RubiCL, an OpenCL Library Providing Easy-to-Use Parallelism.

Aaron Cronin

Master of Informatics

School of Informatics
University of Edinburgh

2014

Abstract

I did some stuff that made the computer words happen faster.

Table of Contents

Chapter 1

Motivation

1.1 Introduction

1.1.1 The need for parallelism

In the previous decade, a trend of ever-increasing hardware clock-speeds fuelled developer complacency. The often-cited "Moore's Law" [1] suggested that our favourite algorithms will scale with demand, as executing systems increase in performance alongside complexity.

The stark truth is that this trend now seems to have lapsed (Figure 1.1). The latest generation of CPUs offer no significant clock-speed increases over the previous. Furthermore, improvements in per-clock performance are lacklustre. Physical hardware constraints are to blame for this disappointment. Namely, higher-than-anticipated levels of interference between subcomponents as a result of vastly increased circuit densities.

To combat this stall, hardware manufacturers have responded by increasing the number of independent execution units, or *cores*, present on produced system components. As a result, the total throughput available on a given device has continued to improve. Today's data-driven economy generates computational problems of steadily increasing size. Therefore, software engineers must adapt to utilise this increased core-count.

Unfortunately, this tactic of improving performance by presenting a greater number of compute units is often incompatible with traditional programming approaches. The inapplicability of many tried-and-tested sequential architectural patterns forces engineers to consider new ones.

Constructing a parallel solution requires the study of new concepts, such as synchronisation and data dependencies. The next generation of software engineers are becoming familiar with these issues, but there is currently a significant knowledge-gap.

The necessary switch to parallel programming is not going as smoothly as desired. A short term solution for easing this transition is providing common developers with the capability to easily utilise all compute units within a system.

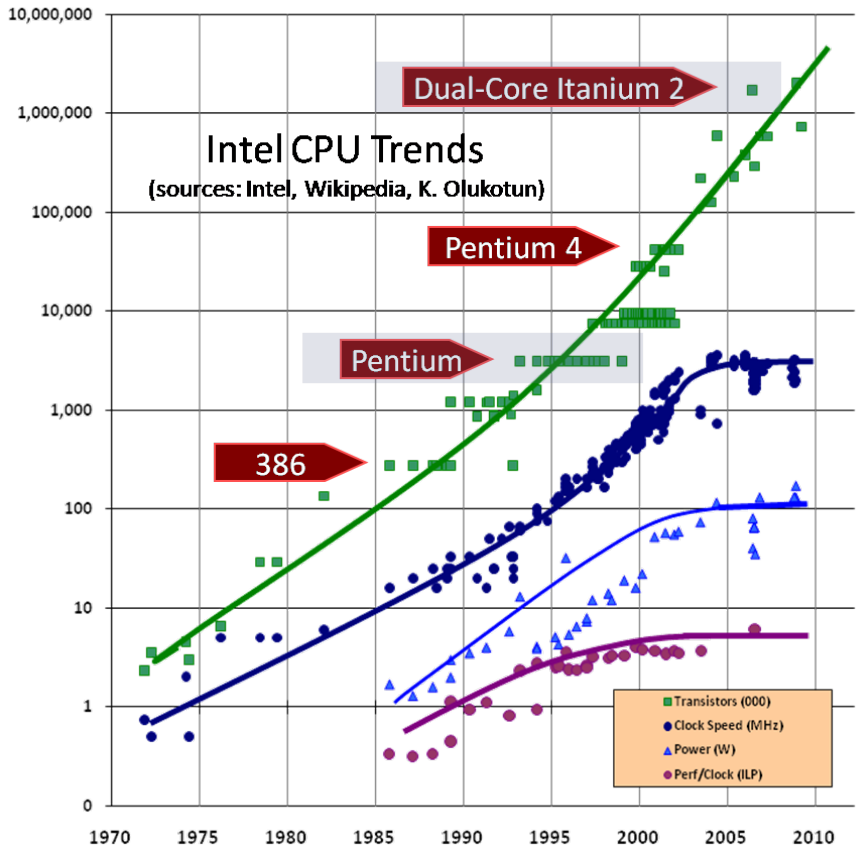

Figure 1.1: Graph demonstrating the recent plateau in clock-speed increase. Source: [2]

1.1.2 Prevalence of parallelism

As of 2014, many desktop machines contain 4-core CPUs, capable of scheduling 8 hardware threads simultaneously. Depending on whether they are aiming for performance or portability, typical laptop systems contain between 2 and 4 cores. Most commodity systems will attempt to improve performance by scheduling a user's tasks across underutilised cores, in order to avoid preemption. This still leaves sequential algorithms facing the bottleneck of a single core's rate of computation.

The other common source of potential parallelism within systems results from advances in computer graphics.

Graphics Processing Units (GPUs) are responsible for performing many computational stages of the graphics pipeline. They are highly parallel devices, tailored for high performance manipulation of pixel data. The popularity of playing games on home computers has led demand for increasingly powerful GPUs, producing a more responsive experience for consumers.

In recent generations, hardware manufacturers have explored combining specialised processing units, such as GPU, along with CPU on a single die. These hybrid devices, known as Accelerated Processing Unit (APU), often boast high transfer rates between components. They often allow modest graphical performance within a portable, low

System	Components	Discrete device count
Desktop (pre 2010)	CPU and GPU	2
Desktop	APU and GPU	3
Portable laptop	APU	2
Headless server	CPU	1

Figure 1.2: Components capable of parallel code execution, present in typical systems.

power device. As such, many laptops will contain an APU as instead of two discrete devices.

Several libraries have been developed to facilitate computation on hardware previously reserved for the graphics pipeline. As a result, GPUs capable of executing custom code in addition to their traditional roles are often referred to as GPGPUs. Lately, there has been a noticeable increase of interest in GPGPU computing and its suitability for common data-driven problems.

In short, most conventional computer systems purchased today will contain more than one available parallel processing device. A selection of common, parallel hardware configurations are detailed in Figure 1.2.

1.1.3 The holy grail of automatic parallelisation

Improvements in language design and compilation are areas of study hoping to increase the magnitude of parallel execution in the wild, without requiring user interaction.

Researchers are investigating the feasibility of discovering parallelism, inherent in user code, through analysis [3].

Whilst some breakthroughs have been made, progress is slow due to the massive complexity of the task. Languages with user-managed memory are hard to perform dependency analysis on correctly. In addition, the seemingly limitless variety of user programs greatly complicates any blanket solution.

It is likely that automatic parallelisation compilers will not become useful to a developer in the near future.

However, automatic parallelisation of code can be achieved if the scope of attempted transformation is reduced. In certain software paradigms, there is low-hanging fruit that can feasibly be parallelised automatically. This provides a stop-gap solution whilst research continues.

1.1.4 Embarrassing parallelism

Some problems are inherently parallel, containing no dependencies between subtasks. They can be dissected into a set of distinct work-units that can be executed concurrently. Such tasks are referred to as “embarrassingly parallel”.

Many *functional programming* primitives, such as `map` and `filter` are embarrassingly parallel. Any operation where the resultant state of each element in a transformed array only depends on a single input can be easily scheduled across many compute units.

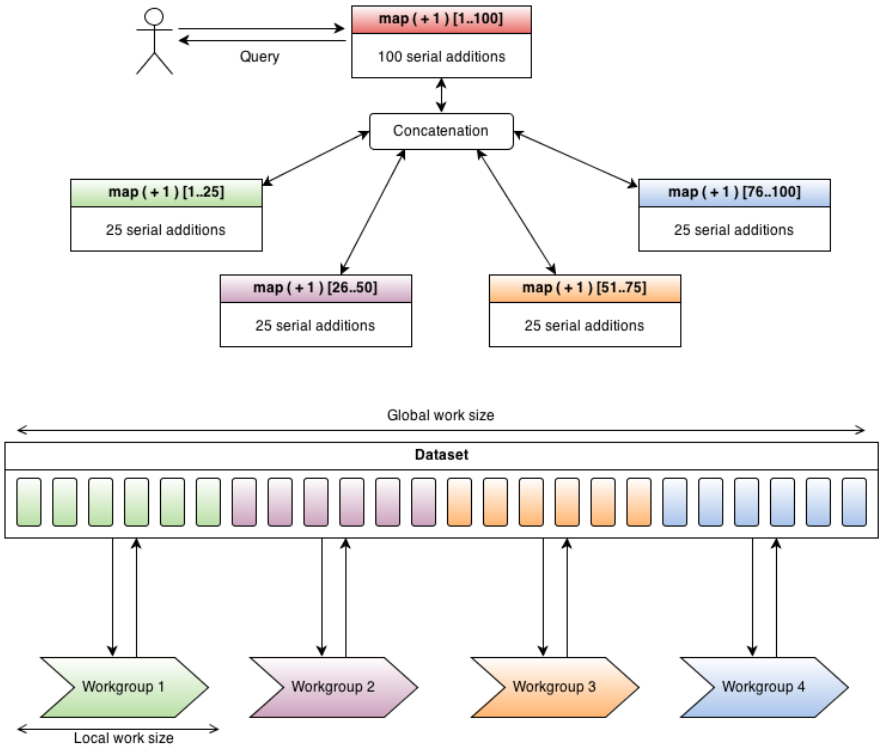

Figure 1.3: A partitioning of *map* computation over several compute devices.

Other tasks are more complicated to structure as the composition of concurrent subtasks. Computing the result to some tasks requires communication and synchronisation required parallel subproblems.

When manually parallelising code, programmers must be familiar with designing multithreaded algorithms. Subcomponents that cooperate must be produced in order to achieve processing speedup.

This effort is often unnecessary. Certain primitives are known to be suitable for parallel execution. These can be algorithmically scheduled across multiple compute units. By automating this translation and scheduling task, programmers can achieve increased throughput without the need for extensive studying and configuration.

1.2 Related Work

1.2.1 MARS

The MARS[4] project provides a MapReduce[5] runtime, executing on GPUs systems. It aims to take advantage of the significant computational resources available on GPGPU

devices. To utilise available graphics hardware, it uses NVIDIA's Compute Unified Device Architecture (CUDA) library. It one of the first research papers presenting the idea of dispatching general-purpose tasks to GPGPU hardware.

MARS attempts to overcome key obstacles, faced when trying to produce a GPGPU computing platform. A GPGPU's high throughput, provided by its massively parallel structure, is only maintained if task idling is avoided. In addition, mapping of work units must avoid cores being under-utilised and producing artificial critical paths. Balancing tasks and scheduling them effectively is important. MARS demonstrates a procedure for load-balancing work-units across GPU devices in order to avoid such idling.

One shortcoming of MARS is its reliance on the MapReduce computation pattern for general purpose tasks. MapReduce is well suited to computation on large quantities of unstructured data. However, when execution is constrained to a single device host, the redundant infrastructure provided by the runtime is no longer beneficial. The communication pattern can produce unnecessary overhead.

Another disadvantage of MARS is the need to write the individual task code as CUDA source files. This is inconvenient for any programmer lacking prior knowledge of parallel programming. To utilise MARS effectively, you must first become familiar with CUDA programming.

1.2.1.0.1 Divergences Instead of taking a large-scale computation pattern and mapping it to GPGPU architectures, this project will start by providing interfaces to primitive operations that such devices are suited for. A suite of expressive operations, composed from efficient subcomponents will then be produced.

Following this work-flow should enable the finished library to achieve a significant performance benefits, as it centres around tasks that the target hardware is well suited to.

1.2.2 HadoopCL

HadoopCL[6] is an extension to the Hadoop[7] distributed-filesystem and computation framework. Again, it provides scheduling and execution of generic tasks on GPGPU hardware. Since it uses Open Compute Language (OpenCL), as opposed to CUDA, it also supports execution on CPU devices.

One benefit, for usability, of HadoopCL over MARS is the usage of the `aparapi` library[8] to generate required task kernels. Often, composing and scheduling custom OpenCL kernels requires significant amount of boilerplate. The purely-Java Application Programming Interface (API) allows programmers to skip a large portion of this boilerplate and focus instead on the task at hand.

The fact that the interface resembles threaded Java programming is another plus. However, it still requires writing functions with logic guided by the notion of kernel execution `ids`. This does not mitigate the need to become familiar with a new paradigm for data-parallel computation. Therefore, the system is still not suitable for novice users.

1.2.2.0.2 Divergences Instead of presenting an interface for programmers to write OpenCL code via shortcuts, the RubiCL project will boast the ability to automatically transform and parallelise simple computational primitives written in native code. This may suffer from reduced flexibility, but benefits from a significantly lower barrier-to-entry for inexperienced users.

Yet, constraining the user to the MapReduce computation pattern also reduces flexibility. The lack of arbitrary kernels for common tasks is not a significant drawback as long as any parallel task primitives are varied and composable.

1.2.3 CUDAfy.NET

The stated goal of the CUDAfy.NET[9] project is to allow ”easy development of high performance GPGPU applications completely from the Microsoft .NET framework”.

CUDAfy completely bypasses the need to write custom kernel code, either directly crafted or indirectly generated through an API. It performs code generation by examining the source code of dispatched methods at runtime, translating the Common Language Runtime (CLR) bytecode to generate equivalent OpenCL kernels.

CUDAfy benefits from significantly increased usability due to generating OpenCL kernels on behalf of the programmer. However, it does not have a high enough level of abstraction to avoid vastly altering the calling code's structure. The programmer's workflow is vastly altered when parallelising calculations. Anyone writing parallel CUDAfy code must still concern themselves with explicitly detecting onboard devices. In addition, the transfer of data to and from a CPU/GPGPU must be triggered manually.

1.2.3.0.3 Divergences Instead of requiring explicit device and memory management, this project aims to ensure that programmers do not have to concern themselves with such concepts in order to parallelise computation. It should be sufficient to solely provide the calculations that are to be executed, after stating that code should run on a particular device. Requiring any more interaction increases the mental taxation resultant from using the library.

1.2.4 Data.Array.Accelerate

Data.Array.Accelerate is a Haskell project[10], and accompanying library[11], providing massive parallelism to idiomatic Haskell code. It aims to approach the performance of 'raw' CUDA implementations utilising custom kernels.

The library introduces new types for compute containers and built-in types are wrapped prior to inclusion in any computation. This allows the runtime to gather information about which datasets need to be transferred to compute devices.

It has received some significant optimisations[12] that target the inefficiencies present when an unnecessarily large number of kernels would be generated and executed, due

to composition of functions.

1.2.4.0.4 Divergences A disadvantage of Data.Array.Accelerate for general-purpose computation is the relative difficulty often associated with becoming a competent Haskell programmer. The language diverges greatly from many mainstream languages. It requires programmers to state calculations in purely functional form.

The Accelerate library offers easy transition into GPGPU programming for existing Haskell programmers. However, people with little Haskell experience may struggle to construct valid code.

To counter this, a more forgiving non-purely-functional language will be chosen for this project. Using a language that is easy for beginners to pick up will allow more people to attempt parallelising execution of their calculations.

1.3 Synopsis

1.3.1 Recap of motivations for research

1.3.2 Brief description of proposed solution

Chapter 2

Overview

2.1 Project Aims

2.2 Leveraged Components

2.2.1 OpenCL

2.2.2 Ruby

2.3 Project Progression

2.3.1 Timeline

2.3.2 Difficulties

2.3.3 Successes

2.4 Completed Work

2.4.1 Features

Chapter 3

Implementation

3.1 Differences between CPU and GPGPU hardware

3.2 System Overview

3.3 Components

3.4 Functionality Testing

3.5 Performance Testing

Chapter 4

Analysis

4.1 Benchmarking

4.2 User Evaluation

4.3 Portability

Chapter 5

Results

Chapter 6

Conclusions

Bibliography

[1] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[2] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[3] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM Sigplan Notices*, volume 44, pages 177–187. ACM, 2009.

[4] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1918–1927. IEEE Computer Society, 2013.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[8] aparapi: Api for data parallel java. `https://code.google.com/p/aparapi/`.

[9] Cudafy.net. `https://cudafy.codeplex.com/`.

[10] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[11] Hackage: Data.array.accelerate. `http://hackage.haskell.org/package/accelerate`.

[12] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 49–60. ACM, 2013.