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.

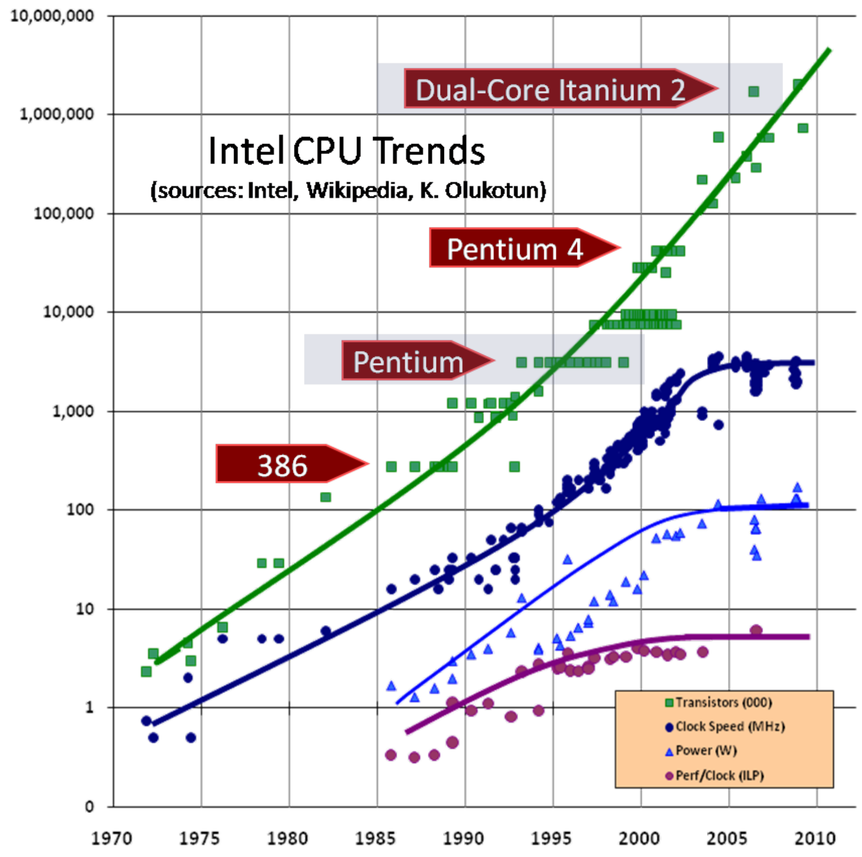

Figure 1.1: Graph demonstrating the recent plateau in clock-speed increase. Source: [2]

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.

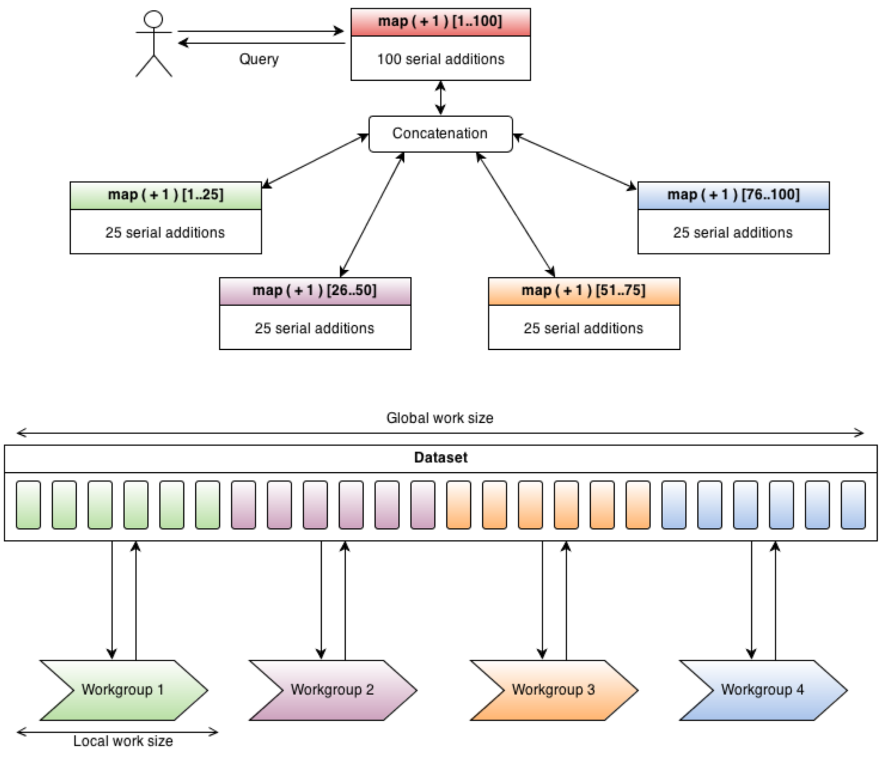

Figure 1.3: A partitioning of *map* computation over several compute devices.

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

1.3 Synopsis

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.