MInf Project Proposal

Practical Framework for Accelerating MapReduce and Functional Programming Primitives via Efficient Utilisation of a System GPU.

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

Wednesday 23rd January, 2013

Abstract

This document outlines my proposal for 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 researchers and students who have access to a high-performance GPU devices 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, and therefore increase the barriers of entry to 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, such as machine-learning algorithms requiring large datasets, that can be efficiently implemented on a highly parallel device whilst remaining programmer-friendly by presenting a collection of simple functions that encapsulate all required complexities.

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 canonically used 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 traditionally purchased for recreational purposes gain theoretical data-processing capabilities that far exceed those of systems solely scheduling instructions on a multi-core CPU.

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 is 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. These disadvantages, along with several others, 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 offer advantages for data-parallel execution 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 other values 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 both trivial and less trivial parallelisation.

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

2 Background

This section provides background information concerning the components and concepts required to studied in order to complete this project, as well as examples and brief evaluation of existing related work.

2.1 OpenCL

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 specific detected on-board platforms and abstract from underlying hardware implementations of operations in order to stay platform-independent across compute devices. Data-processing is distributed across the many cores of a device via the arrangement of kernel instances or *work-items* into *workgroups*.

Work-groups are a matrix of related work-items spanning one, two or 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.

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 release of the OpenCL 1.0 specification NVIDIA produced a platform named *Compute Unified Device Architecture* (CUDA), allowing 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].

2.2 Functional Programming

As outlined in the introduction, Functional Programming is a paradigm involved with 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 primitives that describe meth-

ods of transforming data using a provided function that abstract from the act of iterating through a large vector of data. These primitives 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, leaving 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 dataset. It can be used to concisely describe a uniform alteration. Map is trivial 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 trivial 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 a dataset 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 dataset member input to 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.

2.3 MapReduce

MapReduce is a data-flow model and program specification for distributed processing of particularly large data-sets[3]. It is designed to assist the programmer with automatic parallelisation and scheduling built into the implementation platform, requiring only specification of intermediary functions.

Map

Partition

Compare

Reduce The *Reduce* function takes an input of

Positive aspects of the MapReduce model are mainly attributed to its convenient level of abstraction. No knowledge of underlying parallel programming techniques is required to specify functions for transforming data.

2.4 Previous Work

2.4.1 MARS

2.4.2 Phoenix

2.4.3 Phoenix++

2.4.4 StreamMR

2.5 Ruby

2.5.1 Extensibility

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] MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat

[4] StreamMR: An Optimized MapReduce Frameworkfor AMD GPUs
Marwa Elteiry, Heshan Liny, Wu-chun Fengy, and Tom Scoglandy