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

Wednesday 3rd 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 [1], a viable solution for co-processing purely functional algorithms 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.

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. [4] 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. [10]

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. [5] 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 [6] 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.

2 Project Progress

An extension for the *Ruby* language has been produced that allows parallel co-processing 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 3.

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 simply where the dispatch thread joins 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.

2.1 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 coprocessing capabilities of the GPU is significantly reduced.

2.2 Implementation

2.2.1 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").

2.2.2 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

0	1	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

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

or

0	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.[2]

2.2.3 Fold

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

3 Scope for Improvements

3.1 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.

3.2 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.

3.3 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 coprocessor 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.

4 Future Phase

5 Output and Work Plan

5.1 Immediate Focus - Spring 2013

The first stage of implementation is to produce an underlying library for higher-order functiontional programming tasks. By making it simpler to enqueue OpenCL kernels and get datasets to/from the device, this will enable easy evaluation of necessary scheduling techniques.

Once the library has implemented higher-order functions for basic data types, an initial comparison of its advantages against existing methods shall be produced.

At this point a test-suite with high code-coverage shall be built in order to provide strong assertions that given substitutions to existing methods are correct.

5.2 Later Focus - Late Spring/Summer 2013

A series of implementation possibilities shall be considered after greater evaluation of existing relevant MapReduce frameworks. After shortlisting areas of interest and constructing novel methods to investigate, a modular system shall be constructed whereby combinations of concepts resultant from these decisions can be tested.

5.3 Prior to Release - Winter 2013

After functional completeness, the focus of the project is pareto-efficient optimisation of performance, usability and portability without degrading the correctness or capabilities of the system.

5.4 Final Output of Implementation - Early 2014

The framework will be released with features explained in this proposal alongside relevant documentation.

The project's source should be released, with a suitably clean code-structure, to encourage further experimentation and allow investigation of techniques developed during the lifetime of this project.

5.5 Dissertation - Remaining Time

The research undertaken throughout this project will be presented in a report documenting the problems faced and solved by the choices made. Evaluations of the project's performance against competing platforms shall be presented in detail alongside conclusions drawn.

```
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 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 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 3: Fabricated idea of how syntax might work for MapReduce tasks.

References

[1] MInf Project Proposal

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

Aaron Cronin

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

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

[3] The Khronos Group OpenCL Standard

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

[4] A Comprehensive Performance Comparison of CUDA and OpenCL

Jianbin Fang, Ana Lucia Varbanescu and Henk Sips

2011 International Conference on Parallel Processing

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

Mayank Daga

[6] Parallel programming using functional languages

Roe, Paul (1991

[7] NVIDIA OpenCL Best Practices Guide Version 1.0

[8] Glasgow Haskell Compiler Standard Library

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

[9] MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

[10] Mars: A MapReduce Framework on Graphics Processors

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

[11] Evaluating MapReduce for Multi-core and Multiprocessor Systems

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis

[12] Phoenix++: Modular MapReduce for Shared-Memory Systems

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis

[13] StreamMR: An Optimized MapReduce Frameworkfor AMD GPUs

Marwa Elteiry, Heshan Liny, Wu-chun Fengy, and Tom Scoglandy