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

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 aims to 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. 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 information concerning the components and concepts required to complete this project, as well as examples and brief evaluation of existing related work.

2.1 OpenCL

2.1.1 2.0

2.2 Functional Programming

2.2.1 Map

2.2.2 Filter

2.2.3 Fold

2.2.4 Scan

2.3 MapReduce

2.3.1 Map

2.3.2 Partition

2.3.3 Compare

2.3.4 Reduce

2.4 MARS

2.5 Phoenix

2.5.1 Phoenix++

2.6 Ruby

2.6.1 Extensibility

References

[1] The OSI Model: Understanding the Seven Layers of Computer Networks Paul Simoneau, Global Knowledge Course Director, Network+ CCNA, CTP