Automatic Parallelisation of Programs onto CPU+GPU Hybrid Systems

Benjamin Sago

A thesis submitted for the degree of MSc Computer Science

School of Computer Science and Electronic Engineering

University of Essex

October 2015

Abstract

The advent of Graphics Processing Units being used in addition to the more traditional Central Processing Units has introduced a world of complexity into software development: not only is the core programming model drastically different, but what may be efficient on a CPU may be inefficient on a GPU. Furthermore, any program that contains parallel elements must be substantially re-written in order to run on a GPU architecture.

This research aims to produce a system that will allow programs to be run without specifying which set of devices they can be run on. This will allow programs to be more easily moved between different configurations of processors, but will also allow the system to automatically determine which processor best suits a particular piece of code, producing an efficient implementation without the developer's assistance.

This system uses a custom programming language, PolyLISP, that can define individual kernels with special looping constructs, and a runtime system, PolyCube, that is able to divide up tasks and pass them off to the given processors. The target platform is CUDA graphics cards, and the target programming language is NVidia's PTX, an intermediary assembly language. Programs written in PolyLISP are compiled into kernels of PTX assembly that are connected using the dataflow architecture, which was originally designed for parallel processing.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In the beginning, there was hardware. When you bought a transistor radio, you could open it up and see a circuit schematic printed on the inside. These devices were not only specialised, but *fixed*: it was impossible, and difficult even with electrical engineering equipment, to change the function of a device after it had been assembled.

It was only the 1960s that saw the use of the *microprocessor*. A microprocessor is a *general-purpose* device, rather than a fixed-purpose piece of electronics: it could be programmed to run a different program than the one it came with. These programs were typically slower than their hardware counterparts, but as it was much easier to write software for a processor than design hardware—not to mention that a software program could be updated, while hardware can not—meant that microprocessors won. Since then, the speed disadvantage of software has become negligible as processors have reached faster and faster speeds: the average person has access to *magnitudes* more computational power than they would have decades ago. Clock speeds are getting higher, the chips themselves are getting smaller, and computer have become ubiquitous as a result. Many people are surprised to hear the term "pocket computer" describing their mobile phone.

Moore's Law attempts to formalise this increase, stating that the number of transistors on an individual chip will roughly double once every eighteen months. This approximation has, more or less, held true: as the transistors themselves have shrunk, more of them can be packed onto a single die.

However, efforts to increase the processing power of a single chip have been at odds with the physical limits of the chips being produced. [1] Processors produce heat, and the harder a chip is working, the more heat it will produce. Over the last decade, air-cooled circuits have started to reach their limits of their ability to manage heat. [Pacheco 1996]

[1] While we will eventually reach this limit, this is not the first time it has been used as a warning:

> "For over a decade, prophets have voiced the contention that a single computer has reached its limits and that truly significant advantages can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution."
>
> — Gene Amdahl, 1968, join creator of the IBM System 30 architecture and namesake of Amdahl's Law

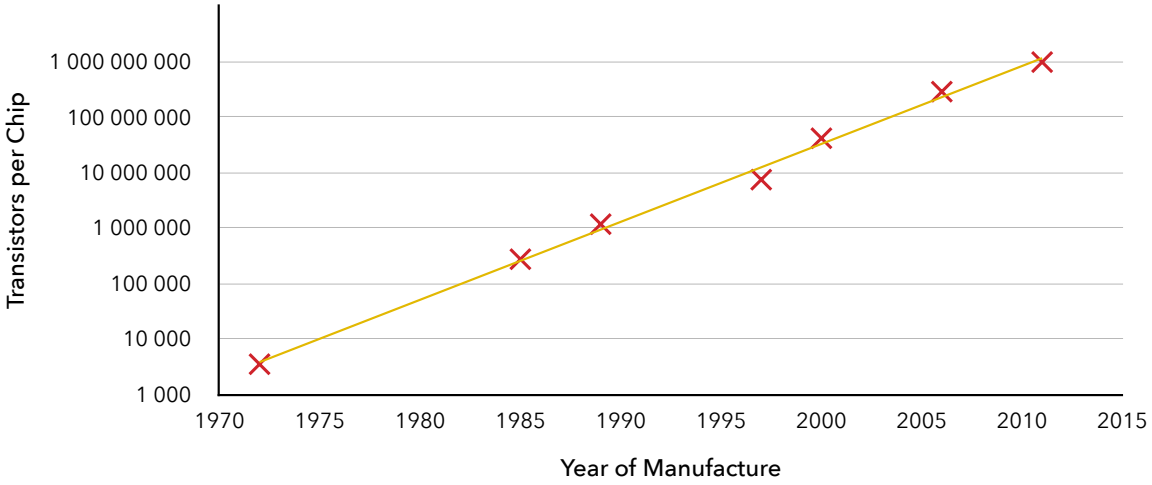


FIGURE 1.1: Moore's Law states that the number of transistors on an individual chip will roughly double once every eighteen months. This is supported reasonably well by the data: this chart shows the number of transistors on several popular consumer-class processors, along with the line of best fit going by Moore's Law predictions.

Architecture	Year	Size of Transistors	Number of Transistors
Intel 8008	1972	10 000 nm	3 500
Intel 386	1985	1 000 nm	275 000
Intel 486	1989	800 nm	1 180 000
Pentium II	1997	350 nm	7 500 000
Pentium 4	2000	180 nm	42 000 000
Core 2 Duo	2006	65 nm	291 000 000
Sandy Bridge	2011	32 nm	995 000 000

And there *must* be a physical limit: at the microscopic level, circuitry works by moving electrons, and at a certain size, wires and circuit traces would become too small for individual electrons to pass through! One solution is water-cooling, but this comes at an increased cost, and would only serve to push the limit back by several years.

So, the trend taken by consumer-level computers is to extend the computational power of a device not by increasing the clock speed of a particular processor, but by using two or more processors working in parallel.

This new trend of parallel computing has brought with it new paradigms of programming, entirely different hardware architectures, and specialised algorithms that offer increased performance when run on more than one processor. With dual-core or quad-core CPUs now commonplace, many pieces of software in use today are reaping the benefits of these new, parallel systems.

But while faster CPUs can run programs more quickly without having to make any charge to the actual program, recent parallel systems are typically only used by those who can understand them. A program written to take advantage of multiple processors may have a different design than the same program written to run on just one. This is not

only due to the fact that there are different algorithms for parallel processing, but also that programming for multiple processors involves getting many more things correct. [2] This research involves the development of a programming language that can translate a high-level program into one that can be run on various parallel systems.

[2] Programming languages with special constructs for concurrency and parallelism are explored in Section 3.4.

1.1 Identifying Parallel Machines

There have been many papers written about programming for concurrent and parallel systems, both due to these systems' long history and the number of possible configurations of parallel hardware that software can use.

The simplest parallel machine is a collection of serial processors with a *task scheduler* that divides tasks equally between the available machines: if two computations are being performed at once, the scheduler can assign one processor to each computation. In modern processors, this is just a multi-core CPU, with the operating system's scheduler occupying one of the cores some of the time.

A variant of this is *distributed computing*, which is one or more machines spread out over a network, receiving commands sent by a remote authority. The most famous examples of distributed computing are *SETI@Home* and *Folding@Home*, which manage a pool of computers over the Internet to perform long-running calculations that can be processed in parts, handing out each part to multiple computers at a time. [3]

[3] The *BOINC Project* (Berkeley Open Infrastructure for Network Computing), a middleware system initially intended to support *SETI@Home* before moving on to other applications, clocked its network of 451 250 active computers at 5.612 petaFLOPs in June 2011. [*BOINC Combined Credit Overview* 2011]

On top of this, individual algorithms can be designed for parallel processing, giving them many times the throughput of traditional, serial algorithms when run on a multi-core processor [Leopold 2001]. The advent of commodity dual-core processors meant that programs that run in multiple threads often had a speed advantage over those that did not. Today, a high-end program such as a graphics renderer that ran on multiple cores would be considered typical.

These three computing architectures—the simple serial architecture, the multiple core architecture, and the distributed computing model—occupy three out of the four quadrants of *Flynn's taxonomy*, a

2×2 grid of classifications of computer architectures proposed by computer scientist Michael J Flynn in 1966. In the taxonomy, the architectures are classified thus:

- **Single Instruction, Single Data (SISD):** This is the simplest architecture, wich parallelism in neither the instruction nor data components. SISD corresponds to the *von Neumann architecture*, which was very common in everyday computers up until multi-core processors became affordable: a single-core processor and a shared bank of RAM is enough.

  Despite not using instruction-level parallelism, some processors may exhibit parallelism by using *instruction pipelining*, which is explained in Section 2.2.

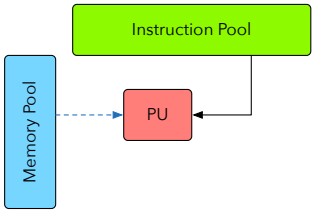


FIGURE 1.2: A diagram of the SISD architecture.

- **Single Instruction, Multiple Data (SIMD):** This is a parallel architecture that is able to perform the same operations on different pieces of data simultaneously.

  Unlike a dual-core computer, the instructions executed on a SIMD system are executed *in lockstep*: the same instructions *must* be executed by each processor this time. If the instructions must vary, then the program must also control exactly which pieces of data are affected by the instructions, or have the computer switch to the MISD model detailed below.

  An architecture does not have to be purely SIMD for this: the Simple SIMD Extensions (SSE) extension to the x86 instruction set contains extra instructions to perform SIMD arithmetical and comparison operators on multiple pieces of data, resulting in better performance than a program without these instructions. It is up to the compiler to both recognise situations where SSE instructions can be used, and to output the relevant instructions during compilation.

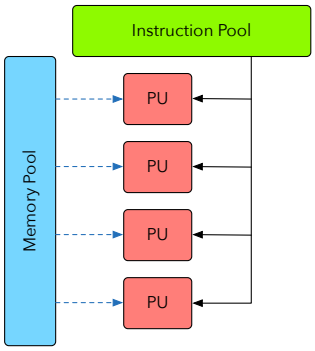


FIGURE 1.3: A diagram of the SIMD architecture.

- **Multiple Instruction, Single Data (MISD):** There are very few instances of MISD architectures; for most common parallel tasks, hardware based around the other classifications is enough.

  When MISD is used, it is usually not for reasons of speed or efficiency, but for fault tolerance: running the same set of instructions on several processors and comparing the results after execution

to check that they match. For example, in *SETI@Home* and *Folding@Home*, the same dataset and instructions are run on different computers and the results compared, with a result only being accepted as correct when enough computers have run the program.

- **Multiple Instruction, Multiple Data (MIMD):** Machines that use a MIMD architecture have a number of different processors, each of which can operate entirely independently.

  The processors in a MIMD system may operate different instructions on different pieces of data at any one time—there is no dependence between any two processors. Because of this flexibility, MIMD systems are good choices for supercomputer setups.

  A bridge between SIMD and MIMD is SPMD, or **Single Program, Multiple Data** systems. This is of particular interest to parallel programmers because it fits well in the parallel model: while the processors are no longer constrained to execute the same instructions at the same time, they are free to execute the same *series* of instructions without needing them to be run at the same time as the others.

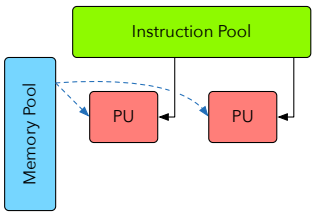


FIGURE 1.4: A diagram of the MISD architecture.

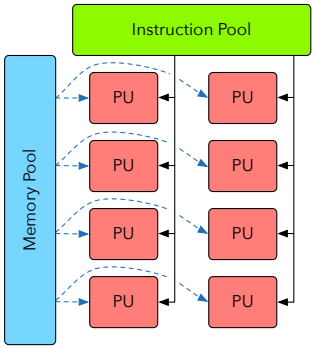


FIGURE 1.5: A diagram of the MIMD architecture.

1.2 Types of Parallelism

When adapting a program to be run on a parallel processor, there are two different approaches to specifying which units of computation are run where:

- **Task parallelism:** The scheduler partitions the various **tasks** carried out in solving the problem amongst the cores, and each core runs the task it's been given.
- **Data parallelism:** The program partitions the **data** input to the program amongst the available cores, and each core carries out similar operations on the data.

These two categories are similar, but the difference is crucial. With task parallelism, it is up to the programmer to design the program in such a way that discrete tasks can be identified and passed to a

processor. In a data parallel system, however, the programmer must design their data structures so that they can easily be handed off to several threads.

When each core works independently, writing a parallel program is much the same as writing a serial one: depending on the programming language used, it may be as simple as using parallel functions on data known to be independent. However, the situation becomes more complicated when the processors involved need to co-ordinate their work, and co-ordination is in fact inherent in many stages of programs. Synchronisation must occur when any two threads interact, in order to ensure that they are both working with the right data; for example, after passing data to a number of threads, the scheduler must wait for each thread to finish its computation before it is able to continue working with the result.

Currently, the most powerful parallel programs are written using explicit parallel constructs: threads synchronised with explicit synchronisation requests, and only tasks that are known to parallelise well are actually run in parallel. But with the advent of multiple types of hardware, the systems are becoming more and more complex, and programmers are looking towards runtime systems to manage this complexity.

1.3 General-Purpose Computing on Graphics Processing Units

More recently, computers have started to offload domain-specific programs to *accelerators*: specific pieces of hardware that can be bought, installed, and upgraded separately from the main computer. These accelerators will have certain operations implemented *in hardware*, which is not only typically faster than an equivalent program running on a CPU in software, and also frees up the CPU for other tasks.

The most popular of these is the *graphics card*, which contains its own memory and Graphics Processing Unit (GPU), which is tailor-made to perform graphical operations such as rendering scenes: turning shape information into a grid of pixels; and filters: adding effects

to that grid. The GPU is especially capable for programs that routinely perform the same instructions on multiple pieces of data due to its specialised architecture for computational-intensive calculation. This has increased the number of cores available to a developer from two or four to several hundred.

Graphics cards were originally designed to be highly-parallel renderers for specific details in computer games. Though entertaining, playing computer games is a graphically-intensive situation, as games require scenes to be rendered at 50 to 60 frames per second in order to have the player keep up. This fast-paced graphical rendering has spurred on advances in hardware to keep up with more and more graphical detail in games. [4]

Programmers realised that they were able to "cheat" the GPU into doing non-graphical parallel work by treating their data is though it were an image, then using the GPU's parallel image manipulation functions to modify it faster than they could with sequential CPU-based code, and finally turning it back into data. This practice is known as GPGPU development, standing for **General-Purpose Computation on Graphics Processing Units**.

Today, programming for one or more graphics cards is facilitated by official software development kits, such as *Common Unit Device Architecture* (CUDA) by NVidia, or the more vendor-agnostic *Open Compute Language* (OpenCL). These allow the program to implement GPU-based algorithms in a language similar to C or C++ by using vendor-specific extensions.

1.4 Research Aims and Limitations

With the prevalence of graphics cards in home computers and the growing interest in the developer community of GPGPU development, more and more programs are reaping the benefits of these new parallel systems. However, because the programming paradigms necessary to utilise the GPU efficiently are vastly different from those to utilise the CPU efficiently, few programmers are able to easily port their programs to the GPU. Even fewer are able to utilise *both* of these processors by letting the CPU and GPU work in tandem.

[4] An even more recent development is the Physics Processing Unit (PPU), which can provide quick collision detection and simulation of dynamic objects faster than a CPU could process them. However, general-purpose PPU development has not taken off, because, according to some GPUs do a good enough job. [Harris 2007]

This research ties together two different computer architectures, both different from the *von Neumann architecture* in use today: one as a more theoretical model that has had few hardware implementations, and one designed specifically for parallel execution on modern-day hardware in order to run programs more efficiently. These are:

- **The Dataflow architecture:** an abstract model of programs, that was once thought to have efficiency benefits by not specifying the exact order of expressions;
- **The GPU architecture:** an architecture based around hardware which can handle many more threads than a standard CPU, with the restriction that they must use the same instructions over multiple pieces of data.

The goal of this research is to develop a system that allows programmers to write programs for both the CPU and the GPU, using a custom programming language that emphasises computation instead of manual management of variables or threads. This would have the computer, instead of the programmer, decide which parts are to be run on which processor, letting parallelisable components of the program be run in parallel automatically.

Automatic parallelisation of *arbitrary* programs has been a long-term goal throughout Computer Science. This research is more specific in many ways: notably, the programs that it aims to parallelise are written in a functional programming language, in contrast to the many existing programs that are written in a very imperative style to match the von Neumann architecture in use today. Compilation from one to the other would be very difficult, and is outside the scope of this thesis.

Another part of this goal is the development of a custom scheduler that can view a program as a series of specialised processors that are designed to do one thing only, dependent on their input. In the dataflow architecture, these programs are referred to as *nodes*. The scheduler could dispatch discrete tasks to each individual processor, allowing not only more than one part of the program to be run at once, but for the GPU-bound tasks to be executed exactly at the moment the scheduler deems most efficient. The scheduler itself will occupy the

CPU for some operations, making the entire system use both processors simultaneously.

Finally, this research concentrates on interactions between one CPU and one GPU, instead of arbitrary configurations of hardware.

1.5 Outline

In **Chapter 2, A Low-Level Look at Parallelism**, I explore the hardware level of parallel programming: differences in the chips used and parallelisation techniques innate to both CPUs and GPUs, and look at case studies of the GPU providing superior runtime performance.

In **Chapter 3, A High-Level Look at Parallelism**, I investigate the intrinsic limitations of parallel programming, describe programming languages with automatic parallelisation techniques, as well as schedulers that are able to disperse programs between a CPU and a GPU.

In **Chapter 4, The Dataflow Architecture**, I discuss the nominal abstract model of parallel processing and list its benefits, drawbacks, and implementation details, and show how it can be used for the optimisation of programs.

In **Chapter 5, PolyCube: A Runtime System**, I explain why such a system had to be built for this research, and provide details on its interpretation and compilation capabilities, as well as the implementation details of such a system.

In **Chapter 6, Case Study: Ray Tracing**, I give results on how PolyCube's dataflow-based processing model compares against programs written for the CPU and the GPU. I use a ray tracer as an example, and list the difficulties of implementing such a program on the GPU and the details of previous GPU-based ray tracers.

In **Chapter 7, Conclusions and Future Work**, I evaluate the results in context, and offer some future work that could be based upon this research.

A LOW-LEVEL LOOK AT PARALLELISM

2

GPUs were not designed as general-purpose computing devices; they were created to give a better experience in games, rather than to change the direction of high-performance computing. The fact that a piece of hardware that was made for rendering pixels can also be used in high-performance computing can be explained by the programming models used.

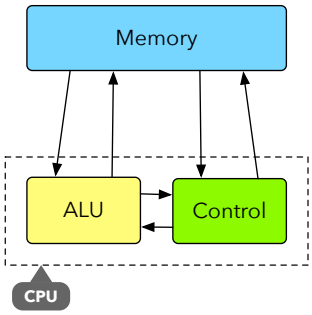


FIGURE 2.1: A simplified model of the **von Neumann architecture**. It consists of a main memory bank, modified by a Central Processing Unit (usually called a *processor* or a *core*) consisting of an Arithmetic Logic Unit and a control processor. The architecture also features an interconnect between the two, and input and output buses for any connected devices.

2.1 Modifying the von Neumann model

Rather than being completely separate branches of architectures, parallel hardware and software are *based* on serial hardware and software, which has the rather simplified task of only having to run one task at a time. When computers were in their infancy, this was literally only one program at a time: the operating system was used to *load* the program and then relinquish all control to it; the program then ran until it exited, upon which it gave the control back to the OS.

Before dual-core processors were developed, *multithreaded* operating systems used a piece of software called a *scheduler* to alternate between running several tasks on the CPU, including the OS and the scheduler itself. [1] It is important that the scheduler does not take up too much of the CPU time itself; instead, it needs to strike a balance between optimising for the most important programs and minimising its own running time.

Advocates of alternative architectures argue that having sequential execution of instructions—that is, having the order in which instructions are executed explicitly defined, disallowing any rearrangement—does not lend itself well to parallel processing, because it fails to provide efficient fine-grained synchronisation support. [R. A. Iannucci 1988] This

[1] This means that the overall performance of all the tasks running on the system is not *quite* perfect.

In fact, the maximum possible speedup is limited by Amdahl's Law, which is explained in Section 3.1, Limitations of parallel programming.

is because the number of *synchronisation events* would grow too large for the processor to handle.

In any concurrent or parallel procedure, there must exist a common ground that all threads must recognise and wait for: a *synchronisation point*. As the parallelism becomes more and more fine-grained, the number of performable operations is limited by not only the number of synchronisation events that the processor must wait for (and hold the details of in memory), but also the cost of each context switch caused by a synchronisation operation.

For this reason, most von Neumann machines employ large-grain parallelism, using *interrupts* for synchronisation between threads. The interrupts allow the processor to keep track of far fewer synchronisation events, as the cost of having larger individual workloads. [Arvind and Robert A. Iannucci 1988]

2.2 Parallelism at the CPU level

The von Neumann architecture is not going away; rather, it is being adapted to fit in a parallel world as more opportunities for parallelism present themselves. For example, *instruction-level parallelism* is a method of improving processor performance by simultaneously executing more than one instruction.

CPU execution is notionally sequential—one instruction's execution must begin after the previous instruction's execution ends. The effects of one instruction, such as assigning a value to a register, may be necessary to set up the correct state for the next instruction to run. Hardware engineers have been able to speed up this process by *pipelining* the latencies of each instruction so that different stages can be executed at the same time. A typical processor has the following five execution stages: [2]

- **Fetch:** Get an instruction from the instruction cache.
- **Decode:** Read the type of operator, and the locations of any registers, from the instruction's data.
- **Execute:** Actually execute the instruction.

[2] Note that it only *may* be necessary to have one instruction before the other: it is easy to imagine programs with some parts that can be executed in any order, but must have *some* arbitrary order. Mathematically, these pairs of instructions have a *partial order* instead of a *total order*: compiling instructions with a designated partial order provides many opportunities for parallelism.

Examples of this in programming languages are outlined in Section 3.4.

- **Memory:** Read from the memory or data cache, if required.
- **Write Back:** Write the result, if any, back to a register.

If each stage were to take one processor cycle, then each instruction would take five processor cycles of time. But by overlapping the times of each stage's execution as it is idle, more than one stage can be run at the same time: while one instruction is being executed, the next instruction could already be fetched.

As an example, consider the sequence of instructions representing the computation $f(c \times (a + b), d)$. Some of these instructions depend on the results of others, but some do not. These instructions can only execute when all the source operands have been loaded into registers; if they have not, they cause a *hazard*, where execution must wait for the data loader to catch up. [Hennessy and Patterson 2003] The result, shown in Table 2.1, illustrates the effect of stalls bubbling down to later stages of the pipeline: instructions are only able to be executed when all of their registers are available.

A modern compiler is able to re-order the instructions to increase the amount of instruction-level parallelism and eliminate hazards to maximise the performance of the program. But even with optimisations like these, the performance of single-threaded programs has plateaued. Standard algorithms that would run efficiently on a multi-core CPU would run inefficiently on GPU hardware.

2.3 Parallelism at the GPU level

The GPU's capabilities lie in its efficiency of executing the same program over multiple pieces of data, instead of running several different programs at the same time. Because of this, the CPU can be left to handle the commonplace tasks, using the GPU when a program requires fast processing of one single task.

There are a number of ways in which the GPU is designed to cope with its task. These range from its different physical hardware, to how it handles hazards (which, like for the CPU, are interruptions in the execution from caused by a busy register or data that needed to

	Fetch	Decode	Execute	Memory	Write Back
1	`ld r0, [a]`	–	–	–	–
2	`ld r1, [b]`	`ld r0, [a]`	–	–	–
3	`add r2, r0, r1`	`ld r1, [b]`	`ld r0, [a]`	–	–
4	`ld r3, [c]`	`add r2, r0, r1`	`ld r1, [b]`	`ld r0, [a]`	–
5	`mul r4, r2, r3`	`ld r3, [c]`	STALL	`ld r1, [b]`	`ld r0, [a]`
6	STALL	STALL	STALL	STALL	`ld r1, [b]`
7	`ld r5, [d]`	`mul r4, r2, r3`	`add r2, r0, r1`	STALL	STALL
8	STALL	STALL	`ld r3, [c]`	`add r2, r0, r1`	STALL
9	STALL	STALL	STALL	`ld r3, [c]`	`add r2, r0, r1`
10	`push r5`	`ld r5, [d]`	STALL	STALL	`ld r3, [c]`
11	`push r4`	`push r5`	`mul r4, r2, r3`	STALL	STALL
12	STALL	STALL	`ld r5, [d]`	`mul r4, r2, r3`	STALL
13	STALL	STALL	STALL	`ld r5, [d]`	`mul r4, r2, r3`
14	`op 4`	`push r4`	STALL	STALL	`ld r5, [d]`
15	`pop r4`	`op 4`	`push r5`	STALL	STALL
16	STALL	STALL	`push r4`	`push r5`	STALL
17	–	`pop r4`	STALL	`push r4`	`push r5`
18	–	–	STALL	STALL	`push r4`
19	–	–	`op 4`	STALL	STALL
20	–	–	STALL	`op 4`	STALL
21	–	–	STALL	STALL	`op 4`
22	–	–	`pop r4`	STALL	STALL
23	–	–	–	`pop r4`	STALL
24	–	–	–	–	`pop r4`

TABLE 2.1: Here, the instructions that do not rely on the result of another will run parallel to it in the pipeline. This is an example of instruction-level parallelism. In processor cycles 1 and 2, the load instructions are independent, so are executed without causing stalls. The add instruction that follows them requires that both of those values are available in registers, causing two stalls to flow down the pipeline, and finally executes them when both are finally available (cycle 7). The following load is *not* a hazard, as it has no data dependencies with the other two loads; however, the multiplication requires the two preceding instructions to be completed, and another bubble of stalls occur.

be loaded), as well as the differences in the programs that are run on it. These ways separate the GPU from the CPU, and they are outlined below.

2.3.1 SIMD Execution

To add more processors onto the von Neumann model, one can give each processor bank memory, or allow every processor to access a shared pool of memory.

The GPU is able to handle such a large number of cores compared to traditional CPUs by specialising its architecture for computational-intensive calculation; it has many more transistors dedicated to data processing rather than caching or flow control. The CPU, on the other hand, uses those elements much more often, so much more of the

actual hardware is dedicated to them. [NVidia 2010] Its restrictions on its thread model also protect against the concerns listed in the previous section.

In other words, the GPU throws away the common case of having to shuffle pieces of data around in memory in order to focus on pure computation instead—specifically, the same program being executed on many pieces of data in parallel—because the CPU can handle the other tasks for it. Flow control is less important when only one program is running at a time.

This is the same model as the SIMD quadrant in Flynn's taxonomy—Single Instruction, Multiple Data. The data is copied in as a texture, and is formatted as a vector of four floating point numbers, corresponding to the red, green, blue, and alpha channels of a pixel. [Wilson and Banzhaf 2008]

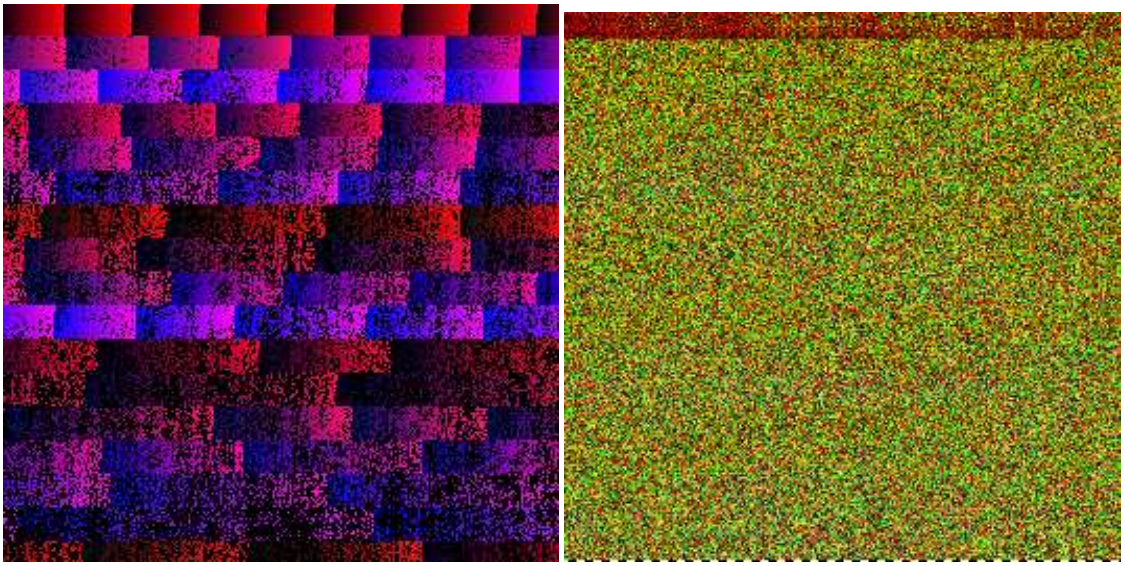


FIGURE 2.2: A *suffix tree* (left) and a reference string (right) encoded as textures by **CMatch** [Schatz and Trapnell 2008], a GPU-based substring matching program used in biological sciences. The developers were able to harness the GPU's speed by using texture memory instead of a standard data structure: the data is encoded into a matrix of colours instead of an array of bytes. In the tree, a pattern is visible, as the nodes are arranged in 32×32 blocks. These blocks are optimised so that those most often accessed by multiple threads lie near the initial position in the top-left corner, and the blocks accessed by individual threads appear further away. The reference string, on the other hand, appears as noise.

2.3.2 Latency Hiding

For most CPU-bound applications, there is no use in running more computational threads than there are available CPU cores. In multi-threaded programs, such as a server than can handle multiple requests, or a program that can analyse data in parallel, a *thread pool* containing a number of threads can be maintained, with only a number of threads less than or equal to the number of available CPU cores actually running. [3]

Threads are useful when hiding large latencies, but are less useful for smaller latencies. On the CPU, switching between threads is a *coarse-grained operation*: it works best when the individual parallel tasks are large in size, and do not have to be switched often. A common use of threading on CPUs is to hide the latency of an IO-bound operation, such as reading from a file or a network socket: other instructions could be executed while the thread waits to finish, or a user interface thread can still be active, responding to user input, while the other thread sleeps. For the CPU, switching between threads involves invoking the OS's scheduler to select a new thread to run, change the execution environment to accommodate this new thread, and restore the values in the new thread's registers from memory. This makes context switching an expensive operation. This works well enough for situations involving multi-core CPUs that assign *long* jobs to each core, minimising the number of context switches.

Fine-grained parallelism, as a counterpart to coarse-grained parallelism, is much more important in the world of GPUs—it enables them to switch tasks much more often, in the common case of one thread needing to wait for the result of a memory access.

The amount of latency caused by stalls that a GPU can tolerate through multithreading depends on the ratio of hardware-managed thread execution contexts to threads that can be executed simultaneously. This ratio is referred to as T, and values for various processors are listed in Table 2.2. A GPU's support for more thread contexts allows it to hide longer, or more frequent, stalls, while CPUs typically maintain only one thread per core.

The core has to execute an instruction from the currently-running thread while maintaining the state on all threads simultaneously. These

[3] A recent development is what Intel calls *hyper-threading*, which presents each core to the operating system as *two* cores, running both threads on the core whenever it can. For example, when one core uses the integer registers, and the other uses the floating-point registers, both instructions can be run simultaneously during this time.

threads called SIMD vector instructions, the exact details of which vary between GPUs. The core also contains a pool of general-purpose registers partitioned among the thread contexts. [4]

At runtime, the GPU copies the program onto each of its thread contexts. The core starts executing the first thread, and keeps running until it detects a stall for memory access—reading or writing to the texture memory on the GPU. While it waits for the result of the memory access, it can switch to another thread, and execute *that* until the result becomes available, at which point it can continue the execution of the original thread.

When executing the program, a minimum of four threads is needed to keep the core arithmetic logic units busy. In practice, however, memory access on GPUs involves latencies of up to hundreds of cycles, so modern GPUs must support more and more threads to remain efficient, as having to run programs at their maximum possible speed requires the core logic units to be constantly in use. [Fatahalian and Houston 2008]

The number of threads that can run on a multiprocessor depends on the number of registers available: each kernel takes up its own set of registers, and with a finite number of registers on the processor, there is a limit to how many kernels can be run at a time. Furthermore, each GPU has a maximum number of potential threads amongst all the

[4] There is little to no standardisation between GPUs, other than their general overall architecture, so each one has to be treated differently. For consumer-class graphics cards, the only standard is the use of a PCI-e bus to transfer data. This lack of backwards compatibility is done on purpose to encourage innovation in GPUs, so they do not need to get held back by supporting older architectures.

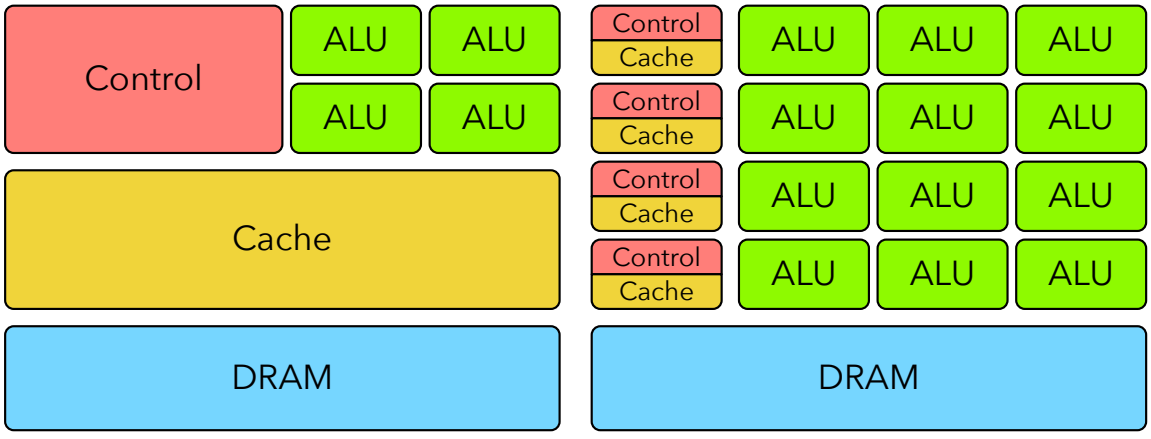


FIGURE 2.3: CPU (left) and GPU (right) architectures. Although not an exact schematic, it shows how much more the GPU favours pure computation: the number of Arithmetic Logic Units (ALUs) eclipse the more generic caching and flow control centres. [Pharr and Fernando 2005]

Type	Processor	Cores per Chip	ALUs per Core	SIMD Width	Max T
GPUs	AMD Radeon HD 4870	10	80	64	25
	NVidia GeForce GTX 280	30	8	32	128
CPUs	Intel Core 2 Quad	4	8	4	1
	STI Cell BE	8	4	4	1
	Sun UltraSPARC T2	8	1	1	4

TABLE 2.2: The maximum numbers of simultaneously-executing threads for a select few processors, both CPUs and GPUs, given their specifications.

kernels; a new kernel will fail to launch, with an error thrown instead, if it tries to break this limit. These potential threads are organised into *blocks* and *warps*. The exact numbers of blocks and warps vary between GPUs, but they obey a common formula. If T is the number of threads per block, and W_{size} is the size of each warp, then the total number of warps in a block, W_{block}, is as follows:

$$W_{block} = \left\lceil \frac{T}{W_{size}} \right\rceil$$

More detail on blocks and warps is given in Section 3.2.

2.3.3 Control Flow and Recursion

Because of its emphasis on pure computation, the GPU has limited support for control flow. When a function running on the GPU branches, such as by taking a side on an `if` statement or running for a variable number of times in a `while` loop, the GPU is forced to have every core either execute the same instruction, or wait until its branch becomes the currently-running branch. This is in contrast to a multi-core CPU, which is able to run both sides of a branching statement on multiple threads simultaneously, without having to wait for one to finish before the next can begin.

One side-effect of the GPU's lack of transistors dedicated to control flow is that they no longer support explicit recursion—a C compiler for the GPU will reject a program that contains a function that tries to call itself, or any combination of mutually-recursive functions, even if the program is guaranteed to eventually terminate.

The reason for this is twofold. Firstly, every function on the GPU is inlined, where the function call is replaced by that function's complete definition with the argument values inserted. A recursive function's definition contains itself, so completely inlining a recursive function would result in an infinitely-long stream of instructions. Secondly, the GPU has no *call stack*: there is nothing telling the GPU which instruction to jump back to when a function returns a value. Ordinary function calls are able to work around this restriction by using function inlining as described above, but this is not possible for recursive function calls.

There are two options for dealing with this restriction. The first is to avoid recursing at *arbitrary* levels, and instead limiting a program to a loop with a maximum recursion depth. [5] This makes long-running loops possible, but will result in decreased performance when there is a large difference between the numbers of threads reaching the minimum and maximum recursion depths: the threads that execute only once will be forced to wait for the threads that execute a maximum number of times to complete before they will be allowed to finish executing.

The second option is to have a recursive function return not a final value, but a half-processed value that can later be processed further, leaving it up to the scheduler to decide when to finish processing.

This second method greatly reduces the performance problems caused by having some threads venture much deeper into the recursion stack than others: with the maximum recursion depth set to a much lower value, the time spent waiting for completed threads to return their values is lessened. The disadvantage is that without a system managing the processing and complexity for the developer, the code becomes much complex and prone to error, as the program would not only have to separate out complete values from the semi-computed ones, but also make the scheduler run all the kernels for each partial execution, instead of just one.

[5] One way of getting around this that gets used in practice is to use the C++ templating system to define several numbered functions, with each one calling the next one, and the final one terminating. This will trick the compiler into inserting a recursive definition because the programmer must specify a maximum recursion depth, meaning it can be inlined into a very repetitive, yet finite, list of instructions.

However, this method can lead to performance problems on GPUs, as every iteration, up to the maximum level of iteration, will get executed every time.

2.4 GPU Case Studies

There have been a number of reports of significant improvements in execution time for programs that can exploit the GPU's parallel nature;

several of them are listed below.

2.4.1 String Matching

There are several examples of common algorithms that had reached their performance limits on the current generation of hardware, only to leap forward in performance after being adapted to running on a GPU. One such example is string matching, which is an important requirement in fields such as computational biology for matching similar proteins and gene sequences against a large database of known sequences.

Schatz and Trapnell, in *Fast Exact String Matching on the GPU*, present a GPU-bound string-matching program called *Cmatch* that achieves a speedup of as much as 35× over an equivalent CPU-bound version. Cmatch encodes the string into a suffix tree, which is stored in texture memory [Schatz and Trapnell 2008] (see Figure 2.2).

By being able to match at more than one position in the string at once, and distributing these positions over the available processing units, an improvement of many times over a CPU-bound algorithm can be reached.

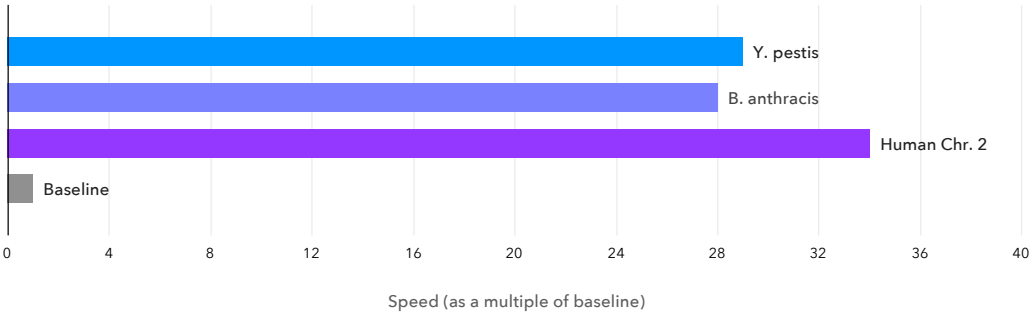


FIGURE 2.4: Comparison between results of matching various gene sequence strings on a GPU, as well as the CPU result as a baseline. These results do *not* include the time taken to transfer the initial large sequences to the GPU before matching can take place.

The average time spent matching the queries on the GPU was only 3% of the total running time, compared to nearly 50% of the total time when running on the CPU. The bottlenecks then became reading the queries from disk and constructing the suffix tree to process.

Dataset	Speedup
Y. pestis	29×
B. anthracis	28×
Human Chr. 2	34×

One element that is *not* included in their workload is the time taken to transfer the initial data set onto the GPU before any searching can take place. This is because this data is expected to be searched through multiple times with many different source strings, so the initial transfer time is amortised over the multiple subsequent runs. Thus, the use of a GPU is most effective when running searches over the same data, rather than with different strings each time.

2.4.2 SQL Database Operations

In *Accelerating SQL Database Operations on a GPU with CUDA*, Bakkum and Skadron used CUDA to increase the performance of several SQL queries. [Bakkum and Skadron 2010]

A mean speedup of 50× is achieved, although this is reduced to 36× when including the time taken to transfer the results from the GPU to the CPU.

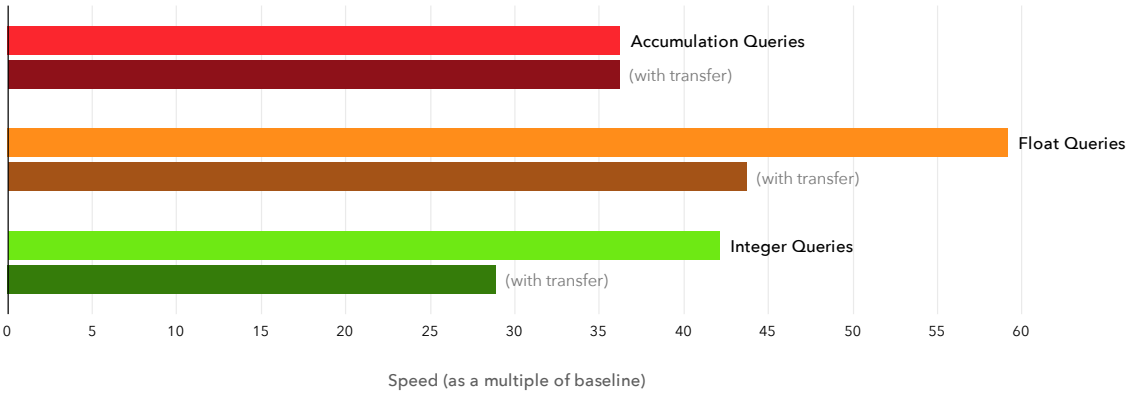


FIGURE 2.5: Comparison between results of running different types of SQL operations on a GPU, as a multiple of the time taken on the CPU as a baseline. The first of every pair show the speed increase against the CPU, while the second shows the increase reduced by including the time taken to transfer the results from the GPU to the CPU. None of the results includes the time taken transferring results *to* the GPU *from* the CPU.

There is a notable difference between the time taken to process the accumulation queries and the numeric (integer and floating point) queries: the accumulation queries return a single number, such as the sum or product of the rows of data, whereas the numeric queries have to return much more data representing the multiple rows of data that are the result of the query.

Dataset	Speedup	Speedup + Transfer
Integer	42.11×	28.89×
Float	59.16×	43.68×
Accumulation	36.22×	36.19×

It also shows that even for a transfer of a single number, there is latency in the communication that is always present despite the small size of the data. [6]

Finally, they put forth the idea of breaking up the data set and running a query concurrently on multiple GPUs. They state that although there would be overhead from co-ordination, it is likely that SQL queries could be further accelerated.

2.4.3 Hash Reversal

A hash, or message digest, is a property of a string that changes drastically even when the source string changes minimally. A hash should be simple to compute but impossible to reverse-engineer without resorting to brute-force methods such as checking every possible input string.

More recently, however, reverse-engineering a hash has become possible due to the significant speedup offered by GPUs. As the number of cores available has multiplied, the time taken can shrink by several orders of magnitude. Results in this area are of particular interest to network security researchers, as password hashes were long thought difficult, if not impossible, to reliably reverse before the advent of massively-parallel hardware such as GPUs.

[6] Even the time taken to transfer a minuscule amount of data is itself not minuscule: the latency of moving *any* data from the CPU to the GPU will take time. For more information, see Section 3.1.3, Latency.

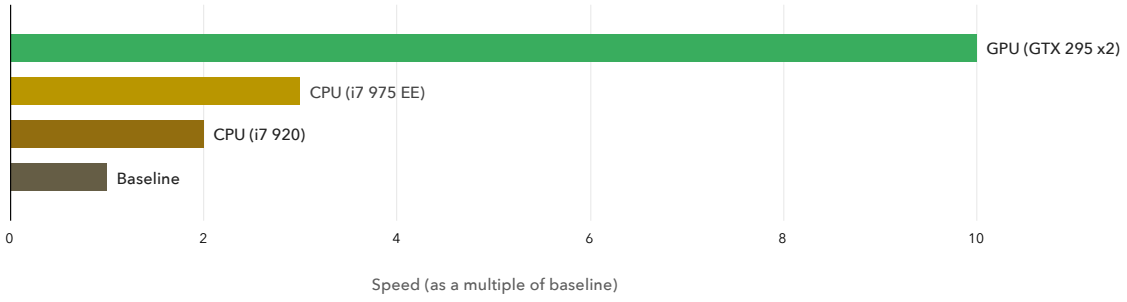


FIGURE 2.6: Comparison between results of reversing hashes of strings on a GPU, as a multiple of the time taken on several CPUs as baselines.

In *GPU-based Password Cracking*, Bakker and van der Jagt put forth the use of GPUs for breaking passwords transformed by the MD5 hashing algorithm. They record the GPU-based systems testing 14× as many passwords per second than the CPU-based systems, stating that

Implementation	Speedup
i7 920	2×
i7 975 EE	3×
GTX 295 x2	10×

with this increase in performance, more complex passwords are becoming feasible to crack.

2.4.4 Conclusions

In order to get the best possible speed increase from the GPU architecture, it is necessary to design your program in such a way that data transferred between the host and the accelerator is minimised. The time taken to transfer data can comprise a significant share of the execution time if data needs to constantly move back and forth.

One way to achieve this is to port the *less* computationally-expensive functions to the GPU as well, and use them to bridge two routines that already exist on the GPU, keeping them on the same device and saving the transfer time.

Both transfer *to* and *from* the GPU must be considered: running a reduction operation—one that reduces many values to one single value, such as a sum or minimum operation—will be more efficient than a transformation operation that has to transfer more values. Similarly, being able to generate data from scratch on the GPU rather than transferring it from the CPU will also be beneficial.

3 A HIGH-LEVEL LOOK AT PARALLELISM

Both concurrent and parallel programming are difficult to get right. At the low-level, a task such as running a loop in parallel on a multi-core CPU involves spawning enough threads to perform the computation, distributing the units of work amongst the spawned threads, and waiting until every thread is complete before continuing with the program. Programming for this architecture is difficult because there are many pieces to get right, and programming for multiple devices is harder still. However, when looking at concurrency and parallelism from a higher level, it is possible to let the computer abstract away some of the details and look at the program in a simplified light.

This section discusses the limitations of parallel programming both those intrinsic to the model and those of the current state of the art, and lists ways in which these low-level details can be automatically managed for the programmer.

3.1 Limitations of parallel programming

In a perfect world, it would be possible to run the program on n different processors and get an n-times speedup; sadly, this is not the case. There are a number of laws that govern the maximum potential speed increase that a program is capable of obtaining, due to serial parts of the program or latency between different processors or machines.

3.1.1 Amdahl's Law

Amdahl's Law states that each program contains an inherent serial component: a part of the program that cannot be parallelised. This part will run at the same speed irrespective of the number available cores,

which, with a high number of cores, will contribute the slowness of the program. [Amdahl 1967]

This has many practical considerations. Even when running on an infinite number of cores, a program would still be limited by having to initially load the data into RAM, a bottleneck that would be difficult to overcome.

Mathematically, with p as the amount of time spent on the parts of the program that can be run in parallel, $(1 - p)$ on the parts that must remain serial, and N as the number of processors, then Amdahl's law gives the speedup S as as: [1]

$$S = \frac{1}{(1 - p) + \frac{p}{N}} \tag{3.1}$$

A large part of parallel programming involves reducing the parts that must remain serial—in this case, $(1 - p)$ in Equation 3.1—to as small as possible.

3.1.2 Gustafson's Law

Another fact that is often overlooked is that a parallel program must often solve larger problems than a purely-sequential one, by virtue of there being potentially-unnecessary work distributed to some processors, making the total running time longer still. This is known as the *Gustafson-Barsis law*, or often just *Gustafson's law*. A program executing on more than one thread will still have performed the computation on the others if one thread returns a sought-after result, effectively making the computations superfluous. In the same way, a program that cannot divide evenly between threads will have to "round up" the time taken, in order to account for the threads that did not execute in the final iteration. [Gustafson 1988; Lewis and El-Rewini 1992]

[1] *Speedup* is defined as the time taken for a program to execute with one processor—in serial—divided by the time taken to execute in parallel.

3.1.3 Latency

The latency of transferring the data between the individual systems is something that can be managed, but not completely avoided. [2] Latency

[2] "Each component of a computer system contributes delay to the system. If you make a single component of the system infinitely fast... system throughput will still exhibit the combined delays of the other components" — Gene Amdahl, joint creator of the IBM System 30 Architecture

can be defined as the time taken to send an empty message; or, for systems with parallel pipelines, the time taken to load the pipelines before the first result is delivered. Thus, it can become a problem when the messages themselves are small enough that the message-sending time becomes a significant part of the overall execution time.

Running time can be modelled by an equation, with T as the overall time, α as the latency time, β as the bandwidth of the system, and N as the length of the message: [Mattson, B. Sanders and Massingill 2004]

$$T = \alpha + \frac{N}{\beta} \tag{3.2}$$

The latency and bandwidth can vary between systems depending on the type of hardware, as well as the quality of the software used to implement the protocols—the time taken for data to reach a graphics card over a PCI bus is tiny compared to transferring data over a network. Both of these can be measured with simple benchmarks. [J. Dongarra and Dunigan 1997]

The most common technique for compensating for high latency or a slow network is to send few large packets instead of many small ones. For parallel programming, this means that it is more effective for each node in a cluster to be given large matches of work to do over small batches.

3.2 CUDA

In November 2006, NVidia unveiled the 8800 GTX, a graphics card built with the *CUDA Architecture*. It included several components designed for GPU computing, which aimed to alleviate the difficulty of GPGPU development: instead of using programmable vertex and pixel shaders, the CUDA architecture included a unified shader pipeline, allowing every ALU on the chip to be used for computation rather than purely for graphics, and added a block of shared memory, allowing programs that require inter-thread communication to run on the GPU as well. [J. Sanders and Kandrot 2010]

CUDA programs are written in CUDA C: a version of C extended with syntax to allow functions to be called on the device. The compiler,

`nvcc`, compiles the host's functions using whichever C compiler is present on the system, and the device's functions itself. It is similar to DirectX or OpenGL in that the programmer does not have to know the model of hardware that the program would be run on: each GPU uses different instructions, and CUDA is able to abstract away these implementation details. [NVidia 2010]

CUDA works with fine-grained data-parallel threads as its units of execution. Launching a CUDA kernel function creates a *grid* of threads that all execute the function. A grid is organised into a 2D array of *blocks*, which is organised into a 3D array of *threads*, giving the programmer five dimensions to work with. The individual threads identify the relevant portions of the data to process using the `blockIdx` and `threadIdx` variables. These appear as pre-initialised, read-only variables to the developer, and are assigned to automatically by CUDA's runtime system. [Kirk and Hwu 2010] The programmer can then write their function roughly as they would ordinary C code.[3]

CUDA allows threads in the same block to co-ordinate by allowing the programmer to specify when they synchronise. This ensures that all threads in a block have completed one phase of execution before moving on to another, by waiting until every thread has reached the same location in the program. [4]

3.2.1 Portability of CUDA

NVidia developed CUDA to allow developers to use their GPUs for more general applications than graphics, but did not provide any way to efficiently run CUDA code elsewhere: the architecture is tied to NVidia's own hardware, and does not even attempt to work anywhere else. [Stratton, Stone and Hwu 2008]

The easiest possible translation is to emulate the GPU's instruction set and model by spawning a CPU thread for every GPU thread that would run, and mapping those threads to the available cores on the GPU. However, this is ineffective, as GPU-based programs often rely on latency hiding, which the CPU is more inefficient at. [5] The result is a program that runs, but much slower than had it been written for the CPU to begin with. NVidia's own CUDA emulation toolkit uses this model, and it suffers from this same inefficiency.

[3] A running thread's co-ordinates can be specified by `threadIdx.x`, `blockIdx.y`, and so on.

[4] This is written `__syncthreads()`.

[5] For more information on latency hiding, see Section 2.3.2, Latency Hiding.

Nevertheless, there have been attempts to translate CUDA programs back onto the CPU. MCUDA, as described in *An Efficient Implementation of CUDA Kernels on Multicores* [Stratton, Stone and Hwu 2008], maps the CUDA programming model onto a multi-core CPU architecture by transforming the source code from CUDA C into standard C, using a library that gets linked at compile time to provide the parallel programming features that get lost in translation.

It appears as though running CUDA programs on the CPU is simple, if not efficient, owing to the fine-grained model of parallelisation used: if all threads within a block occupy the same CPU core, then there is no need to synchronise during the blocks' execution. The threads would have similar control flow paths to each other, and would likely have hazards occur very close together. However, scheduling these threads to run together is more difficult. If a thread is allowed to run on *any* CPU core, then the benefits of running the same program over multiple pieces of data is lost under the large amount of scheduling overhead.

For this reason, the researchers behind MCUDA chose to translate into a programming model that maintains the locality between threads while still using the operating system's scheduler. Their compiler searches for any loops that are run in the kernel, and uses a transformation they call *deep fission* to add synchronisation statements by creating new thread-local loops within the scope containing those statements, treating the scope itself as one more statement that needs to be synchronised.

CUDA also has an official extension for *dynamic parallelism* that enables kernels on the GPU to call themselves. This extension is only able to be used on devices that support a recent version of CUDA.

However, there is a low hardware limit for the maximum recursion depth: 24. This is low enough that certain applications may wish to implement their own strategies for handling recursion. [NVidia 2014]

3.2.2 PTX

The CUDA compiler, `nvcc`, does not compile CUDA C source code into the GPU's native machine code, simply because there *is* no such native machine code: in order not to be hindered by necessitating backwards

```
        .reg.u32 r1, r2,     r3;       // define three registers
start:  mov.u32  r1, %tid.x;           // load the thread ID
        add.u32  r2, r1,     1;        // add one to this value
        mul.u32  r3, r2,     4;        // then multiply it by four
```

FIGURE 3.1: An example of PTX assembly code, that takes an element of an array that starts from 1, and multiplies it by four. The kernel begins with a definition of all used registers, along with their type. In this case, the three registers used are of the `.u32` (32-bit unsigned integer) type.
Inside the kernel, which is designated here by the label `start`, the thread ID is loaded into a register. This number gets converted to the number we wish to operate on by transforming it—in this case, adding one with the `add` instruction. Finally, the value gets multiplied by four with the `mul` instruction.

compatibility between architectures, each graphics card has a similar but not identical instruction set, so compilation would be a moving target. Instead, the compiler produces *PTX assembly code*, with PTX standing for Parallel Thread Execution.

PTX is a register-based, typed assembly language that is used as an intermediary language by the CUDA system. The runtime is able to translate compiled PTX assembly instructions into the *native* instruction set of the GPU much faster than translating the equivalent CUDA code, as much of the work, including parsing, compile-time error checking, and optimisation will have already been performed.

Like CUDA, PTX uses an *explicitly-parallel* model, with a PTX program specifying how a single particular thread should execute, with the system extrapolating this to multiple threads.

A co-operative thread array (CTA) is an array of threads that can execute a kernel concurrently or in parallel, and threads within a CTA can communicate with each other. [6] These threads run in groups called *warps*, all executing the same instructions at the same time. [NVidia 2012] [7] A CUDA *grid* is just a grid of CTAs, enabling them to be run in parallel.

[6] A thread's position in the running CTA can be specified by the `ntid.x`, `ntid.y`, and `ntid.z` PTX variables.

[7] Typically, a warp has 32 threads. This is set as a good default value, rather than a constant depending on the hardware, as it is plausible to maximise performance by changing the number. However, this possibility was not explored, as different warp sizes may have different results on different devices, and there is currently no way to predict the outcome of different warp sizes other than trying many values and seeing which one comes out the fastest, which would likely be enough to offset any potential speed gain.

3.3 Social Aspects of Parallel Programming

In addition to the technical considerations, parallel programming has several "social" aspects that come into play.

As a result of this hardware revolution, the multi-core and multi-device era has put pressure on programmers to re-write their software

for this new hardware. By proposing simpler processing units, architects assume that programmers will now manually take care of mechanisms that used to be performed by the hardware, such as cache checking. Therefore, there is a gap between the traditional model of programming and the style of programming useful for multi-core, accelerator-enabled architectures.

- **Error checking:** When programming with multiple threads on a CPU, any point where the threads interact must be preceded with a call to synchronise the threads, making sure that the correct data has been loaded into each. Synchronising a group of threads potentially halts the entire thread pool until every thread has caught up to that point, using up processing time that could be better spent. However, without this call, the program may be running with incorrect data loaded: a situation that could be difficult to debug.
- **Mathematical optimisations:** Certain optimisations can have a mathematical component, which may not be immediately obvious. For example, the parallel version of reduction over an array, such as finding the sum or product of all elements, can be run efficiently by continually performing the operation on each successive pair of elements, halving the size of the array each iteration until only one element remains. However, for this to be mathematically sound, the reduction operator must be associative, and it is considered outside the scope of a compiler to determine the associativity of a function. This is an example of an optimisation that can only be applied when the compiler has further information about the program.

3.4 Parallelism in Programming Languages

There is a proverb in software development stating that the number of bugs per line of code in any given program is constant: a function written in a verbose language is likely to contain more errors than one written in a terse language. This seems nonsensical at first glance, but there is reason behind it: humans are not perfect programmers and the more lines of code a developer has to write, the greater the chances for an error to creep in.

As the need to process data in parallel becomes a routine operation, programming languages themselves are starting to offer developers ways to parallelise their code without having to write the code to spawn a thread manually. There are several different ways that a language can do this, from working with the language's own features to ensure that only safe code is made parallel, to using extensions built on top of a language to let the programmer decide which functions to run concurrently.

3.4.1 Parallel Functions

One simple way to make parts of a program parallel is to use functions that dispatch threads themselves and do not complete until every thread has finished. These can be offered either by the language itself or as part of a library.

The computational algebra system Mathematica offers functional programming abilities, and has recently added data-parallel variants of these, making parallelism as simple as just using the right functions in places where parallelism is desired.

For example, a trivially-parallelisable problem such as summing together two vectors requires a very small change:

```
Map[Plus, vector_a, vector_b]          (* Serial version *)
ParallelMap[Plus, vector_a, vector_b]  (* Parallel version *)
```

This sort of implementation is best suited for a mathematical system, as concurrency is rarely a problem, and few functions have side-effects. The main disadvantage of this approach is that any compiler or interpreter needs to be aware of these functions—when running the code on an older version of the language, it would complain about the `ParallelMap` function not existing. This limits its use, but remains a very simple solution.

3.4.2 Parallel Directives

In a more imperative language, such as C or C++, different versions of the standard control structures may be used to parallelise code.

With OpenMP, an API for writing portable concurrent code, a standard `for` loop is able to be "tagged" with parallelism. Continuing with the vector addition example:

```
int a[N], b[N], c[N];

#pragma omp parallel for
for (int i = 0; i > N; i++)
{
    c[i] = a[i] + b[i];
}
```

OpenMP is able to sidestep the problem of backwards compatibility by using the C preprocessor's `pragma` statement, which, according to the specification, is not guaranteed to perform any action. This means that a compiler that supports OpenMP is able to see the parallel `for` statement and vectorise the loop, while a compiler that does not support the feature is free to ignore the statement as it would a comment, and proceed to compile it as if it were serial code.

Fortran is another language that has gained a modern version with parallel support. Here is a similar vector addition example that uses Fortran's OpenMP features:

```
INTEGER, DIMENSION(10) :: A, B, C

!$OMP DO do i = 1, 1000
    c(i) = a(i) + b(i)
!$OMP END DO
```

In these instances, it remains up to the programmer to decide whether a function should be run in parallel, and to time the results to make sure there is actually a performance benefit.

For programming in CUDA, NVidia provide an extension of the C language, CUDA C, that has its own set of keywords used to specify the architecture that a function should be run on. This lets the programmer specify whether a function should target the CPU, GPU, or both:

- Host functions are run on the CPU, and can optionally be specified with `__host__`.
- Device functions are run on the GPU and can only call other GPU functions. These are specified with `__device__`.
- Global functions run on the GPU but can be called by the CPU, and can only call device functions. These are specified with `__global__`.

3.4.3 Parallel Data Structures

Another way to add parallelism to a program is to designate certain data structures as being able to have parallel algorithms run on them, and letting the compiler use these algorithms on these data structures only.

The Haskell language is able to use its type system to automatically parallelise operations performed on certain lists with the Data Parallel Haskell extension. The vector addition example in Haskell looks like this:

```
addSerial :: [Double] -> [Double] -> [Double]
addSerial xs ys = [ x + y | x <- xs | y <- ys ]

addParallel :: [:Double:] -> [:Double:] -> [:Double:]
addParallel xs ys = [: x + y | x <- xs | y <- ys :]
```

Any operation performed on the list, such as `map` or `reduce`, is executed in parallel. This allows the programmer to use the same operations on both ordinary and parallelisable lists. [Jones et al. 2008]

The language Perl 6 has several data structures called *junctions*, which represent a range of values that can be tested in any order, including in parallel. To illustrate, here is a simple program that tests a number against a range of values:

```
if $value == any(2, 3, 5, 7, 11, 13, 17, 23, 27) {
    say "$value matches!";
}
```

The `any` function creates a junction value representing the first few prime numbers. When it comes to testing the value, the interpreter can test the values in any order. It can also do this in parallel, using a different thread for each value.

3.4.4 Separation of Data

The main drawback with the above approaches is that it is up to the programmer to determine whether the parallelism is sound.

When dealing with *concurrent* programs that manage several disjoint tasks that may be running simultaneously, it is common practice to ensure that any access to non-thread-local variables is placed behind a *lock*, in order to prevent data races—situations where a variable is accessed by multiple threads at the same time, causing undefined behaviour. If a parallel mapping function was passed a transformation function that had a side-effect, such as modifying a global variable, the result of the computation would thus be undefined, as the threads are not guaranteed to be run in any order, and race conditions may mean that certain mutations are ignored or overwritten!

One solution is to simply forbid individual functions to access global variables—either for reading or for writing—limiting them to their local variables and the values of their passed-in parameters. The CUDA dynamic parallelism extension has this restriction: pointers to local memory are not allowed to be used in recursive calls to the same kernel, and only constant global memory can be accessed by a child grid.

A more lenient solution is to restrict the values that a function can access to only those that have been proven safe. The programming language Rust uses mutability annotations on variables to ensure that each variable can only be accessed mutably by one thread at a time. Here is a naïve example of concurrent access to a shared value, where three threads attempt to mutate it at once:

```
let mut shared_value = 0;

for i in 0..3 {
```

```
    thread::spawn(|| {
        shared_value += i;
    });
}
```

The above code is *not* accepted by the Rust compiler, as it has a data race: the modifications to `shared_value` may conflict with each other as each thread tries to write to the variable at the same time.

Instead, Rust forces the developer to wrap the value in a *mutex* (mutual-exclusion lock), ensuring that only one thread at a time has mutable access to its contents. The `Mutex` type provides a `lock()` method that produces a mutable value for the first thread that accesses it, and will pause for any further threads until the first is done with the value. This means every thread can access the variable safely. A correct version of the program is:

```
let shared_value = Arc::new(Mutex::new(0));

for i in 0 .. 3 {
    let shared_value = shared_value.clone();
    thread::spawn(|| {
        let mut value = shared_value.lock().unwrap();
        value += i;
    });
}
```

The downside of this approach is that it requires a much more sophisticated compiler: one that can track mutability annotations and detect automatically when a lock on a mutable value ends. Rust uses another of its features, lifetime tracking, to detect when a mutual-exclusion lock falls out of scope, and so adding automatic thread safety is not as difficult as adding it onto a language without any related features.

3.4.5 Separation of Tasks

The methods listed above are examples of *opt-in parallelism*, where a program is run serially by default, with parallel threads only spawned

when the programmer opts in to using them. Another approach is to offer *implicit parallelism*, where the programmer merely has to describe the connections between each function or kernel, and the parallelisation opportunities are sought out automatically.

The Occam programming language, originally developed for the Transputer computer architecture, has several features making it suitable for implicit parallelism. It is based on concurrent units of computation it calls *processes*.[8]

An Occam process can be sequential or parallel, and each process is connected to others with channels. Some examples of Occam functions are given in Figure 3.2.

Occam is able to avoid the problems commonly caused by *unsound* concurrently-run functions—such as a transformation function that reads from or writes to a shared state—by disallowing shared variables and memory pointers in its values. This reflects the message-passing Transputer architecture, which has no global memory bank.

[8] An Occam "process" is different from an operating system "process". More than one Occam process need not necessarily be concurrent.

3.4.6 Analysis

In these cases, the programming language is able to offer parallelism by deliberately not requiring computations to be executed in any order—in contrast to traditional imperative programming languages, which assume that the operations *must* be executed in the order in which they are written. When the programmer is able to specify that each iteration of a loop has no effect on the other iterations, or that a series of predicates can be checked in any order, the programming language is then able to run each iteration in parallel.

```
seq
  input ? x
  y := x * 2
  output ! y
```

```
par
  y := x + 7
  z := x * 4
```

```
alt
  input1 ? x
    output ! x
  input2 ? x
    output ! x
```

FIGURE 3.2: The three types of process available to the programmer in Occam: sequential (`seq`, left), parallel (`par`, middle), and alternate (`alt`, right). Sequential processes execute their instructions in the given order, which here is used to read a value from an input channel, multiply it by two, and send it down an output channel—all of which need to be in the correct order. A parallel process is able to execute its instructions in parallel. Finally, an alternate process tests all of its given conditions—which here is checking two channels to see if either has input—and executes only the case where the condition holds.

This opens up opportunities for further optimisation: for example, when checking a series of predicates, the language could check those with cached values first, then move onto the slower checking of the others afterwards. Without the programmer explicitly declaring each check as independent, the language would have to assume that they are in fact dependent, as doing otherwise could be a source of bugs in the program, leaving it with a slower overall running time.

3.5 The Role of the Scheduler

In order to run more than one task on a serial machine at a time, a *scheduler* is needed. This is a small program that governs the launching, pausing, and termination of threads, as well as which threads should get priority in situations where there are more threads than available processors. [Pacheco 1996] Most commonly, this is a function of the host computer's operating system, though there are times when the situation can be more general; for example, with a network of computers controlled by a load balancing machine, the balancer would act as the scheduler.

A scheduler is typically both small and fast to run, in order to minimise the overhead of context switching. However, it does not necessarily have to be straightforward: most operating system schedulers support a feature called *niceness*, which allows the user to specify which processors are more important, and which are less so. The scheduler can then use this information to give less processor time to the *nicer* processes.

Another job of the scheduler is to manage idle processes. Many operations involve latency, especially those involved in communication with memory or another device. The modern scheduler is able to detect when a process is waiting for a response, and puts it into the idle state, running another process in the foreground instead. This way, it can make the most of the computer's available resources.

3.6 CPU+GPU Schedulers

There have been several efforts to introduce scheduling between the CPU and the GPU. Although CUDA is often cited, it divides the program into explicit CPU and GPU sections, rather than partitioning it by analysing the program itself.

The runtime systems specified here, instead of leaving it to the programmer, attempt to discover an individual kernel's most suitable processor.

3.6.1 Qilin

Qilin is a programming system developed for heterogenous multiprocessors. It provides an API similar to OpenMP [9] for writing parallelisable operations, but additionally can target the GPU, using CUDA as a back-end. [Luk, Hong and Kim 2009]

Kernels in Qilin operate on arrays that are automatically dispatched between the different available processing units. This is achieved by training a model for each kernel to determine the amount of time necessary to process the kernel on each type of processor, depending on the input size. This model can be used to dispatch data evenly, ensuring that all processing units can finish at the same time, after the model has been trained with trial runs.

Qilin uses an adaptive mapping technique to find a near-optimal surface of computations mapped to processors. The first time that a program is run, it is used as the training run. It divides the input into two subsets, mapping one part to the CPU and the other to the GPU. It then uses curve fitting to construct two linear equations based on the running times of divided-up segments of each of the inputs. It can then calculate the efficacy of the GPU on a program. Although this is not part of this research's proposed techniques, Qilin provides a nice example of a method which determines the processor to use, which is one of its goals.

Unlike this research, it presents itself as an API for C or C++, rather than a language unto itself. It also makes use of just the running time of a program, rather than a combination of factors such as the amount

[9] OpenMP is demonstrated in Section 3.4.

of data to transfer or the utilisation of other parts of the system, to gauge the target processor.

3.6.2 Charm++

Charm++ is a parallel C++ library that provides load balancing and communication optimisation mechanisms. It is able to target both GPUs [Wesolowski 2008] and Cell processors [David Kunzman et al. 2006]. Charm++ provides a low-level API to offload computation with a task paradigm, rather than automatically parallelising computations with a data paradigm. In the case where the different processing units do not share the same endianness, Charm++ can automatically convert data during the transfer so that they can be used throughout the system. [Kale, DavidM Kunzman and Wesolowski 2010]

Programs written in Charm++ are decomposed into a number of co-operating message-driven objects called *chares*, in contrast to the more traditional *thread*. However, the authors claim that the large grain size requirement for proper utilisation makes it difficult to map kernels onto GPUs directly.

3.6.3 KAAPI

Harmony is the runtime system used in the **Ocelot** dynamic execution infrastructure. [Diamos and Yalamanchili 2008] Ocelot uses NVidia's PTX as its intermediary language, making it able to compile to CUDA-supported GPUs, x86 CPUs using a PTX emulation layer, and on various OpenCL devices using a bytecode translator based on the LLVM compiler. PTX code is obtained by compiling CUDA C with NVidia's own compiler, allowing native CUDA programs to be executed on hybrid platforms, not necessarily with a CUDA-enabled device.

3.6.4 StarSs

The **StarSs** project is a set of language extensions and a collection of runtime systems targeting different types of platforms. It extends C or Fortran with `#pragma` annotations to offload certain pieces of computation onto architectures targeted by the runtime system, including

GPUSs for GPUs, [Ayguadé et al. 2009] and **SMPSs** for multi-core CPUs. [Barcelona Supercomputing Center 2007]

It is a follow-up of the **GridSs** project which provides support for computational grids. [Badia et al. 2003] It implements advanced data management techniques, such as the ability to directly transfer cached data between processors.

However, it remains the duty of the programmer to decide which tasks should be offloaded, instead of having the runtime system pick the most appropriate target automatically.

3.6.5 StarPU

StarPU is a runtime system targeting the CPU and accelerators, including CUDA for NVidia GPUs. Unlike other such systems, StarPU takes into account the size of the data and the transfer speed when deciding which processor to run a kernel on. Additionally, it picks between several different scheduling strategies, and allows the user to add their own strategies if the default set should prove insufficient.

It presents itself not as a new programming language, but as a library that can be included and linked from C code. Additionally, although it does not specifically mention dataflow, it does give each tasks an abstract queue with task submission (push) and request (pop) operations.

3.6.6 Anthill

Anthill is a runtime system designed for clusters of machines equipped with a single GPU but multiple CPUs. [Teodoro et al. 2009] The authors implemented two greedy scheduling policies, one with and one without support for priorities. However, Anthill is similar to Qilin in that it only considers the *relative* speedup of a kernel to select the most appropriate processors, instead of a number of factors including the amount of data and the transfer time.

3.6.7 TERAFLUX

TERAFLUX presents itself as a "manageable architecture design", using dataflow threads (which it calls DF-threads), modifiably by a minimalistic extension of the x86-64 instruction set. These instructions enable asynchronous execution of threads that execute not under the main control flow of the program but under its dataflow, which is scheduled by the system's own distributed thread scheduler.

TERAFLUX does not *entirely* follow the dataflow paradigm: it distinguishes between system threads and dataflow threads. This will allow, it says, a more progressive migration of programs to the dataflow paradigm: parts of a program that have been specifically adapted for dataflow can run its threads on the dataflow-friendly cores and devices, while other parts remain restricted to the CPU. [Yu, Righi and Giorgi 2011] It is also able to repeat a thread's execution on a different core using the dataflow principle in cases where a core has been detected to be failing.

Unlike this research, TERAFLUX is an instruction set architecture that can be utilised by other languages' compilers, rather than being a language in its own right.

3.6.8 Rootbeer

Rootbeer allows a developer to program GPUs in Java. It is similar to CUDA, in that a program can be divided into CPU or GPU parts: a kernel merely implements the `Kernel` interface, which are run by a `Rootbeer` object. [Pratt-Szeliga, Fawcett and Welch 2012]

CUDA provides bindings for many languages, including JCuda [Yan, Grossman and Sarkar 2009]. However, kernels must still be written in CUDA C (or any other language that can compile to PTX and loaded by the CUDA binding library). This is not optimal, as it requires the programmer to learn how to use another programming language and programming paradigm.

3.7 Performance-Tuning Frameworks

Although it may be *possible* to run portable code on any machine that a system supports, it is a different question as to whether code would run *efficiently* on every system. This property is called **performance portability**.

Manually tuning is an option, but there are flaws: not only may targeting every possible target architecture be too thorough a task for a programmer, but some architectures may not even exist: for example, there exist C compilers for a variety of platforms that did not exist when the first C compilers were first developed.

To deal with these situations, automatic tuning techniques that can automatically optimise algorithms for task selection are necessary, so that kernels may take full advantage of such complicated platforms. Auto-tuning covers many aspects of programs, such as selecting the most suitable optimisations depending on the target architecture, or even selecting the best kernel out of many similar kernels, each compiled for a different architecture.

Williams et al [Williams et al. 2008] generated optimised kernels to compute the Lattice Boltzmann Method for fluid dynamics simulations on very different types of architectures including Itanium, Cell, and Sun Niagara processors. The authors were able to obtain significant performance improvements compared to the original code by using a script that automatically generated variants of the same code with varying optimisations applied: loop unrolling, data prefetching, and varying problem sizes. This approach automatically selected the best optimisation set.

Libraries can often optimise for minimum running time by hosting a choice of algorithms and automatically selecting the best choice at runtime. Such techniques are typically based on pre-calibration runs: contributions to test whether the algorithm was a good match for the data set. For example, the kernels implemented by the ATLAS linear algebra library perform initial tests before any data has been read. [Whaley, Petitet and J. J. Dongarra 2000] These optimisations include the number of dimensions to unroll loops over, and how to best permute the data to minimise the number of cache misses. [Li, J. Dongarra and Tomov 2009] [10]

[10] The authors reported that the auto-tuned kernels outperformed their standard counterparts consistently, with a maximum improvement of 27 %.

Another approach is runtime tuning, which provides more flexibility, but requires that the overhead needed to make a decision remains low, as to not impact the overall performance. The FFTW library provides a high-performance implementation of the fast-fourier transform, written in C. The computation of the transform is performed by a number of optimised, composable blocks that the authors call *codelets*. The combination of codelets actually applied is specified by a *plan*, which is determined at runtime via a dynamic programming algorithm. The planner tries to minimise the actual execution time over the number of operations, since the authors claim that there is little correlation between these two performance measures. Thus, the planner tests many implementations and selects the fastest, saving the result to disk to save further tests on future runs.

Finally one of the goals of the StarPU runtime system is to be able to select the best processing unit for each computation. StarPU claims to be complementary to auto-tuning efforts to ensure that dynamically-scheduled kernels are fully optimised. [Augonnet, Thibault and Namyst 2010]

4 THE DATAFLOW ARCHITECTURE

Despite advances in multitasking operating systems, the von Neumann architecture was originally thought by some to be inherently unsuitable for the exploitation of parallelism: its program counter and main memory were both global, which would become bottlenecks when more than one thread attempted to access them at once. [Arvind, Nikhil and Pingali 1989; P. C. Treleaven, Brownbridge and Hopkins 1982; P. Treleaven and Lima 1984] And although the von Neumann architecture was ultimately successful, it was not actually expected to be; a number of alternative architectures were proposed in order to avoid the bottlenecks.

Dataflow programming languages appeared in the 1970s. The main idea behind dataflow programming is that each instruction had precisely-defined data dependencies, an instead of the previous instructions setting up the execution state in a *specific* way, the state was *automatically* set up in whichever way the computer thought would be the most efficient. However, mainstream use of the dataflow architecture was curtained by the development of faster, single-threaded CPUs, making the parallelism faculties of dataflow computers superfluous [Goodman and Lujan 2011; Veen 1986]. The goal of this research is to use a dataflow model to optimise programs for GPUs, an inherently-parallel architecture, where the model may be put to better use.

```
input a, b
y := (a + b) / x
x := a * (a + b) + b
```

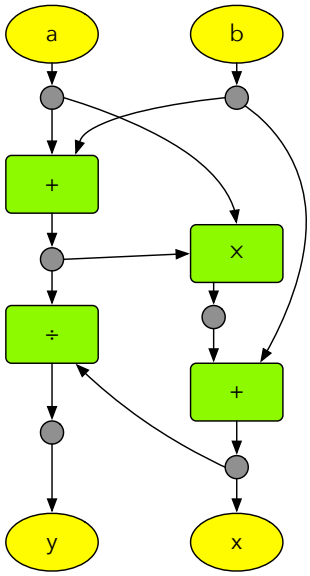


FIGURE 4.1: "An elementary dataflom program", as given as an example in *A Preliminary Architecture for a Basic Data-Flow Processor*.

4.1 Overview of the Architecture

Unlike imperative programs that consist of a list of instructions that must be followed in order, the name "dataflow" comes from the conceptual view of a program as a directed graph, where the nodes represent instructions, and the arcs between the nodes represent the flow of data. [Arvind and David E. Culler 1986; A. Davis and Keller 1982; J. B. Dennis 1974; Jack B. Dennis and Misunas 1975]

Upon execution of the program, data "flows" as tokens between the arcs [J. B. Dennis 1974]: a node produces output, which travels down the arcs towards other nodes. A node with no input arcs can start producing output immediately; a node with no output arcs can store the result of a computation; and the program finishes when each node can produce no more output. [Schauser 1991]

The opportunities for parallelism reveal themselves more readily with dataflow programs than with imperative ones. In the von Neumann model, an instruction can only be executed when the program counter reaches it, and *no earlier*. This is because it is impossible to state for certain whether one instruction requires the state to have been set up by the earlier instructions or not. In the dataflow model, whenever every input arc of a node has data in it, the node is considered *executable*; but apart from that, the execution order is undefined. Formally, von Neumann programs have a *total ordering* of instructions, while dataflow programs have merely a *partial ordering*. [1]

This can lead to the following scenarios:

- A node's input arcs receive data, and it is executed as soon as it can be—the equivalent of a function's arguments being evaluated before the function's body in imperative code. The node's output is then immediately used.
- Two nodes, with no data dependencies, can be executed at the same time, with both results being stored for later. This principle provides the possibility for parallel execution; because it is much simpler to determine whether two parts of the program would not affect each other due to a data dependency, it is trivial to run them both simultaneously.
- A node with more than one piece of data in every input arc can be run twice—a secondary computation can start before the first one has finished.

A special case of the third scenario is called *pipelined dataflow*, where secondary computations can be started before the first computations have completed. One of the more obvious places for parallelisation is in loops: if each iteration is independent, then they can all be executed

[1] Early papers use the term *fireable* instead of *executable*, as a throwback to when computations must be manually started.

at once; without the need to wait for any particular thread to finish, the program can instead wait until *every* computation has finished before proceeding.

4.2 History of Dataflow Processors

There have been a variety of general-purpose stream processors designed, each with its own programming language. An early example of a stream processor is **MIT RAW** [Waingold et al. 1997; Taylor et al. 2002] which has several simple processors on a single chip, using StreaMIT as its programming language. [Gordon et al. 2002]

Imagine is a streaming processor with several ALUs, fast local registers, and on-chip memory. [2] This was written in a language called KernelC, a subset of C; it began the technique of allowing programmers to use a language they already know to program for another device, while still forcing them to program in a stream-friendly manner. However, it exposed the details of the architecture to the programmer, so knowledge of how to program the device could not be carried on to a newer, updated version.

It is important that Imagine uses a *subset* of C, rather than a superset: KernelC is more restrictive than C, disallowing global variables, pointers, function calls, and control flow constructs other than loops. Programmers who wish to use these features must simply re-write their code to comply with the subset of C that KernelC mandates. The reason behind its limited control flow is to maximise instruction-level parallelism.

Finally, **Merrimac** [Dally et al. 2003], written in **Brook** [Buck and E. A. Lee 1992], is similar to KernelC, though it does not expose the architecture—the key benefit of this is that it allows programs to be compiled for other machines without having to be first re-written for a new architecture. This also meant that the architecture—the actual hardware—could be updated, and any new programs would automatically be able to run on it. Brook targeted Intel CPUs, but also NVidia GPUs. [Buck and E. A. Lee 1992]

The Brook language defines kernel functions that are applied to each element in a stream. It makes an explicit distinction between ker-

[2] The official term given to this memory is the "stream register file".

nels and *reductions*, such as sum or product operations, that get applied to multiple elements in a stream instead of just one. The processor then treats these functions specially, picking the processor to evaluate them based on aspects of the function such as their associativity.

4.3 Tying Dataflow to GPUs

Although the architecture of a GPU is a many-core distributed-memory system, there are similarities between the style of programs written for the GPU and the dataflow architecture:

- Both architectures shun side-effects and the writing to a bank of global memory: all functional routines can be considered independent of one another.
- Both architectures allow execution of the same code multiple times at once, through the pipelined dataflow gained by not specifying the order of certain operations.
- Both architectures have their individual kernels scheduled based on the data dependencies of each kernel, and could have their running times improved with a custom, more complex scheduler.

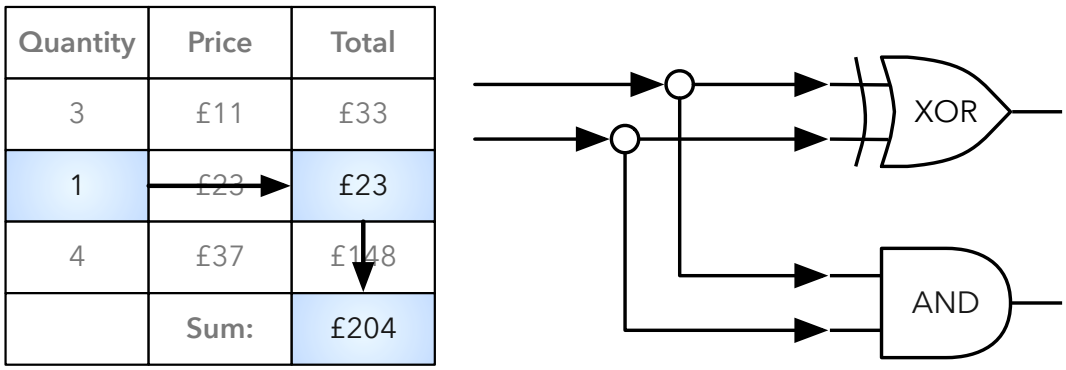


FIGURE 4.2: Two common examples of the dataflow pattern in use. In the spreadsheet (left), when a cell is updated, it sends its new value to the cells that depend on it, finally reaching a result—in this case, the total sum. In the circuit (right), values—bits, or the voltages on the wires—flow from the inputs A and B, through two logic gates, reaching results—in this case, the sum and carry bits. Both examples exhibit behaviour of data *flowing* from an input source to an output source.

- Both architectures require a scheduler to determine which kernels to run, which can introduce latency by moving data from one device to another.
- Both architectures can ignore computations not best suited for parallel processing by having a different processor, such as the CPU, execute them instead. [Bic 1990]

Finally, Goodman and Lujan, in *Scientific GPU Programming with Data-Flow Languages*, [Goodman and Lujan 2011] notice similarities between the uses of the four types of memory used by both dataflow and GPU architectures:

- Read-only memory is read by multiple kernels (in the case of dataflow) or threads (in the case of the GPU) in order to obtain their input.
- Owner-writable memory—memory that can only be written to by a specific thread—will not be read by other threads during the current thread's execution.
- Atomic memory—memory that is protected by atomic sections that allow multiple threads to write to it at a performance cost—is present in both architectures.
- Block-local or thread-local memory is used to store temporary values within a block or thread in both architectures.

Because the dataflow architecture offers the ability for a node to begin execution before its previous iteration has finished (pipelined dataflow), and the GPU's most efficient mode of operation is to execute the same node multiple times, it is plausible that a dataflow program could be run on the GPU more efficiently than on the CPU.

Additionally, the dataflow architecture mandates that each node cannot have side-effects, and cannot affect any other node. This is beneficial for multi-device programming, as it is never necessary for changes made on one device to be copied over onto another: each node can be assumed to start with a blank slate. As there are two devices at work here—the CPU and the GPU—each data transfer would take

additional time; so as the program state does *not* have to be copied over, the overall running time is shorter.

A particular viewpoint regarding dataflow machines as opposed to von Neumann machines is that they are not, in fact, orthogonal architectures, but instead sit at opposite ends of an architecture spectrum that ranges from having an entirely total ordering to having an entirely partial ordering between instructions. These architectures would trade execution order strictness for better low-level synchronisation as they are placed closer to the dataflow end.

4.4 Efficient Dataflow Compilation

Dataflow, being an architecture, is able to provide speed improvements in one of two ways: firstly, by being efficient in hardware, as has been discussed; but secondly, by being a more general platform to target, dataflow can be optimised well by compilers, with the same program able to be compiled to several different dataflow graphs, all equally valid, but with varying levels of efficiency.

There are several ways in which the dataflow architecture can be used in practice. One such concept is "macro dataflow", which begins with the observation that because the costs of dataflow instruction sequencing can be excessive, dataflow should be used only at the inter-procedural level. [D. Kuck and Sameh 1986] This would avoid the inefficiencies of dataflow but still retain certain advantages. [David E Culler 1989] However, this would mean giving up fine-grained parallelism and the ability to context-switch efficiently enough to cover memory latency—meaning this approach is more suitable for multi-processor CPUs than hundred-core GPUs.

In general, a processor capable of supporting multiple simultaneous threads of computation will suffer from more latency when executing a totally-ordered list of instructions than the equivalent partially-ordered list. [R. A. Iannucci 1988] With any multiprocessor architecture, certain instructions can take an unbounded length of time to complete, such as memory or device access, or any other form of communication. A multi-phase operation will minimise latency over a single-phase

operation because the processor idle time can be covered by another operation. [3]

In his paper *Toward a Dataflow/von Neumann Hybrid Architecture*, [R. A. Iannucci 1988] Iannucci describes such a hybrid architecture with its instruction set and programming model. This hybrid architecture uses partitioning to aggregate nodes together into units called *scheduling quanta*. These units can be specifically picked out to maximise a number of benefits:

- **Maximised run length:** With more than one node in a unit, it is more likely that the entire unit will be fireable, since there is more than one node that could receive input at any given moment.
- **Minimised synchronisation:** When an arc crosses the boundary of a node, synchronisation will have to occur. By combining the nodes into units, the scheduler can be left running in the background for longer, removing some of the overhead caused by synchronisation. In this case, there is a further benefit, as two nodes running on the same pieces of data on the same device do not require the data to be transferred onto the CPU for storage for a later time; instead, it can just be run immediately.
- **Maximised machine utilisation:** Here, Iannucci raises the idea of trying to "keep the pipelines full" by examining a set of costs related to instruction execution, synchronisation, and operand access. This can be used to compare two units, based on how well they keep the machine occupied.

[3] A *multi-phase operation* is an operation that can be divided into parts that separately initiate other operations, and later synchronise the threads before obtaining a result. They are sometimes called *split transactions*, as the operation has been *split* into separate phases. In modern programming languages, this is typically done by initialising and starting a *pool* of threads, with the operating system providing the scheduler.

5 POLYCUBE: A RUNTIME SYSTEM

In order to run programs in parallel on multiple processors, it is necessary to provide a scheduler that can both understand the data structures used in order to partition the workload in a data-parallel manner, and also detect the best processor for use for each individual workload automatically.

5.1 Design

My initial approach was to take a pre-existing imperative program and transform it into a dataflow graph, which could then be run on multiple machines. However, there are many problems with this approach:

- The dataflow processor requires that all its nodes be entirely *functional*—that is, that they have no global state, and are guaranteed to produce the same result when given the same arguments. Few programs are written this way, and instead, many programs would have to be substantially re-written in order to work within the dataflow paradigm. This would likely not be worth the time saved compared to limiting the program to CPU cores.
- It is difficult to determine the flow of data in conventional programs, even when removing a global state. It would be necessary to determine the calling points of every function, and analyse the data flow of every declared variable.
- A shortcut solution such as allowing the programmer to declare "hints" as to the nature of individual functions would still have to cater for the instances where these hints were not given. For example, the parallelising compiler extension OpenMP requires

the programmer to place `#pragma` statements before every loop they wish to parallelise. This is only effective as the compiler can handle non-parallelisable loops as well—it does not fail to compile when it encounters something it cannot handle.

Because there are so many downsides to attempting to parallelise an existing program, it is clear that a new language should be developed specifically for the system. Programs written in this language could much more easily be run on both the CPU and the GPU. The language created for the purposes of this research is called PolyLisp.

Making PolyLisp full-featured is outside the scope of this research: there are many implementations of functional programming languages already. All that is necessary is for non-trivial programs to be able to be written in it, even if the language itself is not particularly elegant.

The example that is used in this thesis is a ray tracer, which is heavily mathematical and easily parallelisable, as it has large pieces of code that do not depend on each other. A ray tracer would require the following features:

- **Floating-point data types:** coordinates and colours are specified as floating point values.
- **Mathematical operators:** the arithmetic and trigonometric functions, including `sin`, `cos`, and `tan`. These must be provided, as unlike vector functions, they cannot be provided by a library. The CUDA runtime provides implementations of these.
- **Conditional statements:** Control flow structures such as `if`.
- **Basic vector functions:** `map`, `filter`, or `reduce`.
- **Complex aggregation functions:** reduction functions where the operator is known to be associative, including `sum`, `product`, `any`, `all`, or `none`.

The key element of PolyLisp's design is that each function is given neither a global nor a local scope. The functions that have been chosen to be implemented work within this restriction: whenever they are called with the same arguments, they will give the same result.

The most common side-effect of referential transparency on a program is that it is not able to write output data or read input from the user. In this case, it is only the individual operations that are kept pure: any combination that requires interaction, such as reading a line of input from the user, can be represented as an output-only node in the dataflow graph.

5.2 Parsing

A Lisp-like syntax was chosen for PolyLisp because of its simple structure and ease of parsing. Parsing programming languages is, for all intents and purposes, a solved problem, and no ground was broken with PolyLisp's parser.

Lisp's syntax, S-expressions, is based around two forms:

- **Lists:** a space-separated list of sub-expressions surrounded by parentheses;
- **Atoms:** strings of alphanumeric characters that do *not* contain parentheses. These can be all digits, representing numbers, or a mixture of letters and numbers beginning with a letter, representing variables, functions, or control structures.

S-expressions are named in contrast to M-expressions, which are common in most other programming languages. While an S-expression encloses all its sub-expressions in brackets, such as `(add 2 3)`, an M-expression puts the operator first, such as `add(2, 3)`.[1] Control structures, mathematical operators, and user-defined functions are all called the same way.

The advantage of S-expressions is *homoiconicity*: the code looks like data, because it *is* data. A well-formed Lisp expression can be converted into a tree structure of lists and atoms. This makes it clear how it can be parsed and eventually compiled. Even more complex structures like loops or reductions over lists are closer to their parsed representation in Lisp than in another language.

[1] Although the original Lisp machine was intended eventually to run on M-expressions, its users preferred the S-expression representation, and that syntax stuck.

The main disadvantage is that mathematical formulas are no longer written in infix order: instead, they are written with the operator put before the operands. This appears unnatural and confusing to some users, but means that it is impossible for the operator precedence to be confusing, as the surrounding brackets are mandatory.

Parsing these expressions is assisted with an underlying tokenising stream. This takes an input stream of characters, such as from a source file, and outputs the stream reformatted into *tokens* of open and close parentheses, spaces and newlines, and groups of alphanumeric characters. This allows atoms to be treated as a single token along with the open and close parentheses.

Once tokenised, the parser checks to see whether the first token is an open parenthesis. If it is, it reads as many further expressions as it can, separated by spaces. This call is recursive, so it is able to balance nested open and close parentheses without problems: a list containing other lists is perfectly acceptable, as the parser reads *whole* lists from the stream, leaving only the final close parenthesis at the end of the expression after all the leaves have been parsed.

One other possibility is that the first token is a close parenthesis, which indicates a syntax error: a list has been opened without being closed. Again, because whole lists are read for each expression, it is not possible for there to be stray parentheses in the program for reasons other than this. Otherwise, the token is converted to a leaf in the tree.

5.3 CPU Evaluation

After a source string has been parsed into a tree expression, PolyCube allows it to be evaluated by the CPU. This is helpful even when running on a system with a GPU: not only is it useful during testing, to see if the result differs on the CPU than on the GPU, but as not all expressions are able to be run on the GPU efficiently, it may be preferable to use the CPU instead.

Evaluating an expression requires a lookup table of variables and functions. [2] This, like the variables in the programs, remains the same for each expression being parsed; if a variable has to be declared, then

[2] PolyCube takes the functional definition of *variable*, which is an atom that *may* have a different value every time its name is referenced. For example, when looping over a list, assigning every element to `e` makes `e` a variable. This is in contrast with the imperative program definition, which is an area of memory that can be modified. In PolyCube, mutation is not supported in favour of functional idioms, so this definition would be meaningless here.

a new lookup table is constructed for the scope in which the variable exists. This means that variable and function definitions are always local, never global.

Lists and atoms are evaluated differently. An atom that is entirely digits represents a number, which is evaluated by reading it as an integer or decimal. An atom that is alphanumeric represents a variable, which is looked up by consulting the table of variables in the current scope. If the variable is not present, an error is thrown.

Evaluating a list is done by assuming the first element of the list as the name of a function to run, and the rest of the elements, if present, as the arguments. These elements are then evaluated, in turn, in order to obtain the values to use as the arguments: evaluation is *recursive*. Under the hood, each list, once parsed, is converted to a function object of its own class. This means that the evaluation methods for each function can be kept separate.

However, this recursive nature means that it is only feasible for CPU execution. In order to avoid any form of recursive function call, any code that could be run on the GPU must be compiled instead.

5.4 PTX Compilation

Each function in PolyLisp is able to be compiled into a PTX entry that can subsequently be interpreted by JCUDA to produce a fully-working PTX kernel. Despite being an assembly language, PTX is conceptually simple, and there were few difficulties in compiling PolyLisp into PTX.

Compilation is remarkably similar to evaluation: atoms are evaluated to themselves, and lists evaluate their arguments in turn, before having their own code evaluated. The only difference is that instead of calculations being performed during evaluation, a series of instructions is written to a list.

Once the PTX kernel has been generated, it is exported to a file as text. From here, it can be compiled using NVidia's `nvcc` compiler, which can accept a text file of instructions with a name ending in `.ptx` as PTX assembly. [NVidia 2013] The assembly is compiled into a *cubin* (CUDA binary), which can finally be loaded by a program that supports the CUDA format. [3]

[3] The counterpart to a cubin is a *fatbin* or *fat binary*, which contains the compiled PTX code for multiple devices, instead of just the device present on the system. This is used to avoid the delay or complexity of compiling the PTX code before execution on other machines, in cases where compiled kernels must be distributed.

5.4.1 Branching

One special case that has to be dealt with is *branching*: skipping to another point based on the result of a conditional expression.

Conditional expressions are of great concern to PolyCube because of the way the GPU operates. On the CPU, two threads can execute two different instructions without problems, but on the GPU, every thread must either execute the same instruction, or pause while the other threads do. This means that conditional expressions can often make code take twice as long, as each thread would have to wait for both branches, so they are not to be taken lightly.

However, they must still be included in the produced PTX assembly, because it will not always be more efficient to fall back to dataflow. With short expressions, the time lost by having one of the two threads wait during a conditional may not be worth the cost of having to transfer more data to the GPU and back. This is why the PTX generation code needs to support both labels and branching.

When a conditional expression is reached, the generator creates unique labels before the `else` point and after the entire block of instructions. It then places the code for each of the sub-expressions after the corresponding label. This means that, if the expression evaluates to false, the instruction pointer jumps to the label, skipping some of the code; if it holds true, the program continues as before, only jumping to the second label to skip the other sub-expression.

5.4.2 Memory

PTX provides the developer with a large number[4] of registers for each data type. These registers are later optimised by the compiler, but CUDA's own compiler is able to optimise the number of registers used itself into the number actually in place on the GPU: this number varies depending on the graphics card, so allowing a variable number and compressing it later is the only safe option.

Including registers, there are eight different memory spaces accessible from PTX code. These are listed in Table 5.1.

[4] The limit for each kernel is 32 767, which is highly unlikely to be reached by an individual kernel. The compiler will display an error if this limit is reached.

Name	Description
`.reg`	Individual registers themselves.
`.sreg`	Special, read-only platform-specific registers. Typical kernel code is not concerned about this space.
`.const`	Shared, read-only memory.
`.global`	Global memory, which is shared by all threads.
`.local`	Local memory, which is private to each thread.
`.param`	Any parameters passed to an individual kernel.
`.shared`	Memory that can be shared between threads in a block.
`.tex`	Global texture memory, which has been officially deprecated in recent versions of CUDA by NVidia.

TABLE 5.1: The eight different memory spaces accessible from PTX code.

Out of these eight memory spaces, PolyCube is only concerned with four.

The operands and results of any PTX instruction are loaded into and read from the `.reg` space, which is fast compared to the others. [NVidia 2012] Unlike named variables in traditional imperative programming languages, local variables do not *need* to be stored in their own memory space, as the `.local` space is used for *thread-local* memory rather than *function-local* memory. Instead, it is possible to store intermediate calculations entirely in registers. [5]

Any constants are loaded into the `.const` space. Although these can be global, such as a constant declared at the top level of the program, accessible by all functions, they are kept out of the `.global` space as they do not have to be modified.

Lastly, the values that are passed as input arguments to functions use the `.param` space. Although function calls are inlined on the GPU, the `.param` space must be used for the parameters and return values of functions that use multiple-value structs, which are not supported by the `.reg` memory space.

[5] Storing function-local variables inside registers is also a common practice for register-based virtual machines. Optimisations (including the ones presented in Section 5.5) will ensure that the number of registers does not exceed the number available on the device.

5.4.3 Bottom-Up Compilation

As is said above, compilation is a very similar step to evaluation. A function call is compiled into a list of PTX instructions by first compiling

the expressions that make up its arguments, moving the result into a new register, and adding the resulting instructions into the list. Then, the instructions that make up the actual function are inserted, with their arguments set to the registers that hold the arguments' return values.

For mathematical operators, the function call is a simple one, compiling down into a single PTX instruction: for example, in the case of the + operator, the `add` instruction is used. However, for user-defined functions, the entire function call must be inlined, meaning that the function's entire definition gets inserted into the list of instructions.

If the expression being compiled is a constant value, the value is loaded into the shared read-only memory space and that memory location is used. If the expression is a variable or a parameter, a similar load happens with the register (`.reg`) or parameter (`.param`) memory spaces. This allows compilation to be implemented recursively, using the constant values as the base case, and function calls as the recursive case.

5.5 Optimisation

Each compiled kernel runs through several optimisers to produce code that is on a similar level to that of NVidia's official compiler, which is useful when comparing runtime metrics. While this research is not about investigating compiler optimisations, a few have come in useful. By default, the compiler outputs very inefficient PTX code for the sake of simplicity, leaving it up to the optimisers to generate faster-running instructions.

5.5.1 Register Minimisation

Instead of re-using existing registers when possible, the PolyLisp compiler *always* opts to use new registers for every argument, and another for the return register. This not only makes the code generation much simpler, but also makes it easy to perform optimisations at a later stage, as it can assume that a register will not have been re-used.

Register minimisation involves detecting cases when a register's only use is to have data loaded into it from another register exactly once. If this is the case, then the register can be replaced with the register that its data came from. This eliminates register chains and unnecessary load operations.

5.5.2 Combining Instructions

There exist several instructions in PTX that perform the work of other, simpler instructions, without the overhead of having to execute multiple ones. For example, the `mad` instruction multiplies two values and then adds a third, which would usually take two separate `mul` and `add` instructions.

There are also instructions that perform an operation with one of the operands already loaded, such as `neg`, which negates a number by multiplying it by -1. The optimiser detects when two or three instructions are used in this manner, and replaces them with a call to a single instruction, decrementing the numbers of the following registers by the appropriate amount.

5.5.3 Dataflow Graph Construction

The second stage of compilation is to collect the individual PTX kernels that have now been compiled and link them together to form a *dataflow graph* of computation. This graph will be examined by PolyCube at runtime, and executed in what the system computes to be the most efficient way possible.

Expressions that operate on scalars—single values, rather than lists containing many values—can simply be executed on the CPU recursively, with a function's arguments being evaluated in turn, then fed to the function itself. For the common case of a function simply operating on a scalar, the function will become one self-contained dataflow node.

Then, there are two special situations that the compiler is aware of:

- **Recursive function calls:** A function cannot call itself in CUDA C, as the functions will be inlined, and it is not possible to inline a

Function	Results	Destination
`map`	n	Anywhere
`filter`, `reduce`	Variable	CPU
`sum`, `all`, and others	1	Anywhere

TABLE 5.2: Table of functions and the number of results they produce, given a list of n elements as input. The most common reduction functions produce a single element as output, while `filter` and `reduce` produce a variable amount. The scheduler must then move these results onto the CPU to efficiently concatenate them, leaving the GPU for other computations.

recursive function call. Furthermore, any work-arounds will still be problematic due to the parallel nature of the GPU. PolyCube detects instances where a function is calling itself and, instead of returning another expression, returns a node in the dataflow graph that points to itself.

The arc between the node and itself contains a special value that, internally, contains the values of the arguments that the next recursive call shall be run with. There are limitations with this approach: namely the fact that there can only be one point where the function is called recursively, as the point where the function was called is not recorded. Without this information, the system does not know the point where execution must continue from.

- **Vector function calls:** When a function is called across multiple pieces of data, such as with a `map` or `filter` call, it also creates a dataflow node rather than another expression. This node is then called specially based on what class of function it is, based on Table 5.2.

If the function performs vector operations for part of it, rather than being *entirely* a vector operation, it is desirable to perform as much of it on the GPU as possible. [6] In this situation, the producer of the values will become a node that flows data to another node—the vector operation. Similarly, the consumer of the values, if present, will become another node that takes input from this operation.

With nodes created for these situations, it becomes possible for the runtime to determine when two vector operations are run consecutively, and can run them both on the GPU while keeping the data on that device, eliminating the latency from transferring the data twice.

[6] It is possible to have one function be compiled into both CPU and GPU versions when it is used from other functions that, between them, are run on both devices.

5.5.4 Collecting Data

Having a limited set of functions allows the complier to gain knowledge about the program that would otherwise not be present. For example, it can know for sure that the operator used in the `sum` or `any` reduction functions is associative, and so it can perform the parallel algorithm that requires this to be the case, instead of the slower, serial algorithm. [Pharr and Fernando 2005]

More importantly, it can be aware of the results that each function provides. Table 5.2 lists the classes of functions in the language, and the number of results produced given a list of n elements as input.

Knowing the number of results in advance proves important to the scheduler, as latency between the GPU and the CPU comes into effect. Moving a large number of results between the two devices can slow the execution of the kernel down, and it may be preferable to transfer a single element than many.

The `filter` and `reduce` functions produce a variable number of results. Although these functions are run on the GPU, their results must be processed further on the CPU in order for the data that is no longer relevant to be removed. This is the data that did not match the predicate (in the case of `filter`), or intermediate results (in the case of `reduce`). Without this operation, the GPU would operate on a dataset much larger than necessary, as the threads for the removed elements would still be executed, causing a slowdown. Therefore, it is in the scheduler's interest to remove these elements from execution as soon as possible.

On the other hand, the `sum` and `product` functions are guaranteed to only return a single result, making them a much better candidate for running on the GPU. PolyCube will prioritise running these functions on the GPU if it gets the chance.

CASE STUDY: RAY TRACING

6

Ray tracing is a graphical technique to generate highly-realistic images from 3D scenes that can accurately simulate many real-world optical effects, such as reflection, transparency, diffraction, and shadows.

Although speed is usually considered a virtue for computer programs, rendering engines for computer games are *limited* by needing to perform fast: a computer game engine would use faster, inaccurate calculations over slower, more accurate ones in order to consistently produce 60 or more frames per second. These inaccurate calculations' results would often be indistinguishable, or barely distinguishable, from the accurate ones when playing a game, so they were preferred. The engines would render a scene one polygon at a time, with visual effects added in afterward. Ray tracing, on the other hand, renders scenes one pixel at a time, for the maximum possible accuracy, with some images taking hours or even days to render. [1]

Ray tracing works by creating a *ray* of light for each pixel in the image, and then *tracing* its path to see whether it collides with any objects. If it does, the visual effect is simulated with mathematical formulae, and the colour of the pixel is calculated depending on the object hit and the point of collision. The function to calculate the colour often fires additional beams, allowing real-world effects to be rendered accurately:

- **Reflection:** A ray can bounce off an object, creating a secondary ray travelling in a different direction. The resulting colour is a blend of the colour of the object and whatever the secondary ray hits.
- **Transparency:** Instead of bouncing off a transparent object, a ray can partially pass through it, creating a secondary ray travelling in the same direction. The resulting colour is a blend of the object's

[1] For 3D animators, rendering time is a big issue: the feature film *Shrek 2* needed 10 million hours of CPU time to render.

colour and the colour of the object that the original ray would have hit.

- **Shadows:** Upon hitting an object, additional rays are traced from the point where the ray hit to the light sources, and if there are objects in the way, the resulting colour can be darkened.

The slow performance of ray tracing is usually managed by parallelising the process: in fact, one of the most common uses for a cluster of computers is as a *render farm* for producing animated video. Ray tracing is considered a *trivially-parallelisable* problem, where opportunities for simple parallelisation are easy to spot and implement. [Chalmers, T. Davis and Reinhard 2002] In this case, with each individual component independent, a separate thread can be used for each line or even pixel; additionally, with animated video, the task can be split up further, with each frame being rendered on a different processor. Additionally, many raytracing functions consist of doing similar calculations a large number of times, and with a shared-memory system such as a CPU, balancing the computational load between every processor is straightforward. [Fernando 2004]

6.1 Ray Tracing Challenges

The GPU seems like a good fit for ray tracing, given the highly-parallel nature of the problem. However, the technique does not translate over perfectly. Ray tracing is a *recursive* procedure, where a single ray could bounce around a scene several times before finally resulting in a colour. This is troublesome, as CUDA, as well as GPUs in general, do not support recursive function calls. In CUDA's case, every function run on the device is inlined, and a straightforward recursive function cannot be fully inlined.

This can be avoided in one of two ways, both of which have drawbacks:

- Pre-allocate space for an arbitrary maximum number of rays, and perform that many iterations of the ray-tracing loop each time. This will use more time and memory, as the GPU's nature

dictates that the maximum amount of space must be allocated for each individual thread, even if much of it goes unused.

- Use a stack on the host computer's memory, and co-ordinate the ray tracing from there. The threads may be executed in groups so that only the first iterations of the function are executed the first time, followed by only the threads that need a second the second time, and so on. This technique is more complicated, and requires much more communication between the host and device processors.

A GPU ray tracer that processes each possible secondary ray will waste a lot of time. The threads that complete after one iteration will be forced to wait for the threads that take many iterations to complete, because of the GPU's large overheads when doing state-based computation like this. [Timothy J. Purcell et al. 2002] A ray tracer that offloads parts to the GPU may find the CPU-GPU bridge a bottleneck, negating any possible speed improvements that the GPU offers. [Carr, Hoberock et al. 2006]

Currently, the majority of opportunities for optimisation of GPU-based ray tracing are through the use of more advanced data structures to partition the scene, in order to make sure that *every* object does not need to be tested for *every* ray. A simple grid structure can be implemented on the GPU, [Timothy J. Purcell et al. 2002] but this data structure only works well for scenes with uniformly-distributed objects. A more complex structure can be managed in parallel on the CPU [Carr, Hoberock et al. 2006], but again this is limited by the CPU-GPU bottleneck. Despite these limitations, there have been a number of attempts in solving the secondary-ray problem, and they are outlined below.

6.1.1 Parallel Ray Tracing

In *Photo-Realistic Ray Tracing Kernels*, Ernst and Woop classify a variety of different parallelising approaches to improve ray-tracing rendering speed into three categories, based on the object that is parallelised when the ray-object intersection calculation is performed.

The first of these states that when multiple rays intersect a single object, the rays can be parallelised. This technique is called *packet tracing*. It has been implemented on the CPU by employing Simple Streaming Extensions (SSE) instructions [Wald 2004], but on the GPU, it runs into the secondary ray problem.

In *Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing*, the authors are able to avoid the problem by maintaining a stack of ray iterations in private thread memory, [2] but note that this suffers from a performance decrease by having to constantly access this pool of memory. [Garanzha and Loop 2010] The authors also utilise a ray-sorting procedure to reduce the number of execution branches that the program must take, assuming that rays coming from a similar part of the scene will likely all hit the same object. Although they *do* cite an increase in rendering speed, their technique is specific to their mode of ray-tracing, and is not possible to apply generically to programs that need GPU recursion.

The second category parallelises the objects being intersected with, instead of the rays that intersect with them. This is efficient when the objects are being stored in a suitably-efficient data structure, such as a tree that stores nearby objects together, similar to the ray-sorting technique mentioned above. [Benthin et al. 2012]

The third category parallelises both the rays and the objects they intersect.

6.1.2 Previous Work

Purcell, in *Ray Tracing on a Stream Processor*, abstracts the GPU to a stream processor in order to produce ray-traced images, and claims that ray tracing is most naturally and efficiently expressed in the stream programming model [Timothy J. Purcell et al. 2002].

Purcell's work cites similar ideas about dataflow, which may have originated from the implementation of the stream processor used. There are *kernels*, which are specialised, stateless function calls, and *streams*, that hold intermediate results between different stages of the program. A kernel is invoked on many different records that all require

[2] In CUDA, this would be the `.local` memory space, as described in Table 5.1.

the same processing. This is one of the core tenets of the dataflow architecture, that the same processing should be run on as many pieces of data as possible; however, Purcell does not explore the abstraction capabilities of kernels and streams.

Purcell has implemented his raytracing model as a diagram that holds many similarities to the dataflow architecture explored in this report. The *kernels* are all general functions that can easily be applied to many individual pieces of data at once, and the arcs are used to specify the data dependencies between kernels. Each kernel can be run independently.

Purcell describes an implementation for a stream processor, which is required to allow data-dependent branching: allowing different instructions to be executed for different threads. [Timothy John Purcell 2004]

Carr's method in *The Ray Engine* [Carr, Hall and Hart 2002] offloads the repeatable, ray-triangle intersection code to the GPU, while using the CPU as a scheduler. Although this seems like the perfect use of both systems, this method was found to require constant supervision from the CPU, with large amounts of data transferred over a narrow-bandwidth bus, which lessened the effect of the GPU.

In *Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images* [Carr, Hoberock et al. 2006], Carr recommends a method such as Purcell's method of offloading further computations to the GPU to limit the latency between the host and the device.

One suggestion made by Karlsson, in *Ray Tracing Fully Implemented on Programmable Graphics Hardware*, [Karlsson and Ljungstedt 2004] is that the CPU could be running a program while the GPU is in use, which is one of the core tenets of this research. In this implementation, the CPU is used to run a display loop for the output of the ray-tracing algorithm, but the possibility that multiple pieces of hardware could be used at once is one that has arisen before. Karlsson's implementation leaves out shadows and reflection entirely.

Allgyer, in a Masters thesis [Allgyer 2007], transfers a stack-like structure to the GPU and iterates over it. Allgyer cites problems with this approach, namely that CUDA cannot dynamically allocate memory from within a kernel, so the size of this data structure must be fixed.

Therefore, a middle ground must be found between overestimating the size of the stack (and causing memory to be wasted) and underestimated (causing the program to crash).

NVidia themselves have an implementation of real-time raytracing on the GPU, *OptiX*. It returns to having two programmable shaders instead of a fully-programmable pipeline such as the one CUDA offers. These shaders are specific to ray-tracing: there is no possibility of adapting OptiX to general-purpose computation. [S. Parker et al. 1999] OptiX works by taking user-supplied CUDA kernels that describe how a ray should act in certain circumstances. [S. G. Parker et al. 2013] [3]

The OptiX scheduler explicitly selects a single state for an entire block to execute using a heuristic, thus allowing the threads that cooperate the most to run with the same instructions, decreasing the total running time of the block. Threads within the block do not require the state to be idle during that iteration. This is similar to having a CPU-hosted scheduler pick the threads that work well together, and schedule them to all be run at once.

One final recent development is Samsung Reconfigurable GPU based on Ray Tracing (SGRT), a mobile GPU architecture for real-time ray-tracing. Ray tracing in the mobile environment is difficult because of the lessened computational power that mobile devices possess. [W.-J. Lee et al. 2013] Like OptiX, SGRT divides its processing amongst several different processors, each suited to a particular task. This is reminiscent of dataflow, where a program can be thought of as several processors, each one running only when necessary.

6.1.3 Observations

There are a number of observations that can be read from this body of work:

- It is possible to parallelise a program in multiple ways, with results varying depending on the parallelisation technique used.
- The problem of not being able to implement recursion efficiently is difficult; the most efficient results are from programs specifically written to solve the problem, rather than from a generic solution.

[3] Real-time raytracing has been a commonly-cited long-term goal for graphics technology, due to traditional rendering methods being fast but imprecise. Before NVidia's technology, real-time raytracing has been demonstrated using less-powerful hardware with smaller scenes [Wald 2004; S. Parker et al. 1999] and with more complex scenes using a computer cluster. [Wald 2004] It has also been demonstrated on FPGA-based systems.

- Although the individual threads in a parallel block can be executed simultaneously or in any order, it is possible to run the fastest by running the threads with similar data together, as these threads will often reach a similar number of recursive calls deep, maximising the available time that the GPU's cores are in use.

6.2 Performance

An example scene of multicoloured spheres and a plane (shown in Figure 6.2) was rendered on both the CPU and the GPU using a program written in PolyLisp (described in Chapter B). This section discusses the performance results of these two processors.

The scene was designed not only to have multiple reflective surfaces, which are necessary here to demonstrate the secondary ray problem, but also to have rays reflected varying numbers of times in different parts of the image. The number of rays necessary to render each portion of the image is plotted in Figure 6.1. This chart shows that the vast majority—79.3 %—of pixels in the image take exactly two rays to render, compared to just 2.1 % of pixels taking exactly one. The rest of the graph trends towards zero, with 0.17 % taking exactly ten rays, and 0.000 09 % taking fifty. [4]

[4] The chart also shows a surprising pattern from 5 to about 50 rays, where the rate of change from an odd to an even number of rays is greater from that of even to odd. It is unknown why this happens, but could arise from the image being roughly symmetrical.

This means that, at every level of recursion depth reached while rendering the image, there will *always* be a fraction of the rays that require additional processing after this step—which triggers the secondary ray problem while running on the GPU.

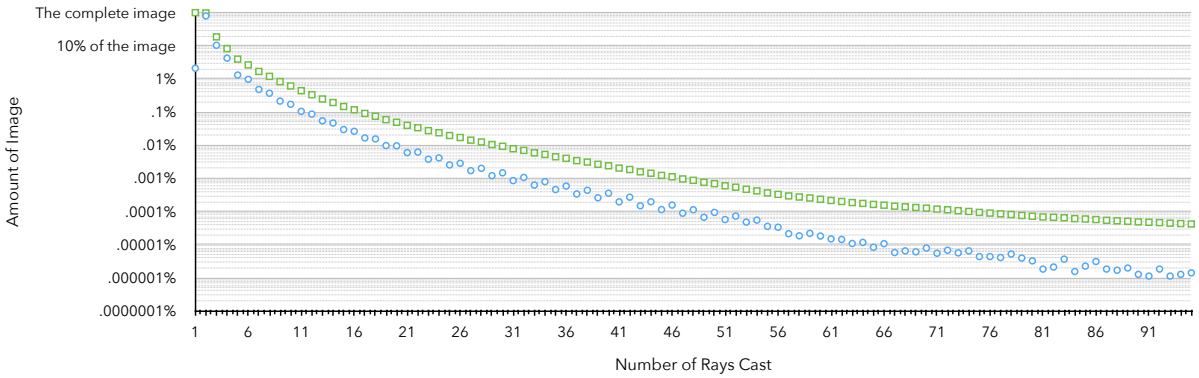


FIGURE 6.1: A graph, on a logarithmic scale, of the region of the image that can be rendered in *exactly* a given number of rays (), and the region that can be rendered in *at least* this number of rays ().

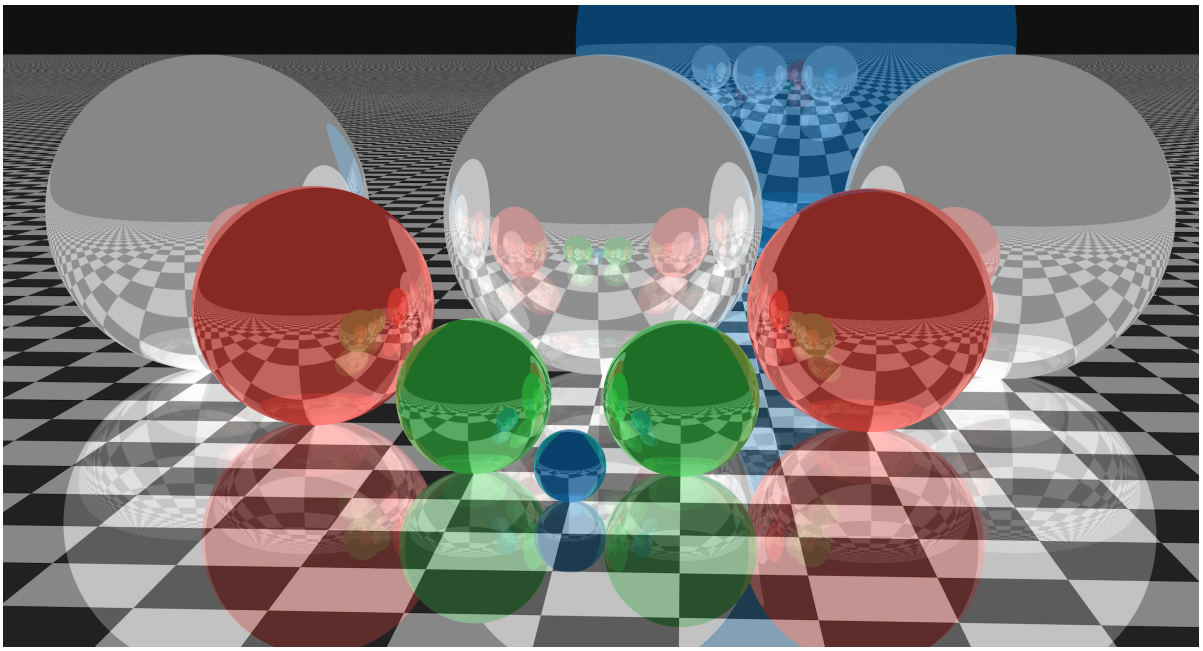


FIGURE 6.2: The scene rendered by PolyCube. The resulting image was identical on the CPU and the GPU. This particular scene was designed not only to have multiple reflective surfaces, but also many regions of the image where rays are reflected varying numbers of times before finally hitting a non-reflective surface (see Figure 6.1). Although more complex shapes and surfaces are possible, simple planes and spheres were chosen to simplify the PolyLisp program and the mathematics involved.

6.2.1 CPU Performance

The ray tracer was initially compiled for the CPU only, and was run multiple times with varying numbers of CPU cores and maximum ray recursion depth. The results are shown in Figure 6.3, plotting the execution time against the maximum depth from 1 to 32 for five numbers of cores.

The results support the fact that the CPU is able to execute different instructions on each core at the same time. For the first five rays, the time taken to render an image climbs quickly, with the additional rays having a large effect on the execution time. After five, however, the execution time increases much more slowly, showing how the remaining cores are not idle for the remainder of a particular level of recursion depth. After approximately fifteen rays, the execution time does not noticeably increase for *any* number of cores, as such a small part of the image requires this may rays to be rendered completely.

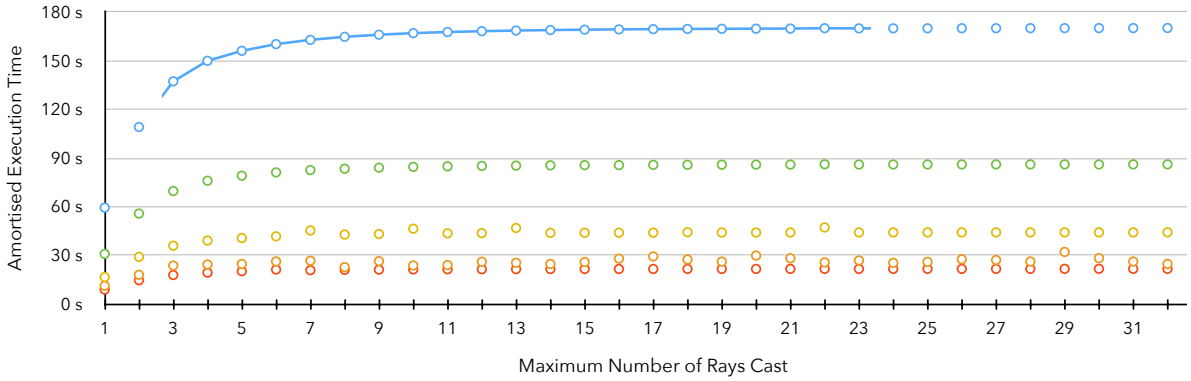


FIGURE 6.3: A graph, on a linear scale, of the running times of the ray tracer run on a CPU against the pre-set maximum recursion depth using just 1 core ( ), 2 cores ( ), 4 cores ( ), 8 cores ( ), and 16 cores ( ) of a 16-core machine. As is expected, the more cores used to render the image, the less time it takes. Also, the steep increase in execution time for the first few rays reflects the growing amount of image that is affected by another level of recursion depth being added (see Figure 6.1). The CPU rendered a 20 480×11520 image four times, in order to average out any discrepancies in individual runs.

The number of cores is also demonstrated to have an effect. Figure 6.4 shows that plotting the running time against the number of cores used for execution results fits a negative hyperbolic curve. Going from 1 core to 2 almost halves the execution time, but going from 15 to 16 has an extremely small effect. This is because ray tracing can be very easily parallelised, with a separate thread for each row, or even pixel of the image.

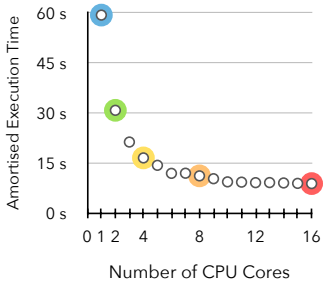


FIGURE 6.4: A graph of the running times of the ray tracer against the number of pre-allocated CPU cores. This is an alternate plotting of the first column of data in Figure 6.3, along another axis. The curve is negative hyperbolic. ($y \approx 72x^{-0813}$).

6.2.2 GPU Performance

Next, the ray tracer was run on the GPU in two configurations:

- **A naïve function-calling implementation:** Both processors are used for execution, but any recursion should have a limit hard-coded during compilation. This uses the technique described in Subsection 2.3.3 to deal with the GPU's restrictions on recursion.
- **The PolyCube dataflow implementation:** Both processors are used for execution, with PolyCube's runtime scheduler examining the program to optimise the recursive function calls.

Both implementations are tested and compared to each other and the CPU version. This is in order to isolate the changes in running time

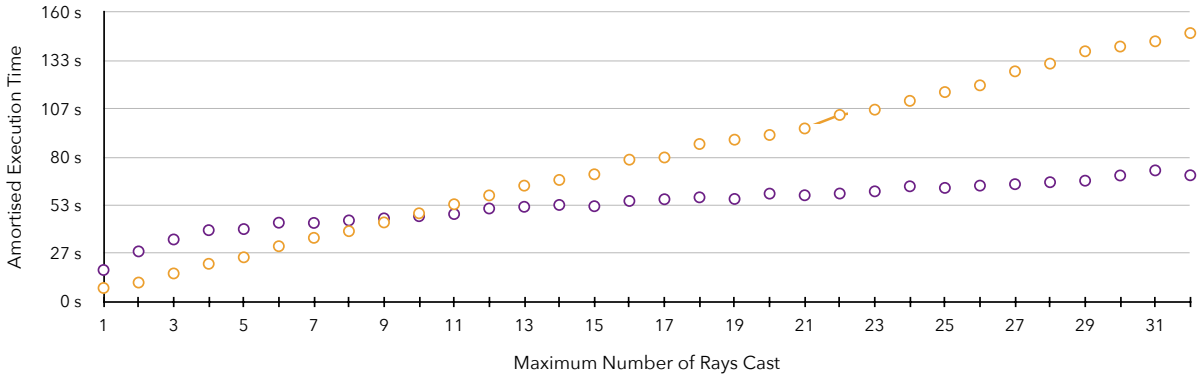


FIGURE 6.5: A graph, on a linear scale, of the running times of the ray tracer run on a GPU against the pre-set maximum recursion depth using an unoptimised (○) version and an optimised (○) version of PolyCube. As with the CPU, the GPU rendered a 20 480×11520 image four times, in order to average out any discrepancies in individual runs.

to only the differences caused by PolyCube, and not any differences that would be inherit in *any* program running on multiple but separate processors.

The results are plotted in Figure 6.5. As is expected, the naïve approach, in general, takes more time than the optimised approach using PolyCube's runtime scheduler.

As the two programs were executed on the same device, it is possible to compare their running times. The naïve approach actually has *less* running time than the optimised version when the maximum ray depth is 9 or less. This is because

Surprisingly, when the lines of best fit are calculated, both curves actually conform to a *linear* fit. The fits are shown in Figure 6.6.

In the case of the naïve approach, this was expected, as each added level of recursion depth would slow down *all* of the available threads as they wait for the final few to finish calculating. This means that each level would add a constant amount of execution time, resulting in a linear graph. These results fit the line $y = 4.582x + 2.543$ with $R^2 = 0.9989$.

In theory, the results for the *optimised* would resemble the CPU graph, with each level of recursion depth adding smaller and smaller amounts of execution time as the number of pixels that require more depth than that to complete get fewer and fewer.

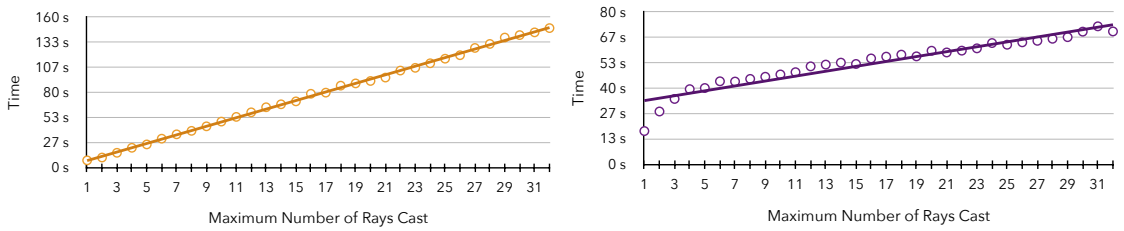


FIGURE 6.6: The best fit curves of the GPU running times in Figure 6.5, again using an unoptimised (, left) version and an optimised (, right) version of PolyCube. Both fit linear curves, but the optimised version deviates from the curve when there are few rays cast.

This curve does fit the graph—but only with a depth of 5 rays and below! This can be explained due to the latency involved. Not only does each rendering step involve the transfer of data to the GPU and back, but it also needs to post-process the data on the CPU before any further work can be done with the results. This time begins to take a length of time approaching the actual rendering execution time, and so the resulting graph is in fact linear.

These results fit $y = 1.289x + 32.136$ with $R^2 = 0.909$. The lower R^2 value is due to the numbers of rays at the beginning of the graph—with depth 4 and below—throw off the line. It most definitely does not take 32.136 seconds for PolyCube to initialise.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

It is not enough to say that the use of parallel processors is growing—they will soon saturate the market, and it will be commonplace to have a CPU+GPU hybrid processor in your computer instead of just a CPU. As GPUs become more general purpose, they will enable users to use the graphics card for general-purpose computation, instead of just for playing computer games.

Problems once thought intractable on the current generation of hardware, such as factorising hundred-digit semiprimes, or trying every possible combination to crack a password, are falling to the sheer processing power that these new parallel systems offer. However, the majority of parallel software is specialist, and many programs do not even try to use the computational power available to them.

In the world of dataflow, there are many papers from the 1980s when von Neumann machines were, relatively, not as fast, and there was further interest in dataflow machines as a competitor to the von Neumann architecture. Although dataflow is still a topic of research, it is most useful here not as a stand-alone hardware architecture, but instead as an intermediary architecture, for its ease of analysis.

An original goal of this research was to be able to parallelise *any* given program by analysing the data dependencies of every loop in order to find a dataflow diagram for the program. This proved to be a bad idea, as the compilation time necessary to compute the dataflow graph quickly exceeded the execution time! Instead, the approach was chosen to let the programmer decide which parts of the program should be parallelised, in order to make the dependency graph smaller, giving the dispatcher less work to do. This is the approach taken by CUDA: programs are run on the CPU, unless explicitly stated otherwise.

7.2 Results

Upon comparing the results of the ray tracer's execution on both the CPU and the GPU, there is evidence that with certain computations, the GPU's processors sit idle for a non-trivial part of the running time, and by re-arranging the order in which certain functions are executed, the the amount of idle time on the GPU can be minimised without this having to be explicitly done by the program itself. Here, it was the scheduler that was able to partition the program into workloads best run on the CPU and GPU by analysing the program, owing to the functional, descriptive style provided by PolyLisp.

Running only on the CPU, a PolyLisp program will have a slower running time than one written in straight C. However, only the bare minimum amount of work was undertaken to improve the CPU running time, instead focusing on compilation to the GPU: the CPU-bound code is evaluated at runtime, instead of being compiled first. Were the target for the CPU a more efficient architecture, there would be much less detriment to writing a program in PolyLisp even for the base case of running on one CPU.

7.3 Future Work

There remain several possibilities for future work that could be built on top of this research. These are outlined below.

7.3.1 Further Programming Language Features

The primary feature of the scheduler is its ability to discern which parts of a program should be run on the CPU or the GPU. To do this, it has had to give up certain language features that don't ease parallelism, such as data structures, a type system, or mutable variables. It should be possible for a sufficiently-smart compiler to still be able to analyse a program that uses these programming language features, keeping the more advanced parts on the CPU while still being able to run the parallelisable components on the GPU.

One drawback of even offering these features is that it becomes very easy to prevent parallelism entirely by, for example, using mutable variables in an inconvenient place. The balance between language features and runtime speed has often been a difficult one to find, and this would be in no exception.

7.3.2 Additional Configurations of Hardware

This research restricted itself to the combination of a single CPU paired with a GPU; this is the most common hybrid architecture based upon consumer-grade hardware, and as such, is the largest target platform for software. But certain problems may require more specific configurations of hardware, such as a render farm employing tens of GPUs to all run the same program, or a computation cluster with multiple high-power CPUs.

It may be possible to adapt the scheduler to target *arbitrary* configurations of hardware based on factors it determines when execution begins, such as picking algorithms depending on the available hardware, or how best to permute data to avoid cache misses. A simple, yet important, example would be having the scheduler prioritise keeping data on *the same* GPU when two are present, rather than to constantly move values between two.

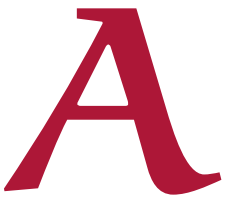
Some examples of performance-choosing schedulers are listed in Section 3.7, Performance-Tuning Frameworks.

7.3.3 Runtime Graph Partitioning

An even further goal is to have the scheduler react to changes in the set of available processors by re-partitioning the graph at runtime to accommodate for additions and removals of processors. This would be accomplished by *trialling* various mappings of individual kernels to processors and seeing which combination proves the most effective.

Runtime tuning is also listed in Section 3.7.

POLYLISP DEFINITION



This appendix describes the PolyLisp language, a dialect of Lisp used to compile and run the ray tracer used in this thesis. It lists the features and limitation of each language function or construct, and details how each is compiled down to PTX assembler instructions.

A.1 Operators

A.1.1 +, −, *, /, %

These arithmetic operators work in the same way as their C counterparts, performing addition, subtraction, multiplication, division, and modulus respectively. They are defined for floating-point types.

The − and / operators have special behaviour when run with just one argument: they negate or reciprocate it. This allows special `neg` or `rcp` instructions to be used instead in the resulting PTX assembly.

This behaviour has no runtime penalty since it is not possible to call functions with a varying number of arguments at runtime—it is merely a compile-time check.

A.1.2 ==, !=, >, >=, <, <=

These boolean operators also work in the same way as their C counterparts, comparing two values of the same numeric type and returning `true` or `false` values of type `.pred`.

A.1.3 `and`, `or`, `not`

These logical operators also work in the same way as the C operators &&, ||, and unary `!`. They are defined for values of type `.pred`.

A.2 Functions

A.2.1 `sqrt`

This function returns the square root of its argument, which must be a floating-point number.

A.2.2 `floor`

This function returns the floor of its argument (rounded down to the nearest integer), which must be a floating-point number.

Despite returning an integer, it will also be a floating-point number. This is to mimic the behaviour of the CUDA implementation of this function.

A.3 Control Flow Constructs

A.3.1 `if`

`if` takes three arguments, but unlike a function, does not evaluate them all. If its first argument evaluates to `true` (as a `.pred`-type value), it will evaluate and return its second argument. Otherwise, it will evaluate and return its third.

This allows conditional expressions to be written.

```
(if (= 0 (floor n) 2)
    *black*
  *white*)
```

A.3.2 when

when, like if, tests a condition and a success case, but if the condition evaluates to a false value such as 0, nil is returned instead of a failure case. This construct is essentially a short-cut for when nil should be returned from a conditional.

```
(when (= 2 2) *up*)
```

A.4 Variables and Definitions

A.4.1 let

The let construct allows the programmer to set the result of one (or several) function calls to variables.

These definitions cannot refer to one another, as that would introduce a possibility for two mutually-recursive definitions to exist, which could not run on the GPU unless specified as separate functions. If one definition must refer to another, the let* construct must be used instead.

```
(let ((one 1)
      (two 2))
  (+ one two))
```

A.4.2 let*

This alternate form of let evaluates its arguments sequentially in order, with later arguments able to access the results from earlier definitions.

This form should only be used when one result depends on another—if they can all be evaluated separately, let should be used instead. It is equivalent to a series of nested calls, with one let per definition.

```
(let* ((one int 1)
       (two int 2)
       (result (+ one two)))
  result)
```

A.4.3 `defun`

Defines a function in a global scope.

This construct takes four arguments: firstly, the name of the function; secondly, a list of the names of the arguments, paired with their types; then the function's return type; and finally the expression to evaluate with these arguments.

```
(defun square ((num int)) int
  (* num num))
```

A.4.4 `defconst`

Defines a constant in a global scope.

Although constants can have any name that does not conflict with another, it's idiomatic to surround constant names within `*asterisks*`.

```
(defconst *pi* float 3.14159265)
```

A.4.5 `defstruct`

Defines a data structure with the given name and fields.

```
(defstruct vec
  (x float)
  (y float)
  (z float))
```

Knowledge of data structures is important to PolyCube, as it must be able to determine the size of every function argument and return type so it fits in the `.param` CUDA memory space.

A.4.6 `defunion`

Defines a *tagged union* type encompassing several structs. This allows any value of one of the types to be passed in as an argument to a function, with the function able to figure out which type it is.

It takes the name of the type as its first argument, and a list of structure names as the rest as union variants:

```
(defunion shape sphere plane)
```

At compile time, this defines the following:

1. A type (such as `shape` above) that allows a value of *any* of the types listed to be passed in as an argument.

   This type must be known at compile time, in order to give it a size: it will be the largest variant's size, plus one byte. This "tag" byte, which occurs after the struct's data, can be queried at runtime to determine which type a value actually has.

2. A predicate function that allows a function to test a value of this type to see which of the variants it is.

   It's idiomatic to end predicate names in Lisp with `-p`, so in the example above, the functions `sphere-p` and `plane-p` would be defined.

Compared to traditional data systems or class hierarchies, this is limited, but it provides the base level of support necessary to write simple programs. For instance, it is not possible for one type to be a member of two tagged unions at once, as the type predicate names would then clash.

A.4.7 `lambda`

Constructs an anonymous "lambda" function.

This function takes a number of input argument names as its first parameter, and when executed, evaluates the expression in its second parameter with those arguments.

These are limited in their use, in that they cannot be returned from functions nor passed around; they can only be passed in as an argument to a list-transforming function such as `filter` or `min`. It is not currently possible for lambda functions to be used on scalar values.

```
(lambda (x) (* x 2))
```

A.4.8 `array`

Creates an array that contains the given elements. The size of the array is fixed at compile-time, and arrays are not mutable after construction. The elements of the array must all be of the same type, which is determined by the type of its first argument.

```
(map (lambda (x) (+ x 2))
     (array 1 2 3 4 5))
```

A.4.9 `loop`

Runs a loop, in parallel on the GPU if possible, over the given half-open range of numbers. The two limits must be given in order.

```
(loop x (0 10)
  (print x))
```

SOURCE CODE

B

This is the source code, in PolyLisp, of the ray tracer presented and demonstrated in Chapter 6.

B.1 3D vectors

```
(defstruct vec
  (x float)
  (y float)
  (z float))
```

A vector is a point or direction in 3D space. It is represented by three floating point values.

```
(defun dot ((this vec) (that vec)) float
  (+ (* (vec:x this) (vec:x that))
     (* (vec:y this) (vec:y that))
     (* (vec:z this) (vec:z that))))
```

The dot product of two vectors is a mathematical formula that gets used to calculate their magnitude.

```
(defun cross-product ((this vec) (that vec)) vec
  (vec (- (* (vec:y this) (vec:z that))
          (* (vec:z this) (vec:y that)))
       (- (* (vec:z this) (vec:x that))
          (* (vec:x this) (vec:z that)))
       (- (* (vec:x this) (vec:y that))
          (* (vec:y this) (vec:x that)))))
```

The cross product of two vectors is another formula. This one is used to calculate a vector perpendicular to both. It gets used in the camera-calibration functions.

```
(defun vec:+ ((this vec) (that vec)) vec
  (vec (+ (vec:x this) (vec:x that))
       (+ (vec:y this) (vec:y that))
       (+ (vec:z this) (vec:z that))))

(defun vec:- ((this vec) (that vec)) vec
  (vec (- (vec:x this) (vec:x that))
       (- (vec:y this) (vec:y that))
       (- (vec:z this) (vec:z that))))
```

Vector addition simply adds the X, Y, and Z positions to those specified by another vector.

Subtraction is also defined, as a shortcut for doing vector arithmetic.

```
(defun vec:* ((this vec) (that vec)) vec
  (vec (* (vec:x this) amount)
       (* (vec:y this) amount)
       (* (vec:z this) amount)))

(defun vec:/ ((this vec) (that vec)) vec
  (vec (/ (vec:x this) amount)
       (/ (vec:y this) amount)
       (/ (vec:z this) amount)))
```

Similarly, multiplication on a vector is defined by multiplying (or dividing, again as a short-cut) each of its elements by a fixed amount.

This is different from matrix or tensor multiplication, as the resulting vector will have exactly as many values as before, and each field is multiplied by the same amount.

```
(defun vec:squared-magnitude ((this vec)) float
  (dot-product this this))

(defun vec:magnitude ((this vec)) float
  (sqrt (vec:squared-magnitude this)))
```

Get the length (magnitude) of the vector, and that value's square. This can be done by computing the dot product of the vector against itself.

The squared magnitude is used as well as the magnitude, and it is more efficient to provide a `vec:squared-magnitude` function than to have to square a value that has already been `sqrt`ed.

```
(defun vec:normalise ((this vec) (that vec)) vec
  (vec:/ this (vec:magnitude this)))
```

Finally, *normalising* the vector contracts or expands it so that its magnitude is exactly 1.

This is necessary because certain intersection and ray-casting routines expect to be dealing with a vector with magnitude 1.

B.2 Rays

```
(defstruct ray
  (start vec)
  (direction vec))
```

A ray is a combination of two vectors: one representing its starting position, the other its direction.

```
(defun ray:extend ((ray ray) (amount float)) vec
  (vec:+ (ray:start ray)
         (vec:* (ray:direction ray) amount)))
```

Extending a ray means finding the position the ray will be at, after it has travelled the given distance.

```
(defun ray:reflect ((ray ray) (pos vec) (normal vec)) ray
  (let* ((old-direction vec (ray:direction ray))
         (new-direction vec
          (vec:- old-direction
                 (* normal 2
                    (dot-product normal old-direction) 2)))
         (temporary-ray ray (ray pos new-direction)))
    (ray (ray:extend temporary-ray 0.01)
         (vec:normalise new-direction))))
```

Reflecting a ray involves calculating the direction of another ray, based on the position at which it hits another object, and the normal direction of the surface at that particular point.

The ray is extended by 0.01 before it is returned. This is to move the ray slightly away from the object that it hits before any more intersections are calculated—it avoids the situation where a ray repeatedly intersects with the same object in the same position.

B.3 Shapes

```
(defstruct plane
  (normal vec)
  (offset float)
  (texture texture))
```

A plane is defined by its normal direction, and how far it has gone in that direction.

```
(defun plane:intersect ((plane plane) (ray ray)) float
  (let* ((normal    vec   (plane:normal plane))
         (distance  float
          (dot-product normal (ray:direction ray))))
    (when (> 0 denom)
      (vec:/ (+ (dot-product normal (ray:start ray))
                (plane:offset plane))
             (- distance)))))
```

Intersect a plane with a ray, returning the distance the ray would have to travel before intersecting, or `nil` upon failure.

```
(defstruct sphere
  (radius float)
  (centre vec)
  (texture texture))
```

A sphere is defined by the position of its centre, and its radius.

```
(defun square ((num float)) float
  (* num num))
```

Helper function to square a floating-point value.

```
(defun sphere:intersect ((sphere sphere) (ray ray)) float
  (let* ((adjusted-centre vec
          (vec:- (sphere:centre centre) (ray:start ray)))
         (scalar float
          (dot-product adjusted-centre (ray:direction ray)))
         (distance float
          (+ (square (sphere:radius))
                     (square scalar)
                     (- (vec:squared-magnitude adjusted-centre)))))
    (when (> 0 distance)
      (- scalar (sqrt point)))))
```

Intersect a sphere with a ray, returning the distance in the same way as the plane.

```
(defun sphere:normal ((sphere sphere) (point vec)) vec
  (vec:normalise (vec:- point (sphere:centre sphere))))
```

Calculate the normal direction of a sphere at the given point.

```
(defunion shape plane sphere)
```

As there are two kinds of shape that must be tested, they must be placed within a tagged union. This allows individual planes and spheres to be used as parameters to functions that accept generic shapes.

```
(defun shape:normal ((shape shape) (point vec)) vec
  (if (plane-p shape)
      (plane:normal shape)
    (sphere:normal shape point)))
```

The first of these functions calculates the normal direction for any shape at the given point.

```
(defun shape:texture ((shape shape)) colour
  (if (plane-p shape)
      (plane:texture shape)
    (sphere:texture shape)))
```

The second returns the colour of the shape's texture, again at the given point.

The point must be specified here because of the checkerboard texture, which returns one of two different colours depending on where it is being drawn.

```
(defstruct intersection
  (distance float)
  (shape shape)
  (ray ray))
```

An intersection holds all the values necessary to calculate the colour at this particular point: the ray that hit the shape, the shape that was hit, and the distance the ray that had to travel.

```
(defun shape:intersect ((sh shape) (r ray)) intersection
  (let ((dist float (if (plane-p sh)
                        (plane:intersect sh r)
                      (sphere:intersect sh r))))
    (intersection dist sh r)))
```

Intersect a ray with one shape, producing an `intersection` object, or `nil` if the ray missed.

```
(defun intersection:position ((isect intersection)) vec
  (ray:extend (intersection:ray      isect)
              (intersection:distance isect)))
```

Calculate the position of the intersection by extending the ray by its distance.

```
(defun intersection:colour ((isect intersection)) colour
  (texture:colour (shape:texture (intersection:shape isect))
                  (intersection:position isect)))
```

Get the colour of the object at the intersection point based on the intersection position.

```
(defun intersection:reflection-ray ((is intersection)) ray
  (let ((pos vec (intersection:position is)))
    (ray:reflect (intersection:ray is)
                 pos
                 (shape:normal shape pos))))
```

Finally get the reflection ray off the object at the intersection point.

B.4 The Camera

```
(defstruct camera
    (position vec)
    (forward vec)
    (up vec)
    (right vec))
```

The camera is defined by how far it can see forward, up, and right, in three dimensions.

```
(defconst *camera-fov* float 1.5)
```

It has a defined field of view. A larger field of view will reveal more of the scene.

```
(defun camera:looking-at ((camera-position vec)
                          (observed-position vec)) camera
  (let* ((forward vec
          (vec:normalise (vec:- observed-position
                                camera-position)))
         (down vec (vec 0 1 0))

         (right vec
          (vec:* *camera-fov* (vec:normalise
                               (cross-product forward
                                              down))))
         (up vec
          (vec:* *camera-fov* (vec:normalise
                               (cross-product forward
                                              right)))))
    (camera camera-position forward up right)))
```

The more useful way to construct a camera is to supply it with not only its position, but the position of the object that it should be looking at.

B.5 Textures

```
(defstruct colour
  (r int)
  (g int)
  (b int))
```

A colour is simply the set of three values corresponding to red, green, and blue.

```
(defun colour:blend ((this colour) (that colour)) colour
  (colour (+ (/ (colour:r this) 2) (/ (colour:r that) 2))
          (+ (/ (colour:g this) 2) (/ (colour:g that) 2))
          (+ (/ (colour:b this) 2) (/ (colour:b that) 2))))
```

Blend one colour into another colour by averaging the values in each of their channels.

This is used when blending the colour of a shape at an intersection point with the colour produced by its reflecting ray.

```
(defconst *red*   colour (colour 255  65  54))
(defconst *green* colour (colour  46 204  64))
(defconst *blue*  colour (colour   0 116 201))

(defconst *black* colour (colour  17  17  17))
(defconst *grey*  colour (colour  51  51  51))
(defconst *white* colour (colour 255 255 255))
```

Some sample colours that are used to construct the scene.

```
(defstruct checkerboard
  (odds colour)
  (evens colour))
```

A checkerboard pattern can be generated by alternating tiles of two colours.

```
(defun even-p ((n float)) pred
  (= 0 (% (floor n) 2)))
```

Helper function that returns `true` or `false` depending on whether the number is in an even-numbered row or column, which is used when calculating the checkerboard tile's colour.

```
(defunion texture colour checkerboard)
```

As with shapes, colours and checkerboards need to be placed in a union to be passed to functions.

```
(defun texture:colour ((texture tex) (point vec)) colour
  (if (colour-p tex)
      tex
    (if (== (even-p (vec:x point))
            (even-p (vec:z point)))
        (checkerboard:odds tex)
      (checkerboard:evens tex))))
```

Get the colour of a texture at the given point.

For flat colours, the texture will be the same all over, so no further processing is required. However, for the checkerboard pattern, the parity of the floor of the row and column numbers of the checkerboard need to be compared—equal and inequal results will form a diagonal pattern of tiles.

B.6 The Scene

```
(defstruct scene
  (camera camera)
  (shapes (array shape)))
```

A scene has only one camera, but an array of shapes.

The sample scene setup, used to render Figure 6.2.

```
(defconst *max-depth* 96)

(defconst *up* (vec 0 -1 0))

(defconst *scene* (scene
  (camera:looking-at (vec 20 4 1)
                     (vec 0.5 0.7 0))
  (array
    (plane *up* 0 (checkerboard *black* *white*)

    (sphere 4 (vec -10 4.2 3) *blue*)

    (sphere 2 (vec 0 2 -5) *white*)
    (sphere 2 (vec 0 2  0) *white*)
    (sphere 2 (vec 0 2  5) *white*)

    (sphere 1.3 (vec 3 1.3 -3) *red*)
    (sphere 1.3 (vec 3 1.3  3) *red*)

    (sphere 0.75 (vec 5 0.75 -1) *green*)
    (sphere 0.75 (vec 5 0.75  1) *green*)

    (sphere 0.33 (vec 6 0.33 0) *blue*))))
```

Trace a ray through a scene, calculating the nearest intersection point to the ray's start position if the ray hits any shape.

This uses the `minimum` vector function to loop through many intersections and return the one that has the lowest distance value out of any, if present. The distance field holds how far the ray has had to travel when it intersects with an object.

If there are no intersections, `nil` is returned.

```
(defun trace-ray ((scene scene) (ray ray)) intersection
  (minimum (lambda ((a intersection) (b intersection))
                   (< (intersection:distance a)
                      (intersection:distance b)))
    (map (lambda ((shape shape))
                 (shape:intersect ray))
         (shapes:scene scene))))
```

```
(defun ray-colour ((scene scene)
                   (ray ray) (level int)) colour
  (let ((isect intersection (trace-ray scene ray)))
    (if isect
       (if (< level *max-depth*)
          (colour:blend (ray-colour scene ray)
                        (intersection:reflection-ray isect)
                        (+ colour 1))
        (ray-colour scene ray))
    *black*)))
```

Get the eventual colour of a ray, depending on which object it hit.

This function is *recursive*, in that if it hits an object, it fires another ray and blends the result of that ray with the original object's colour. It uses the PolyCube runtime system to run recursively on the GPU. Any recursively-fired rays have a depth number one higher than the one this function was called with, in order to prevent rays from bouncing around forever.

Unlike `trace-ray` above, this function cannot return nil, as a ray has to have an endpoint: if it collides with no objects, then the colour of the background of the scene—which, in this case, is black—is returned instead.

```
(defun centre-x ((x int)) float
  (/ (- x (/ *width* 2))
     (* 2 *width*)))

(defun centre-y ((y int)) float
  (/ (- (/ *height* 2) y)
     (* 2 *height*)))
```

Centers a given X position or Y position into a floating-point number in the range -1, for the very left (or top) of the image, to 1, the very right (or bottom) of the image.

```
(defun ray-for-pixel ((scene scene) (x int) (y int)) ray
  (let ((cam (scene:camera scene)))
    (ray (camera:position cam)
         (vec:normalise
          (vec:+ (camera:forward cam)
                 (vec:+ (vec:* (camera:right cam) x)
                        (vec:* (camera:up cam) y)))))))
```

Takes the position of a pixel in the image that should be rendered, as X and Y co-ordinates, and calculates the ray that should be fired to get that pixel's colour.

These rays all begin at the same point—the position of the camera. Their directions are the values that depend on the pixel.

```
(defun render-scene ((scene scene)) image
  (let ((image-buffer (image *width* *height*)))
    (loop i (0 *width*)
      (loop j (0 *height*)
        (let* ((ray ray (ray-for-pixel scene i j))
               (colour (ray-colour ray 0 *max-depth*)))
          (write-pixel image i j (colour:r colour)
                                 (colour:g colour)
                                 (colour:b colour)))))))
```

Renders a scene into an `image`, a built-in primitive, using `write-pixel` to write out the colour values to the buffer.

This is the part that can be easily parallelised, using two nested `loop` constructs to iterate over each pixel's position.

BIBLIOGRAPHY

Allgyer, Michael (2007). 'Real-Time Raytracing using CUDA'. MA thesis (cit. on p. 65).

Amdahl, Gene M. (1967). 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities'. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560 (cit. on p. 24).

Arvind and David E. Culler (1986). 'Annual Review of Computer Science Vol. 1, 1986'. In: ed. by Joseph F. Traub et al. Palo Alto, CA, USA: Annual Reviews Inc. Chap. Dataflow Architectures, pp. 225–253. ISBN: 0-8243-3201-6. URL: http://dl.acm.org/citation.cfm?id=17814.17824 (cit. on p. 43).

Arvind and Robert A. Iannucci (1988). 'Two Fundamental Issues in Multiprocessing'. In: *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*. Bonn, Germany: Springer-Verlag New York, Inc., pp. 61–88. ISBN: 0-387-18923-8. URL: http://dl.acm.org/citation.cfm?id=52797.52802 (cit. on p. 11).

Arvind, Rishiyur S. Nikhil and Keshav K. Pingali (1989). 'I-structures: Data Structures for Parallel Computing'. In: *ACM Trans. Program. Lang. Syst.* 11.4, pp. 598–632. ISSN: 0164-0925. DOI: 10.1145/69558.69562. URL: http://doi.acm.org/10.1145/69558.69562 (cit. on p. 43).

Augonnet, C., S. Thibault and R. Namyst (2010). 'StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines'. In: (cit. on p. 42).

Ayguadé, Eduard et al. (2009). 'An Extension of the StarSs Programming Model for Platforms with Multiple GPUs'. In: *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Euro-Par '09. Delft, The Netherlands: Springer-Verlag, pp. 851–862. ISBN: 978-3-642-03868-6. DOI: 10.1007/978-3-642-03869-3_79. URL: http://dx.doi.org/10.1007/978-3-642-03869-3_79 (cit. on p. 39).

Badia, RosaM. et al. (2003). 'Programming Grid Applications with GRID Superscalar'. English. In: *Journal of Grid Computing* 1.2, pp. 151–170. ISSN: 1570-7873. DOI: 10.1023/B:GRID.0000024072.93701.f3. URL: http://dx.doi.org/10.1023/B:GRID.0000024072.93701.f3 (cit. on p. 39).

Bakkum, Peter and Kevin Skadron (2010). 'Accelerating SQL Database Operations on a GPU with CUDA'. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. GPGPU-3. Pittsburgh, Pennsylvania, USA: ACM, pp. 94–103. ISBN: 978-1-60558-935-0. DOI: 10.1145/1735688.1735706. URL: http://doi.acm.org/10.1145/1735688.1735706 (cit. on p. 20).

Barcelona Supercomputing Center (2007). *SMP Superscalar (SMPSs) User's Manual Version 1.0* (cit. on p. 39).

Benthin, C. et al. (2012). 'Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture'. In: *Visualization and Computer Graphics, IEEE Transactions on* 18.9, pp. 1438–1448. ISSN: 1077-2626. DOI: 10.1109/TVCG.2011.277 (cit. on p. 64).

Bic, Lubomir (1990). 'A Process-oriented Model for Efficient Execution of Dataflow Programs'. In: *J. Parallel Distrib. Comput.* 8.1, pp. 42–51. ISSN: 0743-7315. DOI: 10.1016/0743-7315(90)90067-Y. URL: http://dx.doi.org/10.1016/0743-7315(90)90067-Y (cit. on p. 47).

BOINC Combined Credit Overview (2011). URL: http://boincstats.com/en/stats/-1/project/detail (visited on 30/09/2013) (cit. on p. 3).

Buck, Joseph and Edward A. Lee (1992). *The Token Flow Model* (cit. on p. 45).

Carr, Nathan A., Jesse D. Hall and John C. Hart (2002). 'The Ray Engine'. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02. Saarbrucken, Germany: Eurographics Association, pp. 37–46. ISBN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569046.569052 (cit. on p. 65).

Carr, Nathan A., Jared Hoberock et al. (2006). 'Fast GPU Ray Tracing of Dynamic Meshes Using Geometry Images'. In: *Proceedings of Graphics Interface 2006*. GI '06. Quebec, Canada: Canadian Information Processing Society, pp. 203–209. ISBN: 1-56881-308-2. URL: http://dl.acm.org/citation.cfm?id=1143079.1143113 (cit. on pp. 63, 65).

Chalmers, Alan, Timothy Davis and Erik Reinhard, eds. (2002). *Practical Parallel Rendering*. Natick, MA, USA: A. K. Peters, Ltd. ISBN: 1-56881-179-9 (cit. on p. 62).

Culler, David E (1989). 'Managing parallelism and resources in scientific dataflow programs'. PhD thesis. Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science (cit. on p. 48).

D. Kuck D. Lawrie, E. Davidgot and A Sameh (1986). 'Parallel Supercomputing Today and the Cedar Approach'. In: *Science magazine* 231, pp. 967–974 (cit. on p. 48).

Dally, W.J. et al. (2003). 'Merrimac: Supercomputing with Streams'. In: *Supercomputing, 2003 ACM/IEEE Conference*, pp. 35–35. DOI: 10.1109/SC.2003.10043 (cit. on p. 45).

Davis, A.L. and R.M. Keller (1982). 'Data Flow Program Graphs'. In: *Computer* 15.2, pp. 26–41. ISSN: 0018-9162. DOI: 10.1109/MC.1982.1653939 (cit. on p. 43).

Dennis, J. B. (1974). 'First Version of a Data Flow Procedure Language'. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, pp. 362–376. ISBN: 3-540-06859-7. URL: http://dl.acm.org/citation.cfm?id=647323.721501 (cit. on pp. 43, 44).

Dennis, Jack B. and David P. Misunas (1975). 'A preliminary architecture for a basic data-flow processor'. In: *IN PROCEEDINGS OF THE 2ND ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pp. 126–132 (cit. on p. 43).

Diamos, Gregory F. and Sudhakar Yalamanchili (2008). 'Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems'. In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. HPDC '08. Boston, MA, USA: ACM, pp. 197–200. ISBN: 978-1-59593-997-5. DOI: 10.1145/1383422.1383447. URL: http://doi.acm.org/10.1145/1383422.1383447 (cit. on p. 38).

Dongarra, Jack and Thomas H. Dunigan (1997). 'Message-Passing Performance of Various Computers'. In: *Concurrency - Practice and Experience* 9.10, pp. 915–926. DOI: 10.1002/(SICI)1096-9128(199710)9:10<915::AID-CPE277>3.0.CO;2-C (cit. on p. 25).

Fatahalian, Kayvon and Mike Houston (2008). 'A Closer Look at GPUs'. In: *Commun. ACM* 51.10, pp. 50–57. ISSN: 0001-0782. DOI: 10.1145/1400181.1400197. URL: http://doi.acm.org/10.1145/1400181.1400197 (cit. on p. 16).

Fernando, Randima (2004). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education. ISBN: 0321228324 (cit. on p. 62).

Garanzha, Kirill and Charles Loop (2010). 'Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing'. In: *Computer Graphics Forum*. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01598.x (cit. on p. 64).

Goodman, Daniel and Mikel Lujan (2011). 'Scientific GPU Programming with Data-Flow Languages'. In: *Multi-Core and Reconfigurable Super Computing Conference* (cit. on pp. 43, 47).

Gordon, Michael I. et al. (2002). 'A Stream Compiler for Communication-exposed Architectures'. In: *SIGOPS Oper. Syst. Rev.* 36.5, pp. 291–303. ISSN: 0163-5980. DOI: 10.1145/635508.605428. URL: http://doi.acm.org/10.1145/635508.605428 (cit. on p. 45).

Gustafson, John L. (1988). 'Reevaluating Amdahl's Law'. In: *Commun. ACM* 31.5, pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415 (cit. on p. 24).

Harris, Mark (2007). 'GPU Physics'. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: ACM. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281656. URL: http://doi.acm.org/10.1145/1281500.1281656 (cit. on p. 7).

Hennessy, John L. and David A. Patterson (2003). *Computer Architecture: A Quantitative Approach*. 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558607242 (cit. on p. 12).

Iannucci, R. A. (1988). 'Toward a Dataflow/Von Neumann Hybrid Architecture'. In: *SIGARCH Comput. Archit. News* 16.2, pp. 131–140. ISSN: 0163-5964. DOI: 10.1145/633625.52416. URL: http://doi.acm.org/10.1145/633625.52416 (cit. on pp. 10, 48, 49).

Jones, Simon Peyton et al. (2008). 'Harnessing the Multicores: Nested Data Parallelism in Haskell'. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Ed. by Ramesh Hariharan, Madhavan Mukund and V Vinay. Vol. 2. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 383–414. ISBN: 978-3-939897-08-8. DOI: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2008.1769. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1769 (cit. on p. 32).

Kale, LaxmikantV, DavidM Kunzman and Lukasz Wesolowski (2010). In: Chapman & Hall/CRC Computational Science. CRC Press. Chap. Accelerator Support in the Charm++ Parallel Programming Model, pp. 393–411. ISBN: 978-1-4398-2536-5. DOI: 10.1201/b10376-28. URL: http://dx.doi.org/10.1201/b10376-28 (cit. on p. 38).

Karlsson, Filip and Carl Johan Ljungstedt (2004). 'Ray Tracing on Programmable Graphics Hardware'. MA thesis (cit. on p. 65).

Kirk, David B. and Wen-mei W. Hwu (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123814723, 9780123814722 (cit. on p. 26).

Kunzman, David et al. (2006). 'Charm++, Offload API, and the Cell Processor'. In: *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*. Seattle, WA, USA (cit. on p. 38).

Lee, Won-Jong et al. (2013). 'SGRT: A Mobile GPU Architecture for Real-time Ray Tracing'. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, pp. 109–119. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492057. URL: http://doi.acm.org/10.1145/2492045.2492057 (cit. on p. 66).

Leopold, Claudia (2001). *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. New York, NY, USA: John Wiley, Inc. ISBN: 0471358312 3 (cit. on p. 3).

Lewis, Ted G. and Hesham El-Rewini (1992). *Introduction to Parallel Computing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-498924-4 (cit. on p. 24).

Li, Yinan, Jack Dongarra and Stanimire Tomov (2009). 'A Note on Auto-tuning GEMM for GPUs'. In: *Proceedings of the 9th International Conference on Computational Science: Part I*. ICCS '09. Baton Rouge, LA: Springer-Verlag, pp. 884–892. ISBN: 978-3-642-01969-2. DOI: 10.1007/978-3-642-01970-8_89. URL: http://dx.doi.org/10.1007/978-3-642-01970-8_89 (cit. on p. 41).

Luk, Chi-Keung, Sunpyo Hong and Hyesoon Kim (2009). 'Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping'. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, pp. 45–55. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669121. URL: http://doi.acm.org/10.1145/1669112.1669121 (cit. on p. 37).

Mattson, Timothy, Beverly Sanders and Berna Massingill (2004). *Patterns for Parallel Programming*. First. Addison-Wesley Professional. ISBN: 0321228111 (cit. on p. 25).

NVidia (2010). *NVidia CUDA C Programming Guide* (cit. on pp. 14, 26).

— (2012). *Parallel Thread Execution ISA Version 4.2* (cit. on pp. 28, 56).

— (2013). *CUDA Toolkit Documentation* (cit. on p. 54).

— (2014). 'Dynamic Parallelism in CUDA'. In: (cit. on p. 27).

Pacheco, Peter (1996). *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-339-5 1 (cit. on pp. 1, 36).

Parker, Steven G. et al. (2013). 'GPU Ray Tracing'. In: *Commun. ACM* 56.5, pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/2447976.2447997. URL: http://doi.acm.org/10.1145/2447976.2447997 (cit. on p. 66).

Parker, Steven et al. (1999). 'Interactive Ray Tracing'. In: *Proceedings of the 1999 Symposium on Interactive 3D Graphics*. I3D '99. Atlanta, Georgia, USA: ACM, pp. 119–126. ISBN: 1-58113-082-1. DOI: 10.1145/300523.300537. URL: http://doi.acm.org/10.1145/300523.300537 (cit. on p. 66).

Pharr, Matt and Randima Fernando (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional. ISBN: 0321335597 (cit. on pp. 16, 60).

Pratt-Szeliga, Philip C., James W. Fawcett and Roy D. Welch (2012). 'Rootbeer: Seamlessly Using GPUs from Java'. In: *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. HPCC '12. Washington, DC, USA: IEEE Computer Society, pp. 375–380. ISBN: 978-0-7695-4749-7. DOI: 10.1109/HPCC.2012.57. URL: http://dx.doi.org/10.1109/HPCC.2012.57 (cit. on p. 40).

Purcell, Timothy John (2004). 'Ray Tracing on a Stream Processor'. AAI3128683. PhD thesis. Stanford, CA, USA (cit. on p. 65).

Purcell, Timothy J. et al. (2002). 'Ray Tracing on Programmable Graphics Hardware'. In: pp. 703–712 (cit. on pp. 63, 64).

Sanders, Jason and Edward Kandrot (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional. ISBN: 0131387685, 9780131387683 (cit. on p. 25).

Schatz, Michael C. and Cole Trapnell (2008). *Fast Exact String Matching on the GPU* (cit. on pp. 14, 19).

Schauser, Klaus Erik (1991). *Compiling Dataflow into Threads: Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages*. Tech. rep. UCB/CSD-91-644. EECS Department, University of California, Berkeley. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1991/5470.html (cit. on p. 44).

Stratton, John A., Sam S. Stone and Wen-mei W. Hwu (2008). 'MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs'. English. In: *Languages and Compilers for Parallel Computing*. Ed. by JoséNelson Amaral. Vol. 5335. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 16–30. ISBN: 978-3-540-89739-2. DOI: 10.1007/978-3-540-89740-8_2. URL: http://dx.doi.org/10.1007/978-3-540-89740-8_2 (cit. on pp. 26, 27).

Taylor, Michael Bedford et al. (2002). 'The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs'. In: *IEEE Micro* 22.2, pp. 25–35. ISSN: 0272-1732. DOI: 10.1109/MM.2002.997877. URL: http://dx.doi.org/10.1109/MM.2002.997877 (cit. on p. 45).

Teodoro, G. et al. (2009). 'Coordinating the use of GPU and CPU for improving performance of compute intensive applications'. In: *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289193 (cit. on p. 39).

Treleaven, P.C. and I.G. Lima (1984). 'Future Computers: Logic, Data Flow, ..., Control Flow?' In: *Computer* 17.3, pp. 47–58. ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659081 (cit. on p. 43).

Treleaven, Philip C., David R. Brownbridge and Richard P. Hopkins (1982). 'Data-Driven and Demand-Driven Computer Architecture'. In: *ACM Comput. Surv.* 14.1, pp. 93–143. ISSN: 0360-0300. DOI: 10.1145/356869.356873. URL: http://doi.acm.org/10.1145/356869.356873 (cit. on p. 43).

Veen, Arthur H. (1986). 'Dataflow Machine Architecture'. In: *ACM Comput. Surv.* 18.4, pp. 365–396. ISSN: 0360-0300. DOI: 10.1145/27633.28055. URL: http://doi.acm.org/10.1145/27633.28055 (cit. on p. 43).

Waingold, E. et al. (1997). 'Baring it all to software: Raw machines'. In: *Computer* 30.9, pp. 86–93. ISSN: 0018-9162. DOI: 10.1109/2.612254 (cit. on p. 45).

Wald, Ingo (2004). 'Realtime Ray Tracing and Interactive Global Illumination'. PhD thesis (cit. on pp. 64, 66).

Wesolowski, Lukasz (2008). 'An Application Programming Interface for General Purpose Graphics Processing Units in an Asynchronous Runtime System'. MA thesis (cit. on p. 38).

Whaley, R. Clint, Antoine Petitet and Jack J. Dongarra (2000). 'Automated Empirical Optimization of Software and the ATLAS Project'. In: *PARALLEL COMPUTING* 27, p. 2001 (cit. on p. 41).

Williams, S. et al. (2008). 'Lattice Boltzmann simulation optimization on leading multicore platforms'. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–14. DOI: 10.1109/IPDPS.2008.4536295 (cit. on p. 41).

Wilson, G. and W. Banzhaf (2008). 'Linear genetic programming GPGPU on Microsoft's Xbox 360'. In: *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 378–385. DOI: 10.1109/CEC.2008.4630825 (cit. on p. 14).

Yan, Yonghong, Max Grossman and Vivek Sarkar (2009). 'JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA'. English. In: *Euro-Par 2009 Parallel Processing*. Ed. by Henk Sips, Dick Epema and Hai-Xiang Lin. Vol. 5704. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 887–899. ISBN: 978-3-642-03868-6. DOI: 10.1007/978-3-642-03869-3_82. URL: http://dx.doi.org/10.1007/978-3-642-03869-3_82 (cit. on p. 40).

Yu, Zhibin, Andrea Righi and Roberto Giorgi (2011). 'A Case Study on the Design Trade-off of a Thread Level Data Flow based Many-core Architecture'. In: *Future Computing*, pp. 100–106 (cit. on p. 40).