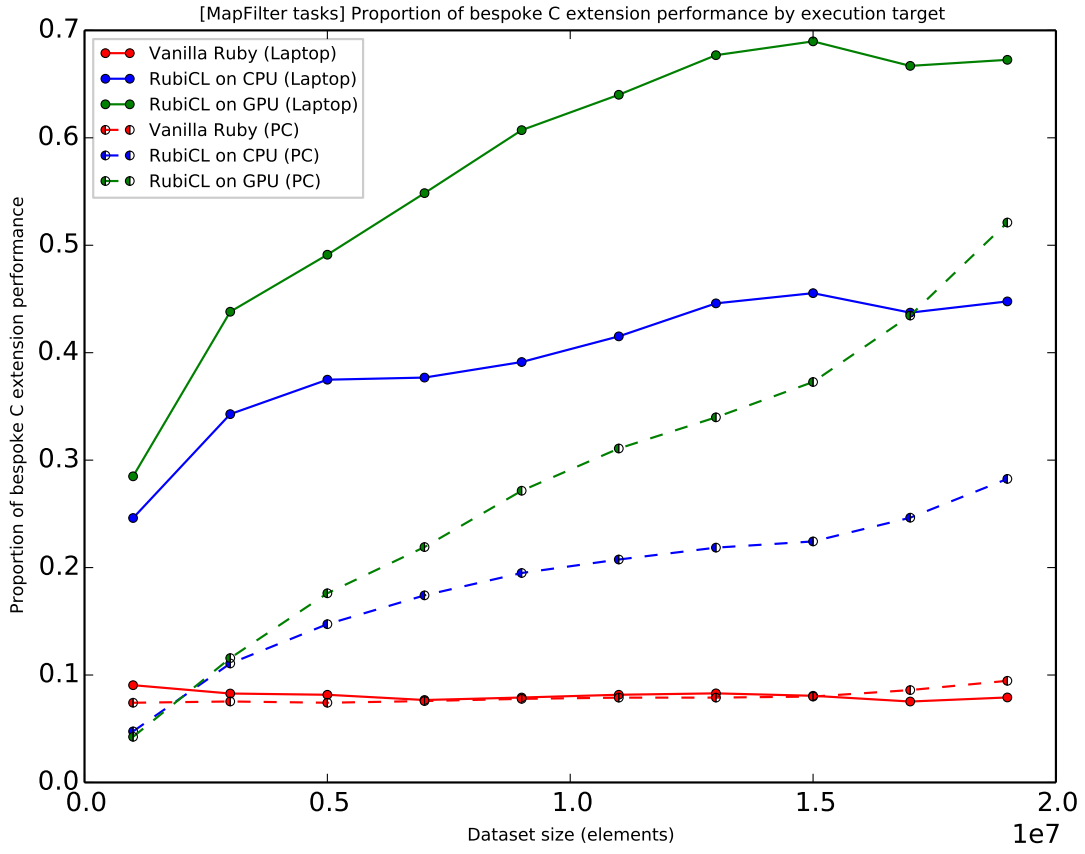


Figure 6.25: Proportion of bespoke C extension performance achieved for *MapFilter*.

Recap of operation performed The computation pipeline for this benchmark consisted of the earlier *Map* task, immediately followed by the dense *Filter* task. This has the combined effect of incrementing all elements and then returning the subset of mutated elements that are even. Again this returns a dataset that is half of the original size. Furthermore, the tasks can undergo *fusion* in order to reduce the number of kernels scheduled and required passes over the data.

Observations and analysis When a *Map* task is followed by a *Filter* task, the project's task fusion optimisation drastically improves performance. Figure 6.22 demonstrates how significantly runtime diverges, with the gap between Ruby 2.2 and competing implementations expanding greatly as larger datasets are introduced. The need for multiple passes over the data greatly delays the unoptimised code, as intermediate yet discarded results for the method pipeline are computed. The optimisation can be seen to reduce the minimum dataset length required for RubiCL's desktop parallelism to provide performance gains over the standard implementation, shown in Figure 6.24. At just 2 million elements, it is less than half of that required when filtering alone.

Figure 6.24 also demonstrates the significant throughput increases provided by RubiCL, compared to the standard library. At over 8 times speedup, combining subsequent *Map* and *Filter* tasks then dispatching the computation to the GPGPU can speed up laptop computation greatly. At nearly 5 times speedup, and the proportional graph again showing no sign of plateau, the same tactic is also highly beneficial on the desktop system used for testing.

GPGPU devices continue to dominate CPU devices, as with other *Filter* tasks. This occurs even though a *Map* task, something that CPU architecture excelled at earlier, has been prepended. It is possible that the simpler task performed earlier did not benefit from the improved device throughput, once the latency cost of context-switching computation was introduced.

Comparison with a bespoke sequential solution in Figure 6.25 shows that a large proportion of tailored solution performance is obtained on both systems. The laptop system obtains, at best, 70% performance and the desktop system obtains 50% with further indications of increasing proportion on larger datasets. This is even more significant as the manual fusion process of custom extension development is conceptually more involved than transcribing distinct tasks. As more mutation and filtering conditionals are brought into the loop body, the compiler is able to eradicate more redundant computation but the source code becomes harder to interpret correctly. With RubiCL's task fusion, separately stated pipeline stages are less concept-dense and therefore easier to understand.

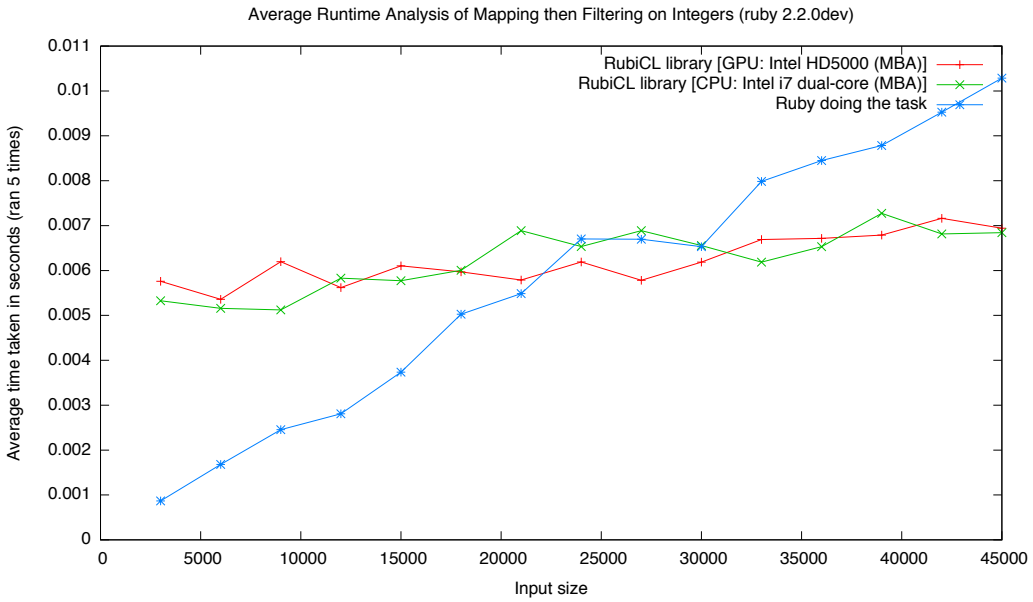


Figure 6.26: Duration for smaller-scale *MapFilter* tasks.

Smaller scale MapFilter tasks Figure 6.26 shows that the RubiCL library, when executing on the laptop system, is beneficial for sparse *MapFilter* tasks containing greater than $25,000$–$30,000$ elements. It also demonstrates that the laptop system experiences task latency of around 50ms.

6.1.4.2 Floating-point performance

Figure 6.27 shows that the lower-bound for beneficial *MapFilter* acceleration by the RubiCL library remains roughly the same for floating-point datasets as what was observed for integer datasets. Comparing OpenCL execution on the CPU to standard Ruby, it is worth outsourcing computation above $20,000$ elements, shown in Figure 6.28.

The RubiCL library offers a speedup of around 3–4 times the traditional implementation. This is nearly identical to the CPU speedup achieved for integer datasets and again suggests that the added cost of object dereferencing and re-creation is insignificant.

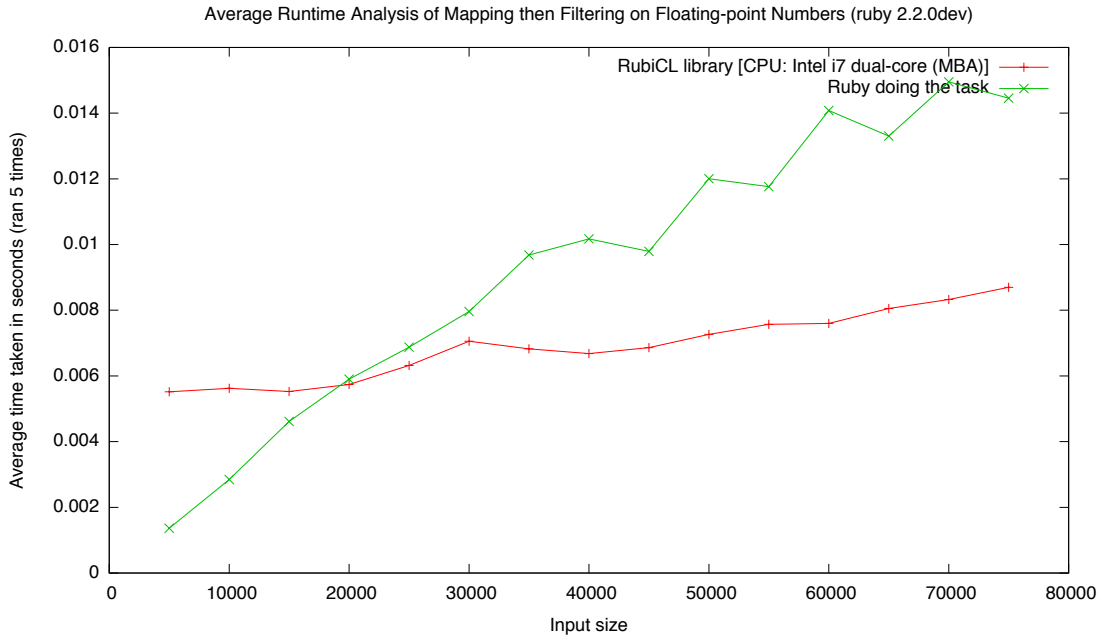


Figure 6.27: Duration for smaller-scale *MapFilter* tasks on floating-point numbers

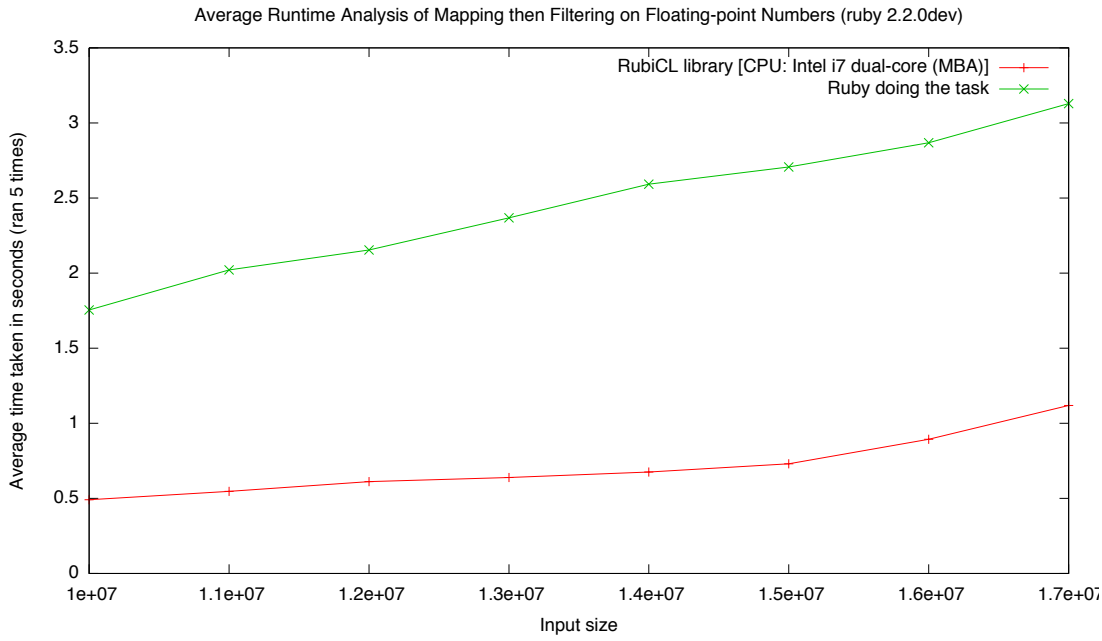


Figure 6.28: Duration for larger-scale *MapFilter* tasks on floating-point numbers

6.1.5 Sort tasks

6.1.5.1 Integer performance

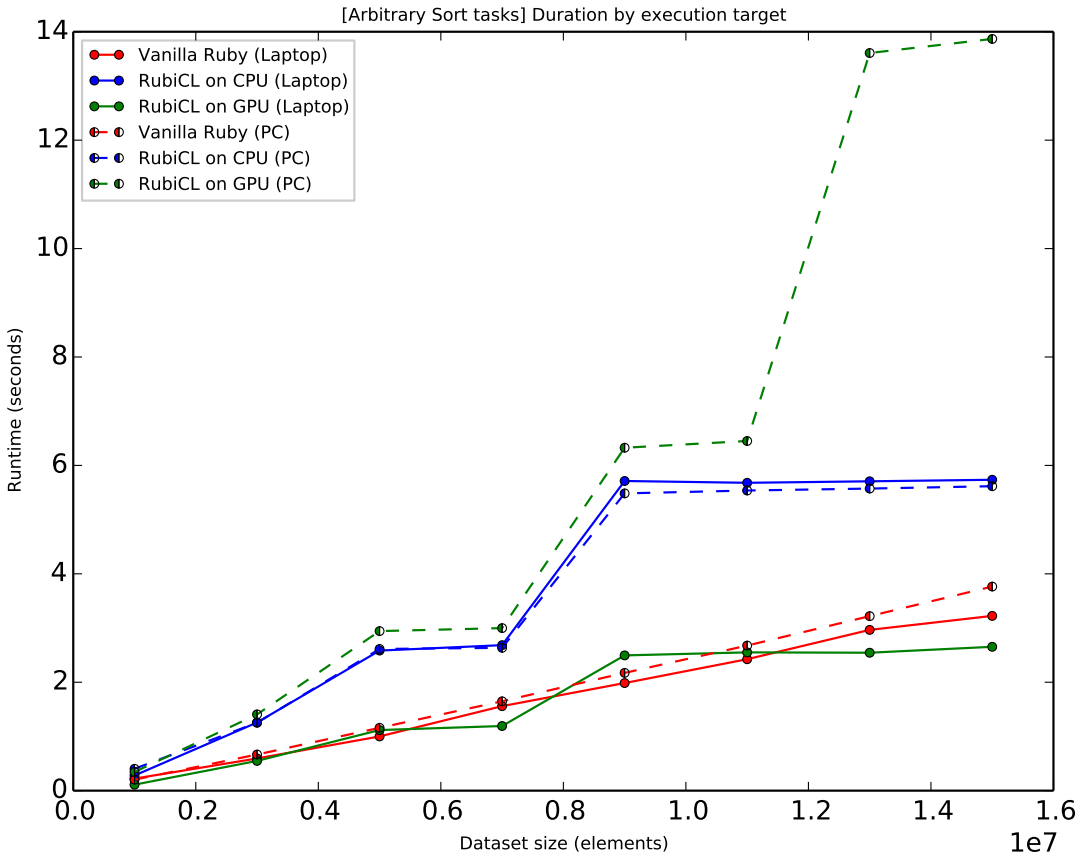


Figure 6.29: Task duration by execution target for sorting arbitrary datasets.

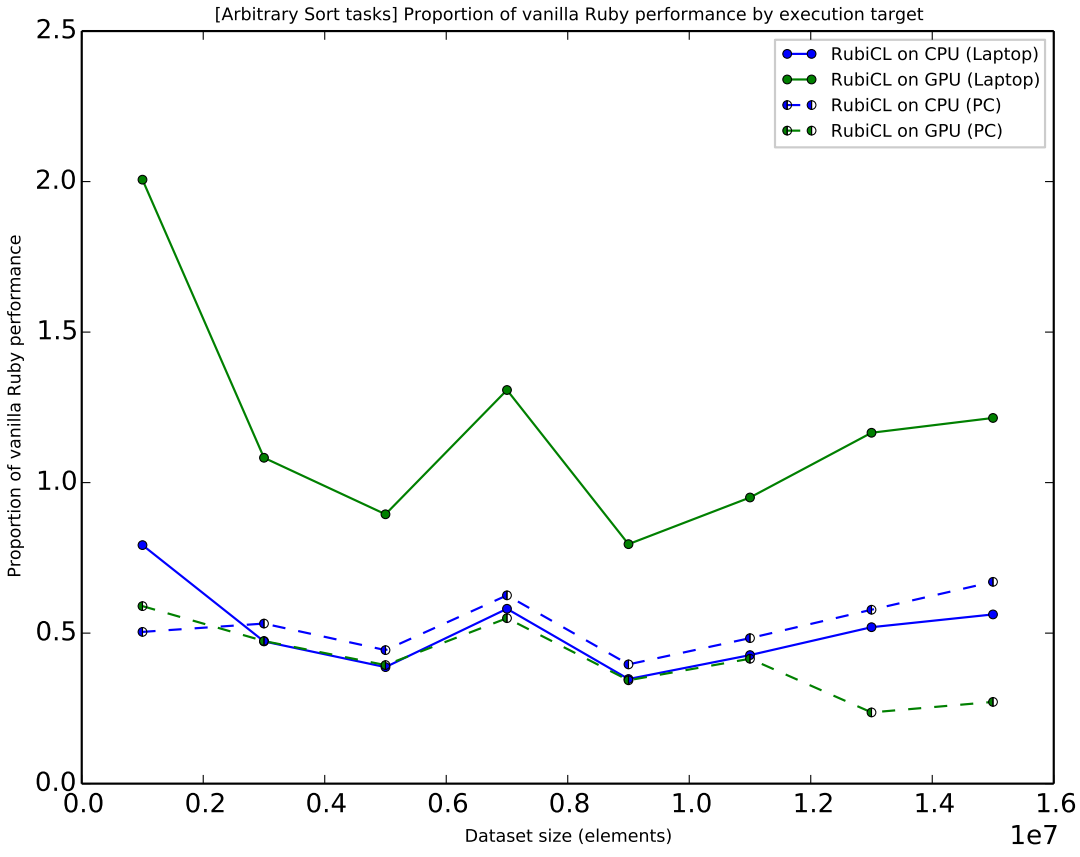


Figure 6.30: Proportion of vanilla Ruby performance achieved for sorting arbitrary datasets.

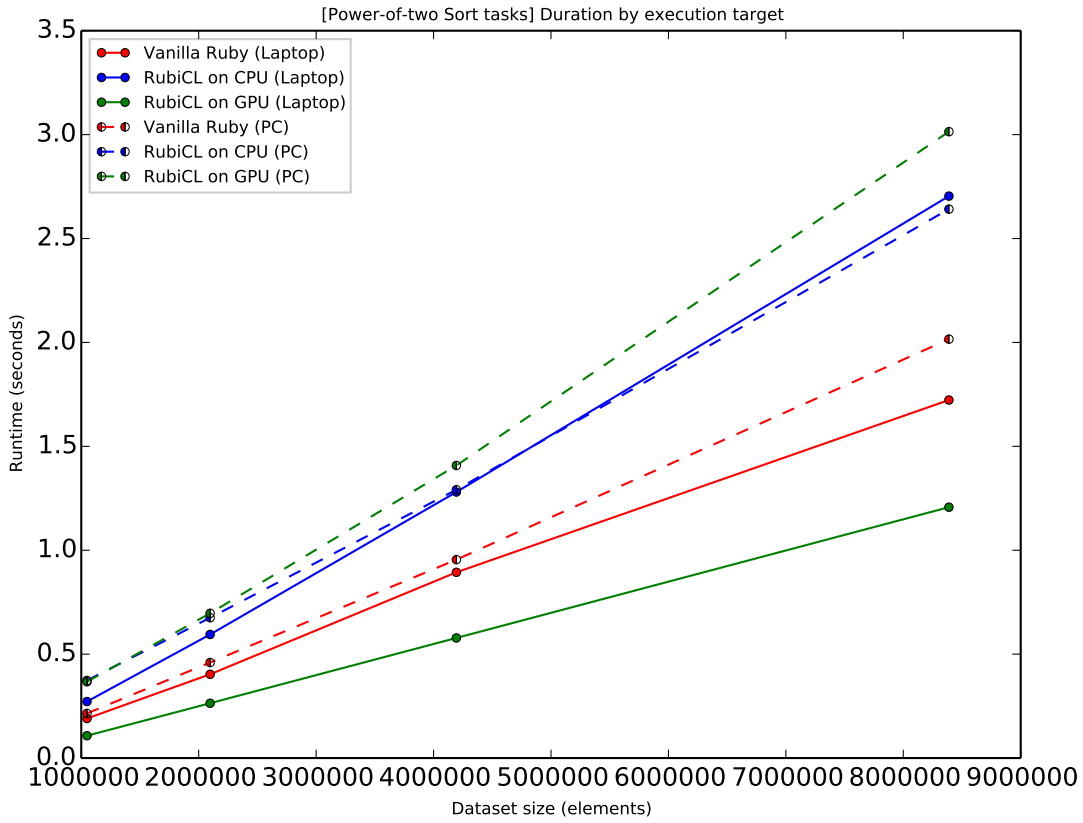


Figure 6.31: Task duration by execution target for sorting power-of-two sized datasets.

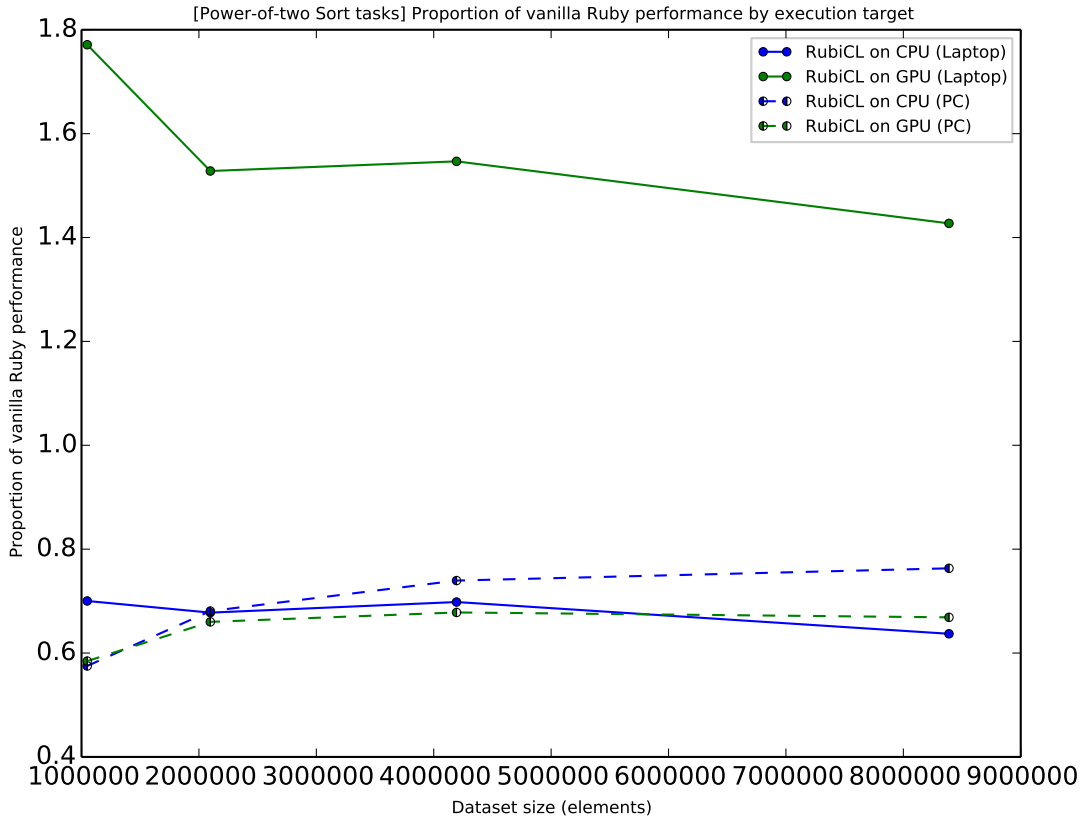


Figure 6.32: Proportion of vanilla Ruby performance achieved for sorting power-of-two sizes datasets.

6.1.5.2 Recap of operation performed

The datasets were sorted in ascending order, by triggering the `sort` method of the dataset. RubiCL's implementation of sorting uses parallel bitonic sort. Standard Ruby 2.2 `Enumerable`s rely on *quicksort*.

6.1.5.3 Observations and analysis

Figure 6.29 highlights a critical flaw within RubiCL's *Sort* task implementation. Parallel bitonic sorting requires power-of-two sized datasets for the sorting network to operate correctly. In order to utilise the algorithm for general-sized sorting, the library pads the input dataset with `MAX_INT` until the next power-of-two size.

The flaw present stems from an inefficient method of padding and unpadding the dataset. First, the input dataset is copied into the trailing segment of a larger buffer that has been prepended with padding. This is far less efficient than resizing the buffer and writing padding into the end segment, although it was unclear how this could be achieved using the OpenCL API at the time that the task was implemented. Finally, the dataset is extracted from the front segment of the buffer, as it can be assumed that all padding has propagated to the end. This action uses a buffer copy again, instead of the far more efficient method of returning a sub-buffer. All this unnecessary transfer of data produces an extraneous delay governed by memory bandwidth, and drastically decreases the performance for certain ratios of padding. The inefficient code was written in an attempt to implement the feature quickly and promptly forgotten about.

A large failing of the project in this regard was the fact that benchmarking of the sorting algorithm overlooked using non-power-of-two datasets. Due to the convenience of raising 2 to a range of powers using the `map` function, this method was used to generate seed sizes within the frequently used benchmarking scripts. Figure 6.31 shows that for power-of-two datasets, sort performance is not hindered. Therefore, the inefficiency was understandably missed.

Programming error aside, the performance of the sort algorithm on the CPU of both systems is unsurprisingly lacking. Although asymptotically identical in cost when performing *compare-split* of values at sorting nodes, *bitonic* sort requires much more work than the celebrated *quicksort* algorithm. The increase in throughput provided by scheduling across many hardware threads of a CPU was insufficient to offset having to perform a greater number of comparisons.

It is more interesting to note that Figure 6.32 shows when sorting an integer dataset using the laptop's GPGPU, performance around 1.5 times that of inbuilt *quicksort* can be achieved. It's important to realise that this does not mean GPGPU sorting will triumph over *quicksort* in other domains, though more efficient GPGPU sorting algorithms have been shown to provide significant speedup over CPU sorting[23]. However, it is more likely that in this case a combination of the lack of boxed variables within the RubiCL execution environment, combined with a still-strong sorting algorithm, resulted in the RubiCL implementation performing the task faster than the RubyVM could manage.

For some reason, the performance of the desktop GPGPU was not as strong as that of the laptop GPGPU. This is disappointing and further work should look into why this is the case.

6.2 User evaluation

Recap of the study performed 7 test-subjects were tasked with answering 5 questions each. Answers to questions were obtained by querying 2 datasets of $10,000,000$ elements each, with the assistance of the project's delivered library. This section presents the raw findings of the study, alongside interpretation and analysis.

6.2.1 Results

All applicants finished the 5 question exercise within 20 minutes. In most cases, the first question took the longest amount of time to complete, with an average of 4.5 minutes. This can be explained by most people choosing to re-read the documentation and become accustomed with library usage before progressing. The next question was generally quicker to solve, with a mean of 3 minutes, as it only involved applying a filtering stage to the previous counting pipeline. The third question took longer for most applicants, at a mean of 4 minutes, as it introduced the new concepts of zipping and summation reduction. By the fourth question, subjects were more accustomed with the library's usage and it took a mean of 2 minutes to complete. The final task was quick for most applicants also, taking a mean of 2.25 minutes. In addition, most of the time was spent de-cyphering the complex requirements of the question.

6.2.2 Test demographics

The level of programming experience present in test subjects was as follows:

- 3 applicants were final-year Informatics Master's students.
- 1 applicant was a final-year Informatics Bachelor student.
- 2 applicants were third-year Informatics Bachelor students.
- 1 applicant was a graduate of an unrelated discipline, with very little prior programming experience.

Effect on performance In general, it was found that more-seasoned programmers completed the tasks sooner. Importantly though, this appeared not to be because those with less experience had difficulty. Instead, it appeared to be the case that experienced programmers could simply transcribe their thought processes into code faster, perhaps due to greater experience of thinking about execution whilst typing.

6.2.3 Observations made

- A common theme throughout all test subjects: They avoided reading documentation as much as possible until they encountered difficulties and became stuck. This highlights the need to produce a library whereby experimentation can often yield the correct answer, as people are reluctant to have to read how something work instead of finding out for themselves.
- Forgetting to annotate a dataset with its type declaration was common for early questions. This was particularly true for subjects with prior experience of the Ruby programming language, perhaps due to the familiarity of the task causing them to write commonplace unannotated code without further thought. Luckily, this mistake became much less common after it had been made a couple of times. All subjects were less likely to make the same mistake repeatedly as the progressed through questions.
- Most people gave anonymous function parameters useless names, like x, when there was only one. In contrast, meaningful names, like $id, amount$, were used when there were several bound variables present. This justifies the project's decision to provide function parsing support, as it allows arbitrary naming of bound variables, used within calculations, by the programmer.
- Experienced programmers were more likely to state filtering predicates over several distinct pipeline stages, citing ease of readability as a motivating factor. This demonstrates the need to support *task-fusion* within a pipeline-based execution environment. Otherwise, there would be an unnecessary penalty associated with orchestrating queries this way.
- There was confusion as to how zipping was achieved in the library, again this was particularly prevalent among experienced Ruby programmers. This was due to them attempting to zip datasets after both had been annotated, despite the documentation stating that only the first dataset needs to annotated and not the method argument. This was less common with subjects inexperienced with Ruby as they read the documentation first. This suggests that, in further work, the `zip` method should be revisited and perhaps redesigned to be more explicit in usage.
- When typos were made within parsed anonymous functions, subjects understood the error message provided. This occurred despite it mentioning "method sending not implemented for `$VARIABLE_NAME`" whenever incorrectly written bound variable were interpreted as a function invocation request.
- The user with little programming experience did not find the library difficult to utilise or the questions hard to answer, after the process of stating computation as a pipeline of operators was explained. This suggests that the library may be suitable for analysts with little programming experience if they can quickly get to grips with how to orchestrate queries.
- Programmers with greater experience in the target language did a better job of highlighting inconsistent behaviour. For example, one user tried supplying a function argument to the `count` operator, something that was unimplemented at

the time as it had not been considered. This inconsistency was later fixed, but its discovery demonstrated the need for expert users in addition to novices when testing usability.

- There was no correlation between prior parallel programming ability and performance in the task. When questioned about levels of previous experience after the task, several participants stated they had no idea that the library was performing queries in parallel at all.
- There was no correlation between prior GPGPU programming experience and task performance. Again, when questioned, no participants were aware that execution was occurring on the laptop GPGPU.

6.2.4 Solution performance

Due to the questions being designed to only have one possible answering technique, all produced solutions took near-identical time to execute. Each question restarted the computation pipeline instead of carrying on, in order to keep answers distinct and easier to reason about.

The mean time of execution, on the testing laptop's GPGPU, for the produced solutions was 12.8 seconds. This compares favorably with the 49.5 seconds required by an identical pure-ruby implementation, giving a speed-up factor of 3.867.

6.3 Portability

Write about installation on desktop system (including all the pain of proprietary AMD drivers).

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. DobbâĂŹs Journal*, 30(3):202–210, 2005.

[3] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] Rspec, a testing tool for the ruby programming language. `http://rspec.info/`.

[22] User evaluation hints, a github gist. `https://gist.github.com/cronin101/9772637`.

[23] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.