MInf Project Report 1

Practical Framework for Accelerating MapReduce and Higher-Order Functions via Efficient Utilisation of a System GPU.

Aaron Cronin
University of Edinburgh
s0925570@sms.ed.ac.uk

Thursday 4th April, 2013

Abstract

This report outlines the evolving aims and progress of a project researching the implementation of large-scale data-processing paradigms on multi-core systems. The initial aim was to provide easily-exploited access to the vast number of programmable cores present in modern systems containing *Graphics Processing Units* (GPUs) and investigate how their usage can be optimised.

The project is constructing a framework that will be of use to anyone who has access to a high-performance GPU device and would benefit from an increase in throughput when processing datasets in a manner that can be suitably parallelised. In addition, the project hopes to avoid the disadvantages of similar parallel frameworks that require large amounts of user-intervention to tune execution. This need for task-specific configuration increase the barriers to entry of otherwise obtainable benefits.

After two semesters of progress following the path set in the project's proposal [2], a viable solution for co-processing higher-order functions on integers has been developed. The project is achieving the previous milestone goals set and is on track to enter the second phase of development over summer.

The implementation is being tested against the hypothesis that there exist use-cases that can be efficiently implemented on a highly parallel device whilst remaining programmer-friendly. This abstraction may be achieved by presenting a collection of simple functions that encapsulate all required complexities.

This document should present the current outcomes of the undertaken project. It should also provide insight into the schedule for activity in the immediate future.

0.1 Introduction

Physical constraints are making it increasingly difficult to continue the trend of increasing clock speed. Hardware manufacturers are responding by adding more cores to *Central Processing Unit* (CPU) chips in order to continue providing improvements to the rate at which instructions can be executed. [5] Each iteration of desktop processor seems to increase the number of hardware threads available; however, the increased core count of a modern CPU is still far lower than that of those found in GPUs.

GPUs are highly parallel co-processors designed for *Single-Instruction-Multiple-Data* (SIMD) execution on a vector dataset. GPUs traditionally use hundreds or thousands of shader units to perform functions required to render a 3D scene; though recently, frameworks such as *Open Computing Language* (OpenCL) have facilitated the usage of shader units to perform arbitrary tasks specified by a subset of C code.

With the capability of user-defined code execution on each GPU core, devices often purchased for recreational purposes gain theoretical data-processing capabilities that far exceed those of systems solely scheduling instructions on a multi-core CPU. [11]

Unfortunately, simply adding more cores in order to increase the speed at which a system can perform computation provides gains that cannot be utilised fully by common sequential programming approaches. In order to exploit the full potential for multiple threads to be executed concurrently, the programmer needs to structure data as a collection of processable entities that share no dependencies. Any calculations within a thread that require knowledge of the state of other threads are unable to continue until the shared state can be synchronised.

In addition to the locality-dependant penalty of memory synchronisation, due to the nature of SIMD execution that runs in lockstep, code written for GPUs becomes inefficient if the work kernels contain significant branching. [6] These disadvantages, force programmers exploring the boundaries of performance to adapt new paradigms and data-flow models when solving otherwise familiar problems.

Functional Programming language-features, and those inspired by them, often offer advantages when exploring parallel execution [7] due to abstraction hiding the concepts of *state* and *mutable data*. Pure functional programming provides *referential transparency* as each function can be replaced by just its result without affecting the correctness of the program. The ability to simplify long chains of computation into a series of values being mapped to outputs greatly increases the ease of verifying an algorithm, as well as highlighting computations that cannot affect the result of others and can therefore be run concurrently.

The purpose of this project is to combine the benefits provided by functional-inspired paradigms with the increased theoretical performance of GPUs in order to provide an easily-utilised library for parallelisation.

Each feature implemented shall be evaluated against existing implementations of functional languages and on similar GPU/OpenCL frameworks.

0.2 Motivation

The motivation for providing a high level framework for fast, parallel computation is to enable traditionally slow dynamic languages to be a powerful tool for scientific investigation on large data-sets.

Interpreted languages that can modify code at runtime are a powerful tool for creative experimentation as the user can modify the calculations being performed and instantly see the effect that this has on the results set. By allowing them to rival or exceed the computational power available in compiled languages using PThreads, a computer scientist can benefit from the ability to test hypotheses at runtime without suffering a noticeable performance penalty.

0.3 Goals

The project has a series of goals, selected after studying previous work in the field and evaluating what is missing from previous solutions.

Production of a Framework The final output goal of the project is to create a feature-rich platform for accelerating MapReduce and Functional Programming primitives via OpenCL co-processing.

The successful completion of this goal will be demonstrated by the framework's abilities to handle a variety of tasks in a manner that is quicker or as quick as existing solutions, without sacrificing ease-of-use.

The project currently focuses on fulfilling the feature-rich aspect of this goal. This is to avoid premature optimisation and allow features to have test-coverage and benchmark results before changes are made.

In addition to the major goal of producing a framework, the following sub-goals will influence the project's direction:

Availability on a wide variety of platforms The resulting framework should be available for use on many different systems. The project aims to provide a shortcut to high-throughput data-processing. It would be adverse to this goal if the shortcut was only available to a small subset of experimental set-ups.

The project currently achieves a wide spread of compatible platforms by using OpenCL instead of CUDA for GPGPU interaction.

The Ruby library associated with the project is simple to deploy on a system as the code can be cloned from a git repository and then native extensions built and linked via the project's Makefile. In future it will be possible to package the code and distribute it via a channel such as *RubyGems*[1] to automate the download, building and linking process even further and to provide ease of updating.

Increasing data-processing throughput The magnitude of improvement depends on the success of iterative optimisation of the framework. It is important to discover the disadvantages faced when offloading computation to co-processors, such as increased latency, so that the effects can be avoided. By allowing

the framework to switch between multiple levels of acceleration by using heuristics may provide the benefit allowing operations with disadvantages to be avoided when the benefit is not worth it.

Providing re-usable components
Functional Programming coding styles often encourage the composition of common functions to implement more complicated operations. By building commonly used primitives into the framework, the code-base can take advantage of the reappearance of certain algorithms. For example, the prefix sum algorithm is used for both folding and calculating offsets for scattered writes. Scattered writes are useful for writing MapReduce values into buckets, and folding is often used in Reduce tasks. Exposing optimised low-level functions that are needed by a variety of tasks should reduce the amount of work that is needed to be done to improve the functionality of the project.

0.4 Methodology

The framework uses a Ruby library with a C native extension that interfaces with the OpenCL API for low level calculations. Presenting a high level abstraction is handled by the Ruby code as its dynamic nature permitting metaprogramming facilitates the construction of abstractions useful for creating Domain Specific APIs.

It is felt that C is the wisest choice for this project's low-level implementation. This is due to the lack of some abstraction features benefiting code-clarity.

It is easy to produce descriptive assertion tests in the *Ruby* language. Good test-coverage of the library's computations will provide greater confidence in the correctness of the code's output each time that changes are made. With a set of tests specifying correct outputs over sample computations; it is much easier to change the architecture when attempting optimisation, as you can assert that the resulting values are unchanged.

Providing access to high-performance computation on data-sets via low-level extensions is advantageous to such languages as they often suffer compared to less syntactically-expressive languages when operating on basic types. The library should allow efficient computation without the programmer having to worry about implementing efficient data-types themselves.

The testing of OpenCL interaction will be performed on a variety of devices throughout it's development, ranging from a laptop's *HD4000* integrated GPU to a high-end GPU in a current generation desktop machine. This makes it possible to investigate how optimisations affect differing architectures and hints at which benefits can be shared throughout all devices.

0.5 Project Progress

An extension for the *Ruby* language has been produced that allows parallel coprocessing of integer vectors. The extension provides the functional primitives *Map*, *Filter* and *Fold* to the main application thread and supports asynchronous dispatch of commands and optimisation of chained tasks.

Example calling code for an operation to Map, Filter and Fold an array of $1,000,000$ integers using the GPU is provided in Figure 0.1.

In the example, the following steps are performed by the extension in order to complete the operations requested:

Conversion of data-types The integer vector input is converted into a vector of C ints or long ints depending on the load function used.

Transfer of data onto device The converted data array is then loaded into a buffer on the OpenCL device. This is done in a background thread so the calling code does not block when this is requested.

Collection of tasks to be performed As tasks are subsequently requested by the user, these are stored internally so that they can be optimised prior to dispatch.

Optimisation of tasks The user signals that no more tasks are required by calling the instance method '!'. At this point the framework examines the list of tasks needed and reduces them into a more efficient yet equivalent set of operations. In the above example, the tasks would be reduced to three tasks: One task to add 11 to all elements in the vector, one task to parallel filter the array (explained later in this document) of elements above $500,000$ and divisible by 4, and one task to fold the array into a single value.

Kernels built for optimised tasks OpenCL kernel code is generated for the optimised set of tasks to be performed. The kernel source strings are then compiled for the target device by the OpenCL environment.

Tasks performed on dataset Once again in a background thread in order to avoid blocking in calling code. The set of built OpenCL kernels are enqueued on the device in order.

Blocking until all tasks have completed When a user calls the 'output' instance method, the framework that has previously been performing commands asynchronously must block until all GPU tasks have been performed. This is achieved by the dispatch thread joining the calling code.

Transfer of data from device Now that the dataset has been processed, the result is transferred back to the host machine and subsequently the calling code.

Conversion of data-types again Before the value is returned to the host code, it is converted back from C int(s) to a suitable data-type for the target language.

Cleanup of required resources Once the value has been returned, the required kernels and the buffers used can be released by the OpenCL environment ready for the next set of tasks to be performed.

0.5.1 Summary

The succinct example code requires a large number of actions to be performed

by the framework. Most of these are required by any non-trivial OpenCL operation. The remaining are a result of optimisations to improve the performance of the system.

These tasks would normally be boilerplate provided by the programmer. With the framework performing them automatically, the difficulty of utilising the co-processing capabilities of the GPU is significantly reduced.

As well as the demonstrated code that would be interpreted from a file on disk, the framework can be used within a *REPL* (Read Evaluate Print Loop) environment in order to allow the programmer to inspect the responses of commands and choose what code should be evaluated next. This gives great power to inquisitive minds that can spot patterns from queries and use theories about the data to inspire future calculations.

0.5.2 Implementation

Map

Performing a Map task consists of generating boilerplate kernel code that replaces an element in the data array with a modification of it.

Generated code for the Map task to add 11 above is as follows:

```
__kernel void foo_task(
      __global int *data_array){
  int global_id = get_global_id(0);
  int i;

  i = data_array[global_id];
  data_array[global_id] = (i + 11);
}
```

This kernel is then executed data-parallel with the *clEnqueueNDRangeKernel* OpenCL function.

The result is each element in the input buffer is mutated in a manner specified by the transfer function (In this case 'i: i + 11', or "i goes to i plus eleven").

Filter

Performing a Filter task is slightly more involved.

Computing the presence array First, a boilerplate kernel is produced that will Map all elements to 1 if they pass a predicate or 0 if they fail, storing the result in a 'presence array'.

```
__kernel void bar_task(
     __global int* data_array,
     __global int* presence_array){
  int global_id = get_global_id(0);
  int i = data_array[global_id];

  if ((i > 500000) && (i % 4 == 0)){
    presence_array[global_id] = 1;
  } else {
    presence_array[global_id] = 0;
  }
}
```

The presence array for the vector

4	5	8	12

with the predicate
'n: n % 4 == 0' ("Keep n if n mod four is zero") is

1	0	1	1

Parallel prefix sum Since the array returned by a useful filter function is shorter than the input array, it is necessary to calculate how much buffer space

to be allocate and the offsets of the original elements within this output buffer.

A *prefix sum* is an operation that applies a binary operator to elements of a vector. In an *inclusive* prefix sum, each element is replaced by the result of applying the operator to all previous elements and itself.

Performing an inclusive prefix sum on the previously calculated presence array produces

1	1+0	1+0+1	1+0+1+1

or

1	1	2	3

The final element gives the size of the output buffer required, in this case there are 3 elements kept.

Each element of the presence array is one higher than the offset within the output array that the input element would be written to if it passes the predicate.

Producing the output vector The pseudo-code for producing the output is:

```
Compute presence array for
    input dataset and predicate.

Parallel prefix sum the
    presence array.

Allocate output buffer of size one
    larger than final sum element.

For each element in the input:
    Scattered write to
    corresponding sum offset
    in output array minus 1
    if the presence element is 1.
```

With a work efficient prefix sum algorithm, the Filter operation on a GPU can vastly outperform the equivalent operation performed sequentially.[3]

Fold

The result of a Fold operation is simply the last element of an inclusive prefix sum.

0.6 Scope for Improvements

0.6.1 Full GPU potential

Due to previously unrealised incompatibility between the initial development platform and the Intel OpenCL library, there was a delay in tuning the code to perform on non-CPU OpenCL devices. This has now been fixed by the purchase of a new development system and the immediate future will consist of tuning the framework to operate on GPUs and collecting statistics of operation.

0.6.2 Iterative Benchmarking and Tuning

Improvements such as performing the data-type conversions in parallel may also be possible if benchmarks show that they are beneficial.

A benchmarking suite is almost complete and then it will be easy to alter the execution of the framework's internals and see the how it affects the runtime over a series of inputs.

0.6.3 Intelligent Device Selection

A GPU has much higher computational throughput than a CPU but the penalty for transferring datasets across the PCI link is much higher than the memory for

a device such as the *HD4000* graphics co-processor within a latest generation CPU. The framework may be aware of when it is beneficial to suffer the penalty of offloading to GPU or when it is better to process on a more local device and use this knowledge to choose which OpenCL device to perform functions with.

0.7 Future Phase

Once previously mentioned work to improve the quality of the functional programming framework is complete, work will initially be focussed on collecting experimental data about the performance of the system against other options.

After it is judged that enough improvement has been made, or that there is no longer any obvious optimisation possible, the focus will shift to completing the MapReduce aspect of the framework.

With both paradigms supported and optimised, the final product should offer a wide variety of use-cases where data-processing can be accelerated.

Figure 0.2 shows an implementation of named tasks dispatch for functional primitives. A similar approach will be useful for MapReduce. However, the challenge is allowing expressive function declarations without overcomplicating the abstraction usage. Experimentation with a variety of styles is the best way forward here as it is foolish to prematurely commit to an abstraction style without understanding the strengths and weaknesses of it against the alternatives.

The work-flow for producing the Domain-Specific Language of the MapReduce framework is as follows: First a possible syntax idea is produced such as in Figure 0.3. If it appears to be sufficiently expressive, a simple sequential back-end solution that passes behavior tests for the functionality is written. Once it is shown that the functionality is present, each component of the back-end is rewritten to use OpenCL and parallel algorithms. The modified code is then retested against the original specifications to make sure that the behaviour has not changed.

0.8 Evaluation

0.8.1 Performance

In order to understand how the implementation performs against alternatives, both as it is being iteratively improved and prior to release, results of various performance tests shall be provided.

For each feature present, the framework will be compared against existing projects or language constructs that provide such functionality. When demonstrating a functional Map implementation, the time taken for a language's default Map implementation, alongside the results of my library performing on both the GPU and CPU as compute devices, shall be shown.

Such comparisons shall be enhanced by providing granulated details of the time taken for various stages of execution, in order to highlight which aspects scale well to GPU environments and which do not.

In addition to 'artificial' benchmarks that show how individual actions perform, several equivalent code samples shall be produced using competing methodologies. Evaluating these shall show if performing some task, such as machine-learning, can be accelerated

when including the project's framework into an existing high-level implementation - without sacrificing clarity.

0.8.2 Usability

Since the project aims for usability of the outcome and not simply the highest performance possible, several user-evaluation trials shall be conducted in order to influence design decisions.

Real-world users shall be observed applying the framework's functions to personal projects or experimental scenarios. Aspects of the provided framework that they find issues with shall be recorded. It will be investigated whether anything can be improved in these cases.

The produced framework shall be packaged in a way that makes it simple to obtain and include into projects by interested parties.

0.8.3 Portability

In addition to ensuring that the project is simple to obtain and utilise, it shall be tested on many compatible build environments and hardware architectures as possible. This shall demonstrate its suitability as a general solution.

0.9 Work Plan

0.9.1 Now - June 2013

Since I will be pre-occupied with exams, I find it unlikely that any significant progress will be made on further implementation in this time-frame. It is possible that several spontaneous ideas that could shape the direction of the project may be recorded but no coding is planned.

0.9.2 June 2013 - July 2013

In the first month of summer without revision taking up free time, I hope to finalize my work on the purely Functional Programming aspect of the framework that has been completed so far. This includes the time taken to gather metrics and optimise any discovered bottlenecks that are reducing the performance of the system. At this point I am to have the functional library in a state where I would be happy releasing it if it were the sole result of this project.

0.9.3 July 2013 - September 2013

Over the later part of summer, significant progress should be made in the MapReduce aspect of the framework. Behavioural tests for the sequential workflow examples should be completed early on and the underlying implementation details should be decided for the MapReduce engine.

The more work done in this period, the less busy the 5th year semesters will be — so large amounts of work will be advantageous in reducing the work-pressure later on.

0.9.4 Semester 1

Depending on previous progress with the framework, missing features and loose ends should be tied up in this time-period. Time should be taken to profile and optimise the implementation and to provide report material about which back-end

decisions were taken and why. By the end of this semester, all coding on the project should be completed so that the final semester can focus solely on report writing and gathering any missing data about performance without concentrating on rewriting the codebase.

0.9.5 Semester 2

Some time shall be spent convincing fellow students to attempt to use the framework for their personal projects and recording information about how successful these endeavours are. Remaining time will be spend writing documentation and the final report for the dissertation. No code modification should be necessary during this period.

If serious problems are encountered at any point during the previous semester's progress towards the framework, the first four weeks of semester 2 may be used as an emergency catch-up buffer. However this is not part of the plan and if work continues at the planned pace this will not be necessary.

```
require 'hadope'

GPU = HaDope::GPU.get

nums = (1...1000000).to_a

# Device is async loaded with ints at point of load_int
# (This can't be optimised)
GPU.load_ints(nums)
# Futher method calls push tasks onto list.
# When OpenclDevice#! is called, these are optimised
# and async dispatched in one thread.
GPU.lmap(x:'x + 1').lmap(y:'y + 10')
GPU.lfilter(n:'n > 500000').lfilter(n:'n % 4 == 0')
GPU.fold('+').!

other_result = some_time_consuming_function(nums)

# Output joins async dispatch thread and then outputs
# result of reading device buffer.
result = GPU.output
```

Figure 0.1: Co-processing integers in the background.

```
before :all do
  FP = HaDope::Functional

  @input_array = (1..100).to_a
  HaDope::DataSet.create(name: :test_dataset,
                         type: :int,
                         data: @input_array)

  FP::Map.create(name: :test_task,
                 key: [:int, :i],
                 function: 'i++;')

  FP::Map.create(name: :inverse_test_task,
                 key: [:int, :i],
                 function: 'i--;')
end

it "allows a map function to be executed on all data" do
  device = HaDope::CPU.get
  device.load(:test_dataset).fp_map(:test_task)
  output_array = device.output
  ruby_map = @input_array.map { |i| i + 1 }
  output_array.should eql ruby_map
end
```

Figure 0.2: Example of named tasks for invocation. This is a reduced snippet from the framework's behavior test suite. This style of declaring tasks is overly verbose for use with simple Map and Filter functions but named tasks will be important for using the MapReduce workflow.

```
HaDope::MapR::Splitter.create( name: :word_unit,
                               in_key: [:string, :s],
                               out_key: [:string, 's'],
                               out_value: '1')

HaDope::MapR::Reduce.create( name: :sum_word_units,
                             in_key [:string, :s],
                             out_key [:string, 's']
                             out_value: [:fold_bucket, '+'])

GPU.load_strings(text_path, [:split_at, ' '])
GPU.splitter(:word_unit).reduce(:sum_word_units).!

word_count = GPU.output_hash
#=> { "the" => 1, "input" => 1, "string" => 1 }
```

Figure 0.3: Fabricated idea of how syntax might work for MapReduce tasks.

Bibliography

[1] RubyGems Ruby community‘s gem hosting service. http://rubygems.org

[2] MInf Project Proposal

Practical Framework for Accelerating MapReduce and Higher-Order Functions via Efficient Utilisation of a System GPU.

Aaron Cronin

[3] NVIDIA GPU Gems Chapter 39. Parallel Prefix Sum (Scan) with CUDA

http://http.developer.nvidia.com/GPUGems3/gpugems_3ch39.html

[4] The Khronos Group OpenCL Standard

http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[5] A Comprehensive Performance Comparison of CUDA and OpenCL

Jianbin Fang, Ana Lucia Varbanescu and Henk Sips

2011 International Conference on Parallel Processing

[6] Architecture-Aware Mapping and Optimization on Heterogeneous Computing Systems

Mayank Daga

[7] Parallel programming using functional languages

Roe, Paul (1991

[8] NVIDIA OpenCL Best Practices Guide Version 1.0

[9] Glasgow Haskell Compiler Standard Library

http://www.haskell.org/ghc/docs/latest/html/libraries/index.html

[10] MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

[11] Mars: A MapReduce Framework on Graphics Processors

Bingsheng He Wenbin Fang Naga K. Govindaraju Qiong Luo Tuyong Wang