MInf Project Proposal

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 24th January, 2013

Abstract

This proposal outlines a project researching the implementation of large-scale data-processing paradigms on multi-core systems. The aim is 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 should provide 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.

These goals concentrate on maximising the performance and usability of a framework that results from evaluation and iteration of optimisations alongside research into existing failings.

The resulting implementation shall aim to verify the hypothesis that there exists 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.

Researching relevant advances, producing the framework, and evaluating it against similar project outcomes should require a lengthy period of management and iteration. This document should demonstrate that there is sufficient depth in research plans, suitable for a long period of focussed activity.

1 Introduction

As physical constraints make 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. [2] 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. [8]

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. [3] 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 [4] 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 Background

OpenCL is an open framework for executing tasks described by a C99 kernel on heterogeneous devices such as multi-core CPUs and GPUs. Kernels are compiled for detected on-board platforms and abstracted from underlying hardware-specific operations in order to stay platform-independent. Data-processing is distributed across the many cores of a device via the arrangement of kernel instances or *work-items* into *work-groups*.

Work-groups are a matrix of related work-items spanning one to three dimensions. They serve to partition datasets into subsets and arrange instances of tasks into sectors that can pass state between each-other without requiring synchronisation outside of their scope. An important challenge when implementing high-performance algorithms in OpenCL is understanding the flow of data between threads so that work-units can be organised into an arrangement that allows efficient memory synchronisation by exploiting locality. [5]

The OpenCL standards are published by the *Khronos Compute Working Group* [1]. Programs utilising the framework can run on any OpenCL-conformant device including Intel *Core* processors, Intel *HD* Integrated Graphics chipsets, NVidia *GeForce* Graphics Cards and AMD *Radeon* Graphics Cards. In short, most modern systems are capable of executing OpenCL kernels on at least one contained device.

Before the release of the OpenCL 1.0 specification NVIDIA produced a platform named *Compute Unified Device Architecture* (CUDA). This allowed compatible GPUs to utilise shader units to perform user-specified calculations. Since the CUDA platform is a proprietary API only designed to execute on NVIDIA hardware, this made optimisations easier to include in the implementation. In contrast to CUDA, OpenCL aims to solve the problem of varying frameworks for each hardware provider causing the portability of code to suffer. Studies have shown that with sufficiently optimised kernel code, OpenCL can perform comparably with CUDA despite it's greater flexibility of execution architecture [2].

2.1 Functional Programming

Functional Programming is a paradigm based on the composition of functions that map data from input to output values without external side-effects. When constructing a program functionally, the programmer is required to reason about the flow of data between each link in the composed chain of component methods. In addition to chaining functions to produce a non-trivial computation, many languages provide highly expressive functions [6] that describe methods of transforming data. This is achieved using a provided function that abstract from the act of iterating through a large vector of data. These entities are called *higher-order functions* as they take a function, requiring first-class function support, as an input. This abstraction is beneficial not only for readability and verifiability but, as implementation is hidden; any correct series of operations can be performed, in parallel or sequentially as long as the output value is correct. This leaves scope for optimisations that do not adversely affect the usability of the language.

Map *Map* is a higher-order function that applies a provided function to all elements in a provided input vector. It can be used to concisely describe a uniform alteration. Map is simple to parallelise since no shared state of threads is required. If other variables are required they can be shared and since no side-effects are permitted, no synchronisation outside of providing read-access to memory dependencies is needed.

Filter *Filter* is a higher-order function that applies a function that evaluates as either *True* or *False* on a dataset, returning the subset of the input vector for which the predicate is true. Once again, it is easy to perform all predicate checking in parallel. It is slightly more involved to then return the subset efficiently as state will have to be shared in order to provide a compact output that does not require a full sequential scan to extract members.

Fold *Fold* is a higher-order function that takes an array and an initial 'result' value (possibility an identity value) then repeatedly applies a combining function to produce an output. The output is equal to that obtained by repeatedly replacing the result with the output of itself and one set member using the combiner, such that each member is consumed once and the result is cumulatively generated. An associative Fold function can be parallelised to increase throughput.

Scan *Scan* is similar to *Fold* in that it takes an input vector and a combining function. However, 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.

2.2 MapReduce

MapReduce is a data-flow model and program specification for distributed processing of particularly large data-sets [7]. It is designed to assist the programmer with automatic parallelisation and scheduling built into the implementation platform, requiring only the specification of intermediary functions.

The use of MapReduce is heavily linked with collections of data that are magnitudes higher in size than any one conventional system could process alone. Its execution model relies on building up and then combining streams of data using pipeline elements that have strict input/output type requirements.

Map The *Map* task takes a (key, value) tuple input and returns a list of (key, value) tuples resulting from its specified function applied to that input. Output tuples can have any key and value, and are not required to match each-other in any way. When presented with a list of input tuples, MapReduce runs the Map task on each input in parallel.

After the Map task is complete, the *Combiner* collects all tuples from all Map tasks that share the same key and emits a (key, list-of-values) tuple that contains all the values associated with that key.

Reduce The *Reduce* task takes a (key, list-of-values) tuple and returns some number of values obtained by applying it's given function to the input.

When multiple post-combiner tuples exist, MapReduce runs the Reduce task on each input in parallel.

With a combination of Map and Reduce tasks, MapReduce can transform a number of inputs into a number of values over many machines and benefit greatly from task parallelism.

Ease of use Some of the success of the MapReduce model is attributed to its convenient level of abstraction. No knowledge of underlying parallel programming techniques is required to specify functions for transforming data.

3 Existing Work in this Field

3.1 MARS

MARS [8] is a project that implements a MapReduce runtime on GPUs. It aims to take advantage of the potential computing power present on Graphics Cards using the CUDA library.

MARS attempts to solve key barriers that will be faced when trying to produce a MapReduce platform using *General Purpose GPU* (GPGPU) frameworks.

A GPU's impressive throughput resulting from its massively parallel structure is only maintained if you can avoid tasks idling and cores being underutilised. Balancing tasks and scheduling them effectively is important. MARS attempts to load-balance work-units across a GPU in order to avoid such idling.

The traditional usage of a GPU does not involve tasks such as string processing or File I/O. In order to make such tasks possible during MapReduce on MARS, efficient methods for providing functionalities to the device kernel are developed.

Analysis Whilst *MARS* is very similar to some aspects of the proposed project, since it is implemented using CUDA it is far more restricted as to which systems can benefit from its reported performance increases over non-GPU MapReduce. This project shall attempt porting performance optimisations present in the CUDA/MARS system and evaluate how well the benefits translate to the OpenCL framework. Furthermore, this project deviates from MARS in that it will test whether there is any benefit to implementing shared CPU/GPU MapReduce purely within the OpenCL runtime, with both types of devices using the same model of data-buffers and kernel dispatch.

3.2 Phoenix and Phoenix++

The original *Phoenix* [9] project provided a shared-memory implementation of the MapReduce runtime, originally designed to distribute around a cluster. It showed that for suitable data-processing tasks, MapReduce on multi-core shared-memory systems can obtain similar performance to PThreads despite being far less involved for the programmer to implement. Although demonstrating potential in some ways, Phoenix suffered from an inefficient implementation of some 'housekeeping' aspects of the MapReduce platform including the *Combiner*.

Phoenix++ [10], a C++ re-implementation of the original Phoenix platform, aimed to rectify some disadvantages by presenting an easier-to-extend framework that allowed critical components to be tuned for greater efficiency. It resulted in a several-times performance increase over the original system via adaption to different usage patterns and a far more aggressive combiner that is triggered after every Map emmision.

Analysis This project's framework should mitigate the need for substantial configuration by the end-user, yet without sacrificing too much performance. The project may follow in the footsteps of *MARS* by showing it is possible to provide greater performance than CPU implementations like Phoenix, despite the expected optimisation difficulties caused by the usage of OpenCL instead of CUDA. The Phoenix++ paper suggests that it is important to make it possible to swap out key functions in the MapReduce pipeline in order to evaluate how some algorithms perform against others.

3.3 StreamMR

StreamMR [11] is an OpenCL MapReduce platform that takes advantage of the AMD Stream SDK to optimise performance on AMD GPUs. This repeats the earlier theme of restricting hardware compatibility in order to make vendor-specific optimisations possible. It also promises improved performance, compared to MARS, when handling intermediate results by using hash-tables instead of sorting by keys. StreamMR, like Phoenix++, allows advanced Combiner functions - in this case to help reduce the overhead of moving datasets to and from the GPU device over a relatively costly PCI-e link.

Analysis Similar to MARS, this project intends to see how much of the optimisations present in this project are AMD specific, as it would be against the goals of the implementation to rely on a particular vendor's optimisations. Experimentation with the hash-table memory model presented may provide performance benefits to this proposal's system at the cost of a more complicated memory allocation routine.

4 Methodology

My project's purpose is to present users of the framework with increased performance when performing parallelisable operations on an input dataset. It has the secondary aim of being unobtrustive to the programmer who may not wish to be troubled by the theoretical issues involved when distributing work over many cores.

I will be interfacing with the OpenCL API using C to perform the underlying function calls that are required for my framework's feature-set. Even though *Phoenix++* is written in C++ and most OpenCL code-samples online are written in C++, I feel that C is the wisest choice for my low-level implementation. I hold this view as I feel that the lack of some abstraction features in C benefits code-clarity and I feel it will not be significantly harder to port other project's optimisations to a framework written it in.

Although I shall be using my constructed library of C code to provide the capabilities of my framework, I intend to link this library to higher level languages such as *Ruby* and possibly *Python* via native extensions. I intend to present the framework to these languages as I feel that ease of use will be greatly improved by the syntactic sugar present in such interpreted languages. I also feel that they greatly increase possibilities for rapid prototyping by concisely allowing changes to framework parameters to be specified.

I have experience in producing descriptive assertion tests in the *Ruby* language. I will ensure that good test-coverage of the library's computations provides greater confidence in the correctness of my code's output each time that changes are made to the underlying framework and its higher level features. 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.

Finally I feel that providing access to high-performance computation on datasets via low-level extensions is advantageous to such languages as they often suffer compared to less syntactically-expressive languages when operating on basic types. By providing a simple method for performing low-level calculations, significant improvements to some use-cases could be provided.

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.

5 Evaluation

5.1 Performance

In order to understand how my implementation performs against alternatives, both as I iteratively improve it and when I intend to release it, I shall be providing results of various performance tests.

For each feature present, my framework will be compared against existing projects or language constructs that provide such functionality. When demonstrating a functional Map implementation, I may detail 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.

I hope to enhance such comparisons 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, I hope to produce several equivalent code samples using competing methodologies and perhaps show that the entire process of performing some task, such as machine-learning, can be performed quicker when including my framework into an existing high-level implementation without sacrificing clarity.

5.2 Usability

Since I do not wish to significantly reduce the usability of my outcome simply to provide the highest performance possible, I shall be conducting several user-evaluation trials in order to influence my design decisions.

I will observe real-world users applying functions that I will develop to requirements of personal projects or experimental scenarios. I will record which aspects of the provided framework they find issues with and investigate whether anything can be improved in these cases.

I will also attempt to package my resultant framework in a way that makes it simple to obtain and include into projects by interested parties.

5.3 Portability

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

6 Output and Work Plan

6.1 Immediate Focus - Spring 2013

The first stage of implementation is to produce an underlying library for functional programming tasks, the higher-order functions detailed earlier. 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.

6.2 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.

6.3 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.

6.4 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.

6.5 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.

References

[1] The Khronos Group OpenCL Standard

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

[2] A Comprehensive Performance Comparison of CUDA and OpenCL

Jianbin Fang, Ana Lucia Varbanescu and Henk Sips

2011 International Conference on Parallel Processing

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

Mayank Daga

[4] Parallel programming using functional languages

Roe, Paul (1991

[5] NVIDIA OpenCL Best Practices Guide Version 1.0

[6] Glasgow Haskell Compiler Standard Library

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

[7] MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

[8] Mars: A MapReduce Framework on Graphics Processors

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

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

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

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

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis

[11] StreamMR: An Optimized MapReduce Frameworkfor AMD GPUs

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