Domain-Specific Languages for Massively Parallel Processors

Luke Cartey

Magdalen College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2013

Abstract

Massively Parallel Processors provide significantly higher peak performance figures than other forms of general purpose processors. However, this comes at a cost to the developer, who needs to deal with an increasingly complicated piece of hardware, for which applications need to be tweaked and optimised to achieve high performance.

Domain-specific languages have been proposed as a potential solution to this complexity problem: generating GPU applications from high-level, declarative specifications. This thesis explores two related ideas: firstly, is it practical to synthesise DSLs from high-level languages, and secondly, how can we simplify the creation of such DSLs?

This thesis proposes a novel approach whereby rather than considering single domains, we consider collections of *collaborative* domains in order to share common features and thus reduce the cost of development. We achieve this using a DSLs-within-a-DSL approach: a custom designed host language, into which extensions may be embedded.

In order to ground our approach in a real case-study, we propose, design and develop a DSLs-within-a-DSL framework for *bioinformatics*. We use a restricted recursive functional language as the host language, and embed new DSLs into this language.

Importantly, we describe how we can use a combination of novel and adopted automatic parallelisation techniques to synthesise a massively-parallel program for a GPU. This automatic parallelisation, achieved through the discovery of a schedule, and program synthesis techniques using the polyhedral model, interacts productively with our embedded extensions.

To further simplify development, we provide a series of customisable heuristics for defining GPU parameters such as the block size (number of threads), grid size and location in the memory hierarchy of data-structures. This encapsulates GPU expertise within the compiler itself.

We finally demonstrate that the total combination of these techniques results in applications with competitive performance, at much lower development cost and greater flexibility than comparable hand-coded applications.

Acknowledgements

I would firstly like to thank my supervisor, Oege de Moor. I am grateful for his unerring support, keen insight, and enthusiasm throughout my dissertation. I hope we get many more opportunities to work together.

I would also like to thank my co-supervisor, Jotun Hein, and the members of his bioinformatics group at Oxford. Developing a domain-specific language requires an extensive understanding of the domain itself, and I have felt like a very welcome member of the group. I would also like to reserve a special mention for Rune Lyngsoe, whose cheery advice, extensive knowledge and excellent sense of humour made our weekly discussions both a vital and enjoyable part of my D.Phil. Many thanks also go to Alex Wood, Balazs Koszegi and Dmitrii Bychov for their hard work during the Bioinformatics Summer School in 2012.

My thanks also go to my examiners Paul Kelly and Mike Giles. Your careful and thorough reading of my work, along with your considered advice and the benefit of your experience and expertise have improved this dissertation significantly. Thanks as well go to Gerton Lunter and Mike Giles for your helpful comments and discussion during both my transfer and confirmation of status, as well as the reviewers of the papers that I submitted during the course of my D.Phil.

Many thanks to anyone who proofread a paper or my dissertation for me, but particularly Michael Howe, who waded through every page of this dissertation in a relentless hunt for typos and errors. Much appreciated!

Finally, but by no means least, thank you to my parents, for their support, and to Shell, for putting up with me whilst writing up, and without whom the last four years would have been much less enjoyable.

Contents

List of Figures

Chapter 1

Introduction

Massively parallel processors represent the culmination of many decades worth of processor development: a very large number of cores on single chip, producing phenomenally high peak performance figures, at the cost of complexity in code development. Taming this complexity is vital if we are to use these chips to their full potential. In order to understand why this complexity has arisen, however, we must first consider its origin.

Moore's Law has encapsulated the increase in processor performance over the last half a century. Simply stated, it predicts that the number of transistors on a chip will double every two years. Whilst the observable progress of processing power has matched this "law", this last decade has seen processor manufacturers hitting a physical limit on both the power consumption and heat dissipation which prevents the doubling of clock-speeds that occurred in earlier hardware iterations (Figure 1.1).

In order to take advantage of the doubling of the transistors, chip manufacturers were forced to take a different approach. Instead of increasing serial performance, they used the extra transistors to provide multiple cores on the same chip, thus multiplying the fundamental performance – the operations per clock – at the cost of increased complexity for the developer[1]. Multi-core processors are now the standard, in everything from desktop and server chips, to mobile phones and tablets.

[1]This argument is summarised succinctly by Herb Sutter in *The free lunch is over* [110]

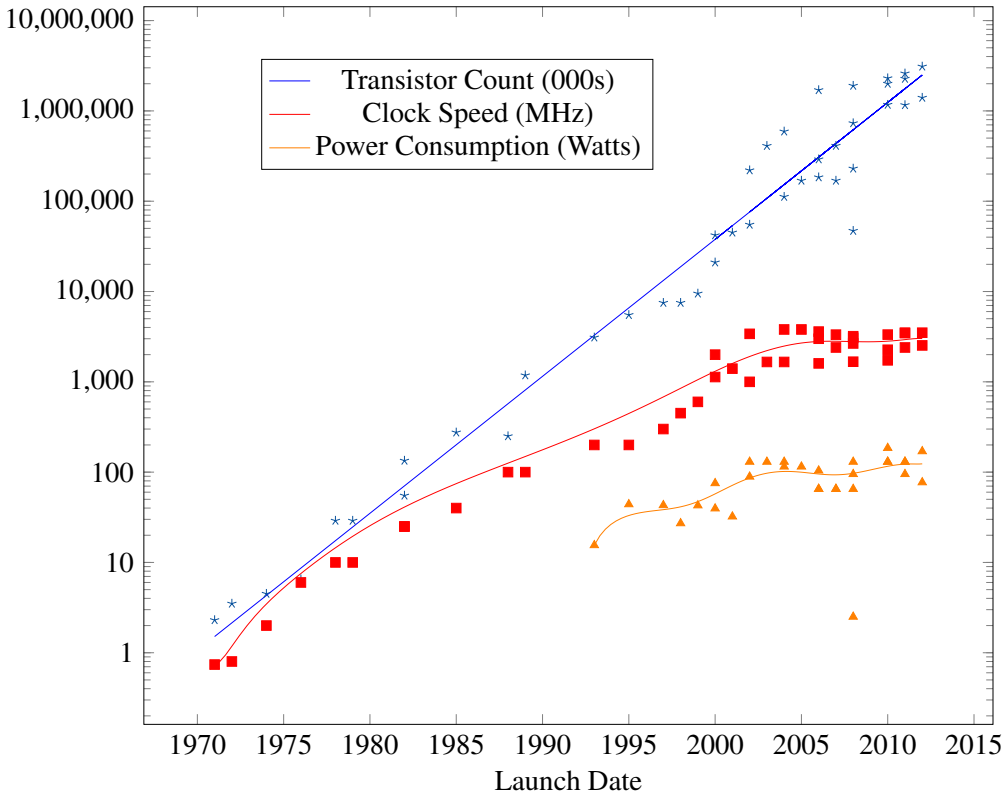


Figure 1.1: Intel CPU changes in transistors, clock speed and power over a period of 40 years (concept reproduced from [110]). Log scaling is used, and trend lines have been added, demonstrating the plateau in clock speed and power consumption.

Massively parallel processors take this logic to the extreme. Rather than increasing the clock-speed or the complexity of each core, massively parallel processors provide many thousands of comparatively simple cores. There is an effective trade-off of core complexity for core count. This trade-off results in better *peak* performance at the expense of *average* performance; consequently, applications must be designed appropriately in order to take advantage of the massively parallel design.

Graphics processors (GPUs) were one of the first devices to realise the mass-market potential of such hardware; computer graphics is a domain abundant with highly parallel workloads. The resulting hardware has a substantially higher peak performance (in floating-point operations per seconds, or "FLOPs") than a comparable CPU (Figure 1.2), but this can only be achieved because the workload is suitably independent.

The high peak performance has naturally piqued the interest of those in high-performance computing. Over the last ten years a growing body of evidence has appeared which suggests that this hardware can be profitably exploited in a wide array of domains to achieve higher performance-per-watt and performance-per-dollar than CPU equivalents. This can be surmised by the rapid domination of the top 500 supercomputers list by GPU-powered machines.

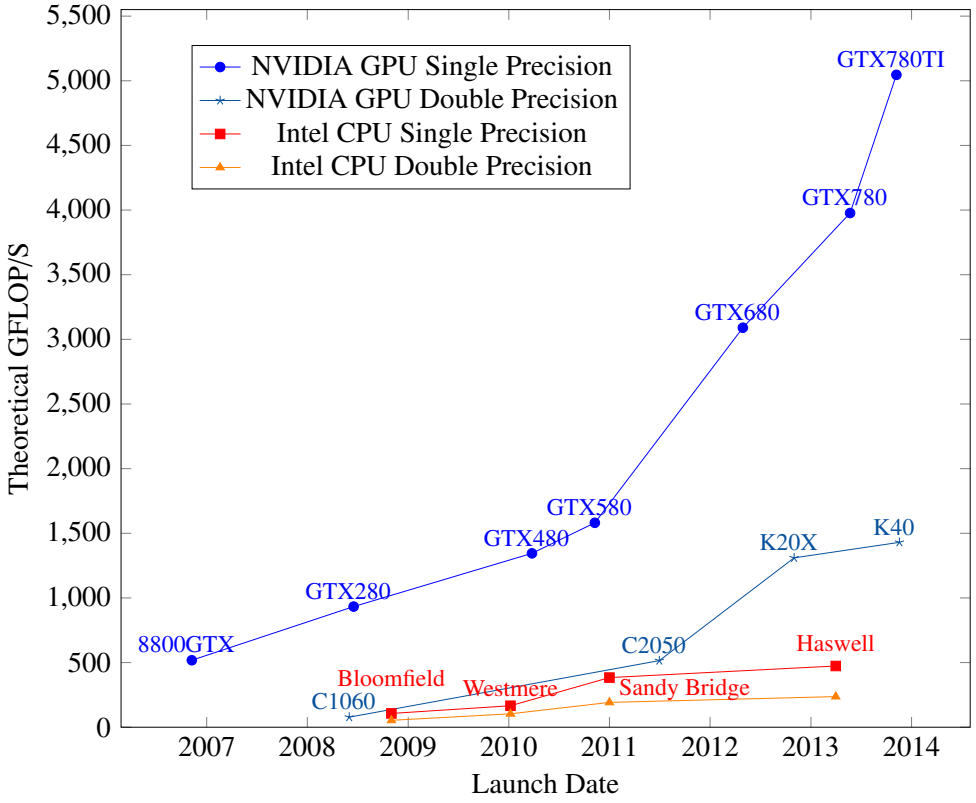


Figure 1.2: Theoretical peak floating point performance, GPUs compared to CPUs. NVIDIA figures are reproduced from [91]. Intel figures are computed from the published instructions-per-clock and the maximum clock speeds achieved by the architecture in a commercial chip.

However, a concern remains that the difficulty of programming such architectures can be a real roadblock to their widespread use. Massively parallel processors of this form are relatively new, and as such the techniques for programming them are still in their infancy. If multi-core desktop chips are notoriously difficult to program for, massively-parallel ar-

chitectures, with hundreds or thousands of cores, are worse. They often require completely different techniques and tools, even a different mindset.

This is particularly problematic for those working in the very domains that would benefit most. Increasingly complex development is at odds with the expertise of the scientific community, which is struggling to get to grips with existing architectures. Considering the complexities involved, it can be extremely difficult to expect domain experts to overcome the challenges involved in GPU computing; yet we wish to enable this technology for a much larger class of application areas. In order to proceed, experts are required in both the domain and in developing high-performance GPU applications.

Domain-specific languages (DSLs) have been proposed as a way to simplify development of GPU applications for domain experts. In this approach, either a single user, or a small group, write a mini-language for describing the problem, and a compiler or interpreter that takes this language and generates an efficient GPU implementation. While I have completed this dissertation, this approach has begun to gain mainstream support, most notably with the Delite framework [20], and the up-and-coming CARP [1] project.

Another approach, also gaining support during the last four years, has been the idea of automatic parallelization for GPU applications, in particular with the use of the polyhedral model and similar scheduling techniques. These techniques have been used to successfully generate code for earlier massively-parallel like architectures such as systolic arrays, and recent work has considered their application to GPUs [10, 6, 116].

This thesis straddles these two research areas: not only do we pursue the idea that GPUs can be profitably supported by developing DSLs, but we attempt to embed automatic parallelization techniques in order that domain-experts can build their own DSLs. This enables experts in scientific domains to focus on applications in their domain, and leave GPU development to GPU experts. Whilst related work has focused either on front-end development for DSLs (e.g Delite) or automatic parallelization of general purpose languages, we focus on combining these aspects to reduce the cost of developing GPU DSLs, enabling domain-

experts to build them with little or no GPU knowledge. In particular, a declarative approach to DSLs and an encoding of GPU expertise in the compiler are two important themes in this dissertation that support this overall goal.

To re-iterate, two key questions we seek to answer are:

- What techniques can we use in order to automatically parallelise DSL applications?
- Developing a DSL can be a time-consuming business, and require a wide array of skills (compiler, domain and GPU knowledge). Can we reduce this cost of development?

1.1 Our approach

The approach taken in this dissertation can be summarised in two ways: *reuse* and *separation of concerns*. *Reuse* – rather than considering *single* domains, we instead consider related collections of domains together. This is achieved by defining a common host language (itself a DSL) into which each of these extra domains can be embedded. By taking this broader approach to development, we can benefit from sharing common features and descriptions amongst the domains, reducing the cost of development by providing much of the scaffolding and common elements.

In order to demonstrate that the approach is both useful and practical, we consider what we call *classical bioinformatics*, a domain where many applications are described using mathematical recursions. By adopting a simple recursive functional language as the host DSL, we can focus parallelisation effort on this language, further reducing the cost of new development.

This illustrates the second focus of the dissertation – a *separation of concerns*. Specifically, we attempt to separate the concerns of the *domain developer* – likely a domain expert wanting to implement front-end support for a particular domain – from those of the *GPU expert*. Automatic parallelisation of the host language is one such way this occurs, but we

also provide techniques for insulating domain developers from the memory hierarchy and launch parameters of the GPU.

To be more precise, the key contributions of this dissertation are:

- We propose a hybrid approach to DSL development, where a custom designed host language is extended by embedded DSLs. This novel *DSLs-within-a-DSL* approach simplifies the creation of collaborative sets of DSLs by providing a unifying host language. For GPU development, we can focus on parallelising this host language, and that effort is amortised across all the embedded DSLs. For new embedded DSLs, very little new code is required.

- We implement the hybrid approach in practice, by creating a framework for DSL development in bioinformatics which significantly reduces the cost of developing efficient GPU DSLs. This framework supports embedded extensions to a simple, recursive functional host language, suitable for describing a wide range of bioinformatics problems.

- Typical applications in bioinformatics are focused on modelling processes and systems. Extensions are therefore based on a *model* system, for representing custom data-structures; domain developers construct a model definition language using the framework DSL API. The model framework integrates seamlessly with the host language.

- A combination of existing and novel techniques are used to *automatically parallelise* functions written using our recursive language. The program synthesis, for generating a massively parallel GPU application, takes the following steps:

  - We use existing techniques [56, 61] to frame the parallelisation problem as one of partitioning the call-graph into an ordered list of groups, where all members of a group are independent. We use a *scheduling* function which determines, for each call, to which partition (time-step) it belongs.

  - We define what criteria a scheduling function must satisfy in order to be valid for a given recursive equation, based on the recursive calls. We use these criteria to formulate a CSP (constraint satisfaction problem) for locating a valid, efficient schedule.
  - We show how the scheduling function can be used to synthesise a graphics card program using a *polyhedral code-generation* tool, CLooG [11]. Specifically, we show how we can provide our schedule and function as input to CLooG, which in turn generates a series of nested loops representing the input domain as transformed by the schedule. We then provide post code-generation transformations to convert the generated (serial) code to a GPU-threaded application.

- We show how embedded extensions interact with this automatic parallelisation in a safe and straightforward way, by defining affine equations representing the range of values for each new expression.

- We show how the framework effectively separates the development of new embedded DSLs from the generation of GPU code. Specifically, our framework ensures only small amounts of GPU code generation are required for new expressions in the language. This separation of concerns ensures that new DSLs can be created with little prior knowledge of the GPU.

- We provide techniques for encapsulating and embedding GPU expertise in the compiler, including heuristics for determining parameters such as block and grid sizes. Importantly, these heuristics may be customized by GPU experts in order to provide improved performance, or support new GPUs.

- Allocating and transferring data to the GPU requires a significant amount of bookkeeping and boilerplate. We simplify the process by providing a *data layout framework* for describing the arrays associated with models, and generating load and copy operations on that data.

  - Different memory locations on the GPU provide different performance characteristics. For domain developers, the choice for their arrays may be overwhelming. We provide a parameterised algorithm for allocating arrays to memory locations, where the parameters are system dependent (e.g memory size, GPU capability), problem dependent (array size) and function dependent (access patterns). Our solution is fully customisable by GPU experts through the definition of various cost functions.
  - To determine appropriate memory assignment requires some consideration of access patterns. We describe a set of program analysis tools for determining such use patterns.

- Finally, we evaluate the performance of the techniques described above, and the overall performance achieved.

1.2 Outline

We start in Chapter 2 by exploring the origins and architecture of massively parallel processors, and in particular GPUs. We consider the complexities involved when developing for GPUs in order to understand what problems face a potential developer, as well as considering important optimisation techniques.

Chapter 3 gives a brief tour of applications from the domain of bioinformatics, our chosen case study area. Exploring these applications finds common ground, which helps to define the scope and reach of the DSLs.

Combining observations from both the previous chapters, we proceed by exploring the construction of suitable DSLs in Chapter 4. We propose a method of simplifying domain development for bioinformatics by providing a recursive functional host language, into which new extensions can be written as embedded DSLs.

Once we have defined a DSL language for describing these problems, our next focus

is on generating GPU code for our DSLs. In Chapter 5 we provide a method for automatically parallelising the recursive host language, and subsequently generating GPU code using the polyhedral model. We discuss and implement a number of optimisation and automatic parameter setting techniques based on the understanding of the GPU developed in Chapter 2.

In Chapter 6 we bring together the themes of this dissertation in an example case study, a DSL for Hidden Markov Models. We describe the implementation of the HMM extension, from the construction of the front-end to the interaction with the automatic parallelisation routines and code generation.

In Chapter 7, we attempt to evaluate the success of our approach; both in terms of the success of the language and extension systems, as well as the overall performance results. Whilst we justify the methods and techniques used on a theoretical basis, the GPU is a complicated architecture and performance results are hard to predict. We therefore benchmark the framework against a number of full applications, and provide micro-benchmarks for a number of our optimisation and heuristic routines.

Finally we provide concluding remarks in Chapter 8, summarising the success and limitations of this DSL approach, as well as describing where further work may be profitably pursued.

1.3 Audience

Pursuing this thesis has entailed straddling a number of domains, exploring ideas from compiler development, automatic parallelisation, HPC (High Performance Computing), massively-parallel processing and bioinformatics. As such, this dissertation is potentially useful for interested parties in any of those areas – knowledge sharing is one of the distinct advantages of such a broad thesis. However, in writing this document we have to keep in mind a particular reader or set of readers whom it will most benefit. In this case I have

written the text with a particular focus on those who are developing language or library support for graphics cards or other massively-parallel processors. For those experienced with GPU development, Chapter 2 may be optional, as it introduces the GPU architecture, and GPU development.

1.4 Typographical conventions

Code is written using a `fixed-width` font. Mathematical recursions are formatted as equations, e.g $f(x, y) = f(x - 1, y - 1)$. Type names are written using a sans-serif font.

Chapter 2

Massively Parallel Processors

Massively parallel processors are a rapidly expanding area of research, making use of vast numbers of cores on a single chip for both specific and general purpose applications. Growth in this area has been driven by the development of massively parallel graphics cards, resulting in commodity hardware that is widely available, scalable and cost-effective. The adoption of general-purpose hardware for graphics applications has opened the door for non-graphics applications to use the potential of the graphics card.

Graphics cards manufacturers, in particular NVIDIA, were quick to identify the potential market for such applications, leading to the introduction of general purpose frameworks such as CUDA and OpenCL. The target audience has grown from high performance applications in the physical sciences, life sciences and financial sectors to applications across research and beyond. Desktop software in diverse areas such as image editing with Photoshop, video encoding and decoding and web browsing have benefited from the use of a massively parallel co-processor.

New players are rapidly entering this space, albeit with different takes on the architecture. The recent launch of the Intel Xeon Phi demonstrates a commitment from Intel towards the development of massively parallel processors, and shows that massively parallel processing is beginning to become a significant market in its own right. On the other

end of the scale, smaller companies have taken the opportunity to provide innovative architectures which combine cheap cost-effective parts in a massively parallel way. A great example of this is Adapteva with their Kickstarted Parallella project.

In principle, any application can be ported to a massively parallel architecture. In practice, we must find applications, or sections of applications, that are suited to parallelisation across a relatively large number of cores. As always with parallel computations, we are constrained by *Amdahl's Law* – we can only speed up applications relative to the percentage of the work that is parallelizable.

In massively parallel processing, a degree of independence or a repetition of work is a key criteria for acceleration, and may vary by the target hardware. We must caution, therefore, that this approach is no silver bullet – the massively parallel architecture must instead act as a true co-processor, working in tandem with the CPU to compute suitable workloads.

We will use this chapter to explain the motivation of the work that follows, by determining why massively parallel processors are useful, how they achieve such high peak performance, and how to best exploit them. By identifying the challenges involved in developing applications for our chosen device, the GPU, we can hope to improve the lot of domain-experts interested in building applications for the GPU.

2.1 Background

2.1.1 The genesis of the massively parallel co-processor

The development of a drop-in card for graphics began in the early 1980's with the advent of IBM PC video cards, however it wasn't until the 1990's that true graphics co-processors become common-place. These early graphics co-processors were simple affairs with fixed functions designed for implementing common 2D graphics operations. The development in the mid 1990's of combined 2D/3D cards led to hardware with **fixed pipelines**, in which

a series of pre-determined computational steps were applied to a set of vertices and pixels [81].

The potential of this new hardware became rapidly apparent – not only did a specialised co-processor for graphics processing alleviate work from the CPU, it allowed for a custom hardware architecture, fully optimised for computing these fixed pipelines.

In particular, it allowed the designers to take advantage of the concurrency inherent in these pipelines – much of the data could be computed independently. Additionally, the hardware could be designed to make frequently performed operations extremely fast at the expense of less frequently used operations for general purpose computing. These factors combined to provide hardware with high peak performance – typically four to five times higher theoretical Floating Point Operations Per Second (FLOP/s) than an equivalent CPU.

As game and graphics developers required more control over the exact function performed on the GPU, manufacturers moved towards a more flexible hardware. The development of *programmable shaders* in the early 2000's for the first time allowed developers to access the graphics hardware in a meaningful way. Each shader represented something akin to a general purpose processor, which could be used in combination with the fixed function hardware. Originally programmed in low-level assembly languages, high-level languages quickly appeared including Microsoft's HLSL, NVIDIA's Cg [77] and OpenGL's GLSL. It was rapidly discovered that these high-level languages could be exploited for general purpose applications as diverse as linear algebra [59], physical simulation of clouds [44] and ray tracing [98]. As these applications proliferated, these languages evolved into *hardware-oriented*, *general-purpose* languages.

The BrookGPU [15] project was one of the first to take advantage of these low-level hardware languages for a *general-purpose* computing framework; this formulation is the starting point for nearly all General Purpose GPU (GPGPU) computing today. They adopted the *Brook Stream Programming Model* previously developed for streaming supercomputers and adapted it for a massively parallel GPU. The model consists of data described as

streams, essentially multi-dimensional arrays, which are manipulated by *kernels*, special functions that act on streams of data by applying the body of the function to each element in the stream. BrookGPU provides a limited memory model – kernels are required to specify which streams are input and which are output to prevent side-effects, data can be *gathered* from different locations only if described as a gather stream, but cannot be written *scattered* i.e they must write in a regular pattern.

2.1.2 Mainstream general purpose computing

Since the release of BrookGPU, general purpose computing on graphics hardware has become a major selling point for all of the large graphics hardware manufacturers. NVIDIA have become market leaders with the Compute Unified Device Architecture (CUDA) compute framework, whilst AMD (formerly ATI) have developed the APP (Accelerate Parallel Processing) SDK for developing massively parallel applications in conjunction with OpenCL. OpenCL, initially developed by Apple, is a general purpose framework for executing applications on heterogeneous architectures, and was specifically designed to support GPUs. Microsoft developed DirectCompute, a cross-manufacturer framework included with DirectX, with similar execution patterns to CUDA and OpenCL.

These technologies all adapt the Brook approach described above, adopting the *kernel* system whilst dropping much of the focus on streams. Kernels can be specified to run over a certain grid of values, which are then used to access data as required, supporting full scatter/gather on the GPU. Hardware improvements as well as software support have been a vital part of this effort.

Intel Xeon Phi chips may also be programmed using OpenCL, however, their x86 heritage and heavy-weight cores mean that they can also be programmed using existing and custom CPU parallel tools, such as pragmas in the code for offloading to the co-processor and parallel CPU frameworks such as OpenMP.

2.1.3 High-level frameworks

The perceived complexities with this computational model have also led many to consider "higher-level" frameworks. These frameworks attempt to simplify GPU development by using simpler computational models.

Microsoft have developed the *Accelerator* [114] framework, which aims to make general-purpose use of parallel, massively parallel machines and FPGAs accessible to the average programmer. Available on a wide variety of .NET languages, and running on a wide-variety of GPU platforms, it is implemented through parallel array functions that perform data parallel functions across all items within an array. Another array focused language is SAC, a high-level language with multiple targets, including GPGPUs [43]. Similar work has been undertaken for GPU data-parallelisation using array computations in Haskell by defining DSLs [75, 22, 111]. Copperhead [18] is a similar embedded data-parallel language which uses a subset of Python including many data parallel primitives, permitting compact descriptions of GPU algorithms.

A different approach to this problem is to use *directive-based* libraries to speed up code by annotating e.g parallel loops, for example OpenACC [3]. The advantage is that the original code can remain in place, effectively providing quick access to multiple execution targets at minimal cost, in particular for legacy code bases. The downside is the lack of flexibility and optimisation – in particular, code is limited to pre-programmed parallel patterns such as loops with independent body execution and simple reductions.

These examples illustrate two trends: firstly, the growth of high-level frameworks targeting multiple types of parallel and custom hardware (GPGPUs, multi-core, FPGAs, etc.) and secondly the focus on array computations and basic parallel patterns as the basis for implementing parallel algorithms. The latter should not come as a surprise; array-style processing through frameworks such as map-reduce has become common place in computing across clusters and it provides a cheap and easy way to develop such applications.

A recent interesting development of the first trend has been language virtualisation [20],

where a parallel runtime is accessed via a series of user-created domain-specific languages for different application areas. This approach has the benefit of providing easy access to parallel hardware for domain experts.

2.1.4 Application growth

At the front of the queue for these new technologies were the physical and life sciences, and modelling communities, with diverse target applications including financial modelling, bioinformatics, computational chemistry and medical imaging, amongst others. For these areas, the low cost of development, wide availability and excellent scalability all contributed to movement away from traditional data parallel high performance computing techniques.

Applications in life sciences are typically driven by large data-sets – in particular, bioinformatics deals with enormous, and growing, datasets of DNA and protein sequences, that require complex analysis. Development in other application areas requires a large amount of processing on smaller datasets.

One advantage of GPUs is that they are ubiquitous across the board. Many desktop applications have therefore benefited from the widespread installation of GPUs. This particularly includes workstation tools; classic applications such as rendering and visual effects, but also image editing and other mainstream uses such as 1080p decoding and web browsing.

2.2 Parallel thinking: a worked example

As a first approximation, we will consider a simplified, abstract version of the massively parallel architecture: a large collection of relatively slow cores grouped into a series of *multiprocessors*, with discrete memory. This simple description will allow us to develop the ideas surrounding development on such hardware, before we expand to detail the intricacies

of the particular implementations.

We will use a common example in massively parallel programming [57, 91], matrix-matrix multiplication. This is an excellent example of an application that is suited to porting to a massively parallel processor. In practice, library support exists for performing linear algebra computations on the device - CUBLAS, MAGMA and FLAME amongst others.

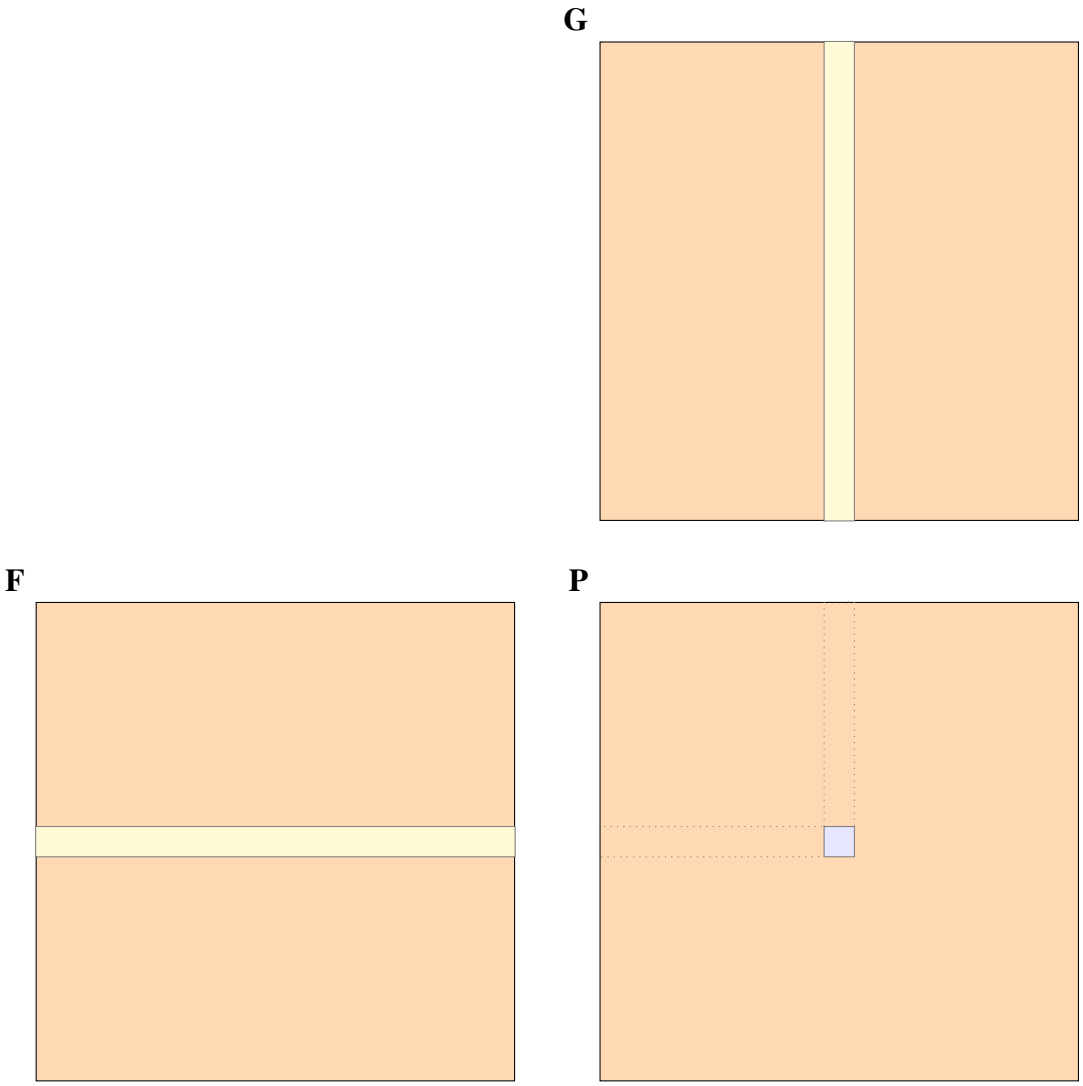


Figure 2.1: Matrix multiplication - computing a cell in **P** requires the dot product of a row in **F** and a column in **G**.

In matrix multiplication we take a pair of matrices **F** and **G** and perform a series of dot products to compute a new matrix **P**. Each cell in the matrix is computed by loading the

equivalent row from **F** and the equivalent column from **G** and computing the dot product. Each cell is computed independently from any other - the result of any other cell will not change the result of this cell. This *independence* will allow the computation of any combination of the cells of **P** in parallel. Figure 2.1 illustrates the point.

Our observation that each cell may be run in parallel with any other is key to placing the algorithm efficiently on the device. In general, algorithms that display limited dependencies will be easier to parallelise than those that have an intricate set of dependencies.

2.2.1 Device basics

The device we are considering for the rest of this chapter is a massively parallel co-processor, most commonly in the form of a GPU connected via a PCIe slot. Execution on the co-processor or "device" follows these steps on the host:

1. Initialise the co-processor.
2. Allocation of memory on the device.
3. Copying of source data to the discrete memory of the device from main memory
4. Transfer and execution of a *kernel*, which describes the code to run on the device.
5. Copying of destination data from device to the host memory.
6. De-allocation of memory.

The kernel describes the code to be executed on each multiprocessor on the device. Each multiprocessor is independent – there are no synchronisation functions between multiprocessors on a device. The cores within a multiprocessor are co-operative in that only a single instruction may be executed at each step - so called SIMT (Single-Instruction, Multiple Thread).

Threads are grouped by *blocks*, with each block assigned to an arbitrary multiprocessor on the device, according the decision of the on-device scheduler.

2.2.2 Thread granularity

The first step to parallelisation is to determine the granularity of the thread – what level of the algorithm should we divide in to threads? Threads in the massively parallel model tend to be light-weight, so the trade-offs will be different from typical multi-core programming. The granularity should take into account factors such as the number of computations, the intensity of those computations – the ratio between I/O and instructions – and the typical scale of the problem.

The easy way out is to appeal to *data parallelism* – apply the serial algorithm to large amounts of data, assigning a thread for each sequence. However, this approach leaves limited scope for the finer grained optimisations described later on, and may also be impractical for many applications. For our worked example we are going to assume we cannot appeal to this sort of parallelisation.

Based on our observation that each cell is independent, we are instead going to assign a single thread to each cell. A finer granularity – say, computing the dot product in parallel – would require some form of inefficient reduction on the device. A broader granularity - say a collection of cells – may not produce enough threads to fully saturate the device.

We will group threads into blocks according to their two-dimensional spatial locality - selecting what is often described as a tile in the output table. This is a very common pattern in massively parallel applications. In theory we could group threads by rows or columns instead, however grouping by tiles will allow us to make better use of the memory hierarchy on the device when we deal with optimisations.

2.2.3 Kernel implementation

Now we have decided on the granularity of the threads, the implementation of the kernel is fairly straightforward. The kernel is written from the perspective of a single thread. In this case, each thread should compute the dot-product for a single cell in the table. These will be grouped into thread blocks by tiling the output table; each thread will then compute the

dot-product in tandem with other threads representing near by cells.

Figure 2.2 gives the description of such a kernel in the CUDA framework, although it only differs slightly in syntax to the equivalent implementation using OpenCL. One important feature exploited in this description is the ability to lay threads in a 2D or 3D grid within a thread block. By defining the width/height of a block when calling the kernel, each thread will be given a unique co-ordinate, accessed through the `threadIdx` variable. This notation simplifies the description of many algorithms that deal with a grid of values. Also note the use of local variables, stored in registers, to compute temporary values.

```
__global__ void matrixMultiplication(float * m, float* n, float* p, int width) {
        float output = 0;

        int x = threadIdx.x;
        int y = threadIdx.y;

        for (int i = 0; i < width; i++) {
            float mval = m[y * width + i];
            float nval = m[i * width + x];
            output += mval * nval;
        }

        p[y * width + x] = output;
}

// Kernel can be called using the following code
dim3 threads(width,height);
matrixMultiplication<<<blocks,threads>>>(..<parameters here>..);
```

Figure 2.2: Example kernel for matrix multiplication.

2.3 Architecture

The hallmark of a massively parallel architecture is that it contains a very large number of cores on a single chip. Recent introductions to the market have, however, created a fragmented design space. In general, the architectures are based on a collection of *vector processors*. Each vector unit implements a variant of *SIMD* (Single Instruction, Multiple

Data) to work co-operatively on a problem, and work is spread over different vector units.

In a graphics card each of these processors is known as a *streaming multiprocessor* (SM). Each multiprocessor is, in one sense, much like a CPU chip, with a number of cores that can work in tandem. However, it differs both in the number of cores - typically 32 or more - and in the method of co-operation. Where a CPU chip will schedule so-called "heavy-weight" threads to each core, with each thread executing in a completely independent way, a streaming multiprocessor will execute all threads synchronously.

Such a system is a natural extension to the paradigm, and so is often described as a *SIMT* (Single Instruction, Multiple Thread) or *SPMD* (Single Program, Multiple Data) architecture. In addition, each multiprocessor is independent from the rest; there are no synchronisation primitives across multiprocessors, and only limited support for communication through atomic operations.

The MIC (Many Integrated Core) Architecture at the heart of the Intel Xeon Phi is slightly different. Each vector unit is described as a *core* within the MIC processor. The SIMD vector units within each core are traditional SIMD units rather than GPU SIMT units. The processor also provides synchronisation primitives across cores, in contrast to the GPU. This reflects the differing aims of the project; it intends to provide easy access to a massively parallel architecture by recycling existing x86 cores with SIMD operations.

2.3.1 Kernel

The device code is defined by a *kernel*. This kernel provides an imperative program description for the device code. This code is executed on each active thread in step. The code for each thread differs only in the parameters passed to it - such as a unique thread id. These allow loading of different data, through indirect addressing, despite each thread executing the same instructions.

Whilst it is true to say that all active threads on an SM must co-operate, it does not mean that they must all follow the same execution path. Where active threads take a different ex-

ecution path, they are said to have *diverged*. A common strategy for fixing such divergence is to divide the active threads when we reach such a branching point, and execute one group followed by the other, until both groups have converged. However, this may result in an under-used SM, where some cores remain fallow for much of the time.

The actual hardware implementations introduce more complexity. An SM may have more threads allocated to it than it has cores. As such, it may need to schedule threads to cores as and when they are required. As a result, they are designed with "light-weight" threads, which provide minimum overhead, including low cost of context switching between threads. This is achieved by storing the data for all threads assigned to a SM at once - each core provides a program counter and a minimal amount of state, including the id of thread. To switch threads, we simply replace the program counter and the minimal amount of state. Register contents remain in place for the lifetime of the thread on the device.

2.3.2 Comparison to CPUs

It is illuminating to compare the massively parallel architecture described so far to a typical multi-core chip. The most obvious differences lie in the layout of the chips – a desktop processor typically has 2 to 8 cores on a single chip, whereas a GPU will contain many more, often thousands, organised into vector multiprocessors comprised of tens to hundreds of cores. Another difference, driven by the need for high data throughput, is the memory bandwidth available – main memory to the CPU is typically around 12-50GB/s; GPUs can support anywhere up to 200GB/s.

An interesting comparison can be made between the cores. GPU cores are optimised for single-precision floating point computations; typical floating point operations may require fewer clock cycles than on an equivalent CPU. Additionally, the SM is configured to support low-cost memory accesses by providing vast numbers of registers. One key difference between the two is the way in which they configure and use the cache. Most GPUs provide user configurable *shared memory*, which has similar latency to a cache but is managed by

the user. The latest NVIDIA GPUs allow this to be split between user-configurable and a true cache.

2.3.3 CUDA

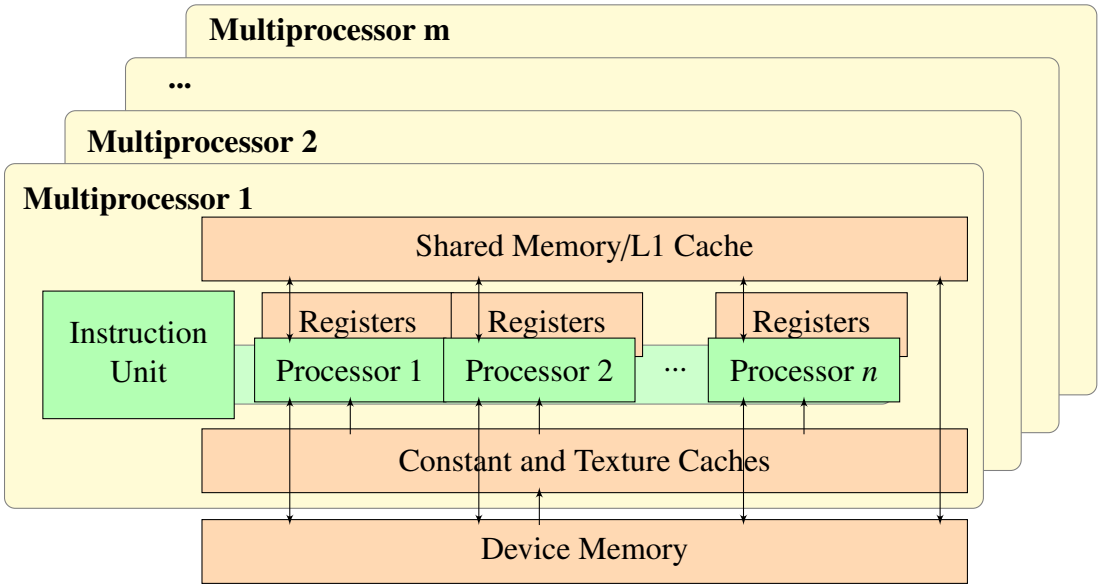


Figure 2.3: The general purpose architecture of a typical CUDA based GPU.

CUDA [91] is one such concrete implementation of GPGPU framework on top of massively parallel hardware. It has been developed and designed by NVIDIA in conjunction with the graphics card hardware resulting in a coherent and usable framework. We will describe the details of CUDA as an example of how a modern massively parallel framework works. Figure 2.3 provides an outline diagram of the CUDA architecture.

For the remainder of this thesis will focus on using CUDA devices; this is a pragmatic decision, based on the availability, and quality, of the frameworks and tools at the start of the research, as well as the widespread install base of NVIDIA GPUs.

2.3.3.1 Blocks, threads and warps

CUDA describes a collective group of threads as a *block*. Threads in a block may work co-operatively, and are placed together on a single multiprocessor, which may be shared with

other blocks. Threads are scheduled to the cores of the multiprocessor in smaller groups known as *warps*. Warps may be considered to be a constant width that abstracts from the true execution width of the device; the block may be considered as a high-level grouping of threads that may co-operate.

Whilst warps are how threads are scheduled to the device, the execution of each warp depends on the device itself. A CUDA SM has between 8 and 192 cores, depending on the exact card; a warp is 32 threads. When a warp is executed, we may execute it directly, or on older devices, the hardware divides the 32 thread warp into half or quarter warps. However the hardware divides (or not) the warp, the threads are executed in parallel on the device – one thread per core – until the whole warp has been processed. Cards with 32 cores typically run two half warps simultaneously; the half-warps may be from two different full warps. More recent cards, with anywhere between 48 and 192 cores, use around one third as a *super-scalar* extension; any half-warp whose *next* instruction may be executed in parallel with the current instruction can use the super-scalar cores.

Such a system ensures that all threads within a warp remain in step. However, since each warp may execute at its own pace (or that of the scheduler), threads within the same block, but in different warps, may be executing different pieces of code altogether. NVIDIA provide a `__syncthreads()` function to ensure all warps in a block reach a given place in the kernel before continuing.

2.3.3.2 Memory features

As an artefact of its origin and development as a processor for graphics workloads, the GPU architecture features a complex memory hierarchy. Using this memory hierarchy appropriately is vital to achieve good performance. Each memory location is optimised for a different type of work load, and latency, bandwidth and caching policy vary. The different available locations are:

- *Registers* – Each SM has a register file shared amongst all the resident blocks, which

is between 8192 and 65535 registers on current devices. There is little to no latency involved in accessing registers. The registers of a thread are maintained, even when that thread is suspended, in order to maintain low-latency context switching.

- *Shared Memory* – On-chip memory, *shared* amongst all the cores on a single multiprocessor. It is designed to allow co-operative computation amongst the cores of the SM. Each block may allocate an area of shared memory, and each thread of the block may read and write to all locations within it. Latency is typically much lower than global memory – of the order of 40 clock cycles. However, the shared memory is maintained even when a block is suspended, in order to preserve low latency switching between blocks. High use of shared memory can therefore inhibit the number of blocks allocated to a multiprocessor, which can impair performance. The shared memory is organised in consecutive *banks*. Bank conflicts can result in serialisation of requests. On later devices, it shares memory with the L1 global cache.

- *Global Memory* – This is the main memory of the GPU. Whilst it has a very high bandwidth – around 100-200 GB/s – it also has a very high latency, around 400 to 800 clock cycles. Minimising global reads and writes is critical to achieving high performance, including *coalescing* data access to ensure data is stored in thread access order. Data loaded to and from global memory may be cached in L2 or L1 caches, depending on the device – where it exists, the L1 cache has a latency and bandwidth similar to shared memory. Global memory consists of the large part of the device memory advertised for the device – typically in the range of 128Mb or 256Mb for low-end cards, up to 4GB or more for specialist workstation cards. Only the host may allocate global memory – no dynamic memory allocation is allowed on the device itself. Data is usually copied from host-to-device or device-to-host by the host, but more recent devices allow some device initiated transfers. However, the device may access the data using a pointer passed as a parameter to the kernel function.

Memory accesses are dispatched across either a warp or a half warp, depending on the *compute capability* (architecture revision) of the device.

- *Texture cache* – Designed for caching the loading of textures in and out of global memory. Has the unusual feature of optimising texture loads in a 2D locality. On some devices it provides lower bandwidth than the L1 cache, and so can be less frequently used. The latest NVIDIA cards, based on the "Kepler"architecture, and first released in 2012, improve the ease with which this can be used as a read-only cache for un-coalesced work loads, as well as improving throughput. Use of this cache does not necessarily reduce latency from global memory, but can reduce memory bandwidth. It can also help prevent coalescing problems with older devices without L1 caches.

- *Constant cache* – A small area of cache designed for small, read-only, contiguous arrays. Very low-latency.

2.4 Performance considerations

Despite the distinct differences in hardware and architecture design across different graphics cards, there are a set of common features that often require careful programming to utilise correctly.

2.4.1 Minimise data transfer to and from the device

The majority of co-processors are developed using the PCIe bus as a method of communication to the main memory and processor. It is a fast method of data transfer, however it is an order magnitude slower than data transfer between main memory and the CPU or global memory and the GPU. We must therefore ensure that data is transferred efficiently to and from the device, by only transferring what is strictly required. In addition, there is

an overhead associated with each PCIe transfer. This is typically minimised by performing memory allocation and data transfer in large blocks, rather than small sections.

Page-locked (or pinned) memory is another feature that improves performance of data transfers, by *locking* the host pages in memory. This ensures that memory transfer from the host to the device can be achieved without the CPU becoming involved, as the device can make a DMA (direct access request) for the physical pages locked in memory. Without pinned access, pages may be swapped out, and transfers require CPU intervention to copy data to pinned memory, before transferring to the device. Furthermore, pinned memory often provides higher bandwidth than non-pinned memory. A disadvantage of page-locked memory is that it prevents the host swapping these pages in and out of memory, thus potentially leading to poor host performance, and a limit on the amount of host memory that can be allocated.

2.4.2 Tiling: making use of shared memory

Our example problem described a scheme for grouping threads into blocks by tiling the destination matrix. One advantage that this has over other groupings is that for $m \times n$ cells we only need to load $m + n$ distinct rows or columns from the other matrices. By default, the hardware will load these values without broadcasting them, meaning memory access is much higher than necessary. This is particularly relevant on older devices with no caches between global memory and the multiprocessor.

In this case we can use the shared memory of the multiprocessor to store the necessary row and column values – acting as a *user-configured* cache. Before computing the dot products, the cells in a thread-block work co-operatively to build the shared memory representation of the input matrices, before continuing on to use the values to compute the individual dot-products. This can have a notable efficiency saving over each cell loading the same values independently.

This technique is often seen in other pieces of hardware with multiple cores – the Cell

processor, for example, uses explicit Direct Memory Access calls to copy data from a main memory to local, low-latency memory on the device.

Functionally, shared memory on CUDA devices is split into a number of distinct banks. These banks can be accessed simultaneously, thereby increasing bandwidth to the multiprocessor. The ideal memory access pattern is therefore that every thread reads consecutive 32 bit (or 64 bit) memory locations. If two threads in the same warp access the same bank, the request is serialised and there is said to have been a *bank conflict*. Depending on the device, this may result in lower bandwidth performance. We should therefore ensure that our tile size and ordering is designed with device in mind, to prevent serialisation due to bank conflicts.

2.4.3 Coalescence

When accessing device memory we can reduce the number of different memory requests that are dispatched by *coalescing* memory reads or writes. Coalescing considers the requests from threads across the warp, and attempts to bundle them up into a smaller number of 32-, 64- or 128-byte memory requests. The conditions for this coalescing differ between different chips and architecture versions, but in general coalescing is best exploited when threads in a warp are accessing consecutive memory locations. Note, this is similar to the ideal memory access patterns displayed in the shared memory banks described above.

2.4.4 Constant values

For a small quantity of unchanging values, devices support a *constant cache*. This provides a low-latency cache that is similar in access times to shared memory.

In order to support low-cost warp switching, all active warps retain their register and shared memory allocation whilst other warps are running. Registers are therefore a finite resource that, when considered with the high-latency of device memory, must be carefully allocated. Constant memory can help alleviate this register pressure, in addition to improv-

ing access times for frequently accessed data. Constant data must be allocated on the host and cannot be modified during device calls. When all threads in a warp access the same constant value, the value is loaded and broadcast to all threads. If different constants are accessed, the requests are *serialised*. Constant memory is thus best used when there is little divergence between threads over which constant values are loaded.

In CUDA devices, the constant memory size is 64KB, with each multiprocessor containing an 8KB cache for the current working set. Thus, constant cache is best used where warps on a multiprocessor use a similar subset of constant values.

2.4.5 Latency tolerance

One of the fundamental design features of GPUs is that they are *streaming architectures* – operations have a high-level of latency. This conscious decision to provide simple cores is a necessary trade-off in order to place a high number of cores on the device. For memory operations to global memory, this may be as much as 400-800 clock cycles; on-chip memory is typically around 40 clock cycles and arithmetic latency around 5-10 clock cycles. Furthermore, *register dependencies*, where consecutive instructions write then read a register introduces a latency of around 25 clock cycles[1]. Synchronisation primitives may also cause threads to halt whilst waiting for other threads to reach the barrier. Hiding these latencies is fundamental to achieving high performance on the GPU.

Latency hiding is primarily achieved by ensuring multiple instructions can be executed at any one time. This takes two forms: *thread-level parallelism* (TLP), running more threads on a single multiprocessor to allow the scheduler to switch to a different warp whilst waiting for other operations to complete or *instruction-level parallelism* (ILP), where we ensure multiple instructions on the same warp can be dispatched simultaneously.

TLP is enabled by providing low-cost context switching. When warps are resident

[1]Quoted latency figures are from both the CUDA Programming Guide [91] and empirical data. For an example of empirical results, see *Demystifying GPU Microarchitecture through Microbenchmarking* [122], which uses micro-benchmarking to empirically discover the latencies of the GT200 architecture.

on the GPU, they retain their register and shared memory allocations between scheduled executions. Switching therefore requires no copying overhead as on CPUs, and allows the scheduler to cover even small latencies by switching to other threads. To ensure TLP occurs we need to launch a sufficiently large number of threads, so that latencies are covered appropriately.

ILP, on the other hand, will often require changes to the kernel to take advantage of it. Kernels must have *consecutive* independent instructions that can be dispatched consecutively i.e without waiting for the previous instruction to finish. In some cases the compiler may be able to re-order computations in order to achieve this. A combination of ILP and TLP is typically the best approach in order to hide latency effectively.

2.4.6 Occupancy

The determining factor for TLP is the *occupancy* for each multiprocessor, defined as the percentage of the theoretical maximum threads that are allocated. Given the block size, number of registers used per thread, shared memory per block and the compute capability of the device, we can determine the occupancy using the CUDA GPU Occupancy Calculator [89].

Occupancy has been an important metric for optimisation since the early days of CUDA. In the NVIDIA Cuda C Best Practices Guide [88], the advice regarding occupancy is this:

> Higher occupancy does not always equate to higher performance – there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

Recent research, however, has suggested that a narrow minded focus on occupancy can be damaging to performance. A relevant presentation "Better performance at lower occupancy" [119] from GTC 2010, coherently and convincingly argued that performance was

often *improved* by reducing the number of threads per multiprocessor. Examples of matrix multiplication and real world tests demonstrated that high performance can be achieved through lower occupancy.

In this case, the devil is in the details, and the thrust of the argument is often misunderstood. It is better to *trade* occupancy for greater shared memory and register usage per thread, offsetting the loss of latency hiding using TLP with ILP instead. Thus the mantra is "more *independent* work per thread, more resources to compute that work". A straightforward way of achieving this is to take work that would have been evaluated by n separate threads, and instead perform the work on $n/2$ or $n/3$ threads, with each thread doing two or three times the work. Consequently, each thread has two or three times more scope for independent operations which may cover latency. In certain devices ILP may be required to achieve peak performance, and in most devices it appears that ILP is more effective at covering latency.

What this more subtle interpretation of the utilisation of the GPU emphatically does not say is that given a *fixed* program we should reduce occupancy. Instead they recommend *trading* occupancy for increased number of registers and shared memory per thread, in order to maximise independent workloads and therefore the amount of available ILP.

2.4.7 Spatial memory access

Where memory access is not regular enough to support a high-level of coalescence, there are other techniques for improving memory throughput from the device. One such technique is to use spatial memory access, typically known as *texture caching*. In this form, we take a two or three dimensional array and store it according to 2D or 3D location rather than in the typical row-by-row or column-by-column ordering. In graphics applications, this speeds up operations involving textures, where a group of threads will work together co-operatively to compute a square or cube section of the texture.

2.4.8 Optimisations based on hardware differences

Massively parallel processors tend to be developed with a high degree of hardware-variance. This is primarily due to the fast pace of GPU development, where each manufacturer is straining to improve gaming performance. This has led to a rapid and widespread development of the architectural features. From the point of view of an application developer, these differences between hardware targets must be considered to achieve speed improvements across the board.

2.4.8.1 Core density and speed

In principle, the configuration of massively parallel hardware is straightforward – a large number of relatively slow in-order cores. In practice, the exact configuration can vary greatly between manufacturers and even between models. The number of individual cores must be balanced against the cost of the core, the speed and therefore the heat output as well as the power requirements. Different cards will make different trade-offs.

The most obvious example of such differences in hardware is the variance between how the cores are grouped into multiprocessors. In the latest NVIDIA "Kepler" cards (released in 2012), for example, each multiprocessor contains 192 cores, compared to the earlier 48 and 32 core chips of the "Fermi" cards (released in 2010), and the 8 cores of the G80 and G200 architectures. The practical influence of such differences will relate to exactly how we divide up the work – where we have a larger number of cores we will need to ensure we have dispatched enough work (e.g warps) to fulfil the parallelism potential, and ensure that kernels are amenable to ILP where it is necessary in order to make full use of those cores.

2.4.8.2 Memory size and configuration

Typical memory size for new desktop cards are now around 2GB, but can be as little as 128MB for older devices or 6GB for top of the range cards. For graphics applications, larger memory provides little performance improvement except when using very high res-

olutions. Special purpose hardware, such as Tesla, tend to move toward the larger ranges to support more general purpose applications. This memory is an absolute limit – no form of paging usually exists on such devices. Instead, we have to *chunk* data to support larger datasets.

2.4.8.3 Cache configuration and size

Caching is an important part of speeding up reads and writes for desktop processors, however, early massively parallel graphics cards featured very little in the way of caches. This reluctance stems from the type of applications, which use a high compute intensity to hide memory latency. Where caching was useful, it was recommended as a manual memory allocation to the on-chip shared memory, as in the tiling example described previously. This approach has the advantage of supporting the processors in a predictable and accurate way.

The latest cards from both NVIDIA and AMD support a small amount of L1 cache on each multiprocessor. The addition of a cache demonstrates a move towards a more general purpose architecture, and an attempt to improve the performance of programs which have not been heavily optimised.

NVIDIA allow some limited customisation of cache size on the latest Fermi cards, where 64 KB is available for both cache and shared memory, and can be configured in either 16KB shared/48KB cache or vice versa. In addition, an L2 cache of 768KB shared across all multi-processors is available. AMD use 8kiB for an L1 cache for each multiprocessor, along with a number of areas of shared memory that can be used to mimic caches. The differences in cache size compared to typical desktop processors can be put down to the relatively slow speeds of each individual multiprocessor, combined with the ability for applications to hide memory latency in other ways.

2.4.8.4 Call stack

Since the origins of the multiprocessor in the GPU are in implementing fast hardware pixel and vertex shaders, CUDA devices before the Fermi architecture, released in 2010, do not contain a call stack. If we wish to target the vast majority of GPUs, we must therefore do away with many of the comforts of programming on a standard architecture - in particular, all functions are implemented by inlining the definitions in the appropriate places. A corollary of this approach is that recursive functions cannot be defined directly within CUDA on devices before Fermi. This condition is not as onerous as it first appears – in practice, many recursive algorithms will achieve higher performance when re-written as a serial loop, in order to take advantage of the massive parallelism available.

2.4.8.5 Super-scalar execution

A recent advance in design for graphics card is the addition of a number of cores dedicated to *super-scalar* execution. Often known as *instruction level* parallelism (ILP), it allows two or more consecutive instructions to be executed in parallel, on the condition that the two instructions are independent. For massively parallel GPUs, the implementation occurs by providing the ability to dispatch either multiple warps or multiple instructions per warp on each clock cycle, usually by providing more function cores per MP.

The latest graphics cards with super-scalar execution – such as the GTX 600 series, and certain GTX 500 and GTX 400 cards – are set up to dedicate a certain portion of the chip to only super-scalar execution. We must therefore design our kernels with some independence between consecutive instructions in order to exploit these devices. In the worst case scenario, where no super-scalar execution is possible, we may use only a certain percentage of the true performance of the device (typically around 66% on current NVIDIA devices).

2.5 Challenges

To GPU experts the challenges facing the inexperienced initiate may not be obvious. But consider the questions any potential application developer might need to consider before, and during, development:

- How can I divide my task amongst the threads? Do I use block level parallelisation techniques? Place independent work on different threads?
- Where shall I put my data? Should I make use of cached global memory, shared memory, texture cache or perhaps constant memory?
- How can I set block size and grid size to maximise performance? How do these factors even effect performance?
- How does occupancy affect performance, and should I optimise to increase occupancy?
- How can I make use of ILP on suitable devices?

Furthermore, tuning applications often requires an intimate knowledge of the hardware, and the investment of time on the cycle of tweaking and bench-marking. That is not to say that GPU development, as it stands, is solely in the domain of the expert; a determined and capable programmer, willing to invest a reasonable amount of time, will be able to produce a decent GPU application. It is this time investment, however, that is the challenge, and often rules GPUs out as a development platform for many who would find it useful. An important and valuable goal, therefore, is reducing the time taken to develop new applications on the GPU.

Chapter 3

A domain case study: Bioinformatics

In order to explore and develop ideas for constructing domain-specific languages for GPUs we will adopt a case study. Using a case study provides a concrete example, real world motivation and, ultimately, practical implementation experience. Bioinformatics makes an ideal case study for both the viability and utility of domain-specific languages for GPUs, for three primary reasons:

- It is full of large-scale, long-running applications that would benefit from GPU acceleration.
- Many applications in bioinformatics have a wealth of *variations* on a single concept.
- Practitioners have diverse programming experiences; most do not have the time, programming experience or inclination to develop for GPUs. A GPU DSL would lower the barriers for entry, and provide access to an efficient platform for a much wider variety of users.

Over the last three decades biology has gained a distinct computational focus. The "central dogma" of molecular biology states that "DNA makes RNA makes protein", but the precise mechanism remains shrouded in complexity. By considering DNA, RNA and protein sequences as characters ("bases") in a fixed alphabet, we may perform statistical

analysis to determine biologically interesting features such as familial similarity, structural alignment, structure identification and the creation of trees of relations or phylogenies. Whilst computational analysis may not supplant the development of theories through empirical results, it can work to focus experimental efforts on important or significant areas.

The growth in the number of available sequences – with databases in the range of gigabytes of data – has emphasised this issue, making it increasingly important to develop efficient and effective techniques for analysing sequence data which simply isn't possible to analyse through careful empirical study. In particular, this growth has outstripped the development of the desktop CPU, leading to exploration of alternative methods of computation, e.g clusters, multi-core server CPU's and co-processors such as GPUs.

Almost all algorithms in bioinformatics are based upon choosing the most probable explanation of the data as a function of the underlying model and parameters, whether that be building phylogenetic trees, aligning sequences or identifying structural or functional features. These are classic optimisation problems; many make use of dynamic programming to achieve respectable performance [29].

In this chapter we shall briefly describe a number of applications within bioinformatics that make use of the dynamic programming technique, and attempts thus far to port such applications to the GPU or other massively parallel processors. This will place the work of future chapters in context, and provide the basis for the DSL decisions we later take.

For more information on these (and other) bioinformatics applications, the interested reader is directed to e.g. *Biological Sequence Analysis* [29]. For a gentle introduction to bioinformatics for computer scientists see [26, 73].

3.1 Pairwise sequence alignment

One of the earliest computational challenges in bioinformatics was to compare two sequences to determine how related they are. This is performed by *aligning* parts of the

```
HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
           G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN  GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
```

Figure 3.1: A sequence alignment between a fragment of human alpha globin and human beta globin.

sequences that are statistically similar, termed a *pairwise sequence alignment*. Computer scientists may recognise this concept as similar to that of the edit-distance problem; the minimum number of edits from one sequence to another.

Figure 3.1 shows an example alignment. The centre line is the alignment between the two sequences, where characters represent values that are identical in both sequences, and a + represents two characters deemed similar. An alignment will be described in terms of *substitutions*, *insertions* and *deletions*. Similarity in this case is determined by a substitution matrix – a table of values detailing the cost of substituting one character for another. A more likely than pure chance substitution is given a positive cost – or score – whilst a less likely than chance substitution is given a negative score.

This scoring is important; pairwise alignment techniques must choose between multiple alignments by determining which alignment best fits according to a scoring scheme. Substitution matrices commonly used include BLOSUM50 and PAM [29].

3.1.1 Needleman-Wunsch

The simplest form of alignment is *global alignment*, where we assume that we need to match the entirety of both sequences together, allowing for some gaps. This algorithm is known as Needleman-Wunsch [85], with a more efficient version described by Gotoh [37] using an affine gap algorithm.

The algorithm is usually described by a recursive functional equation:

$$
\begin{aligned}
F(i,0) &= -id \\
F(0,j) &= -jd \\
F(i,j) &= \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}
\end{aligned}
$$

where $F(i, j)$ is the score of best alignment of the first sequence and second sequence up to and including positions i and j respectively, s is a substitution matrix and d is a *gap-penalty* i.e. the cost of introducing a gap rather than a match. Informally, we say that the best alignment between the two sequences up to that point is the maximum value from choosing between substituting, inserting or deleting the characters at i and j having computed the best score up to those points.

3.1.2 Smith-Waterman

Smith-Waterman is similar in character to Needleman-Wunsch, however it instead focuses on *local alignment*, that is alignment between *sub-strings* of the two sequences:

$$
\begin{aligned}
F(i,0) &= 0 \\
F(0,j) &= 0 \\
F(i,j) &= \max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}
\end{aligned}
$$

The essential difference here is the ability to reset the score to zero if at any point all previous options have provided a negative score. Thus, to retrieve the score of the maximum local alignment between two sub-strings we must find the maximum value of $F(i, j)$ for all values of i and j.

One problem with this simplistic formulation is that the *gap penalty*, the cost of a series of insertions or deletions, is proportional to the length of the gap (e.g $-d$ multiplied by the length of the gap) which does not reflect the biological reality. This algorithm can be refined by using *affine gap penalities*, where the cost of a gap is defined by an affine function, which can be set so that extending a gap has a smaller cost than initiating a gap. Given a gap function γ, the recursive part of the equation becomes:

$$F(i,j) = \max \begin{cases} 0 & \\ F(i-1,j-1) + s(x_i, y_j) & \\ F(k,j) + \gamma(i-k) & k = 0, \ldots, i-1 \\ F(i,k) + \gamma(j-k) & k = 0, \ldots, j-1 \end{cases}$$

Note that we are now maximising over a newly introduced variable, k. This clearly increases the runtime complexity from $O(n^2)$ to $O(n^3)$ (when using dynamic programming). By assuming a simple affine gap penalty (with a gap opening penalty, ρ, and a gap extension penalty, σ, we can reduce the complexity back down to $O(n^2)$ by using three mutually recursive equations, equating to matching(H), insertion (E) and deletion (F) like so:

$$\begin{aligned} H(i,j) &= \max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + s(x_i, y_j) \end{cases} \\ E(i,j) &= \max \begin{cases} E(i,j-1) - \sigma \\ H(i,j-1) - \rho - \sigma \end{cases} \\ F(i,j) &= \max \begin{cases} F(i-1,j) - \sigma \\ H(i-1,j) - \rho - \sigma \end{cases} \end{aligned}$$

This form – mutual recursion to implement a gap opening and gap extension penalty – is the most commonly implemented version of Smith-Waterman. Conceptually, it may also

be considered, and implemented, as a vector recursion with three components.

3.1.3 Implementation

By necessity, both pairwise-alignment techniques described here use dynamic programming to ensure that the computation involved is feasible even for large sequences. The recursion is *tabulated*, with i and j on the axes.

3.2 Hidden Markov Models

Hidden Markov Models are a form of *generative* statistical model, where a set of *hidden* states determine the output sequence. They can be used to determine various probabilities associated with that sequence. They have an increasingly wide variety of practical uses, within bioinformatics and the wider scientific community, including DNA and protein sequence alignment and profiling.

Informally a Hidden Markov model is a *finite state machine* with three key properties:

1. The *Markov* property – only the current state is used to determine the next state.

2. The states themselves are *hidden* – each state may produce an output from a finite set of *observations*, and only that output can be observed, not the identity of the emitting state itself.

3. The process is *stochastic* – it has a probability associated with each *transition* (move from one state to another) and each *emission* (output of an observation from a hidden state).

The primary purpose is usually annotation – given a model and a set of observations, we can produce a likely sequence of hidden states. This treats the model as a *generative* machine – we proceed through from state to state, generating outputs as we go. The probabilistic

nature of the model permits an evaluation of the probability of each sequence, as well providing answers to questions such as "What is the probability of reaching this state?".

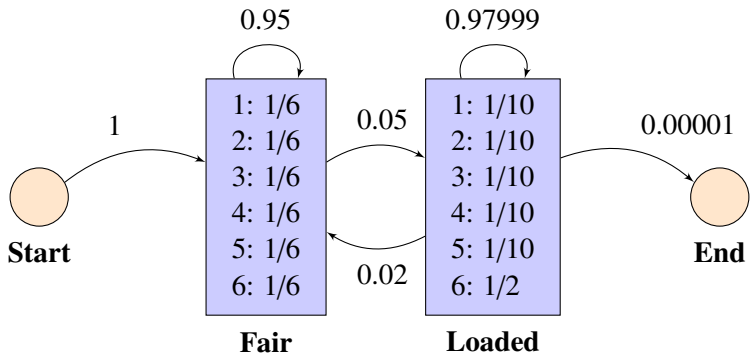


Figure 3.2: The occasionally dishonest casino

As an example, consider the *Occasionally Dishonest Casino* problem (Figure 3.2). In it we assume we are playing some form of game where we role a dice repeatedly, but that the casino is occasionally dishonest and will, with some fixed probability, switch to a loaded die in order to gain an advantage. We can use a HMM to model the probabilistic nature of the problem. We have a state that represents the case when the die is fair, and a state for when the die is loaded. Each transition has an associated probability – the likelihood of performing that transition. Each emitting state – fair and loaded – specifies the likelihood of emitting each character in the *alphabet* – in this case the sides of the die. The model is *hidden* in the sense that we never know which state emitted the character – in other words, which die was used – only the sequence of output characters.

3.2.1 Definition

Precisely defined, a Hidden Markov Model is a finite state machine, $M = (Q, \Sigma, a, e, begin, end)$, with the following properties:

- A finite set of states, Q.
- A finite set of observations Σ, known as the *alphabet*.

- A set of *emission* probabilities, e from states to observations.
- A set of *transition* probabilities, a, from states to states.
- A *begin* state.
- An optional *end* or termination state.

The process begins in the start state, and ends when we reach the end state. Each step starts in a state, possibly choosing a value to emit, and a transition to take, based on the probabilities. We will denote the probability of a transition from state i to state j as $a_{i,j}$, and the probability of state i emitting symbol s as $e_{i,s}$.

For the model to declare a distribution, it must relate a distribution over each transition. The emissions must also be a distribution over a specific state: however, this distribution may include a "no-output" emission.

A state which has no emissions is said to be *silent*; the start and end states are examples of this. Silent states are normally introduced to a model in order to reduce the number of transitions between states. A typically example is a forward connected graph, where states are ordered linearly, and each state has a forward transition to those beyond it; this leads to a number of transitions quadratic in the number of states. Introducing a silent state for each emitting state, which forwards on to either the next emitting or the next forward state, results in a number of transitions linear to the number of states.

HMMs are widely used in pattern recognition tasks, such as speech [101] and optical character recognition [113], part-of-speech tagging [60], melody classification [96] and crypt analysis [55]. This type of pattern matching task is very common in bioinformatics, and HMMs have been used for a wide variety of such tasks. These include pairwise sequence alignment [29], homology search [29] and gene finding [58]. Their advantage in these tasks is that they place the problem on a full probabilistic basis by asserting that the transitions and emission from states should describe distributions.

3.2.2 Key Algorithms

The key algorithms are designed to answer three questions of interest for real world applications. They all have similar properties – in particular, they are defined using a recursive function, and can take advantage of *dynamic programming*. Using a dynamic programming approach, we take a recursive function and store the results of each recursive call into a table, with each parameter as an axis in the table. In algorithms which require the repeated computation of sub problems, we can save time by using the stored result in the table, rather than actually implementing the recursive call. In practice, we tend to compute all entries in the table, in an appropriate order, before reading the result from the desired row and column. More background on these algorithms can be found in *Biological Sequence Analysis* by Durbin et. al. [29].

3.2.2.1 Viterbi

Problem: Given a sequence of observations, O, and a model, M, compute the likeliest sequence of hidden states in the model that generated the sequence of observations.

The *viterbi* algorithm provides an answer to this question. It makes use of the *principle of optimality* – that is, a partial solution to this problem must itself be optimal. It decomposes the problem to a function on a state and an output sequence, where the likelihood of producing this sequence and ending at this state is defined by a recursion to a prior state with a potentially shorter sequence.

We can compute $V(q, i)$, the probability that a run finished in state q whilst emitting the sequence $s[1..i]$ with the following formula.

$$V(q,i) = \max_{p:a_{p,q}>0} \begin{cases} a_{p,q}V(p,i) & \text{if } q \text{ is silent} \\ a_{p,q}e_{q,s[i]}V(p,i-1) & \text{if } q \text{ emits} \end{cases}$$

The result will be the *probability* of the sequence being emitted by this model. To

compute the list of hidden states, we will need to backtrack through the table, taking the last cell, determining the prior contributing cell, and recursing until we reach the initial state.

For models with silent states we assume, in this and other algorithms, that there are no cycles of such states. In the absence of cycles the algorithms are extended to handle silent states in a straight-forward way – the paths are finite, as the model itself is finite. Models with loops of silent states have potentially infinite length paths. They can, however, be computed by eliminating the silent states, computing the effective transition probabilities between emitting states [29]. This may lead to a rapid increase in the complexity of the model; it is usually easier to produce models without loops of silent states [29].

3.2.2.2 Forward & Backward

Problem: Given a sequence of observations, O, and a model, M, compute P(O|M), that is the probability of the sequence of observations given the model.

The *forward* algorithm performs a similar role to the Viterbi algorithm described above – however, instead of computing the maximally likely path it computes the sum over *all* paths. By summing over every path to a given state, we can compute the likelihood of the arriving at that state, emitting the given sequence along the way.

$$F(q,i) = \sum_{p:a_{p,q}>0} \begin{cases} a_{p,q}F(p,i) & \text{if } q \text{ is silent} \\ a_{p,q}e_{q,s[i]}F(p,i-1) & \text{if } q \text{ emits} \end{cases}$$

A similar equation can perform the same computation in reverse – starting from the start state with no sequence, we consider the sum of all possible states we could transition to. This is the *backward* algorithm, and the form is very similar to that of the Forward algorithm. The dynamic programming table would be computed in the opposite direction in this case – we start with the final columns, and work our way to the start.

$$B(q,i) = \sum_{p:a_{p,q}>0} \begin{cases} a_{p,q}B(p,i) & \text{if } q \text{ is silent} \\ a_{p,q}e_{q,s[i]}B(p,i+1) & \text{if } q \text{ emits} \end{cases}$$

Not only may this be used to compute the probability of a sequence of observations being emitted by a given model, it also allows the computation of the likelihood of arriving in a particular state, through the use of both the Forward algorithm, to compute the likelihood of arriving in this state, and the backward algorithm, to compute the likelihood of continuing computation to the final state.

3.2.2.3 Baum-Welch

Problem: Given a sequence of observations, O, and a model, M, compute the optimum parameters for the model to maximise P(O|M), the probability of observing the sequence given the model.

Clearly, the models are only as good as the parameters that are set on them. Whilst model parameters can be set by hand, more typically they are computed from a training set of input data. More precisely, we will set the parameters of the model such that the probability of the training set is maximised – that is, there is no way of setting the parameters to increase the likelihood of the output set. This is known as *maximum likelihood estimation* or MLE.

The solution can be determined easily if our training data includes the hidden state sequence as well as the set of observables: we simply set the parameters to the observed frequencies in the data for both transitions and emissions.

However, it is more likely that we *do not* have the hidden state sequence. In that case, we can use a MLE technique known as *Baum-Welch*. Baum-Welch makes use of the forward and backward algorithms described in the previous section to find the *expected frequencies* for $P(q,i)$, the probability that state q emitted $s[i]$. This is computed in an iterative

fashion, setting the parameters of the model to an initial value, and continually applying the formulae until we reach some prior criterion for termination. After each run of the forward and backward algorithm, we use the frequency counts maintained in $P(q, i)$ to compute the model parameters.

The emission parameters can be computed using the following formula:

$$e'_{q,\sigma} = \frac{\sum_{i:s[i]=\sigma} P(q,i)}{\sum_i P(q,i)}$$

Essentially, we are computing the number of occasions q emits character σ, divided by a normalising factor – number of times q emits any output.

The transition parameters can be set by considering the forward and backward results in the following fashion:

$$a'_{p,q} = \begin{cases} \frac{\sum_i F(p,i)a_{p,q}e_{q,s[i+1]}B(q,i+1)/P(s)}{\sum_{r:a_{p,r}>0} F(p,i)B(r,i)/P(s)} & \text{if } q \text{ emits} \\ \frac{\sum_i F(p,i)a_{p,q}B(q,i)/P(s)}{\sum_{r:a_{p,r}>0} F(p,i)B(r,i)/P(s)} & \text{if } q \text{ is silent} \end{cases}$$

3.2.3 HMM extensions

There are a number of interesting extensions to the simple HMM structure previously described. An obvious extension is to provide multiple-tapes; that is, each state can emit to multiple sequences in the same step (with the option of a blank emission). This allows the construction of more complicated examples e.g alignment of multiple sequences.

A second, relatively common, extension is to support a form of higher-order HMM which relaxes the Markov constraint from depending on the previous state to depending on the previous n states, for some fixed n. In practice, we may implement this as a form of look-ahead, where the emission depends on the next n elements in the sequence. One reason for considering such an extension is to model *codons* – sets of three DNA bases – which are important in many biological applications (e.g gene-finding).

3.2.4 Example applications

A wide-range of biological statistical applications may be framed using HMMs; they are sufficiently well studied to be adapted to many different types of problems. Examples of these four different kinds can be found in *Biological Sequence Analysis* [29].

3.2.4.1 Pair Hidden Markov Models

Pairwise alignment can be placed in a fully probabilistic setting by using a form of Hidden Markov Model known as a *pair HMM*. This HMM differs from that described above by including multiple output sequences – each state (or transition, depending on whether the model represents a Moore or Mealy machine) may emit characters to multiple sequences.

A pair HMM allows output to two sequences at once, which allows the model to emit an alignment. We model the process of aligning the sequence by providing a state for each alignment method e.g substitution, insertion and deletion. The probabilities can be set using observed data; the final score also represents the likelihood rather than an arbitrary score as in the standard models. An extended model allows for local alignment by including states for modelling random emission before and after the alignment states. [29]

3.2.4.2 Multiple Alignment Hidden Markov Models

We can extend the pair HMM to support multiple alignment by providing support for *multi-tape* (e.g multi-sequence) HMMs. These permit each state to emit to multiple different output tapes. In multiple sequence alignment, each tape represents a different sequence.

3.2.4.3 Profile Hidden Markov Model

A common case in bioinformatics is to have a group of related DNA or protein sequences, perhaps discovered empirically, and to subsequently find other related sequences in an existing database. Pairwise aligning each candidate with all the sequences in the family is expensive; we instead want to identify the statistically important features of that family.

A *profile hidden markov model* is a type of HMM that describes a family of sequences as a *profile* [29]. A profile models the likelihood of a given sequence corresponding to the described family, by performing an alignment to that profile.

3.2.4.4 Gene Finder

Gene finding or gene prediction is the process of identifying genes in regions of DNA. They are often described using Hidden Markov Models, which are trained to recognise statistical features of particular genes, and used for likelihood estimation and prediction.

3.3 RNA Secondary Structure Prediction

As well as having a primary structure (e.g the sequence of bases of the RNA), RNA also interacts with itself to create a *secondary structure* by pairing bases across the sequence. Interesting RNA sequences often *conserve* their secondary structure more than they conserve the primary sequence. That is, changes in the sequence may occur more frequently than changes in the secondary structure. Furthermore, the secondary structure of RNA may affect biological functions elsewhere in the system, for example as a catalyst or in genetic regulation. Thus, predicting the secondary structure from the primary structure can be an important task.

A simplistic approach to secondary structure prediction is to select the structure which maximises the number of base pairs. This is known as the *Nussinov* [87] algorithm, and is a simple, recursive algorithm for finding the maximum number of base pairs:

$$N(i,j) = \begin{cases} 0 & \text{if } j <= i+3 \\ \max \begin{cases} N(i+1,j) \\ \underset{\substack{i+3<k<j \\ \text{bp}(s[i],s[k])}}{max} \{1 + N(i+1,k-1) + N(k+1,j)\} \end{cases} & \text{otherwise} \end{cases}$$

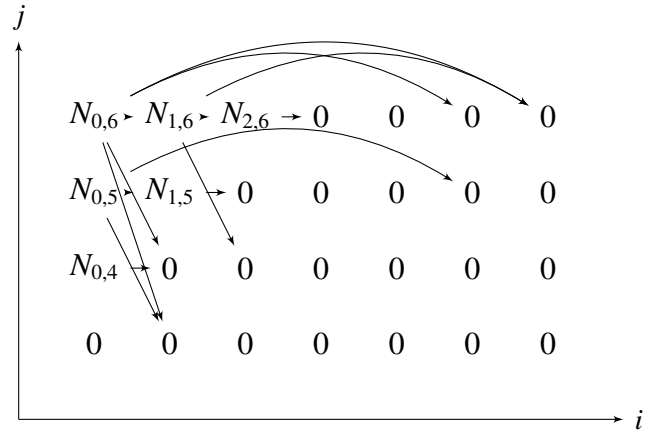


Figure 3.3: The dependencies in the Nussinov algorithm for a 6 by 6 example. Rows 0-3 have been eliminated for clarity, and those that do not recurse are simply assigned zero. Importantly, this exhibits a *diagonal* dependency pattern – elements only depend on those which are below and/or to the right of them.

.

The condition $j <= i + 3$ ensures that each base-pairing is separated by at least three other bases, enforcing a physical constraint observed in the real world, and, additionally, fills in the base-case cells of the table. If there are at least three bases between i and j, we compute maximum number of base-pairs between i and j by computing (a) the maximum amount by pairing i with nothing – in the $N(i + 1, j)$ case and (b) the maximum number by pairing i and k for each k between $i + 3$ and j.

Unfortunately, this method is insufficient when considering real world RNA secondary structure. There are a number of factors that this approach ignores; firstly, secondary structures often pair bases into a *stack*. Furthermore, we have tacitly assumed that base pairs cannot overlap. These are known as *psuedoknots*, and do not form a constraint on real RNA secondary structures. We can extend our recursive approach to consider maximising the number of base pair stacks; pseudoknots, however, require a different approach since recursive descent cannot represent such overlapping entries.

A different approach altogether is to optimise not the number of base-pairs, but the more biologically accurate value of *free energy*. This energy minimisation approach, embodied in the Zucker [124] folding algorithm, assumes that the correct structure is the one with the

lowest equilibrium free energy. This is approximated by summing the individual energy contributions of secondary structure elements (e.g loops, base pairs). Once again, this algorithm can be described using a (somewhat more complicated) recursive algorithm, but for reasons of space and complexity we will omit the description. Accurate prediction using this method is dependent on accurate experimental figures for the energy parameters.

3.4 Stochastic Context-Free Grammars

Hidden Markov Models belong to the class of *regular grammars*. As a consequence, applications in bioinformatics that require palindromic or context-sensitive features, such as RNA Secondary Structure prediction, cannot be modelled by HMMs.

In the same way Hidden Markov Models are *stochastic regular grammars*, we can create a stochastic equivalent to a context-free model, known as *stochastic context-free grammars* or SCFGs. We can also create equivalent algorithms for the three applications; the *CYK* algorithm for Viterbi, and the *inside-outside* algorithm for Forward-Backward. Like HMMs, the SCFG algorithms (CYK and inside-outside) are defined using recursions (for details of the exact algorithms, see *Biological Sequence Analysis* by Durbin et. al. [29]). The existence of these algorithms enables the description of problems such as RNA Secondary Structure Prediction as parsing problems over fixed grammars.

3.5 Phylogenetic Trees

Tree structures play an important role in bioinformatics applications. A *phylogeny* is a binary tree structure representing the evolutionary relationship between species. Interesting applications include tree construction and progressive alignment using guide trees.

3.5.1 Example: Weighted Parsimony

A classic bioinformatics problem over trees is *weighted parsimony*. In this problem we are given phylogeny with characters observed on each leaf only. The goal is to find an assignment of *characters* to each internal node, such that the sum of the cost of changes between nodes is minimised. We can use this as an operation in building trees, by choosing the tree with the lowest cost.

Assuming we can refer to left and right children of a node u using $\mathrm{L}(u)$ and $\mathrm{R}(u)$, and d is a substitution matrix, the lowest (most *parsimonious*) score for assigning character σ to node u is

$$w(u, \sigma) = \begin{cases} 0 & \text{if observed character is } \sigma \\ \infty & \text{if observed character different from } \sigma \end{cases}$$

if u is a leaf node and

$$w(u, \sigma) = \min_{\tau \in \Sigma} \{w(\mathrm{R}(u), \tau) + d(\sigma, \tau)\} + \min_{\tau \in \Sigma} \{w(\mathrm{L}(u), \tau) + d(\sigma, \tau)\}$$

if u is an internal node.

This example contains features common to many tree based algorithms – in particular recursion over the children of the tree.

3.6 Existing Implementations

In order to inform our designs for DSLs for bioinformatics, it is worth considering the existing applications in the area, what strengths they have and what weaknesses may be addressed by using a DSL. To that end we will take a brief survey of some of the most commonly used applications in bioinformatics.

The most frequently used applications in practice are those dealing with sequence align-

ment and homology search tools. The most popular are BLAST [5] for DNA and protein homology search, FASTA [94] for protein alignment and ClustalW [115] for multiple sequence alignments, which all use score based methods for alignment and search. MUMmer [27] is another score based tool for sequence alignment, based on suffix trees. HMMeR [30] is an equivalent statistical tool for alignment and search built on profile HMMs, which has eventually been widely adopted after initial concerns that statistical methods were too slow in comparison to older scoring based methods. This is a recurring pattern in bioinformatics; more sophisticated approaches are often trumped by simpler, higher performance applications. This is primarily because of the size of the data sets.

Hidden Markov models have great utility beyond simple profile and alignment tasks. HMMoC [72] is a Hidden Markov model compiler that takes a high-level definition of a HMM and generates an efficient C++ implementation of the three classical algorithms. Performance on profile HMM applications is on a par with HMMeR 2.0, but it provides increased flexibility for describing different types of models.

One problem with the software described above is that they have limited scope; applications cannot be customised or extended with great ease. Algebraic Dynamic Programming [36] mitigates this customisation problem by building a domain-specific language for describing dynamic programming problems in the form of searches over context-free grammars. This context-free approach makes it particularly suitable for describing RNA Secondary Structure prediction applications. One problem with the approach as it stands is that it is difficult to include more sophisticated applications that require extensive data models, and the limitation of sticking to context-free grammar applications; both in form and function this may place restrictions on the way algorithms are described. Markov Chain Monte Carlo (MCMC) is another important technique in bioinformatics. StatAlign [86] is an example of software that uses MCMC for multiple alignment and is heavily customisable.

Another problem with the state-of-the-art is that individual applications for different

application areas are hard to combine. A recent project that has tried to mitigate these problems by providing a workspace for bioinformatics software development is Unipro UGENE [92]. This allows the user to combine any number of tools into a *workflow* in order to automate tasks. Whilst this works well at coordinating existing applications, it does not aim to provide the ability to define new algorithms in the existing data. BioJava [97] is another example of an integrated framework, a Java library providing functions for sequence alignment, phylogenetic tree creation, classification and sequencing. Whilst this provides more flexibility in defining combinations of algorithms, it comes at the expense of performance.

3.6.1 Massively Parallel Implementations

The focus on high-performance applications has led to a number of forays into the parallel processing world. However, massively parallel hardware, such as GPUs, have faced a slow adoption period in this domain – pioneering work has occurred but wide-scale use is still distant.

An early piece of work [67] used OpenGL to implement Smith-Waterman using a method of *diagonalisation*, where all values on a diagonal in the dynamic programming table are computed in parallel. In practical terms, this was achieved by indenting each row using an offset that forced the independent cells to appear in the same columns.

CUDA-SW++ [69], a CUDA-based Smith-Waterman program, uses a similar diagonalisation technique, which the authors describe as *intra-task* parallelism (task-per-block), where multiple threads work co-operatively to compute a single table. They provide an alternative in *inter-task* parallelism (task-per-thread), where each thread works independently on a problem – in this case a single table per thread. The paper found that inter-parallelism was more efficient, but intra-task better supported longer sequences. SW-CUDA [76] is an early CUDA Smith-Waterman implementation that took an inter-task parallelism approach, evaluating each table column by column using a single thread. CUDA-SW++ itself takes a

hybrid approach; shorter sequences are computed using inter-task parallelism, and longer with intra-task. A follow on paper [71] proposed improvements to the technique using a vectorised approach.

Given the popularity of existing score-based alignment tools, it is no surprise that concerted effort has been made to port them directly to GPUs. Examples include GPU-BLAST [120], which uses a sequence-per-thread approach, GPU-ClustalW [68], an OpenGL based port of ClustalW and MSA-CUDA [70], a CUDA port of ClustalW which uses both inter-task and intra-task parallelisation on similar lines to CUDA-SW++. MUMmerGPU [105] is another sequence alignment program ported to the GPU, reporting 10-fold speed-ups.

Needleman-Wunsch has been used as one of a number of case studies comparing implementations on FPGAs, GPUs and CPUs, with researchers finding the GPU to be more efficient that CPUs, and significantly easier to develop than optimised FPGA implementations [23]. This result was corroborated through the development of SWCUDA, a Smith-Waterman algorithm for GPUs, and subsequently comparing against BLAST and SSEARCH, finding performance improvements of 2 to 30 times faster [76]. Both SWCUDA [76] and CUDASW++ [69] found that ordering the sequences by length improved inter-parallelism performance. Algebraic Dynamic Programming has also been ported to the GPU architecture [107], which takes a slightly different intra-task parallelism approach. Each element in a diagonal is computed simultaneously, with the loop over diagonals executed on the host.

As well as cost based methods, statistical methods have also received some GPU parallelisation effort. Hidden Markov models have been considered, first with cuHMM [66]. HMMeR was first ported to a streaming GPU implementation in Claw-HMMeR [47] using the BrookGPU language for AMD cards and improved upon in GPU-HMMeR [121], using a sequence-per-thread approach. More recently, a hybrid sequence-per-thread and sequence-per-block (intra-task parallelism) port has achieved further speed-ups [35].

3.7 Challenges and patterns in application development

The survey of the existing literature has led us to identify some challenges and patterns in application development in bioinformatics, which will inform the construction of our DSL in the next chapter. A first observation is that the most popular, widely used applications are for very narrow domains (e.g BLAST, ClustalW, HMMeR, etc.). However, what they lack in flexibility they gain in outright performance. Furthermore, when they have been ported to the GPU, they have seen significant speed-ups, suggesting this an area ripe for GPU development. An ideal GPU DSL would provide performance similar to those customised options with much greater flexibility.

A further problem with this highly-focused approach is that customising these applications can be hard work; variations on these algorithms can be hard to implement due to extensive optimisations in the existing code base. Furthermore, the background of the users can be extremely varied; bioinformatics is a cross discipline domain with expertise from the life sciences, statistics and computer science. End-users may therefore not have the requisite programming ability (or inclination), and the idea of customising these applications would be a daunting one.

The idea of using a DSL for bioinformatics problems is an under-utilised one; a few prior attempts (e.g HMMoC and Algebraic Dynamic Programming) have been successful in encoding and executing algorithms, but not widely adopted. There may be a few reasons for this; complexity of the input language, lack of flexibility in the type of programs accepted, and lack of performance, in the case of HMMoC. A suitably designed language might mitigate some of these issues, bringing together a large number of application areas.

Exploring the case studies, we can make some clear observations of common patterns. Firstly, recursive functions, amenable to dynamic programming, are clearly the primary way to describe existing algorithms in the cases we've considered. Furthermore, these functions often recurse over structures, such as statistical models (e.g HMMs, SCFGs) or data representations such as trees or substitution matrices. The key features of the functions

describing these applications are:

- Recursion over indices of sequences, or elements of a model.
- Optimising (e.g sum, max and min) over explicitly specified and ranged elements.
- Structures such as trees, sequences on biological alphabets (e.g DNA, RNA, protein), graph and networks.
- Exploring related elements of a structure feature (e.g children of a tree node, or transitions in a HMM).
- Important types include probabilities, sequences, indices on sequences, characters and alphabets (DNA, RNA, protein).

We should note here that we do not attempt to cover all potential bioinformatics applications under this umbrella – primarily what is known as *classical bioinformatics*. More advanced statistical techniques (such as Markov Chain Monte Carlo) are not amenable to this approach; they are not easily described as recursive applications, and so trying to find a unifying language is perhaps unwise. Building a DSL is as much about what is left out, as what is left in, and our choice of domain covers a broad spectrum of bioinformatics applications.

In future chapters we will, for brevity, consider "bioinformatics" as a consistent domain made up of applications that fit our observed criteria e.g classical bioinformatics, those applications defined by recursive functions.

Chapter 4

An extensible approach to Domain-Specific Languages

We all encounter domain-specific languages on a regular basis, whether we appreciate it or not. From web tools such as HTML and CSS, database query languages such as SQL, regular expressions for pattern matching and TEX for publishing, they have become a vital part of our language toolkit. The majority of DSLs embrace the following tenet: by providing a focused or limited language, we can ease development in that domain. When used effectively, the technique renders the language virtually invisible, by allowing users to describe their problems seamlessly using the primitives of the domain.

Some DSLs have a further aim; to provide an efficient implementation in a domain where such a thing is either time-consuming or difficult to produce. Examples here include regular expressions and parser generator tools, such as YACC and its derivatives. This approach is particularly relevant when the market for the domain goes far beyond those who would be able to program it. Regular expressions are a excellent example of this; if you were forced to write a string matcher by hand every time you needed some simple pattern matching, it would certainly restrict the proper use of that domain, and lead to fragmented and incompetent attempts to recreate the functionality using string matching

and other torturous tricks.

Of course, regular expressions are a part of many programmer's toolkits, no matter which domain they reside in. The most demanding users of computing resources today, however, are typically those in specialised areas such as the sciences, engineering or finance, many of whom have no formal background or experience in computer science or without programming expertise. It is a rare person who combines both the technical aptitude and the domain knowledge required to produce high-quality, efficient solutions. A recent survey, reported in the journal Nature, of 2000 scientists in a variety of disciplines found that the majority, when questioned, were either not familiar with standard software engineering concepts or considered them unimportant [82]. These are users who want to, and should be able to, put all their focus into the problems of the domain rather than wrestling with tough-to-write-for architectures. This is particularly relevant when we consider the difficulty involved in programming the latest high-performance architectures, such as multi-core CPUs, clusters and especially GPUs.

Thus, in this chapter we come to the central thesis of this dissertation; that DSLs provide a way of harnessing massively parallel hardware in an effective manner, opening GPUs specifically to a much wider community.

4.1 Why a DSL?

The key benefits of developing a domain-specific language for a problem are:

- Programs in that domain have a reduced cost of development.
- The language can provide a fluent interface that matches the domain experts' understanding.
- A single program can target multiple architectures with no extra work on the part of the end-user.

- The compiler can use the domain specific knowledge to produce efficient implementations.

- These combined benefits should lead to increased longevity of programs.

A large amount of research has been completed on the productivity benefits provided by developing a DSL. In this dissertation, the focus will instead be on the construction of efficient DSLs. The contention is that by narrowing the domain, the compiler can make deductions and assumptions[1] that are impossible in a general purpose language, and use them to generate efficient code with minimal effort on the part of the domain user.

4.2 Identifying a domain

The art of developing a successful DSL is as much about the choice of the problem domain as it is about the nuts and bolts of the language. A clumsy, over-reaching DSL will likely be difficult to implement and difficult to use. Identifying a suitable problem domain is key to making the most efficient use of development time.

The essence of a DSL is a set of *primitive* operations that represent concepts in the domain which can be combined to great effect. These operations may represent concepts that are computationally complex, but are considered as building blocks for applications in the domain. This will be most effective with a small number of primitives, reducing the learning curve.

Our first criterion for a suitable domain is therefore that it must be representable by a limited set of clear and concise primitives. The advantages are that the DSL becomes fluent[2], clear and easy to learn for experts in the domain.

[1] Deductions are properties derived from the domain and could, for example, take the form of identifying, dependencies between components automatically, which is only possible because the domain is restricted. Assumptions take the form of optimisations we take based on our knowledge of the input – for example, we might assume in bioinformatics that we are dealing with small alphabets, or smll functions, and optimise accordingly.

[2] A fluent language for a domain is one in which writing programs is natural for a domain-expert – in other words, the choice of the primitives, and their syntax, is such that it is generally obvious how to write

A DSL for a high-performance architecture such as a GPU naturally has higher cost of development than a simple DSL wrapper around a library or model. Development requires a significant amount of knowledge of the target architecture. As such, and in contrast to a standard DSL, an important criteria for selecting a domain for a GPU DSL is that it requires a *high-performance* implementation. Creating, say, a GPU DSL for typesetting would be uneconomic because standard DSL techniques will probably suffice. We'd much prefer to use simple, existing, high-level techniques for construction that sacrifice performance for usability.

Furthermore, an ideal domain is amenable to reuse. This is an important consideration for any DSL, but particularly for GPU DSLs because the cost of development is so high. This cost must be amortised across many different applications in order to be successful.

4.3 Implementing DSLs: approaches and problems

When implementing a domain-specific language we are faced with a plethora of options, from low-cost solutions in existing languages, to full-blown compiler stacks. Each option has a set of trade-offs, and so different domains and circumstances may lead us to adopt one scheme over another

In this section we are specifically concerned with the applicability of these techniques to GPU DSLs, in particular for broad scientific domains. By exploring the existing techniques, we can motivate the decisions on the design and implementation of a framework for building GPU DSLs. For a more general focus on the trade-offs between different implementation strategies, see the comparative study "When and how to develop domain-specific languages" by Mernik et al. [83].

any given application in that domain, and it is straight-forward to understand the meaning of the program from the text itself.

4.3.1 Library

By far the simplest way to develop a domain-specific language is to construct a *library* in an existing language. This approach is tempting, because it provides many of the advantages of a DSL at a fraction of the cost. For example, programs in the domain still have a reduced cost of development, can target multiple architectures and generate efficient applications. In contrast to a true DSL, all domain creation development is within the existing language, and no compiler knowledge is required to develop the "language".

Given these advantages, why should we consider using a full DSL? Firstly, a DSL provides a level of custom syntax that can be difficult to achieve when using a general purpose language, with general purpose syntax. Another difference is that a DSL typically provides access to some form of representation of the program to the creator of the DSL, providing more opportunities for advanced analysis techniques such as re-ordering of instructions and global analysis. This is particularly important for developing GPU DSLs, where the high-level language will typically be a long way from the low-level implementation.

The latter part can be addressed by an *active library*, which provides a standard library interface, but actively modifies or adapts the application in order to produce highly optimised code. OP2 [84] is an excellent example of this approach, in which source-to-source translation is used to generate highly optimised code for a number of architectures, notably CUDA. This is part way to a DSL; the important distinction is that it does not attempt to include custom syntax, relying instead on standard calling conventions to describe the API.

4.3.2 Internal DSL

If a library proves insufficiently flexible, then an alternative might be to develop an *internal* or *embedded* [48] DSL. These differ from library implementations in that they go beyond the standard library calling mechanisms of the language in order to create a clear syntactic, interface. The idea is that we can reuse (some might say abuse) the primitives and libraries of the host language in order to produce a clear and effective syntax.

There are various methods for implementing internal DSLs. Firstly, the syntax of the language may be flexible enough to support the desired semantics of the problem concisely. This may still not be enough to have a fluent interface. In more advanced languages, we may be able to provide custom syntax, or reuse sufficiently flexible syntax from other parts of the language. Unfortunately, it can be difficult to perform high-level analysis and optimisation on such DSLs, because there is no internal representation of the DSL programs to manipulate.

A number of different implementation methods have been suggested in order to avoid this performance penalty. Active libraries, particularly with source-to-source translation, may overcome these issues when combined with an internal DSL. Compile-time metaprogramming techniques are a popular method for implementing internal domain-specific languages; Template C++ and Template Haskell [106] have both been used to generate efficient implementations of (restricted) high-level interfaces. However, they remain static representations and cannot provide dynamic optimisations.

Partly in response to the perceived inefficiency of standard embedded DSLs, *multi-stage languages* [112], such as Lisp, MetaOCaml [41] and MetaHaskell [74], have been an significant area of research for internal DSLs. A *staged* representation of the DSL program is provided that can be used to generate code for multiple compilations targets. In such a suitable language, using a suitable method of embedding (e.g Scala/Delite [20]), we may even be able to access an abstract syntax tree or other representation of the program in order to perform a deeper level of analysis and optimisation.

For our purposes, this approach has a number of issues. Common languages amongst scientific domain experts often provide poor support for flexible DSL development. In bioinformatics the lingua francas are Java, Python and C/C++ – none of which have extensive support for syntactically fluent internal DSLs. Languages such as Scala, Lisp, MetaOCaml and (Meta-)Haskell all provide excellent ways to develop internal DSLs, but the adoption of such languages puts a heavy requirement on the end user to learn the tools and

techniques for that language. Furthermore, constraints may be placed on the syntax and semantics of the embedded DSL, and they may adopt quirks of the host language, which can be hard for a domain expert to fathom.

4.3.3 External DSL

Perhaps an external DSL provides a better choice? The clear advantage is that there are no constraints on the syntax of the language or the evaluation/code generation steps. The obvious problem is the increased cost of development. In a large domain, this increased cost is multiplied across each new DSL created. Furthermore, the increased cost takes it away from the capabilities of experts in the domain; development will therefore be limited to compiler experts.

Another problem facing external DSLs is scalability and composition. In an area such a bioinformatics where a cluster of DSLs are required to effectively cover the domain, it is vital that each DSL focuses on a single cross-cutting area, and that these can be combined effectively to solve different application problems. Repeated, single DSLs are wasteful, repeating effort and decreasing overall quality. Furthermore, either of these approaches still requires domain developers to have a close familiarity with the underlying architecture if they hope to develop an efficient DSL.

4.4 Our approach

Whatever implementation method we choose, the cost of developing a GPU DSL is high, and therefore we must maximise the return on our investment. Furthermore, there are domains for which GPUs would be helpful but the domain is too broad for a single DSL (e.g bioinformatics).

In order to address these concerns, and mitigate some of the problems associated with the implementation strategies described in the previous section, we have developed a hy-

brid, *incremental* approach to DSL development. Rather than building a single DSL, we cover a broad domain using a *family* of co-operating DSLs, built on top of a core host language – "DSLs within a DSL". Each DSL should be orthogonal, addressing a small, clearly defined aspect of the larger domain. By choosing orthogonal DSLs we can combine the results in effective ways.

This is achieved by developing a custom language and system for hosting and combining these DSLs. The host language provides tools and language features that may be reused across all domains hosted on that language. Typically this will be a language for describing algorithms within the domain. We can subsequently focus effort on generating an efficient parallel implementation from this language; thereafter, any further DSLs may reuse this initial domain and achieve the parallelisation for free.

The consequence of this design is that we can distinguish two different types of user for the system:

- A domain developer – they create extensions to support a specific domain. They use the framework to define the syntax and semantics of their extension. These extensions can add new primitives to the host language. They may provide library routines written in the host language. In addition to expertise in the domain itself, a basic background in Computer Science is advantageous – in particular, basic knowledge of the standard compilation steps (lexing and parsing).

- A domain user (expert in the domain) – they use the host language and extensions to develop their applications, reusing library routines or writing their own applications.

We describe this as a hybrid approach because it combines aspects of both external DSL development, in that we develop a custom host language, and internal DSL development, in that we embed new DSLs into this host language. We therefore get the advantages of an external DSL (custom syntax, full control of the host language for optimal code generation) with the advantages of an internal DSL for incremental domain developers (low-cost

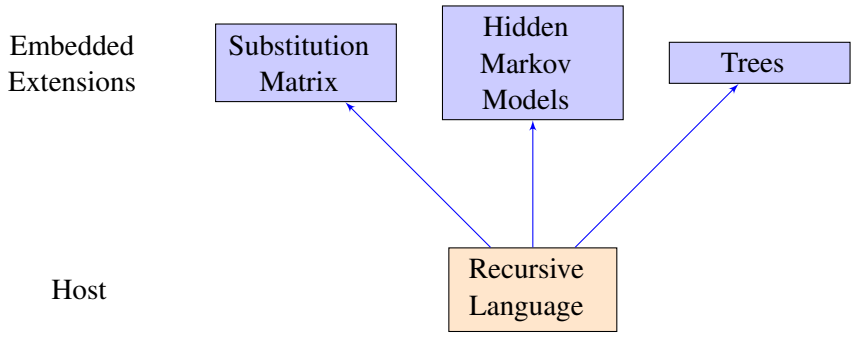


Figure 4.1: Overall system design.

development). Furthermore, the system can insulate the domain developer from the complexities of GPU development by encoding their algorithms through the base language; domain developers are only concerned with how to represent their area of expertise in the language.

As a consequence of these advantages, our domain developers are not limited to those with extensive compiler experience. Once the initial domain and extension support is constructed, new extensions can be developed cheaply and effectively. The rest of this chapter will be focused on how we can reduce the cost of domain development so that it is feasible for domain experts to create GPU DSLs. The next chapter will focus on how an exemplar host language, and language extension mechanism, for bioinformatics can be leveraged to generate efficient GPU code.

This incremental approach to DSL development shares much with existing techniques in embedded language development. The key difference is that the host language is, itself, a DSL, designed specifically to describe applications in the domain and to be easily parallelisable. In comparison to hosting the language in a general purpose language such as Scala, the constraints that our focused host language imposes greatly simplify the parallelisation, as we shall see in the next chapter.

4.4.1 Implementation for bioinformatics

Our analysis of bioinformatics as a domain, described in Chapter 3, leads us to conclude that many algorithms in bioinformatics are best described using recursions over models or structures. We adopt the incremental approach described in the previous section by developing a pure, functional language for describing recursions as a host language, and an extension system that focuses on models and structures (see Figure 4.1). The intention here is that end-users write their algorithms in the functional host language using language extensions provided by domain developers.

The extension system takes the form of the development of embedded DSLs representing *models*, which are defined as sets of related components along with associated methods of exploring those components. These may include linked or nested components, and components with data. Examples from the previous section include statistical models, such as Hidden Markov Models and Stochastic Context Free Grammars, as well as commonly used structures such as trees or substitution matrices.

For domain-development, we provide an *extension toolkit* for extending the compiler. This toolkit simplifies the process of developing new extensions for the domain by providing extension points for describing new syntax, and tools for generating GPU code. The idea is that the toolkit should permit willing domain experts to rapidly build efficient and fluent domain extensions. The toolkit provides two specific areas of extension

- Model definitions – Support for creating new model definitions. This includes creating new syntax for the model definition, for defining how the model is represented on the device, and automatic analysis tools that make some GPU representation decisions (e.g memory location) automatically.

- Host language – Extension mechanisms for the host language. This includes new syntax for exploring models, new model types for iterating over and accessing models, as well as GPU code generation tools.

$$
\begin{array}{lcl}
Stmt & ::= & varname = funcname\ (Arg*) \\
 & | & varname = \mathsf{map}(PArg) \\
 & | & varname = \mathsf{load}(type, filename) \\
 & | & Func \\
 & | & \mathsf{print}\ varname \\
 & | & \mathsf{define\ alphabet}\ name\ string* \\
Arg & ::= & varname \\
 & | & string \\
 & | & integer
\end{array}
$$

Figure 4.2: The grammar for the imperative part of the host language describing the permitted operations

4.5 A tour of the host language

The host language is developed with two aims in mind; firstly, it must be easy to use for those familiar with tools and languages used in bioinformatics. Secondly it must be easy to extend, providing hook points for extension that give rise to a large number of potential applications.

The language is a pragmatic mix of pure function definitions for describing recursions, composed using an imperative language. This system will be familiar to those working in bioinformatics where imperative "scripting" languages are widely used and recursions are dealt with commonly.

4.5.1 Scripting statements

Programs are composed of a series of statements that are evaluated in order. For reference purposes, a simplified grammar[3] describing the syntax is provided in Figure 4.2. The primary method of execution is through the definition and execution of functions. Function definitions are considered as statements; functions must be defined prior to use and may be redefined later in the script. Function definitions consist of a body and a set of parameters, where the body is an expression as described in the next section.

[3]For the interested GPU or domain expert, a *grammar* describes the syntax of a language; that is the set of syntactically valid programs for the language. Each rule defines how the syntax is decomposed, e.g function calls are statements, strings and integers may be arguments, and so on. For a detailed introduction consult a compilers text book [4, 34].

Functions may be called directly, or over a list of parameters – the latter uses the map keyword to describe a parallel map over a set of values. Arguments to map use a cross-product operator (*) to combine multiple parameters. This is primarily used with variables assigned from the load operator for loading data onto the device. For example, assuming we have a function prototype f(model t, seq x), then

```
seqfile = load("protein.db")
result = map(f, mymodel (*) seqfile)
```

first loads a set of sequences onto the device, then runs the function f for each sequence in seqfile paired with mymodel.

Finally, we may define new alphabets in order to provide options for users working with different sets of characters, such as a DNA sequence or protein sequence, and print variables using the print statement.

4.5.2 Defining functions

$$
\begin{array}{lcl}
\mathit{Func} & ::= & \mathit{funcname}\ \mathit{Param}*\ \mathit{Expr} \\
\mathit{Expr} & ::= & \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} - \mathit{Expr} \mid \mathit{Expr} \times \mathit{Expr} \mid \mathit{Expr}/\mathit{Expr} \\
 & \mid & \mathit{Expr} < \mathit{Expr} \mid \mathit{Expr} > \mathit{Expr} \mid \mathit{Expr} == \mathit{Expr} \mid \mathit{Expr}! = \mathit{Expr} \\
 & \mid & \mathit{Expr}\ \mathit{min}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{max}\ \mathit{Expr} \\
 & \mid & (\mathit{Expr}) \\
 & \mid & \mathit{Var}[\mathit{Expr}] \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{funcname}(\mathit{Expr}*) \\
 & \mid & \mathit{integer} \\
 & \mid & \mathit{if}\ \mathit{Expr}\ \mathit{then}\ \mathit{Expr}\ \mathit{else}\ \mathit{Expr} \\
 & \mid & \mathit{sum}(\mathit{var}\ \mathit{in}\ \mathit{Expr}\ :\ \mathit{Expr}) \\
 & \mid & \mathit{min}(\mathit{var}\ \mathit{in}\ \mathit{Expr}\ :\ \mathit{Expr}) \\
 & \mid & \mathit{max}(\mathit{var}\ \mathit{in}\ \mathit{Expr}\ :\ \mathit{Expr}) \\
 & \mid & \mathit{Expr} \ldots \mathit{Expr} \\
\mathit{Var} & ::= & \mathit{varname} \\
\mathit{Param} & ::= & \mathit{Type}\ \mathit{varname}
\end{array}
$$

Figure 4.3: The grammar for the function part of the host language. Binary operations are all left associative, with the arithmetic operators having the standard precedence.

```
int d(seq[en] s, index[s] i, seq[en] t, index[t] j) =
   if i == 0 then
      j
   else if j == 0 then
      i
   else if s[i - 1] == t[j - 1] then
      d(i - 1, j - 1)
   else
      (d(i - 1, j) min d(i, j - 1) min d(i - 1, j - 1)) + 1
// Function can be called like so
// Note, only the sequence parameters are required
result = d(s,t)
// ‘‘result’’ represents the full dynamic programming table
// printtable prints the full result
printtable result
```

Figure 4.4: The source code of a simple edit distance recursion

Functions are defined using a pure, functional language – no operations have side effects – which contains common arithmetic operations, basic conditional expressions and support for recursive calls. The grammar describing the syntax for defining new functions is provided in Figure 4.3, and an example of a function for the edit-distance problem is given in Figure 4.4. We provide only an informal definition of the semantics here; beyond noting that all expressions are pure, the precise semantics are not vital to either understanding the language or the parallel analysis.

The majority of applications in bioinformatics are based on analysis of sequences of characters, and so we provide a *sequence* primitive type which can be queried by index using `Var [ Expr ]`. No further operations are provided on sequences – thus, the sequences are immutable. In particular, recursion on sub-sequences is only supported through indices in our pure language.

The language provides a standard *if* expression, which works as in other functional languages, evaluating the conditional as a boolean and returning the result of the relevant branch. Boolean operators are provided for numerical comparison and for equality comparison on characters.

The language has a looping mechanism which combines over *iterable* elements. The default provided operations are *sum*, *max* and *min*, which perform their respective combination operators. The argument to these combination operators can be any iterable expression. The only default iterable expression is the *integer iterator*, represented using the ... operator. This works by providing an inclusive range, given by a start and end value, e.g. 1...4 would iterate from 1 to 4 inclusive.

Recursive calls can only be made to named functions – no higher order or first class functions are allowed. This limitation ensures that the language remains simple for programmers unfamiliar with functional programming, and ensures the analysis remains straightforward. Furthermore, bioinformatics applications do not typically need such expressive power.

By constraining the allowed functions we permit a tractable analysis – we do not preclude an analysis of a more comprehensive definition. We simply take a pragmatic approach to support the most commonly found forms for scientific applications.

4.5.3 Standard library

We provide a number of pre-defined values in a *standard library*. This includes common alphabets – DNA, RNA and protein alphabets – as well as bundled functions for extensions, such as the HMM algorithms, and pre-defined matrices (e.g BLOSUM50).

4.5.4 Type system

Our base language has a simple type system which includes the following primitive types: integers, constants, characters, sequences, indices on sequences, floats, probabilities, booleans and alphabets. We justify the introduction of a probability type as well as a general floating point type by the frequency with which bioinformatics applications use probabilities. By introducing a high-level type, we can determine an appropriate low-level representation. For example, underflow is a frequent problem when multiplying small floating point prob-

abilities. We can counter this by converting probabilities to log space, where we can use addition instead of multiplication. Alternatively, we could introduce a custom extended exponent implementation. Either option is possible in our implementation.

Alphabets define a set of characters; sequence and character types are specified with reference to a particular alphabet. In the edit distance example (Figure 4.4), sequence s and t refer to the English alphabet, denoted by *en*. Indices specify the sequence to which they belong; this is necessary so we can determine the dimensions of the recursive problem for analysis and tabulation. In the edit distance example, index *i* references string *s* in this way, as does *j* with string *t*.

To support recursive functions with some invariant parameters, we introduce two classifications – calling and recursive. These classifications are baked in to the compiler – with each type belonging to either, neither or both. For a type to occur in the parameter declaration of a function, it must be either a calling or recursive type. The classifications are:

- *Calling* type – a type is a calling type when it must be instantiated before the problem can begin, and remains *constant* over a single recursive run.
- *Recursive* type – a type is a recursive type when it must be specified each time we recurse - it is therefore *varying*. The prototype for a recursive call is simply all the parameters whose types are annotated as recursive.

Sequences are a calling type, so in the edit distance example the strings (the sequences) *s* and *t* are defined once and remain constant over a single recursion, while we vary the indices *i* and *j*.

We have included two types in the framework that can act as recursive parameters – integers and explicit indices on sequences. Types may be both calling and recursive – for example, integer types, where the initial value determines the size of the domain. This is because the domain of all parameters must be finite for the method to be applicable, so

we must specify an initial value for the integer domain. In the case of indices, they are naturally bound by the size of the sequence they reference.

We use this finite nature to our advantage to state that all recursive types must define a mapping between elements in their domain and the natural numbers. We can therefore assume in our analysis that parameters can be considered as natural numbers.

4.5.5 Java integration

In order to reduce the cost of developing the host language, we provide comprehensive integration with Java, in which the runtime framework for the host language is written. This integration occurs in two ways; either Java to host or host to Java. We provide a library interface for Java, where scripts or functions may be evaluated on demand. Vice-versa, we provide a statement for calling arbitrary Java code from within scripts. The statement evaluation process occurs on the CPU; calling arbitrary Java code on the device, from within function definitions, is beyond the scope of this thesis.

This approach mitigates some of the problems with using an external DSL, in that the library and routines written in Java can be used from our host language and provide standard library features for free, as well as tight integration with existing tools written in Java.

4.6 System implementation

Before we continue to discuss the extension mechanisms available, it is worth considering how the system is constructed. This is particularly relevant because the extension mechanism is not available from *within* the language, as with many embedded systems, but is instead achieved by extending the *compiler* using the compiler tools of the original compiler. Beaver and JLex are, respectively, the parser and lexer generator systems used.

4.6.1 JastAdd

The system is developed using JastAdd [31], which is a *meta-compilation* system for implementing compilers and related tools. It is an *aspect-oriented* system for defining attribute grammars.

As a brief introduction for non-compiler specialists, the steps involved are to create a structure for representing an Abstract Syntax Tree (AST), then specify new methods on those nodes. The AST represents the program in memory, and can be defined and developed using inheritance. These methods are defined in an aspect-oriented fashion: for example if we have a node Expression we might define a type method – or *attribute*, as it is known – like so:

```
syn Type Expression.getType();
```

Notice that we have not provided an implementation; when nodes inherit from Expression, they must provide an implementation. We will later use this to define an *interface* for the extension mechanism, describing what new nodes must implement within the system. JastAdd is succinctly described in a recent tutorial [45].

4.6.2 Evaluation mechanism

Each script is evaluated by interpretation on the CPU. Each function call is implemented on the graphics card device, with the functions compiled on-the-fly. This is a reasonable approach because these function calls are typically long running; short applications are not well suited to GPU, and so compilation time is negligible compared to the overall run time. Secondly, we can make use of the runtime values of the parameters to create an efficient runtime solution, as we see in the next chapter.

4.6.3 Intermediate language

We provide a custom *intermediate representation* (IR) designed for describing abstract GPU programs, and a custom API for generating such IR code. The purpose is to provide an abstract low-level language for describing the output of our parallel and code analysis techniques, and providing a compilation language target for domain developers. The key part of this API is a Builder class, which provides static methods for creating operations and primitives of the IR.

The primitives of the IR are those of a fairly low-level imperative language, and constructed to ensure a smooth translation from the IR to GPU assembly code. In particular, many of the operations have direct equivalents in PTX, the NVIDIA GPU low-level instruction set. However, unlike PTX, it is not a three-address code, and multiple expressions may be combined. The PTX ISA is described in NVIDIA CUDA Parallel Thread Execution ISA document provided by NVIDIA [90].

In order to simplify the process of generating IR code, the Builder class supports basic looping and conditional statements in addition to the GOTO/label mechanism provided by PTX. The language itself is strictly typed, matching the PTX types, and provides casting mechanisms between types as detailed in the PTX document. As with PTX code, data may be stored and loaded from any of the memory spaces on the GPU, whereas locally scoped variables are defined by the prefix "REG" and may be accessed freely, and used as arguments to any expression, function or statement.

A program in the IR represents a complete and functioning program that can be evaluated on the GPU. Compilation occurs by producing an equivalent PTX program from the IR, and passing that to the NVIDIA binary display drivers, which include a PTX compiler which in turn creates a binary which can be executed directly on the users GPU. In this way, we do not require the CUDA toolkit or SDK to be installed; nor do we require a C or C++ compiler. Eliminating these steps from the toolchain reduces compilation time dramatically, and permits compilation of GPU functions on-the-fly using only the binary

display drivers.

4.7 An extensible system

A keystone of our incremental approach is that the host language be extensible in order to support new applications. This is achieved by the embedding of new definitions into the host language. Extension writers are first-class citizens in this world; they write their extensions using the full gamut of compiler tools, and with full access to the original source code. In this way, they are free to implement new statements and expressions in whatever way they see fit, so long as they do not modify the existing language.

For the majority of applications, this freedom is overkill – extensions are typically restricted to a class of applications we call *models*. We provide direct support for implementing models through the definition of new syntax, code generation and data location and transfer, greatly simplifying development. This extension system in fact provides enough flexibility to define all the examples described in the bioinformatics chapter.

Note that our system is an extensible compiler for an extensible language; it cannot be extended within the language. Writing an extension with new syntax and semantics requires developing a plugin for the compiler.

4.7.1 Extension development steps

Our system makes heavy use of existing compiler tools, as well as providing our own DSL APIs in order to reduce the cost of development. The process for constructing a new DSL has the following steps:

1. Decide on a syntax and semantics for each new model definition to be introduced. Implement the AST in JastAdd [31], extending ModelDefinition and ModelType, and define a parser using the parsing framework provided. Provide type system extensions for any new concepts (Section 4.7.2).

2. Provide constructs to access this model in the function definition. For each construct, implement type analysis, recursion analysis and model analysis features as required by the API (Section 4.7.4).

3. Specify arrays to be generated by the model to represent the model on the device. These are defined using an abstract framework, where the implementation may be modified by the compiler for optimal performance (Section 4.7.5).

4. Implement code generation for new constructs. The work here is simplified greatly by referring to the arrays above, and using a code generation API (Section 4.7.6).

4.7.2 Model definitions

Extensions may define one or more *model definitions*, each designated by a globally unique keyword. Each model definition represents a set of related concepts or *components*. End-users create new model instances using the define statement, which delegates parsing and lexing to the domain extension registered for that keyword. Using this technique, extension writers exist in a sandboxed environment, where they can freely define the syntax and semantics of their models. In essence, each model definition is an embedded DSL for describing new models of that form.

The grammar is extended like so:

Stmt ::= ...
| define *extensionname modelname* { *Contents* }

The Contents grammar production is dispatched to the appropriate grammar based on the `extensionname`. In this way, new extensions cannot interfere with existing extensions, and the writer is still free to define whatever syntax they desire within the extension.

The new model definition is represented in the AST by extending the abstract `ModelDefinition`. The implementation interface does not place onerous demands: it must firstly define the

type of that model (see the following section), and secondly it must return a DeviceModel which returns the device representation of that particular model.

For an example of how an extension is written see Chapter 6.

4.7.3 Type extensions

The construction of a new model definition usually requires the introduction of new *types* to the system in order to be able to describe functions that act on those models. Extensions are free to implement new types as they see fit; however, the process is greatly simplified by inheriting from one of two particular abstract "types" provided with the extension toolkit.

The first is `ModelType`, which is a *calling*, *non-recursive* type. Recall from Section 4.5.4 that a calling type is passed statically to an original call. It is a type designed to represent a single instance of a model, for example a `HMMType` or `TreeType`. By extending `ModelType` and specifying it as the type of the `ModelDefinition`, the new type provides automatic type analysis of any variable which points to that model.

A common feature of extensions is that we pass a `ModelType` statically to a function, and use that to determine the range of valid values for the set of *recursive* types. Recall that recursive types, as the name suggests, are those from the function prototype that we recurse over. For example, we may pass a tree as the ModelType, then recurse over the nodes of that tree.

In the general case of a model with components, where we wish to recurse over the components, we can extend from `ModelElementType`, a *recursive*, *non-calling* type. `ModelType` and `ModelElementType` work analogously to `seq` and `index`, in that the `ModelElementType` is parameterised by the name of another parameter, and is related to that parameter. Each `ModelElementType` must also define a size, which is usually computed from the model instance of the associated `ModelType`.

To take a concrete example, we consider an extension for representing tree structures. Typically, we would have a `ModelType` called `TreeType`, represented by the string `tree`,

representing the tree itself. Trees contain nodes, which would be represented by a `ModelElementType` named `NodeType` and represented by the string `node[t]`, which is parameterised by the name of a tree. A function prototype using such an extension might look like so:

```
int recurse(tree t, node[t] n)
```

In this case the function `recurse` would recurse over the nodes of the tree argument passed to the function.

4.7.4 Host language syntax extensions

Defining a new model definition language is worthless unless we can extend the function expression syntax of the host language to explore that model. To that end, domain developer are free to implement any form of expression they desire, provided the expression is *pure*, e.g has no side-effects. New expressions must, however, implement the following JastAdd interface. Recall that the syntax `syn TypeName Expression.functionName()` means that all `Expression`s should implement a function called `functionName()` returning a value of type `TypeName`.

- `syn Type Expression.getType();` – computes the type of the expression.
- `syn IRExpression Expression.gen(DeviceCall call);` – generates the IR representation of this expression. We generate code on-the-fly, on a per call basis, which is why we pass a particular device call to the generation routine.
- `syn AffineEquation Expression.toAffineEquation();` – computes an affine equation representing this expression, if it is intended to be used in recursive definitions. The necessity of this is discussed in the next chapter.

In order to simplify the construction of commonly found expressions, we provide two convenient abstract extension points:

- *Accessors* - Elements of the model can be explored using accessors on existing elements of the model. Examples might include finding child nodes on a tree or the parent of this node. We use a . record style syntax, which is prevalent across other languages, and so this provides a natural way to access the model.

- *Iterators* - The base language provides some elements for combining or searching over all the elements of an abstract *iterable*. Examples include sum and max. DSL writers can implement this iterator interface by extending IterableExpr and implementing the loop initialisation and increment functions. DSL owners could also implement new Iterator expressions by defining the operation required when folding together the elements of the iterator.

These features are common in other languages, and should be familiar to audiences in the scientific domain who have experience with languages such as Python or Java.

4.7.5 Data storage, access and transfer

One of the biggest factors in the performance of a GPU application is how we store and use our data. For our DSL applications this comes down to data *layout* and data *location*. The *layout* is the way in which we order the data onto the device, either in a single piece of contiguous memory or multiple pieces of memory. The *location* specifies where on the device the data is to be stored – GPUs tend to have a wide variety of on-device memory locations and caches, each with their own advantages and disadvantages. In order to produce an efficient solution, we are going to have to choose the appropriate area of memory for each array described by the DSL.

We make the pragmatic decision to let the domain developer specify the layout and, only optionally, the memory location. An automatic derivation of the layout would be difficult to achieve without unnecessarily restricting the framework. The expected expertise of a domain developer is that they have solid programming experience, but not necessarily

GPU or compiler knowledge. This is feasible – we expect a small group of people to develop DSLs, which are then used by a much larger majority. Consequently, expecting domain developers to provide a layout for their data is a reasonable assumption.

4.7.5.1 Data layout framework

We provide a *data layout framework* for describing the layout of data in an abstract way. This allows the domain-developer to describe and interact with their arrays at a high-level, where the framework will automatically generate the low-level implementation. The benefit of this system is that it creates a uniform interface for different locations on the GPU, leaving the framework free to select, automatically, an appropriate location.

The framework is unashamedly simple and direct. Domain developers specify their structures using abstract arrays of a specified dimension. During code generation, the domain developer can reference these structures to generate code for loading data.

Each model definition type (e.g "trees") defines how a model (e.g an instance of a particular tree) is converted to a DeviceModel, which is a set of arrays representing that model. Arrays are generated dynamically at runtime; this gives the extension full control over the type and kind of arrays generated. For example, it may generate different types of arrays depending on what operations are used, the number of a particular type of element or the absence or presence of an element in the model.

Each model must implement the `generateDeviceModel()` method, which creates a DeviceModel, a container for arrays on the device. Arrays may be added to the model as shown in Figure 4.5, using the addArray method. The use of this method is demonstrated for the firstChild array in the tree extension. The location specifies which area of memory the array should be stored in, and may be selected from valid locations for this device, or an AUTO option.

Arrays are considered to be homogeneous data stores, with a single specified container type. These can be one of a set of specified low-level types provided – int, float, double or

(a)

```
public void DeviceModel.addArray(Location loc,
           String name, int array, int... dims);
```

(b)

```
DeviceModel model = createDeviceModel();
int[] firstChild = getFirstChildList();
model.addArray(Location.SHARED, "firstchild",
               firstChild, firstChild.length);
```

Figure 4.5: Adding arrays to the model. (a) is the function prototype for adding arrays to a model, and (b) is an example from the tree extension

char. Multi-dimension arrays are supported; however, the framework is free to change the ordering of the dimensions of the array for better performance.

4.7.6 Code generation

The majority of code generation is handled by the framework itself. However, domain developers will need to implement code generation for each of the high-level expression primitives they have added. This is achieved by providing an implementation of the gen(DeviceCall call) function, which returns an IR expression using the IR API (Section 4.6.3).

Importantly, extension writers must adhere to the rule that the expression is *pure*: it must have no side-effects, and it must return an IR expression that itself returns a value.

In addition to the standard IR API we provide code generation tools for accessing arrays in models. Importantly, this abstracts away from the differences of loading from each memory location, providing a uniform method for data access. An example API call from the tree extension is:

```
Builder.generateAccess("firstChild", index);
```

During code generation, any array that is specified using the AUTO location is generated with a dummy load operation. Once the code for computing the expression of the

function call is determined, and all uses are tallied up, we execute the auto-allocation algorithm, described in the next chapter, to allocate arrays. We then reify the dummy load operations to use the correct memory location, and generate the initialisation code required on the GPU.

4.8 Extensions

During the course of development on the framework, three extensions have been created. We will describe a first, simple, extension here – substitution matrices. After covering the parallel analysis and code generation steps in the next chapter, we will consider the development of a more complicated example in Chapter 6 – Hidden Markov Models. Finally, we describe how a set of students developed the final domain extension for describing trees in Section 7.2.

4.8.1 Domain extension example: Substitution Matrix

Substitution Matrices are a simple data-extension to the language. They allow the user to describe the cost of substituting one character in an alphabet for another. They are most commonly encountered in sequence alignment algorithms, such as Needlemam-Wunsch and Smith-Waterman.

Support for specifying substitution matrices is created by introducing a new statement to our language for defining the matrix with a straight-forward table. This model is represented by a new type, matrix, which is a *static* type – that is, values of that type can be passed to functions but cannot be recursed over.

The final step is to extend the function definition language in order to access the data stored by the substitution matrix. We do this by extending the expression syntax with a simple two dimension access.

$$Expr \quad ::= \quad \ldots \mid Var[Expr, Expr]$$

4.9 Related work

Domain-specific languages have often been proposed as a means of taming the complexity of development for graphics cards, and are a rapidly growing area of research. Notable examples include OptiX [93], a DSL for ray tracing; a GPU backend for *FLAME* [42, 79] a DSL for describing dense linear algebra problems; *STARGATES* [95], a parallel DSL for describing partial differential equations (with code generation for CUDA, MPI and TBB); *Diderot* [25], a GPU DSL for biomedical image analysis and *Physis* [80], a DSL for stencil computations, a GPU backend for the finite element solver DSL called UFL [78], amongst many others. These DSLs have a common approach: they encode the *primitives* of the domain, and synthesise efficient solutions from these high-level definitions. They can make extensive use of domain knowledge in order to produce highly optimised output. The flip-side is that these DSLs can have a high cost of development, requiring expertise in GPUs, the domain and compilers.

A bioinformatics DSL with a similar focus to ours is *Algebraic Dynamic Programming* [36], designed specifically for dynamic programming problems in bioinformatics using strings, and has recently been extended to GPUs [107]. Input is restricted to problems described by context-free grammars; in contrast our input language is more flexible, allowing a wide-range of dynamic programming problems and providing an extensible system to support other data-structures in the form of models.

DSLs have also been proposed as a way of simplifying development of general purpose GPU applications. Notable examples of this approach include *Chestnut* [108], a DSL which provides explicit parallel primitives (e.g parallel loops and array primitives), and *Conflux* [16], which embeds a DSL into C# for describing kernels as classes, and annotations for describing the memory location of data. Python is also a popular choice for embedding such languages, either compiling subsets of the language [18, 54] or dynamically generating CUDA code from annotated python [19, 104]. These simplify the process of describing explicit parallelism, rather than attempt to automatically derive such paral-

lelism.

On a similar theme, array programming DSLs embedded in functional languages as a means of generating efficient GPU code have also become fashionable over the last five years. Examples here include GPU.gen [64], *Nikola* [75], *Obisidian* [111], *Accelerate* [22] and *Barracuda* [63, 62]. These all follow similar patterns, describing *explicit* parallelism by providing array primitives, such as *map* and *reduce* operations, which are automatically translated to GPU equivalents. Parallelisation of a program is achieved by the user explicitly calling the parallel primitives of these embedded DSLs from within full functional languages. In contrast, our work focuses on designing a front-end for a specific custom domain (in this case bioinformatics), and *automatically* parallelising any functions written in that domain.

As well as being array languages, these are also examples of a wider trend in the functional programming community – embedding DSLs for hetereogeneous systems. How this is to be achieved is a thriving area of research, with particular focus on how to build DSLs with a single representation which have multiple compilation targets (e.g [53, 103, 74]), whilst retaining type safety, and expressiveness. *Active libraries*, which have many of the same goals, and achieve many of the same benefits as DSLs, are also an active area of research, with examples like OP2[84] for defining unstructured grid applications to be run on multiple platforms.

A practical outcome of this focus, and one with a similar motif to our approach, is the concept of "Language Virtualization" provided by the Delite framework [20, 21, 14]. Delite has been used to create *Liszt* [28], a DSL for physics simulations and *OptiML* [109] for machine learning. New DSLs are *embedded* within Scala; the Delite framework simplifies this process by providing reusable components (such as parallel patterns like Map, Reduce, ZipWith, Scan) across DSLs and leveraging Scala features for DSL front-end development. In particular, it uses a technique (*lightweight modular staging* with *polymorphic embedding*) to provide a staged AST on which domain-specific and compiler optimisations

can be executed, and from which we can generate code for multiple compilation targets.

Where our approach differs is that we develop a *custom host language* designed from the ground up to (a) explicitly describe bioinformatics problems and (b) provide automatic parallelisation features, simplifying development of new DSLs in our chosen domain. Developing new bioinformatics DSLs in our system requires little to no GPU knowledge, whereas Delite requires back-end development that consists of either custom GPU development, or the composition of existing parallel patterns for explicit parallelism. Our focus also differs in that they are attempting to produce code generators for heterogeneous systems, whereas we focus solely on GPUs. ALPHA[118] is a language with a similar concept to our core language, which targets systolic arrays (which are similar to GPUs) and allows the definition of recursive functions.

Chapter 5

Generating GPU Code

The work in the first part of this chapter has been previously published, in shortened form, in the paper "Synthesising Graphics Card Programs from DSLs" [17]. The work described below is entirely my own.

Domain-specific languages have obvious benefits for end-users. An effective DSL makes programs in the domain clear and quick to write, by bringing the language closer to the user. Such a language is high-level, in the sense that it abstracts away from the underlying model of execution, further than a general purpose language.

This level of abstraction leaves a compilation *gap* between the definition and the underlying hardware. Paradoxically, this can work in our favour when considering unusual execution models. A carefully constructed DSL can ensure the dependencies between different computations are more clearly defined than in a general purpose language. It is the contention of this chapter that such a carefully designed DSL can be used to generate graphics card applications comparable in performance to hand-coded tools.

The work in this chapter acts as *enabling* work for the incremental DSL approach detailed in the previous chapter. The incremental approach to DSL development is only effective if we can derive enough information from the base language and extensions to generate efficient graphics card code. The work in this chapter is provided as a proof of that con-

cept, using bioinformatics. This is by necessity; there can be no overriding mechanism that works for all domains, and each must be considered in turn. However, this does not mean that the techniques detailed in this chapter are only of use in bioinformatics – many have applications beyond that. We will consider, as we go along, how the techniques may be adapted to tackle other domains.

The techniques are separated broadly into two categories. Firstly, we consider the available methods of execution on the graphics card to decide how to generate the *parallel algorithm* for the given problem. We consider the trade-offs to be made between different models of execution, and provide heuristics for deciding when each model is appropriate. We consider *scheduling* as one way of parallelising the previously designed recursive language, unrolling operations in order to make use of the available cores.

The second category is that of the *parameters and settings* required for execution on the GPU, including memory location, kernel launch parameters and so forth. These parameters may have a significant effect on performance, as we will see in Chapter 7. We consider the setting of block and grid parameters using heuristics. We will also look at how the memory hierarchy is to be used effectively, providing a cost model for deciding between different uses of the memory hierarchy.

An important theme in this chapter is the *encoding* of specialist GPU knowledge within the compiler; providing heuristics and cost models that match the experts' intuitive understanding of the hardware. In many cases, this embedded knowledge can be customised by GPU specialists at a later date, or modified to provide better performance on new hardware.

5.1 Anatomy of a recursive problem

In order to introduce the concepts in this chapter, we will take a simple example that is, nonetheless, representative of typical problems in bioinformatics. The *edit-distance* problem is a straightforward example of a recursive problem. It asks us to compare two strings

to find the minimum cost of making a series of edit or match operations to transform one string to another. It has relevance in bioinformatics in the *Smith-Waterman* and *Needleman-Wunsch* algorithms for aligning two biological sequences, which were discussed in Section 3.1.

The *Principle of Optimality* allows us to frame the problem as a combination of a series of sub-problems, and so we can provide a recursive definition to determine the solution. For the edit-distance algorithm, as for many optimisation algorithms, this provides a natural and intuitive way of describing the solution (see Figure 5.1).

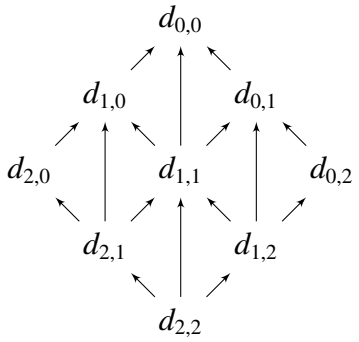
$$d(x,y) = \begin{cases} x & \text{if } y = 0 \\ y & \text{if } x = 0 \\ d(x-1,y-1) & \text{if } s[x] = t[y] \\ min\begin{pmatrix} d(x-1,y), \\ d(x,y-1), \\ d(x-1,y-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

Figure 5.1: The edit distance recursion on strings s and t.

This form of declaration has an implicit method of evaluation, a recursive function, which is most naturally solved in a serial fashion, one call at a time. However, for many problems, this is not the most efficient solution. For example, we may repeat computations unnecessarily when two different calls depend on the same computed value. *Dynamic programming* permits us to store those repeated computations, or even tabulate the result bottom up – $d(0,0)$ first, followed by dependent computations.

This leads to another example of redundancy in the edit-distance problem – once we have computed $d(0,0)$, it does not matter whether we compute $d(1,0)$ or $d(0,1)$ as they are *independent* – that is, they do not depend on each other's values. This opens up the possibility of concurrently executing both $d(1,0)$ and $d(0,1)$. Similarly, we can observe that any set of cells on an $x+y$ diagonal line are independent, and can be computed concurrently provided that all prior dependencies are solved.

5.1.1 Dependencies



(a) Edit distance (b) Fibonacci

Figure 5.2: An example of the difference between a linear (5.2b) and a wide dependency graph (5.2a)

.

It is clear that the constraints on a parallel solution for recursive problems come from the dependencies between different computations. Some problems will be well suited to parallelisation, whilst others will not provide any opportunity for parallelisation. Figure 5.2 contains the contrasting *dependency graphs* for the edit distance algorithm on a 3x3 problem and a straightforward recursive definition of the Fibonacci sequence for *fib*(5). It is clear that the Fibonacci recursion permits no parallel solution – analysing each sub-problem only "unlocks" one other sub-problem. In comparison, sub-problems in the edit distance recursion can clearly unlock multiple other sub-problems – e.g once we compute $d_{0,0}$, we can compute either $d_{1,0}$ or $d_{0,1}$. We say that the computation of $d_{1,0}$ is *independent* of $d_{0,1}$ when the problem provides no constraints on the order of evaluation between the two. Consequently, we can evaluate them one after another, or even evaluate them simultaneously. It is this independence property within problems that we hope to exploit when identifying a parallel solution.

5.1.2 Scheduling

The vectorised, SIMT, nature of the GPU architecture is such that parallel algorithms must find sets of work that can be executed simultaneously across a number of lightweight cores. A straightforward way of achieving this is to have multiple independent problems (e.g a large number of edit distance problems), with each problem given a single thread. However, the GPU can often be best utilised by exploiting the co-operative nature of the GPU, where multiprocessors contain caches that optimise local sharing of data, to evaluate a single problem using a block of threads. This equates to finding elements of the problem we can evaluate simultaneously.

In a finite recursive problem, such as the edit distance problem, we might think of this as dividing the elements in the domain – that is, the recursive calls in the dependency graph – into sets (*partitions*) of computations that can be concurrently executed. Our informal understanding of "can be concurrently executed" here should be that the computations are independent, that is, the elements in the set should not depend on any other elements in the set, either directly or indirectly.

It is clear that a single problem may be partitioned in many different ways. Furthermore, each partition may also require some dependencies to have been computed before the partition itself can be evaluated. To this end we define a *schedule*, a group of partitions that together cover the entire domain and do not overlap. As implied by the name, the schedule also defines a total order between partitions in the domain, providing a fixed sequence of execution and simplifying the dependency graph. Figure 5.3 illustrates a valid scheduling of the 3x3 edit distance problem. We can think of each partition as a *time-step* of the computation.

Typically a schedule for a function f is defined as a mapping or function from the original domain to the partitions of the schedule [56]. We denote this schedule as S_f – the *scheduling function* for f.

For example, if we have two sets of arguments to f, $\bar{x}$ and $\bar{y}$, then if $S_f(\bar{x}) = S_f(\bar{y})$

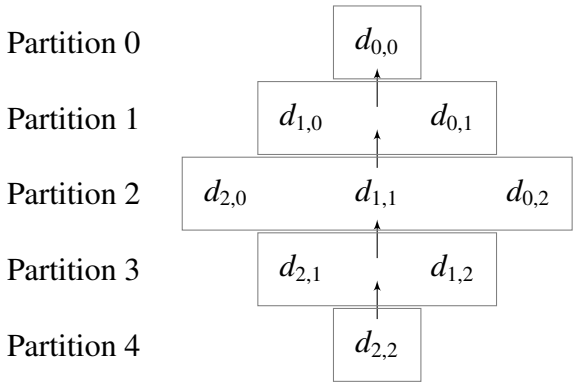


Figure 5.3: A valid diagonal schedule for the 3x3 edit distance problem. This schedule has five partitions.

we know that we can independently evaluate the results $f(\bar{x})$ and $f(\bar{y})$. In our edit distance example, if $S_d(0,1) = S_d(1,0)$ then we can deduce that $d(0,1)$ and $d(1,0)$ can be computed independently of each other.

We note that any technique that defines S_f explicitly as a direct enumeration of the cells of each partition is doomed to fail on these problems, due to the vast number of cells that require calculation. To do so would be both memory intensive and difficult to determine efficiently a priori on problems of varying sizes .

Instead, we make the analysis tractable and the implementation possible by restricting the schedule to be an affine function. Affine functions allow us to describe a wide variety of *regular* partitions of the dependency graph with a single function, which can then be used to synthesise a parallel solution.

In the edit distance example (Figure 5.3) it is clear that an appropriate scheduling function is $S_d(x,y) = x+y$ – the parallel lines which represent independent values are described by the equation $x + y = c$, where c is the current partition. Figure 5.4 illustrates some schedules commonly found in scientific problems.

It is clear that not all schedules are created equal. We must therefore define some criteria for choosing an appropriate schedule such that it is both *valid* and *efficient*.

As we previously observed, the validity of a schedule is determined by the dependencies described in the recursive function. In other words, a valid schedule is one that ensures that

any dependency between two elements is fulfilled by executing the dependent before the dependee. We can do this in an inductive fashion, by proving any direct dependencies of an element are fulfilled before evaluating that element. What are the dependencies of f? The recursive calls of f; each different set of arguments to a recursive call constitutes a potential dependency.

We will formalise this choice by identifying from each recursive call a *criterion* on valid schedules. The key condition which we wish to maintain is that the partition – e.g time-step – of the recursive call is less than that of the current call. Take, for example, the edit distance algorithm in Figure 5.1. It has three recursive calls $d(x-1, y)$, $d(x, y-1)$ and $d(x-1, y-1)$. We will therefore need to prove that the following equations hold:

$$S_d(x, y) > S_d(x-1, y),$$

$$S_d(x, y) > S_d(x, y-1)$$

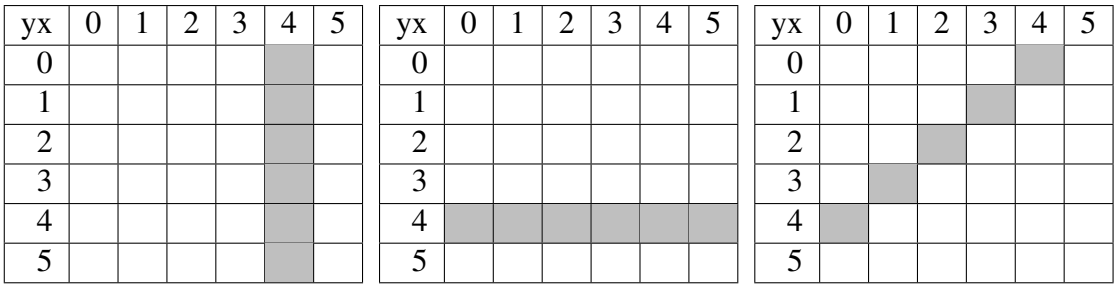
$$S_d(x, y) > S_d(x-1, y-1)$$

If, as before, we select $S_d(x, y) = x + y$, it is clear that all three are true, therefore this is a valid schedule.

An efficient schedule is one which will make best use of the resources available. In an ideal world, we would take into consideration the exact configuration of the hardware available – including the number of cores and the relative performance. However, it may not be practical to determine this precisely. We will instead aim to minimise the number of partitions we need to evaluate, as that will provide a reasonable proxy for an efficient solution. This maximises the average size of each partition.

Again, if we consider the edit distance example, an equally valid schedule would be $S_d(x, y) = 2x + y$, however this results in more partitions (time-steps) and would therefore be considered a less efficient schedule. There are very few occasions where a schedule with more partitions will be more efficient. These arguments are further developed in the

following sections.



(a) $S_f = x$ (b) $S_f = y$ (c) $S_f = x + y$

Figure 5.4: An example of three parallel strategies for the two dimensional case. Each case highlights the partition where $S_f = 4$.

5.2 Parallelisation Overview

The GPU provides a novel architecture which supports a number of different parallelisations strategies. Each strategy considers the deployment of *tasks* – items of self-contained work. In our framework, tasks equate to a single instance of *calling* a top-level function – for example, `editdistance("foo","bar")`. This task would include any subsidiary functions which may be called in the process of evaluating the top-level function. In our example, each task relates to an instance of the edit distance problem on a different set of inputs.

The only ways to dispatch tasks are through either a direct function call or the `map` operation – our tasks are therefore *independent*, and can be scheduled in any order. This is in contrast to other systems that focus on modelling dependencies between serial tasks – we consider dependencies *within* a task. This difference is not purely semantics – although we could model our larger tasks as sets of smaller dependent tasks, our system is optimised for larger tasks with small dependent sub-problems. Task-dependency systems are often optimised for large, serial, tasks instead.

We can broadly classify the strategies for evaluating tasks into three approaches:

- **Task per block**[1] – threads within a block work co-operatively on one or more tasks.
- **Task per thread**[2] – each thread evaluates one or more tasks.
- **Task per multiple blocks** – Work co-operatively across different blocks to compute task results.

Selecting a strategy from these options can result in more or less efficient code, depending on the program in question. There are a number of reasons for this: problem size (e.g the size of the recursion) may make it suited to a smaller or larger number of cores, whether the code is amenable to ILP (Instruction Level Parallelism) or TLP (Thread Level Parallelism). As an example, if we have only a single problem of a significant size, we will need to use multiple blocks in order to fully utilise the GPU. In addition, we may wish to divide the input into sets which get evaluated by one method or the other.

In this thesis we will focus only on the first two, as the applications we have considered tend to have a large number of small tasks, suitable for task-per-thread and task-per-block. We will also delay discussion of how to choose a method until after describing the code generation process. Regardless of the method chosen, a large portion of the analysis and code generation remains the same.

In order to make informed decisions, our compilation system occurs at *runtime*, generating GPU code on-the-fly. This may at first appear to be a strange decision for an optimising compiler. However, analysing and generating code at runtime ensures that we can exploit the runtime parameters to achieve the best possible parallelisation. Since these applications typically have a long runtime, the overhead of compilation (typically under a second) is negligible.

A good example of the type of optimisation this enables is the automatic allocation of data to locations on the GPU, which exploits both a detailed understanding of use-patterns

[1] Sometimes known as an intra-task parallelisation
[2] Sometimes known as an inter-task parallelisation

of the application, but also runtime parameter sizes to accurately predict where data should be place on the GPU.

To summarise our compilation approach, we:

1. Identify a schedule for the function (Section 5.3);

2. Generate a set of nested, serial, loops honouring this schedule. This problem is both well known and well researched. We use a *polyhedral code generator* called CLooG [11], which models the domain as a *polyhedron*, and considers a schedule to be a *transformation*, which it applies in order to generate a set of nested loops to iterate over the transformed domain. We provide our function domain and schedule to CLooG to generate our serial nested loops. (Section 5.3.11);

3. Determine, based on the size and type of input, how to distribute work e.g per-thread, per-block (Section 5.5.1);

4. Create a different kernel for each type of workload (Section 5.3.11):

    - **Task-per-thread kernel** – each thread will evaluate one task, so we keep the serial loops provided by CLooG.

    - **Task-per-block kernel** – transform the inner, independent, serial loop to a parallel loop by distributing the work across a block of threads. Each thread in the block evaluates one position in the loop in parallel with the other threads, based on the thread index. Each thread is responsible for every position nt in the loop, where n is the block size and t is the thread offset.

5. Generate CUDA code for the functions inside the CLooG loop.

6. Optimise data access by finding an allocation of data structures to memory locations on the GPU (Section 5.4).

7. Implement, where appropriate, the *sliding window* optimisation, an array contraction style optimisation which stores only the intermediate results required for calculating the following cells, rather than all intermediate results (Section 5.5.4).

8. Use heuristics to determine parameters for block size and grid size (Section 5.5.2, Section 5.5.3).

5.3 Synthesising parallel algorithms

Recall that our applications are defined as recursions written in a narrowly defined functional language. Recursions naturally define the domain and application of the function. In particular, we limit recursions to *fixed and predictable bounds*, based on the input, as each recursive parameter is of a finite or bounded type with respect to the input.

Our language is therefore amenable to dynamic programming using *tabulation*. The dynamic programming table must be computed with respect to the dependencies of the function. We proceed by defining a *schedule* (5.3.3) that generalises such dependencies. By synthesising a program that adheres to the schedule we can unroll the recursion to execute it bottom up. A polyhedral model code generator is the natural way to generate code for such problems.

Whilst recursions are the most natural form of definition for applications in bioinformatics – matching the definitions commonly provided in that domain – other domains may benefit from similar methods. Any application that describes regular dependencies is theoretically open to the type of dependency analysis we will describe in this chapter.

Our program synthesis method begins by determining whether there are recursions in the call graph. All functions that are part of recursive loops must identify a *schedule* which defines the order in which elements of that recursion must be computed. Program synthesis proceeds by generating code using the polyhedral model, adhering to the discovered schedule. The steps are:

1. Identify the call graph for f, the function being called on the GPU.

2. We encode the dependencies of the recursion as a series of criteria on the scheduling function (Section 5.3.5).

3. We use these criteria and a suitable goal function to automatically find, using a constraint solver, the coefficients of a valid scheduling function, which is optimal with respect to the number of partitions required (Section 5.3.7).

4. Using our globally defined schedule we synthesise a massively parallel program using CLooG (Section 5.3.11).

5.3.1 Definitions and input

Given a function f our task is to place the computation of f on the GPU. f is a function in our recursive language (described in Section 4.5.2), which may be mutually recursive. Each of the recursive parameters of the function f must define a mapping between elements in their domain and the natural numbers. We can therefore assume in our analysis that all recursive parameters are natural numbers without loss of generality.

We construct a call graph, $G(f)$, representing the calls made either directly or indirectly from f. $G(f)$ is defined using $C(f)$, the set of functions which may be called in the evaluation of f, and $E(f)$, the set of calling edges $(c_i, c_j) \in (C(f), C(f))$. Thus, $G(f) = (C(f), E(f))$.

The absence of first class functions and function aliasing in the language ensures this analysis is simple and accurate. Furthermore, our recursive functions are typically designed to execute every possible path at least once during tabulation.

5.3.2 Similarities to other approaches

It is important to note at this stage that the techniques described here tie into a much larger body of work, starting with schedule analysis of uniform recurrences [56] and leading on to

work on systolic arrays [118, 24], loop scheduling (wavefronting) [61] and tiling [65] and applications of the polyhedral model [11]. Affine schedules have long been understood to be a succinct way to describe and implement parallelisation, and recent work has applied some of these techniques to GPU applications [9, 10, 6, 116]. To clarify, the contributions of this section are:

- The formalisation of the constraints on the schedule placed by our particular language definition.
- A runtime algorithm for finding a schedule (in contrast to prior work on static source-to-source schedule identification), which uses a unique formulation of a Constraint Satisfaction Problem (CSP).
- The extension mechanism that allows domain developers to define new expressions, and extend the parallel analysis to those expressions by defining affine ranges representing their values.
- The program transformations we execute after generating code using CLooG to produce a GPU program (although these transformations are similar to work that was completed in parallel, e.g [10]).

5.3.3 Scheduling

Conceptually, the *scheduling function* for a function f maps the elements in the domain of f – the possible calls to f – to *partitions*. Each partition is a set of elements that are *independent* with respect to each other and are identified by a natural number. This one-to-one mapping between partitions and natural numbers ensures that there is a total ordering over partitions. This total ordering equates to a *time-step* for the function.

We will define a schedule for function f to be an affine function, named S_f with integer coefficients:

$$S_f = a_1 x_1 + \ldots + a_n x_n$$

Where $a_1, \ldots, a_n \in \mathbb{Z}$ and the *recursive* or *source* domain of f is defined as $X = X_1, \ldots, X_n$, where $\bar{x} \in X$ when $\bar{x} = (x_0, \ldots, x_n)$ and $x_i \in X_i$. n is therefore the number of parameters to the function. For schedule identification there is no theoretical limit to the size of n, however, due to the practicality of solving the constraint problem, and of generating code for higher dimension cases, we provide an artificial limit of three.

These constraints on the function ensure that the analysis is tractable and the implementation efficient; they are not, however, fundamental limits and the extent to which they can be lifted will be discussed later in the section.

5.3.4 Formalising function dependencies

In order for a schedule to be useful it must be *valid*. Informally, a valid schedule is one which adheres to the dependencies described by the calls between the functions. Put another way, we must ensure that when we come to evaluate $f(\bar{x})$ that everything it depends on has already been computed.

All our functions are pure – the result of each element is dependent only on the arguments. A call, c, is therefore uniquely defined by a pair of the function and its list of arguments, $(g, (y_0, \ldots, y_n))$, where each y_i is a value in the domain of the equivalent parameter of g.

Recall that $G(f)$ is the call-graph starting at f. The edge $(c_1, c_2) \in E(f)$ represents the case when c_1 *may* call c_2. We do no branch analysis based on conditionals – instead, we consider all recursive calls in the function to be part of the edge set. We will say $c_1 \to c_2$ when $(c_1, c_2) \in E(f)$.

Using $E(f)$ we can define the relation $\rightsquigarrow$, the transitive closure of edges of $G(f)$. We can think of this as the transitive closure of the dependency graph, encoding the indirect and direct dependencies of any call.

Assume that we have defined a schedule S_g for each function $g \in C(f)$ which is recursive – that is, there exists a pair of calls c_1, c_2 such that both are calls to function g and

$c_1 \rightsquigarrow c_2$. We can build a *composite* schedule, S, from each component using:

$$S(g, (y_0, \ldots, y_n)) = S_g(y_0, \ldots, y_n)$$

This schedule is then valid iff

$$c_1 \rightsquigarrow c_2 \Rightarrow S(c_1) > S(c_2) \tag{5.1}$$

We will label this equation the *partition ordering* condition. It states that dependent calls should be evaluated before their dependees.

The definition encompasses two important properties. The first is that there is an implicit ordering over partitions, such that if we evaluate the lowest time-step partition followed by the next time-step partition and so on, we are guaranteed to maintain the order of dependencies required by $\rightsquigarrow$.

The second is that two elements in a single partition must be *independent*. A pair of calls, c_1, c_2, are described as *independent* iff there is no call path between them, that is $c_1 \not\rightsquigarrow c_2$. If two calls are independent, they can be computed without reference to each other, and therefore can be computed synchronously. The partition ordering condition implies just that:

$$S(c_1) = S(c_2) \Rightarrow c_1 \not\rightsquigarrow c_2 \tag{5.2}$$

By finding integer coefficients $a_1, \ldots, a_n$ of $S_g = a_1 x_1 + \ldots + a_n x_n$ for each $g \in C(f)$, such that our composite S adheres to to 5.1, we can use the implicit independence to generate efficient parallel implementations.

One consequence of our partition condition is that there may be many valid schedules for each function; different schemes of parallelising the problem are described by different schedules.

5.3.5 Efficiently computing a valid schedule

We have now defined the properties of a valid schedule. However, as formulated in (5.1), it is difficult to efficiently apply the constraints directly to a schedule. In this section, we derive a set of *criteria* that valid schedules adhere to.

This has two practical consequences – firstly, end-users can specify a schedule that we can then verify. Secondly, we will use the validity criteria to automatically determine a schedule (Section 5.3.7).

For a S to be valid, it must adhere to the implication in (5.1). We will show this inductively, by considering the direct dependencies of the call. Recall that $\rightsquigarrow$ is the transitive closure of $\rightarrow$ (i.e $c_1 \rightsquigarrow c_2 \Rightarrow c_1 \rightarrow^\star c_2$). We can thus satisfy our condition by proving that for all $c_1 \rightarrow c_2$ the following holds:

$$c_1 \rightarrow c_2 \Rightarrow S(c_1) > S(c_2) \tag{5.3}$$

So far we have simply stated that $\rightarrow$ is the set of all direct dependencies of calls. We will need to define $\rightarrow$ in terms of the recursive calls of the function.

For a function $f(x_1, \ldots, x_n)$, a recursive call will consist of $g(xr_1, \ldots, xr_m)$, where xr_i are linear combinations of the initial parameters, e.g $xr_i = b_{i,1}x_1 + \ldots + b_{i,n}x_n + c_i$. These xr are the *descent functions* of the recursive call. g may be f here, if this is a recursive call to itself.

Each recursive call therefore represents a *set* of dependencies, described by:

$$\{(f, (x_1, \ldots, x_n)) \rightarrow (g, (xr_1, \ldots, xr_m)) | \forall x_1, \ldots, x_n\}$$

For each recursive call site, we will need to verify that this set satisfies (5.3). Substituting in our S equations:

$$S(x_1, \ldots, x_n) - S(xr_1, \ldots, xr_m) > 0, \forall x_1, \ldots, x_n$$

Recalling our pre-condition that S_g is affine, for all g, and evaluating by variable substitution, this is equivalent to the following condition:

$$a_1 x_1 + \ldots + a_n x_n - b_1 xr_1 - \ldots - b_n xr_m > 0, \forall x_1, \ldots, x_n$$

Where a and b represent the coefficients of the scheduling functions for the respective function calls. This equation denotes the *criteria* on a valid schedule. We derive one criterion for each recursive call in the function, and confirm that a S_f is valid by confirming it satisfies all the criteria.

If the two functions are the same, then the equation can be simplified like so

$$a_1(x_1 - xr_1) + \ldots + a_n(x_n - xr_m) > 0, \forall x_1, \ldots, x_n$$

When the descent function xr_i is uniform (e.g of the form $x_i + c_i$) – the majority of practical cases – this equation will simplify to $(-a_1 c_1) + \ldots + (-a_n c_n) > 0$, which is straightforward to analyse. For any descent function that is affine (e.g of the form $A\bar{x} + \bar{c}$), the criteria will require the runtime range of x to determine the validity. Descent functions that are neither affine nor uniform are not supported.

5.3.6 Mutual recursion

In the next section, we will derive a CSP for automatically finding the schedule. However, the criteria for mutually recursive functions is significantly more complicated than that of simple recursion, and is often unsolvable using a CSP. Furthermore, complex mutual recursions are less common in bioinformatics. We will therefore simplify our approach by allowing *only* those mutual recursions where every function in the call graph has the same

parameter list. In this case we will derive a *single* schedule which applies to every function: essentially we are modelling the recursions as a single recursion.

Doing so has one stumbling block: recursive calls in the same partition to other functions. For example, a pair of recursions $f(x) = g(x)$ and $g(x) = f(x - 1)$ cannot use a scheduling function $S(x)$ as it stands, because $f(x) \rightarrow g(x)$. However, if we provide an ordering over functions, such that for a particular partition, the elements of g are evaluated before those of f, we can relax the criterion for inter function calls to:

$$c_1 \rightarrow c_2 \Rightarrow S(c_1) \geq S(c_2)$$

Using this approach, we first derive a schedule by *assuming* that such an ordering exists. Once we have a schedule, we compute the ordering by considering each cross-function call, factoring in the schedule, and for those whose partition or time-step doesn't change, record an ordering dependency. When we generate code for mutually recursive functions, we will use this ordering to generate the functions in the appropriate order.

For the rest of this chapter, we will assume that this simplification is in place, which allows us to work only with a single schedule for a single function, that may represent multiple functions in practice.

5.3.7 Automatically determining a schedule

In the previous sections, we derived a series of criteria which we used to verify a pre-existing schedule, perhaps provided by a user. In this section, we will use the same criteria to derive a schedule on behalf of the user. This is fully automatic – no further user input beyond the recursion is required. In this way, we can support both automatic and user-specified parallelisation schemes.

Our aim is to describe a Constraint Satisfaction Problem (CSP) that combines the criteria of the previous section – which will enforce the validity of the scheduling function –

with a selection measure which will help choose an efficient schedule. We can then pass these conditions and goal into a suitable CSP solver to find the coefficients $a_1, \ldots, a_n$ of a solution S_f. We use the JOPT JS CSP solver [2], which implements the AC-5 arc consistency algorithm.

By necessity, the selection measure will need to be a heuristic – the various factors that affect the execution time are difficult to predict prior to execution, and may include which memory operations occur and when, the number of instructions, the size of the problem and the exact hardware used.

As such, the heuristic we have chosen is to minimise the number of partitions required to evaluate the entire table. By minimising the number of partitions, we are maximising the average size of a partition, and therefore maximising the ideal amount of parallelisation provided by an "ideal device" with infinite synchronous cores. In practice, we have a fixed number of cores, so we will have to evaluate partitions in blocks, regardless of the size of the partition.

If we have a schedule, $S_f(\bar{x}) = a_1x_1 + \ldots + a_nx_n$, where $\bar{x} = x_1, \ldots, x_n$, then we can compute the minimum number of partitions by minimising the difference between the largest and the smallest partition in the range, by aiming to minimise the following expression:

$$min_{a_1,\ldots,a_n}(max_{\bar{x}}(S_f(\bar{x})) - min_{\bar{x}}(S_f(\bar{x}))) \tag{5.4}$$

Our intention is to construct this as a CSP over $a_1, \ldots, a_n$. However, it is clear that equation (5.4) is not a linear problem – we are trying to solve for both a and x. How can we encode this as a tractable CSP problem if it is not linear?

The solution to this apparent problem comes from the observation that S_f is linear, and therefore the maximum value of S_f occurs when each component of S_f is maximised. How do we maximise a_ix_i? As we run our algorithm at runtime, we allow ourselves the luxury of knowing the range of x_i, e.g $0 \leq x_i < n_i$. We can therefore maximise the component by maximising x_i, if a_i is positive, and minimising x_i if a_i is negative. We can do the reverse

to minimise the component, and thus find the minimum expression.

With this approach we will need to evaluate up to 2^n different constraint problems to find the solution. This is practical for two reasons: the majority of applications are between two or three dimensions, and in most of cases, the criteria may predispose us to eliminate a subset of these problems – for example, if we already know $a_i > 0$ then $x_i = n_i - 1$, as it is the largest value in the range.

For each constraint problem, we specify the maximum and minimum values of $x_1, \ldots, x_n$, use equation (5.4) as the goal we wish to minimise, and add a criterion for each recursive call, as discussed in the previous section. If a result is found, we store the minimum goal value found, and the first set of solution coefficients, and continue to the next problem, until we have checked or eliminated all dimensions.

5.3.8 Affine recursive arguments

We previously specified that all recursive parameter types must define a mapping between elements in their domain and the natural numbers to be amenable to our technique. However, we must also place a further restriction on the arguments to the recursive calls: that they can be converted to an affine expression stated in terms of the original recursive parameters.

The constraint is a matter of practicality; recall that each argument (recursive descent function) is used in the CSP, and we can guarantee that we can encode affine expressions with ranges as a CSP problem. In theory we might extend this to anything encodable in our CSP solver. In practice, however, we keep to affine parameters as this gives the solver a better chance of achieving a solution.

The method of achieving this is that each expression must define an equivalent affine expression that represents the expected value. Computing the affine expression for each argument occurs in a recursive manner; each valid descent expression generates the appropriate affine expression from the affine expressions of its arguments. For example for the

expression $x - 1$, the subtraction node determines the affine expression for x, and the affine expression for 1, then returns the subtraction of the two.

We allow the constant value of the affine expression to be a range to support extensions; the value may therefore be an over-estimation of the exact arguments. This is acceptable as long as all valid values held by this expression are within the over-estimation.

There are a number of expressions in the language that fall foul of the affine restriction. Examples include expressions with a conditional result, such as if and max, as well as those which are not affine in nature, such as the multiplication of two variable expressions. These cannot be used as arguments to function calls.

Rather than restrictively preventing the user from describing such recursions, we permit them to exist, assuming for the purpose of the parallel analysis that they may take any value in the domain. In these cases, because we cannot guarantee that these recursive calls will always be valid, we generate an implicit range check during code generation. This is important, because if there is a form of parallelism in one of the other dimensions, we would like to be able to exploit that.

One kind of expression which complicates the system is the iterable expression. This is an interface with combine operations such as sum, max and min over customisable ranges. Recursive calls within the iteration may depend on the range of the loop, which cannot be described with simple affine expressions. We therefore make one small concession: we allow the addition of new, ranged variables, where the range of those variables is described in affine terms of the original parameters. The IntegerIterable expression is thus encoded by taking the affine expression of the start and end values, and using those as the range. This relaxation can be legitimately represented as a CSP. To take an example, given a (simplified) recursion such as `f(x) = sum(t in 1...(x-1): f(t))`, the affine expression of the `sum` node will be t, where $t = 1..(x - 1)$.

5.3.9 Supporting language extensions

A key feature of our technique is that we can define *structural recursions* over extension models, by introducing new recursive types. In our framework this is typically a ModelElementType, as described in Section 4.7.3. Examples include "nodes" of a tree, where we can recurse of the children of the node, and "states" in Hidden Markov Models, where we may look at transitions to other states. In order for the parallelisation method we have described to work on these structural recursions, they must be mapped to a total ordering; that is, have a one-to-one correspondence with the natural numbers. In other words each structural element has a unique natural number as an identifier. The recursive parameters can then be treated as natural numbers for the purposes of recursion analysis. It will also provide the layout for the dynamic programming table.

Furthermore, when extensions introduce new expressions on their new types, they must provide *affine expressions*, as described in the previous section.

To take a simple case, we consider an index over a sequence. An index has a natural total ordering (same as the integers), and expressions that manipulate the index do so in the same way as an integer. It is clear how we can identify a natural number with each entry, and provide an affine value for each expression.

More often, when structural recursion occurs, the arguments to the recursive calls are typically *accessors* on previous structural values, for example children of a node for a tree, or edges of a graph, or transitions in a HMM. To support this use accessors must provide a method for converting their value to an affine expression. In addition, when an extension implements an iterable expression it must follow the rules stated above, e.g that the start and end ranges must be expressible using affine functions of the original parameters.

A simple example involves trees. We can iterate over the children of a node in a tree, and we can recurse on those nodes. If we adopt a *depth-first, post-order* assignment of the total ordering of natural number IDs to the nodes, then the children of any node are guaranteed to have smaller IDs than their parent. We can therefore constrain the range of

the child node c to lie in the range $[0 \ldots x)$, where x is the number associated with the current node. In this case the start and end values of the range are easily expressible as affine equations, and act to constrain the possible recursion from all nodes, to only those child nodes. In turn, this may result in finding a schedule which serialises over the nodes of the tree e.g. $S_f(n, \ldots) = n + \ldots$ where n is a tree node.

5.3.10 Multiple problems

Our technique so far discusses how to map a single problem to a single multiprocessor – rather than employing all multiprocessors in a GPU. Typically a single problem is *tiled* across the multiprocessors, however bioinformatics often deals with large numbers of small problems. The compilation algorithm above makes use of the precise bounds of each domain to determine the best S_f and a_f. In theory, this means that different problem sizes may require different schedules to run efficiently. Consider a function with a recursive case of $f(x, y) = f(x-1, y-1)$ which may require a schedule of either $S_f(x, y) = x$ or $S_f(x, y) = y$ to minimise the number of partitions.

In practice, we find that for regular dependencies of the sort common in bioinformatics, the relative sizes rarely make a difference to the "minimised" schedule. For the common case, we simply use the known *ranges* of the runtime parameters as the ranges of those parameters in the CSP. If there is a single minimal schedule which applies to all possible ranges, then, if the CSP terminates, it will find a single, unambiguous answer. If there are multiple minimal schedules, it will produce an ambiguous answer, where at least one a_i may be assigned multiple values.

When an ambiguous answer is found, we arbitrarily choose one of the schedules. If a more sophisticated mechanism was required, the following proposal is one which would fit with our system. The principle is for *dynamic conditional parallelism*. This method derives multiple, minimal schedules, then determines at runtime which schedule is appropriate for each task. The advantage is that each task is given the optimal schedule.

The key to this technique is the derivation of conditions under which each particular schedule is *minimal*, which we then evaluate at runtime on a problem-by-problem basis to determine which generated parallelisation code to use. For example, given a function $f(x, y) = \ldots f(x-1, y-1)\ldots$, the minimal schedule depends on the length – when $n_x < n_y$ then $S_f(x, y) = x$ otherwise $S_f(x, y) = y$.

Our solution proceeds by adapting the CSP for the schedule to remove the goal value. Instead of deriving a single, minimal, solution, we instead propagate the constraints to derive the valid range of each variable. The descent functions must be uniform, as we can no longer consider runtime ranges.

Given the ranges of values found by the CSP, we could loop over all the possible coefficients for the scheduling function. However, we would identify many schedules that are valid but never minimal for any value in the domain. e.g, in our previous example we might find the following non-minimal schedules $(2, 1)$,$(2, 2)$,$(3, 3)$, when it is clear that the only minimal schedules are $(1, 0)$ and $(0, 1)$. Formally, a schedule is *minimal* iff the coefficients a of the schedule satisfy $\exists \bar{x} \in X : \forall \bar{b} \in P : \bar{a}\bar{x} < \bar{b}\bar{x}$, where P is the set of valid schedules.

Identifying the complete set of minimal schedules a priori is tricky. Instead, we determine a subset of the minimal schedules with positive coefficients using the following method:

1. Create all $n!$ permutations of the domains e.g in two dimensions $1, 2$ and $2, 1$

2. For each permutation, we minimise each dimension in turn, propagating the constraints until we find a solution which we add to the set. We therefore find the first lexicographical solution with respect to the permutation of dimensions. Each defined solution is minimal for some $\bar{x}$. We note that in practice the majority of problems have a single schedule, so the set size is far smaller than $n!$.

It may seem that this method is potentially expensive. To limit this effect, we restrict the coefficients to a small fixed number (10) that is customisable by the end user and note that

n is usually small. Additionally, unlike the runtime analysis described for single problems, this analysis can be performed during compile time.

For each schedule found, we generate code (using the same mechanism described in the next section, Section 5.3.11). We use the conditions to determine which of the generated code paths to take. When using task-per-block, this does not lead to any unnecessary code branching on the GPU, as a single problem will be allocated to a single multiprocessor, and each thread on that multiprocessor will choose the same schedule.

5.3.11 Program synthesis using the Polyhedral Model

```
parfor threads t in 0..tn
   for p in 0..max_partition:
      for elements of p assigned to t
         x0,...,xn = I(p, t)
         farr[x0,...,xn] = f(x0,...,xn)
      sync
```

Figure 5.5: The program synthesis template

Scheduling functions provide a succinct way to describe the dependencies of a problem. However, to have any practical utility, we must be able to use them to synthesise a practical and efficient program.

Figure 5.5 outlines our code generation approach: we will loop over the time-step partitions of the schedule in the domain, computing each partition entirely before synchronising (in task-per-block) and continuing on. Each thread will be allocated a number of elements to compute. For each partition, each thread will loop over all assigned elements.

We will use the *polyhedral model* [65] to generate code from our schedules. In this approach, we consider the original domain of the recursion as a polyhedron – specifically, a convex polyhedron representing the elements of $X_1, \ldots, X_n$ that the recursion visits.

In this model the scheduling functions are an *affine transformation* of that source polyhedron to a target polyhedron. The polyhedral model has been extensively researched in

relation to loop parallelisation, but prior work has also been completed on uniform recurrence relations ([56, 118] amongst others).

Code generation takes place by constructing a set of nested loops which iterate over every element in the domain. These loops take into account the dependencies defined by the schedule. The computation of each element proceeds by calling some function I to determine the indices of x associated with the current partition, thread and thread-step. We can then calculate the value and store it in a dynamic programming array.

CLooG [11] is a code-generation tool that can determine iterations over dimensions in this way. It takes, as input, the source polyhedron and the schedule – or *scattering function*, as it is described in CLooG. The output of this is a series of nested loops that iterate over the entire domain. In our case, the first, outer, loop will be over our partition time-steps. Each further nested loop represents a new space dimension of the target polyhedron, and the collection of all those inner nested loops iterates over all elements in the partition. This transformation should ensure that each element of the source – recursive – domain has an equivalent element in the target domain. Assuming that the dimensions are linearly independent, the number of dimensions in the target polyhedron will match that of the source polyhedron.

Using a scheduling function alone as an affine transformation creates a mapping from the recursive domain to a single dimension – the time-step. The elements of that time-step still need to be enumerated. CLooG generates a set of appropriate inner loops for these cases.

Consider the edit distance example, where $S_f(x, y) = x + y$. We use this as an affine transformation of the domain to one ordered by time-step then iteration (Figure 5.7 for the transformation, Figure 5.6 describes the output given by CLooG).

Depending on our choice of execution model, we may need to perform further transformations on these loops in order to provide the appropriate parallelisation. In a *task-per-block* system we will need to map the elements within a time-step to threads, by unrolling

```
for (p=0;p<=m+n;p++) {
 for (i=max(0,p-m);i<=min(n,p);i++) {
   S1(i,p-i);
 }
}
```

Figure 5.6: CLooG output for the edit distance problem with a scheduling (scattering) function of $S_d(x, y) = x + y$.

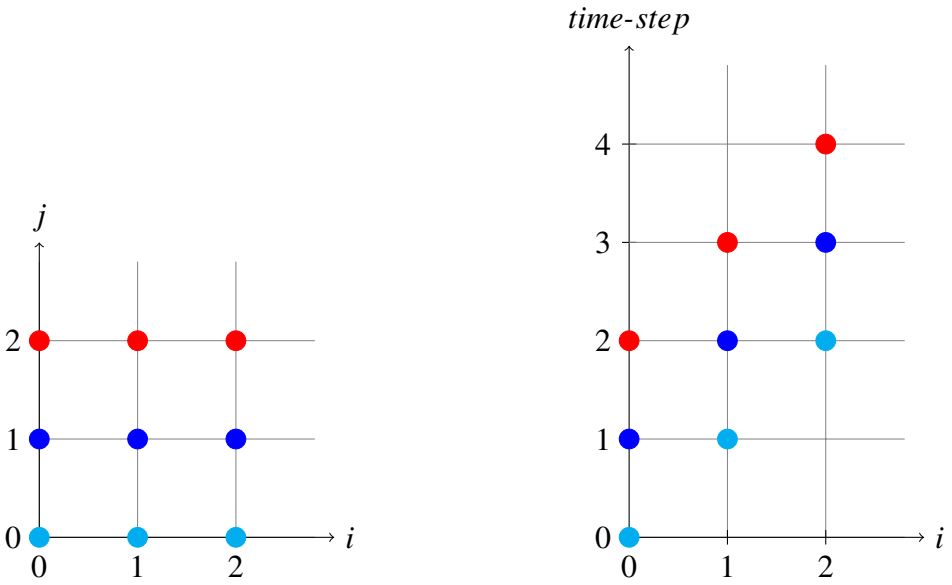


Figure 5.7: Visualisation of the transformation for the 3x3 edit distance, the original polyhedron and the transformed polyhedron.

one of the space dimension loops. We choose the inner loop. Given *tn* threads, with the thread designated by *t*, we can convert this loop:

```
for (int v = <start>; v < <end>; v+=<inc>)
```

by computing the entire range in groups of *tn* threads, like so:

```
for (int v = <start> + t; v < <end>; v+=(<inc>*tn))
```

Figure 5.8 gives our transformed parallel version of the loop for the edit distance example problem.

```
parfor threads t in [0..tn) {
 for (p=0;p<=m+n;p++) {
  for (i=t+max(0,p-m);i<=min(n,p);i+=tn) {
    x0,x1 = i, (p - i);
    farr[x0,x1] = f(x0,x1);
  }
  sync
 }
}
```

Figure 5.8: Converting the CLooG loop.

5.3.11.1 Determining the Task ID

Whichever strategy we use, we need to determine a *task_id* to decide which task we should be evaluating in this particular thread or block. In a *task-per-block* model, we can compute the task_id using the identifier of the current block. Assuming we can refer to block indices by gx, gy, gz and block sizes by ngx, ngy, ngz, then $task\_id = blockOffset$ where $blockOffset = firstBlock + gx + (gy * ngx) + (gz * ngx * ngy)$.

In a *task-per-thread* system, we need to further refine the task_id by considering the thread offset within in the block. This higher level of granularity can be computed using $task\_id = blockOffset * blockSize + threadOffset$ where $threadOffset = tx + (ty * nx) + tz * nx * ny$ and $blockSize = nx * ny * nz$.

In both systems, we can provide a task offset in order to start evaluation at a particular task. This allows us to divide execution into multiple kernel launches, where appropriate.

5.3.12 Limitations of the approach

We have made it clear that we are only considering affine scheduling and descent functions – that is, linear combinations of the coefficients. We justify this limitation in three ways:

1. Good solvers are available which can determine solutions in linear cases – if the recursive parameters or the scheduling function were not linear, we would not be able to use CSP or linear programming techniques.

2. We can synthesise good target code in the linear case; it is not clear how a non-linear function would be synthesised because it would be difficult to find the inverse transformation.

3. Problems in bioinformatics, our target domain, infrequently require non-linear cases. All the examples included in Chapter 3 are linear.

Our approach is optimal with respect to the number of partitions we evaluate. However, this does not guarantee a minimal execution time. In particular, we will typically evaluate a partition in *blocks* of threads. Doing so can lead to wasted execution on the GPU, where a schedule with a non-minimal number of partitions might, in fact do better.

5.4 Memory location allocation

One of the most important decisions a developer in our framework has to make is where their data is to be stored on the GPU – the *memory location*. The precise location of data in the complex memory hierarchy can make a significant difference to the runtime for the generated application – a good allocation can potentially halve the runtime.

This decision is normally made by a GPU expert according to a certain amount of intuition and word-of-mouth information on appropriate settings for particular scenarios. Less expert users are often lost in the mire of complexity surrounding the many choices available.

One standard solution is to provide *auto-tuning* tools. Auto-tuning proceeds by repeatedly running the same application with different parameters on the end-users machine until finding a good set of parameters. One advantage of this approach is that it will find parameters that are suitable for the precise hardware of the end-user. One disadvantage, however, is that the auto-tuner must be run for each different application.

For a "typical" GPU application this may not be a problem, as a single auto-tuning run upon, say, installation may be sufficient. For applications written in our language,

this becomes more problematic, since each application may differ substantially in terms of read/write access patterns, size, data structures and so forth, necessitating new auto-tuning runs. Furthermore, a focus for us is on incremental development of new applications – in these situations, you may not want to run auto-tuning on each situation. Whilst auto-tuning may still play a role, we instead focus here on determining useful initial defaults.

As a domain developer, using the data layout framework simplifies the task of allocating output arrays to certain memory locations. In normal C for CUDA, the NVIDIA API for GPUs, different initialisation, load and intermediate operations are required for each location. In our framework, a single parameter change is all that is required.

Since the choice of memory location is detached from the rest of the high-level DSL development, it leaves the system free to implement an *automatic memory allocation* algorithm. This can be enabled by setting the parameter to AUTO. This is a further example of abstraction enabling the system to make low-level GPU decisions automatically.

Automatic memory allocation occurs by leveraging existing GPU expertise, in the form of cost based optimisation algorithms which can be specified, and customised, by GPU experts. We are essentially trying to encode the GPU expert into the compiler. It is our choice of a simple host language, along with our DSL API, that makes this derivation possible.

5.4.1 GPU memory architecture

As we discussed in Section 2.3.3.2, the appropriate use of the complex memory hierarchy of the GPU is vital in achieving the maximum performance from a particular kernel. Each location provides a different sort of performance characteristic with respect to latency, bandwidth, caching policy and scope.

This choice of memory location is far from irrelevant to the overall performance. On a simple benchmark of coalesced loads in a single multiprocessor we have observed performance differences of a factor of two. To further complicate matters, each iteration of

the hardware provides a different balance between these memory areas. Simple allocation schemes are therefore unlikely to work well.

In this section we consider allocating structures to the following memory locations: *uncached global memory*, *cached global memory*, *texture cache*, *constant cache* and *shared memory*. Cached global memory is cached in L1 and L2 cache; uncached global memory is only cached in L2, a feature available in PTX on a per-load basis. Memory copy operations, for example from global to shared memory, are automatically generated.

5.4.2 Problem outline

Our problem is this: we have a number of arrays and a number of memory locations. We must allocate the arrays to our memory locations in order to produce an efficient program. In this section we describe one algorithm for achieving that mapping. Our system is, however, flexible enough for a GPU expert to define their own allocation algorithm (e.g first fit etc.).

Our system is to determine the cost of each possible allocation, and therefore choose the lowest cost allocation. The aim is to provide a comparable cost between different allocations on the *same* program. Many factors influence the performance characteristics and modelling them all is infeasible. Instead, we ask the pragmatic question "how does the performance cost on *this* program vary by allocation". This simplifies matters greatly; we do not need to consider cost of arithmetic operations, number of problems etc. because these remain constant no matter the allocation.

The outline of our algorithm is as follows. We pre-compute the cost estimate for placing each array in each memory location according to a customisable cost function. We then generate each permutation of allocating arrays to memory locations, determining a total cost by applying a customisable summation function over the cost of each of the independent allocations. We then use the lowest cost allocation. The algorithm itself is completely customisable: the cost and summation functions may be tweaked or modified in order to

improve the performance across more scenarios.

The set of locations are determined based on the particular device from the following set: SHARED, TEXTURE, CONSTANT, GLOBAL, GLOBALL1. Note that we treat global memory and global L1 cache as two different memory locations in order to model the vastly different performance. GLOBAL data may or may not use the L1 cache, depending on what other arrays have been allocated – this provides the option for large, single-use data loads to avoid polluting the L1 cache. Locations with on-chip caches, but backed by larger memory locations, are penalised when the size of data allocated is larger than the cache.

5.4.3 Input parameters

The input parameters for our algorithm fall neatly into two categories. The first category is the precise device we are running on. The data in this category is either queried from the device directly, or in the case of performance characteristics that can't be queried on demand (e.g latency) we pre-compute values that are associated with the compute capability, which can be queried. Pre-computed values are determined according to the results and methodology of [122]. The second category is the function and the use of the arrays.

The input parameters used are

- Memory locations (cached/uncached size, latency, bandwidth)
- Warp size, and memory dispatch size (warp or half warp)
- The size and type of each array. We can determine this because we provide a runtime system, compiling on-the-fly.
- The sites at which each array is used (see Section 5.4.6.1 for details), including:
  - An estimated repetition count, based on the outer loops
  - The size of the load

– The indices into the array, in terms of local variables.

5.4.4 Allocation algorithm

The allocation algorithm itself is akin to a bin packing problem, but with a fixed number of bins. Each array has a cost associated with being placed in a specific memory location, determined a priori. We then select the placement of data that minimises the overall cost according to some summation function (Figure 5.9).

We compute the lowest cost solution by considering all possible allocations. This is therefore an $O(k^n)$ algorithm, where k is the number of locations and n the number of arrays. We justify this by noting that n and k are typically small, neither reaching above 5 in our example extensions – a typical runtime for 5 elements is around 50ms in our benchmarks on an Intel Sandy Bridge i7 2700k (single core). In practice, if larger models are required and run time is an issue, we could easily swap out our exact version with an approximate version which hueristically considers a smaller subset of all allocations. A corollary of this is that, if required, we could feasibly implement an auto-tuning system.

```
public Allocation alloc(Device dev, Array[] arrays) {
  Allocation best;
  long bestCost = Long.MAX;
  costMap = preComputeAllocCost(dev, arrays);

  for (Allocation alloc : generateAllocs(arrays, costMap)) {
    long cost = summate(alloc);
    if (cost < bestCost) {
      bestCost = cost;
      best = alloc;
    }
  }
  return best;
}
```

Figure 5.9: The allocation algorithm

5.4.4.1 Cost function

The cost function provides an easy way to specify the cost of allocating a particular array to a particular location. This function can be modified, or even replaced, by a GPU expert to provide a better mapping, and can be customised to support new devices. This function is used to pre-compute the cost map for each array.

```
public long cost(Device dev, Array array, Location l) {
  long totalCost = 0;

  for (UseSite us : array.getUseSites()) {
    long execCount = us.getExecCount();
    long sizeOfElement = array.getSizeOfElement();
    // This is memory allocation warp size
    long warpSize = dev.getThreadsPerWarp();

    long byteSize = warpSize * sizeOfElement;

    // In clock cycles.
    long latency = device.getLatency(l);
    // Bytes per clock
    long bandwidth = device.getBandwidth(l);

    // Compute the cost by combining latency and bandwidth
    // Note that will we normally give latency a significantly
    // smaller weighting
    long cost = combine(ceil(byteSize / bandwith),latency);

    cost *= execCount;

    totalCost += cost;
  }
}
```

Figure 5.10: The default cost function

The cost function computes a comparable figure for estimating the cost of all memory accesses in **one warp** of a function call (e.g dynamic programming table) for array array in memory location m on device dev. We compute a figure we call *ACC*, or *Abstract Clock Cycles*. These are loosely comparable to clock cycles on the device. We choose this as a measure because it is of the right granularity. Note that we factor in both the bandwidth and

the latency of the memory location. For well optimised code the latency is usually hidden, so we can give much greater credence to the bandwidth of the location.

The default cost function (Figure 5.10) computes the cost by considering each location at which the array is used, considering the read size, the execution count and the latency and bandwidth of each location. We will delay discussion of how we determine these factors to the next section.

5.4.5 Summation function

The summation function considers the total cost of the allocation. This is required because arrays are not independent – in particular, we cannot allocate more data to a location than it can hold, and performance may suffer if we put more data in a location than can fit in the cache.

Our default summation function (Figure 5.11) loops over each memory location, summing the total amount of data for that location, and eliminating or penalising allocations that do not fit the size of that location. An example of such a penalty is for shared memory – data in shared memory is local to a block, and so arrays that are allocated in shared memory are repeated for each block. Thus, if the size of arrays allocated to shared memory is too large, it can reduce the occupancy, and therefore latency hiding. We therefore scale the cost according to the predicted occupancy reduction.

5.4.6 Static analysis techniques

Our system of automatic allocation requires not only detailed information about the size and type of arrays, but also how it is to be used. In this section we discuss a few analysis techniques for determining such pieces of information.

```
public long summation(Alloc a, Device dev) {
  long totalSize = 0;
  for (Location l : a.locations()) {
    long cost = 0; long size = 0;
    for (Array a : getArrays(l)) {
      // The cost is precomputed
      cost += getCost(a, l);
      size += a.getArraySize();
    }
    long locationSize = layout.getSize(l);
    if (size > locationSize) {
      return Long.MAX_VALUE;
    }
    cost = penalise(cost, size, l);
    totalSize += cost;
  }

  return totalSize;
}
```

Figure 5.11: The summation function

5.4.6.1 Use analysis

A GPU is typically connected to the processor through the PCIe bus; transfer between main memory and GPU memory is therefore many orders of magnitude slower than accessing the GPU memory directly. In addition, our model scheme allows the domain developer to generate a large number of arrays for many different operations. A simple optimisation, then, is to generate, and transfer, only those arrays associated with operations which actually occur in the function to be run.

This is relatively straightforward – we traverse the IR code generated for the function in order to determine the collection of arrays that may be visited during execution. We can then request only those arrays which are required for this particular run. Furthermore, since there may be multiple function calls to the device during execution, we leave those arrays that may be used again on the device, and only transfer any new arrays that might be required for the new function call.

5.4.6.2 Estimating read cost

An important factor in the auto-allocation scheme is estimating how often each array is used. From our use analysis we develop a second, more sophisticated analysis for determining the *average execution count* of each expression. This can then be used to estimate the total read cost for each array.

We start by identifying the set of *use sites* for each array by instrumenting the data layout API so that it records the use when an access is generated. For each use, we record the execution count and the index of the access. The total read cost is dependent on the chosen memory location. We pass this *use* information to the cost function, to help it determine the cost of placing this array into a particular location, and therefore determine the lowest cost allocation. Thus arrays with a high-frequency of use will be more likely to be promoted to a lower-latency memory.

Formally, the execution count analysis is an estimate for each expression of the number of times it is executed per execution of a single warp, where each thread in the warp is computing a single cell of the dynamic programming table.

The analysis is as follows:

$$
\begin{aligned}
execCount(Func\ f) &= 1 \\
execCount(\_\ x) &= execCount(parent(x)) \\
execCount(Iterator\ i) &= execCount(parent(x)) \\
&\quad * i.iterableExpr.iterableCount
\end{aligned}
$$

Due to the simplicity of our language, the majority of expressions are executed once per execution of a cell, and thus have the same execution count as their parents. Expressions associated with a function have an execution count of one. Since we tabulate the result using dynamic programming, function and recursive calls are not considered to change program flow – their values will have been pre-computed.

The only expression that changes the execution count is the iterator expression. In

order to estimate the execution count for the body of an iterator, any *iterable expressions* must implement an `iterableCount` that provides an estimate of the number of elements in the iterable. `iterableCount` is implemented by reference to the *actual* call because our system is a runtime system. We therefore know the actual parameters to the function call when analysing the function and generating GPU code.

5.5 Other optimisations

Efficient automatic parallelisation is primarily dominated by the two factors previously discussed: the parallelisation algorithm and the allocation of data to memory locations. There are, however, a number of other choices that can have a significant impact on the eventual runtime of the generated code, such as the strategy chosen, the block and grid sizes as well as optimising the amount of data we are required to store.

One limitation of the work in this section is that we have made an implicit assumption of independence of these choices, except where explicitly specified. In practice they can have an effect on each other; we have only consider this effect when it appears to have a significant impact.

5.5.1 Strategy code generation and analysis

In Section 5.2 we identified two different parallel strategies for code generation:

- **Task per thread** – each thread evaluates tasks.
- **Task per block** – The threads within a block work co-operatively on tasks.

In principle, different strategies may be better suited to certain tasks, or certain GPUs, or even certain tasks at certain sizes. A significant amount of evidence in the domain of bioinformatics (see Section 3.6.1) suggests that task-per-thread is often better suited to

smaller sequences, whilst task-per-block is better suited to longer sequences. The progression is typically implemented using a *threshold* on sequence length; sequences longer than the specified threshold are evaluated using task-per-block, rather than task-per-thread. This is particularly noticeable in Smith-Waterman and Hidden Markov Model applications.

Why should this be so? Task-per-thread strategies can often hide latency more successfully, because they make better use of ILP (more independent work), and there is no synchronisation required between threads. Furthermore, with a suitably arranged input, the traversal of the dynamic programming table can be achieved so as to maximise work per thread, and improve the reuse of input characters and so forth, thus reducing the required memory bandwidth. On the flip side, on larger problems, the better sharing and reuse of data achieved by task-per-block, especially using shared memory as a scratchpad, can provide improved performance. Furthermore, recall that each SM has a fixed number of registers, and a fixed quantity of shared memory. If each task already requires a significant amount of resources then by selecting task-per-thread we may have to comprimise by either reducing the number of tasks resident on the SM, or reducing the number of registers or amount of shared memory available to each task, either of which may result in a reduction in performance.

In practice, we have found similar performance from both our task-per-thread and task-per-block implementations. The major reason for this is that we have focused our efforts on task-per-block parallelisation. Consequently, we use the same schedule for code generation for both task-per-thread and task-per-block. This schedule is optimised to maximise the amount of intra-task parallelism. This results in the same traversal of the dynamic programming table – for example, along the diagonal in Smith-Waterman – for task-per-thread as task-per-block. There is no fundamental reason why the code generator could not generate a more efficient traversal for task-per-thread, using the polyhedral model with an appropriate transformation.

5.5.1.1 Memory Coalescing

In order to achieve high performance for either strategy, we must try to maximise the number of coalesced memory operations (as discussed in Section 2.4.3) and thus reduce the amount of memory bandwidth required to service load and store requests. Our choice of strategy affects what will be executed on each thread at each step; where *task-per-block* will typically access sequential cells from the data for the same task, *task-per-thread* will typically access the same cell in different data for multiple tasks. Accordingly, we must adjust how we store our data.

There are two specific places where such memory coalescing is particularly important:

- *Input Sequences* – e.g those loaded from files. In task-per-block, consecutive threads will typically load consecutive elements of the sequence, whilst in task-per-thread, consecutive threads will typically load from the same offset, into different sequences. We reorder the input sequences based on the strategy.

- *Sliding Window table* – as we discuss later in this chapter (Section 5.5.4), we can store intermediate results into a smaller table, based on the length of dependencies required. We provide one such window per task; in task-per-block, we access one table per block, reading consecutive cells to and from the table. In task-per-thread, we access multiple tables per block, accessing the same elements in many tables. We therefore place the task as the inner dimension on task-per-thread, and the outer dimension on task-per-block.

5.5.2 Block size heuristic

Recall that the *block size* of a CUDA kernel is the number of threads allocated to a single block, and scheduled together to a multiprocessor. A unique feature of the GPU is that each resident block retains any shared memory and register allocations even when the scheduler context switches to another warp. This allows context switching to be very low-cost, but

may limit the number of blocks that can be allocated to each multiprocessor, due to shared memory, register, or warp pressure.

Limiting the number of blocks consequently limits the number of threads per multiprocessor (known as the *occupancy*), which can have a profound effect on performance (as described in Section 2.4.6), by reducing the latency hiding potential through TLP.

Recall that our code generation produces a set of nested loops, where the outer loop is the synchronised partition loop. The inner loops represent iteration over a subset of our parameters. We allocate our threads to the innermost loop, representing the innermost recursive parameter.

Unfortunately, there is no clear relationship between the block size and performance for all applications and parallelisation strategies. We instead describe here a *heuristic* method for determining an appropriate block size for our particular applications. The main contributing factor is our choice of parallel strategy: task-per-block or task-per-thread. A further factor is the average size of the problem. Our performance findings are detailed in Section 7.3.5.

In order to improve the flexibility of the framework the block size heuristic is customisable by GPU experts, by providing a new function for computing the block size, based on the execution parameters. This enables GPU experts to improve on the heuristics provided here, adding support for new cards or establishing better heuristics. The intention is not necessarily for the domain developer or end-user to customise this function.

Task-per-block For task-per-block we use two contributing factors to determine the block size:

- The *average* partition size – the size of the partition is the size of the inner dimension of the loop, that is the loop that is evaluated in parallel. The number of threads should be no bigger than this in order to reduce wasted threads. When the partition size is large, we empirically find that maximising the number of threads is the most effective

approach.

- The number of blocks resident on the GPU – this, rather than the occupancy, is empirically more important for determining performance on the GPU for our applications. We propose that this is because a lower occupancy, with a higher block count, and smaller block size will be better at hiding latency, as each block is functionally independent of every other, whilst many warps in the same block may be suspended whilst waiting for a final warp to complete.

Accordingly, when the average inner dimension (e.g partition) size is below 500, we maximise the number of blocks resident on the GPU. When it is above, we maximise the number of threads.

Task-per-thread We pragmatically choose 64 threads as an appropriate block size for task-per-thread. Empirically this maximises performance for our task-per-thread applications. With further optimisations to the generated code, this balance might change.

5.5.3 Grid size heuristics

On desktop machines, kernels typically have a timeout of around five seconds to ensure that they don't monopolise the GPU. Furthermore, long running kernels have a significant impact on the responsiveness of the desktop. Since desktop machines are an important part of our target audience, we divide the work of the kernel into multiple grids. Launching a grid, however, has a small overhead, and larger grids generally provide better performance, so we will need to choose a grid size that is appropriate in order to balance these two considerations.

NVIDIA [88] only has this to say about setting the grid size:

> The number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute. Further-

> more, there should be multiple active blocks per multiprocessor so that blocks that aren't waiting for a __syncthreads() can keep the hardware busy....To scale to future devices, the number of blocks per kernel launch should be in the thousands.

This ties in with the idea of *occupancy* discussed in the previous chapter – each multiprocessor should have multiple blocks running in order to hide latency. By dividing work into multiple grids, we may inadvertently reduce the occupancy on the GPU, by providing too little work to fill up each multiprocessor on each launch.

In order to ensure we reach the maximum occupancy (at least initially, before any threads terminate), we will need to dispatch a minimum number of blocks determined by the follwing formula:

$$\textit{Minimum grid size} = \textit{Number of SMs} * \textit{Blocks per SM} \tag{5.5}$$

As we previously noted, we may not need to achieve a high-level of occupancy in order to cover the latency in the application, so this represents an over-estimate of the minimum number of blocks required to cover latency. To achieve 100% initial occupancy requires around 100-300 blocks, depending on the device.

Through empirical testing (see Section 7.3.6 for further details) we make a number of observations:

- Larger grid sizes, in general, provide better performance. In the absence of kernel launch timeouts, and whilst respecting memory constraints, we are better off providing no limit on the grid size.
- If we are to provide a limit, then it is notable that the performance is smoothed out beyond 256 blocks per grid, which matches with our intuition provided above of the number of blocks we might need.

- Blocks sorted by workload size prove to have odd performance characteristics, where performance improves up to a multiple of 8, 16 or 32 (depending on the particular device), then dramatically drops, before gradually improving as it nears another multiple. We are not aware of any reason for this to be the case, but can only speculate that the block/warp scheduler is sensitive to such workloads. Fortunately, this effect flattens out once the GPU is initially fully occupied.

Taking these factors into consideration, we take a pragmatic decision to choose 8192 as a default grid size; it ensures maximum occupancy for all known devices, as well as matching the advice provided by NVIDIA. Furthermore, it is large enough that kernel launch overheads are not problematic. Exceedingly long running kernels may still fall foul of the kernel timeout; in this case (and for optimising performance reasons), the end-user can still specify a different grid size. Users running on cards without a timeout may remove the limit altogether. Finally, we may reduce the grid size if memory constraints are a problem; we will run with the largest grid size below the specified limit which fits on the desired GPU.

5.5.4 Sliding window

The computation of each *cell* of the dynamic programming table typically requires a number of reads – from dependent cells – and a single write operation. These dynamic programming tables are often large – sequences of interest range from hundreds to tens of thousands of characters long – which means we are unable to place them in high-bandwidth shared memory, and instead they have to be placed in the slower global memory.

Furthermore, the access patterns of these operations are often not coalesced, accessing multiple cache-lines and causing poor performance on both devices with cached global memory and without. A diagonal schedule, as in the edit-distance problem, using task-per-block, will potentially require accesses across all the different rows or columns of a table at each time-step. Whether the table is stored in column or row order, this will produce poor

cache access patterns, often thrashing.

We can both improve the access patterns and reduce the quantity of data required by noting that typical recursive applications do not require the results of all the previous time-steps. In fact, they often require only the results from a few previous time-steps – we denote this as the *offset*. Exploiting this offset, we can arrange for a *sliding window* to be placed over the dynamic programming table, indexed not by the original parameters of the domain but by time-step offset and thread id, which moves across the overall dynamic programming table as we proceed. This is an array contraction style optimisation which is commonly applied to GPU applications; importantly, we will show how we can automatically derive this offset from our language so that this optimisation can be applied when appropriate.

To take a concrete example we again consider the edit distance problem. Note in the dependency graph (reproduced in Figure 5.12) that the longest dependency is two partitions, or time-steps, previously – $d_{1,1} \rightarrow d_{2,2}$, for example. In this case we therefore need a window of size 3 – one to store the current time-step, and two for the previous two time-steps. Since the widest partition is three elements, we will need a 3 by 3 storage table. Figure 5.13 gives the table during the third time-step.

Note that although in this case the sliding window is no smaller than the original table, in general the sliding window size grows at $O(min(n, m))$ versus $O(nm)$ for the general table size. Furthermore, this difference in growth size is why it is acceptable that the sliding window leaves some cells empty with skewed schedules, for example, in the edit distance problem described above – the sliding window will require less memory, even with the empty cells.

5.5.4.1 Sliding window analysis

The first step to identifying a sliding window is to identify a fixed offset – the sliding window size. This is the upper bound on the offset over all possible recursive calls. As this analysis occurs after the scheduling function has been chosen we assume the schedule

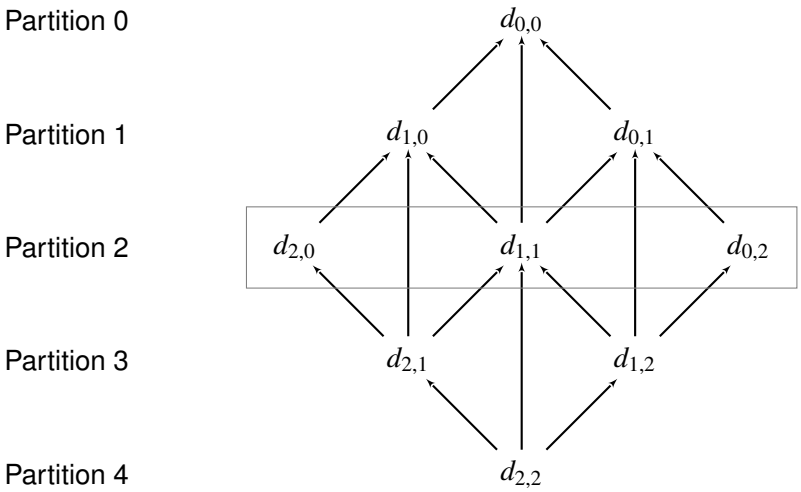


Figure 5.12: The edit distance dependency graph, with the third partition highlighted.

	Offset -2	**Offset** -1	**Offset** 0
T1	$d_{0,0}$	$d_{0,1}$	$d_{0,2}$
T2		$d_{1,0}$	$d_{1,1}$
T3			$d_{2,0}$

Figure 5.13: An example of a sliding-window table, during evaluation of the third partition i.e filling in the Offset 0 column.

is both correct and considered efficient. The analysis considers each recursive call in turn, computes the offset for that call by substituting the recursive descent functions into the schedule. The sliding window size is then the maximum of those offsets, computed by:

$$max_{xr_1,\ldots,xr_m}(S(x_1,\ldots,x_n) - S(xr_1,\ldots,xr_m)), \forall x_1,\ldots,x_n$$

This will produce an exact result when all recursive descent functions are uniform. In the more general affine case, we will not be able to determine a suitable offset.

5.5.4.2 Sliding window implementation

Implementing the sliding window optimisation requires us to consider to different steps:

- Modifying the accesses for each recursive call with new accesses to the sliding window.

- Allocating an appropriately sized piece of memory, in an appropriate memory location.

Modifying the accesses Modifying the accesses is not just a case of changing the base address of the array – we must also change the indices used to access that array. If the schedule, and nested loops produced by CLooG, are thought of as a *transformation* of the domain, the sliding window can be thought of as the transformation of the dynamic programming table. We therefore need to access this table by the *schedule adjusted indices*.

The dimensions of the sliding window are those of the transformed domain; the outer dimension is the partition or time-step, the further dimensions are those of the inner nested loops around the elements of the partition. In Section 5.3.11 we noted that we do not change the number of dimensions in our transformation, as each recursive parameter is linearly dependent. For this reason, the sliding window will have the same number of dimensions as the original table.

Recall that the outer loop is that of the time-step or partition; the $(n-1)$ inner loops each represent one of the n parameters, with the nth parameter linearly derived from the other parameters and the time-step. Each of these inner loops is shifted so as to visit the correct indices for that dimension in that time-step; this is done by setting an initial value and a step or stride value. For this reason we cannot use the raw transformed indices; we instead map them to consecutive natural numbers in order to minimise the required space.

To formalise this approach, consider a function $f(x_1, ..., x_n)$, with a recursive call $f(xr_1, \ldots, xr_n)$. Furthermore, assume that the scheduling function for f, S_f, has been implemented in CLooG, returning a set of loops L, where $|L| = n-1$. Each loop is uniquely associated with a parameter of the original call, x_i, and is of the form:

```
for (int xi = lowerbound(i); xi < upperbound(i); xi += stride(i))
```

Using this form, we can compute the sliding window index for this dimension, for this

recursive call, using the equation:

$$(xr_i - lowerbound(i))/stride(i) \tag{5.6}$$

This provides the indices for accessing the entry in the sliding window for the recursive call. Parameters that do not take part in the schedule are unaffected by the transformation. They will be accessed using their arguments directly.

Mutual recursion is also supported by providing a sliding window table per function.

Memory allocation Recall that the system may evaluate multiple tasks simultaneously. Each task has different inputs, and therefore different problem sizes; these result in different sliding window sizes for each task. On a NVIDIA GPU we have two primary options for storing the table – global memory or shared memory. In the case of the former, we need to allocate a contiguous block at least big enough to fit a sliding window table for every task. In the case of the latter, memory is allocated *statically*, and is the same size for each block[3]. It cannot, therefore, depend on the size of the dimensions for a particular problem. For correct execution we must therefore allocate enough shared memory to store the *largest* sliding window we encounter, at any point in any problem.

For either method, we must be able to identify the largest partition size in each of the problems. This will be multiplied by the window offset, which is fixed for any problem size. The identification of the largest partition is very much linked to the generation of the iteration code for that domain. This is because iterating over each element in a partition is the same as enumerating them. As we have seen in the edit distance example, this may be wasteful, leaving empty cells for some partitions, but is a necessary over-estimation.

This size problem has been extensively studied in the form of counting integer points in *parametric polyhedrons*. The loops produced by CLooG may be, together, considered as parametric polyhedrons, that is polyhedrons parameterised by the dimensions of the

[3]Using the runtime library, the size may also be defined at runtime, but only once per kernel invocation. The size is still static on the device, and cannot be specified per block.

specific problems. A number of libraries, such as `barvinok` [117], exist to compute the cardinality of such parametric polyhedrons for specific parameters i.e specific problem sizes. Whilst this approach is theoretically sound, for practical reasons – primarily that the framework is written in Java – we have not yet integrated any such library for computing the maximum size of the partition.

An alternative simplified method is adopted, that computes a conservative (i.e guaranteed larger) value for the maximum partition size. Recall that each loop is associated with a single parameter. The number of iterations in each loop is computed by $(upperbound - lowerbound)/stride$. It is clear that the upper bound on this expression is the maximum size of the associated dimension in any problem. We can compute the total upper bound of the partition size by multiplying the maximum size of each loop together.

This simplified method overestimates the result when different problems contain the largest dimension sizes. A pathological case would be a pair of two-dimensional tasks of size (n,1) and (1,n). In this case, we would derive a size $O(n^2)$, when, in fact, a size of $O(n)$ would be sufficient. Fortunately, most workloads on applications of interest scale linearly.

The next step is to determine whether to allocate shared or global memory. Using shared memory is generally desirable as it provides much lower latency and higher bandwith operation than uncached global memory accesses, and is user managed, ensuring that data is not expelled automatically, unlike a cache. The downside is that it is allocated on a *per-block* basis and remains in the limited shared memory (typically 8KB to 48KB) whilst the block is resident on the GPU. Excessive use of shared memory can therefore limit the number of blocks which can be resident on a single multiprocessor, and subsequently reduce occupancy and impair performance. At the limit, evaluating large problems on older devices, we may not be able to use shared memory at all.

Certain global memory operations can achieve the same amortised latency by using the L1 cache on applicable devices; in fact, the L1 cache and shared memory reside in the same memory, and can be explicitly divided. Effective use of the available memory is achieved

by exploiting both shared memory and cached global memory.

If the sliding window does not fit into shared memory, or would adversely affect the occupancy, we place the sliding window in global memory. In this case we can try to make use of the global cache in order to improve performance. The sliding window aids this aim by improving the locality of reads and writes to the table, thus increasing the likelihood that the data will remain in the cache. When allocating global memory for sliding windows, we allocate *pitched* memory in order to improve coalescing of reads and writes.

Note, this approach may not improve *coalescing* of reads – the original program may have had scattered reads for recursive calls that we cannot fix through this approach – but should reduce memory usage, and improve the locality of reads and therefore cache use.

Other optimisations A naive implementation would require some copying between each time-step to maintain our sliding window invariants (e.g each column would be copied left). We can eliminate this copying overhead by rotating which column is the current column, and computing all indices modulo the window size. This modulo may be computed efficiently using bit masks if the window size is rounded up to 2^k.

The sliding window is useful for problems where we are only interested in the end result – typically a corner of the recursion. In this case we simply store only the final result. However, we can also use it for problems where the end user might want to inspect the entire table, or we might need to do some back-tracking. In these cases we still use the sliding window for accessing values, but we also store values to the dynamic programming table. This is a small increase in memory usage, but still maintains the advantage of placing the recently accessed data in low-latency memory.

5.6 Related work

The work in the first part of this chapter has been previously published, in shortened form, in the paper "Synthesising Graphics Card Programs from DSLs" [17]. This chapter ex-

tends the work on code generation and schedule analysis with respect to the custom extensions. A key new section is that of the memory location allocation algorithm, along with detailed descriptions of important optimisations, such as the sliding window.

5.6.1 Polyhedral Model

Schedules were first proposed as a means of parallel analysis of recursive functions in the seminal paper by Karp et. al. [56]. A practical implementation was first provided by CRYSTAL [24], where recursive functions were characterised by *space-time mappings*, equivalent to our schedules. A period of refinement occurred in the late 80's, in particular determining schedule search algorithms for systolic arrays [99, 102, 123, 100]. A practical implementation of this work was MMAlpha [118], a language for describing recursions which could generate systolic array implementations. The computation of schedules from recurrences was summarised by Paul Feautrier [33]: however, much of the subsequent work has ignored recursions as a form of input, in favour of nested loops for parallelization.

For parallelisation of loops, the mantle was taken up by Leslie Lamport [61] with the concept of *wave-fronting*, but with the advent of the first massively parallel processors, systolic arrays, it propagated through a large amount of research, culminating in the development of the polyhedral model (summarised by Lengauer [65]) as a formal technique, and resulting in frameworks such as LooPo [38].

Recent work has refined the technique, developing large-scale practical frameworks for polyhedral code analysis and generation, such as CLooG [11], PLUTO [13] and the Polyhedral Compiler Collection framework PoCC [12]. These all maintain the focus on code generation for loop schedules. Polly [39] is a project that aims to integrate polyhedral model optimisations to LLVM, which may, in turn, activate GPU applications, as LLVM has recently acquired a NVIDIA PTX backend.

5.6.2 Polyhedral Model for GPUs

The advent of GPGPU has led to recognition of similarities between systolic arrays and GPUs, and thus that the polyhedral model may be used profitably on GPUs. This has concentrated on leveraging existing tools to develop end-to-end automatic parallelization frameworks.

One of the first papers to consider GPU parallelisation in terms of the polyhedral model was Baskaran et. al. [9], work that was later further refined and incorporated into a source-to-source framework using Pluto and CLooG that generates CUDA code [10]. Baghdadi et. al. [6] provided similar work based on the PoCC collection of compiler tools, and also generating CUDA code. A more recent effort, PPCG [116], works on similar lines, but provides more advanced techniques for placement of data on the GPU.

These share some similarities with our approach, so we will highlight the differences. First, and foremost, these take as input *nested loop applications* written in a general purpose language, generally C – our input is a specifically constructed *recursive language*. As a consequence, our scheduling analysis runs along slightly different lines. Furthermore, our language is JIT compiled at runtime, using runtime information in a way that is impossible for a source-to-source translator. Finally, the DSL nature of our applications is far more constrained, and as such our further optimisation techniques are more focused.

5.6.3 Domain-Specific languages for GPUs

We noted a number of DSL approaches to GPU development in Section 4.9. The majority of this related work either focuses on trivial parallelisation e.g map-reduce style array computations, with independent computations mapped to threads, or hard-coded parallelisation techniques, where user-code is implemented in some parallelisation routine pre-defined explicitly by the creator of DSL. In the former, translation tends to occur using *templates* or *skeleton* code snippets, instantiated using appropriate parameters; the results of these snippets are then merged to form the program. In the latter, e.g the *Delite* [20, 21, 14]

framework, the DSL developer has to explicitly described the parallel equivalent for each operation. In contrast, the parallelisation in our approach is not pre-defined – it is extracted from the recursive function implicitly, without any user or DSL developer input.

A recent research project on very similar lines to our research is CARP [1]. In CARP, Domain-Specific Languages are compiled down to an intermediate language, PENCIL, a loop-based language designed specifically for parallel analysis, which retains important information from the DSL. PENCIL is analysed using polyhedral model techniques, and parallel code in the form of OpenCL is generated.

In many respects, this is a natural progression of the research documented in this dissertation. By adopting an intermediate language to which many DSLs can compile, they achieve parallelisation over a wide range of domains. There are, however, advantages to proceeding with the approach we have adopted. By describing a fully integrated system for a single domain, we can provide further levels of optimisation based on our DSL knowledge (e.g cost model analysis).

In addition, the recursions we described are not naturally described as loops over a domain – there is no natural lexical ordering. In order to place our recursions into a language such as PENCIL we would need to pre-analyze the recursions to produce a correct loop, which would take the same steps as the parallel analysis. Furthermore, recursions provide a more declarative way of describing the dependencies between computations; the lack of execution order simplifies the parallel analysis, as well as simplifying the creation of programs in our DSL.

5.6.4 Parallelisation of functional languages

We discussed parallelisation of embedded DSLs within functional languages in Section 4.9.

Parallel implementations of fully fledged programming languages have been extensively researched over the years. For example, a recent tutorial in Haskell is [52]. One feature of our system is that we have consciously restricted the input language in order

to apply more aggressive optimisations, and in particular the polyhedral model. Another example, in the context of graphics cards, is [32], which demonstrates how to generate efficient GPU code for graphics problems. This also uses a restricted functional input language; however it does not address the parallelisation of complex dynamic programming recursions.

5.6.5 Parallelisation of bioinformatics applications

We have summarised the prior GPU parallelisation work in the bioinformatics chapter (Section 3.6.1). In essence, the majority take a manual *wave-front* approach, for example evaluating along columns for HMM applications, on a diagonal for edit distance problems, or the anti-diagonal for the palindromic RNA Secondary Structure prediction problems. These are typically hand-coded applications, where the parallelisation has been manually identified. A notable exception is the GPU parallelisation of Algebraic Dynamic Programming [36, 107], which generalises the palindromic approach by defining programs as searches over context-free grammars and thus execution is in the form of context-free parsing; parallelisation on the anti-diagonal is subsequently derived from the definition.

It is important to note that for each application, we automatically derive the same orientation of parallelisation as the majority of these hand-coded applications.

The polyhedral model has also been applied to certain dynamic programming bioinformatics algorithms [51], in the context of FPGAs, by deriving very efficient systolic array implementations for both the Nussinov [49] and the Zucker [50] algorithms. This is a *manual* approach, where for each different algorithm a series of systematic polyhedral model transformations are described and applied by hand from the definitions, and the results used to create FPGA implementations which are extremely efficient for those particular cases. In contrast, our approach is fully automatic, taking a broad range of dynamic programming recursions as input and automatically deriving the implementations for the GPU.

5.6.6 Other GPU optimisations

Our cost model work has some similarities to existing work in GPU simulation [7] and analytical cost frameworks [46] for GPUs. What is unique about our approach is that we attempt to predict the *comparative* cost of making important GPU decisions, rather than predict the total performance. In this sense we pragmatically tackle an easier problem to determine practical information on a real world choice.

Baskaran et. al. [8, 9] have developed an interesting approach which (a) optimises coalescence of global memory through loop transformation using the polyhedral model (b) uses tiling to parallelise loops, again using the polyhedral model and (c) estimates the total memory loads and reads in a tile, and uses that to guide an empirical search for optimum parameters. The latter approach is an interesting comparison to our heuristics for block size, grid size and memory locations, because it is applied *experimentally*, in a form of auto-tuning to determine the most appropriate values. We provide a much more restricted input, and thus we can better determine suitable heuristics which eliminate the need to empirically determine the best approach.

Our sliding window approach is similar to approaches that have been taken for dealing with other "scratchpad" like memories, similar to shared memory. It is important to note that the sliding window is also a common optimisation across GPU applications; importantly, we derive our window size automatically from our parallel analysis of the DSL.

A system that uses the polyhedral model in a similar way to ours is [40], although that focuses on code generation from loop nests, and includes loop reordering optimisations as well as array reordering. Optimised use of scratchpad memory was also considered in [8], where portions of arrays are selectively copied to shared memory, based on polyhedral model analysis, in order to improve performance.

A further observation of these techniques is that they assume a two memory system (global, shared); our approach is more nuanced, in that it supports the complete memory hierarchy of the GPU (cached global, shared and constant memory, texture cache, etc.).

Chapter 6

A worked example: Hidden Markov Model

In this chapter we will put together the various techniques and methods of the previous two chapters from the point of view of an implementor of a DSL extension. The aim is to demonstrate both the need for, and the design of, the extension mechanism on a practical example.

Recall from Section 3.2 that a Hidden Markov Model (HMM) is a model for random stochastic processes, constructed as a probabilistic finite automaton. Models consist of a set of states, where transitions between states are associated with a probability. Each transition only depends on the current state - the *Markov* property. The *hidden* part of the name derives from the fact that we cannot observe the states - instead each state has a distribution over some *emission alphabet*.

Their widespread use, simple structure and recursively defined algorithms make them an ideal candidate for a domain-specific extension to our language. To that end, we have constructed an extension for Hidden Markov Models on top of the recursive framework, which is called *HMMingbird*.

6.1 A DSL for describing Hidden Markov Models

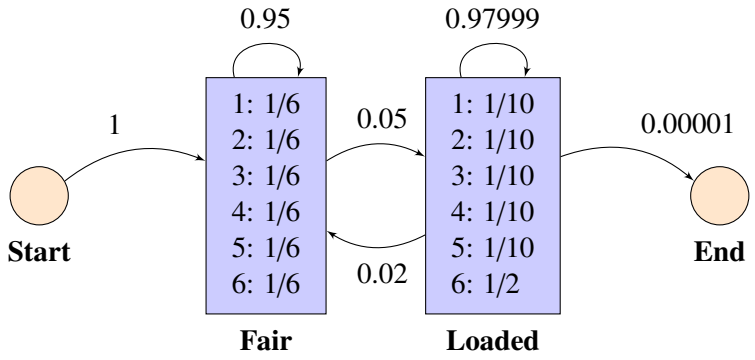


Figure 6.1: The occasionally dishonest casino

```
define alphabet dice [1,2,3,4,5,6]

define hmm casino {
   alphabet dice;
   startstate start;
   state fair emits fairemission;
   state loaded emits loadedemission;
   endstate end;
   start -> fair 1;
   fair -> fair 0.94999;
   fair -> loaded 0.05;
   fair -> end 0.00001;
   loaded -> loaded 0.97999;
   loaded -> fair 0.02;
   loaded -> end 0.00001;
   emission fairemission = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6];
   emission loadedemission = [0.1, 0.1, 0.1, 0.1, 0.1, 0.5];
}
```

Figure 6.2: The occasionally dishonest casino code

If we consider the HMM algorithms described in Section 3.2, it is clear that our language as-is is insufficient to describe the algorithm. First and foremost, we need a way to describe and encode a specific HMM, preferably in a simple and easy to read way. Furthermore, we will need to be able to extend the type system in order to pass instances of this HMM around, and recurse over states, as well as providing new expressions in order to use

the values of the model, such as transition and emission probabilities.

The model extension system is ideal for building HMMingbird. The model framework allows us to build a fluent and simple interface for describing HMMs, unconstrained by the existing language. It provides support for describing the data representing the model, and placing that on the GPU automatically, reducing the code and effort required. Furthermore, the extensions to the expressions syntax are quite straight-forward, and support the automated parallel analysis. To emphasise the benefit: developing the HMMingbird requires minimal code generation, and little knowledge of the GPU.

The first step is to create a new model definition for describing a single Hidden Markov Model, registered to the extension point `hmm`. The syntax has been designed to be not only human-readable and human-writable, but also clear and concise, simplifying program development and understanding for domain experts. This is in contrast to existing techiques for describing HMMs, such as Gerton Lunter's HMMoC [72], which uses an XML based format, and the profile HMM tool HMMeR [30], whose syntax is primarily designed to be machine-readable, and is limited to only HMMs of the profile HMM form. Whilst our HMM syntax is not part of the framework, it is important to note that the access to the full compiler toolchain ensured that we were able to develop this clear and concise syntax.

In Section 3.2 we consider a small example, the *occasionally dishonest casino*. Recall that it represents a dice rolling game where, occasionally, and without the players knowledge, the die will be switched for an unfair one. The diagram is repeated in Figure 6.1. Figure 6.2 contains the language definition for the model. To highlight the most interesting aspects, we can first see that it provides a straight-forward description of the model by permitting the user to declare states, transitions and emissions on the model. We use an arrow (`->`) to represent transitions in a fluent way. States that emit must be declared, as must start and end states. All other state declarations are optional. Emissions can be defined on states or on transitions, and are declared separately so that they may be reused.

Another interesting point is that we reutilise the alphabet defined *outside* the model,

using base language primitives, to link to the alphabet within the model. This firstly saves redeclarations, and can ensure that the type system is properly connected. Other inter-model declarations are possible.

To increase the fluency of the model, we permit probabilities to be entered as decimal values, fractions or floats. We provide tools for checking the model that aid HMM development, including checking for a distribution over transitions – do the probabilities sum to one? – and ensuring no states are unreachable.

6.1.1 Implementation

The HMM definition language provided with HMMingbird is built using a standard compiler tool-chain provided within the system. This is relatively straight-forward DSL development; develop a parser and lexer, and describe the AST. No code generation is required in this step. The extension toolkit is then used to create a representation of the model on the GPU.

Behind the scenes, the custom-built parser will build an AST, perform checking and validation of the model and analysis on the AST, before populating a *semantic model*. This is a powerful approach, because it allows us to expose the semantic model directly for modification by expert users. By supplying a suitable Java class file, users can manipulate models through a simple API before they are transferred to the GPU.

One example of a desirable model manipulation is the removal of *silent* states from the model. Silent states do not contribute characters to the output sequence, and can be eliminated through a lossy procedure. This is advantageous for running on the GPU as silent states complicate the dependencies in the algorithms. Recall from Section 3.2.2 that silent states introduce *inter-column* dependencies. For example, in the Viterbi algorithm, the standard recursive step calls $V(p, i-1)$ which would result in a scheduling function of $S_f(x)$. When silent states are considered, we introduce a recursive step of $V(q, i)$, where p may be any state in the model. Without further refinement, this recursion cannot be

analysed by the system. By eliminating the silent states we can use the simple recursion, and therefore provide an optimised parallel implementation.

The low-level representation of the Hidden Markov Model is generated directly from the semantic model, and guided by the context in which the HMM is used. Most notably, the automatic data transfer framework builds structures on the GPU for those elements of the model which are used in the functions that the HMMs are applied to.

We generate the following arrays:

- `transProb` – the transition probabilities, in a 2D array, ordered first by state, then by transition position.
- `initState` – stores the initial state of each transition, in a 2D array ordered as for transProb.
- `finalState` – stores the final state of each transition, in a 2D array ordered as for transProb.
- `finalTrans` – since each state can have a different number of transitions, this records the final transition position for each state.
- `emission` – stores the emissions for each state.

6.2 Types and their implementation

The type system plays an important role in the automated parallelisation, by determining the structure of the recursion. In order to support recursions over HMMs we provide a hmm type for representing HMMs, which extends from `ModelType`. We also provided state[hmmname] for modelling a state of a particular HMM and transition[hmmname] for modelling the transitions of a HMM, which both extend from `ModelElementType`. The hmm type is a *calling* type, provided when we call the function initially. The state and

transition types are *recursive*, *non-calling* types; they cannot be passed directly, and are used in all recursive calls.

6.2.0.1 Parameter ordering

Both the parallelisation mechanism and the tabulation of the recursion require a natural ordering over new recursive types (see Section 4.5.4) e.g a mapping to natural numbers. The precise ordering we choose may have a significant effect on the possible parallelisations – recall that the schedule analysis is performed with reference to the arguments of each recursive call, and how they relate to the original parameters. Typically, a logical layout will respect relationships between components of a model.

For our first recursive type, `state`, we do not have any clear relationship between elements; states do not in any sense "contain" states, nor do they have a natural ordering. In this case the ordering is somewhat arbitrary, even more so because no recursion depends on the precise ordering. A logical option in this case would be to order states by their *depth*, that is the smallest number of transition steps from the state. In any case, we place the start state at position 0, and the end state in the last position, for easy access and identification.

However, perhaps somewhat unexpectedly, the ordering can have an effect on performance in another way. It determines, indirectly through the schedule, the mapping of cells in the tabulated recursion to cells, and in particular the co-location of threads. Depending on the particular recursion, we can take advantage of this colocation by ensuring that cells with similar access patterns are grouped together. For example, if the recursion uses nested loops over the state transitions, we can order them by the number of transitions, so threads have similar execution times.

The ordering of the other recursive type in HMMingbird, `transition`, is equally arbitrary, for the similar reasons to the `state` type. We choose to order it by the pairs (from-state,tostate), on the basis that this might promote reuse of state data.

In both cases, note that the arbitrary nature of the ordering is a representation that

parallelising over the elements of the model is not viable. An extension that may, in certain circumstances, parallelise over the elements of the model, is the tree extension, where there is a clear relationship between nodes (e.g parent, child).

6.3 Host language extensions

Expressions for accessing elements of the model are divided in to those that act on transitions, and those that act on states. Those that act on transitions are:

$$
\begin{aligned}
Expr \quad ::= \quad & \ldots \\
| \quad & Var.start \\
| \quad & Var.end \\
| \quad & Var.prob
\end{aligned}
$$

We can find the start and end states of a transition t using $t.start$ and $t.end$, and the probability using $t.prob$.

Those that act on states are:

$$
\begin{aligned}
Expr \quad ::= \quad & \ldots \\
| \quad & Var.isStart \\
| \quad & Var.isEnd \\
| \quad & Var.emission[Expr] \\
| \quad & Var.transitionsTo \\
| \quad & Var.transitionsFrom
\end{aligned}
$$

We can determine whether a state is the start or end state using $s.isstart$ or $s.isend$ – these are booleans which can be used in conditional expressions. We can access the emission array of state s using $s.emission$ – `emission[Expr]` is the probability of a state emitting the Expr value.

For many of the HMM algorithms we will need to iterate over transitions to or from a state, so states provide *s.transitionsto* and *s.transitionsfrom*. These implement the IterableExpr interface, and therefore can be used in iterator expressions such as sum and max.

6.3.1 Typing & code generation

For each new expression we introduce we must implement a type analysis, `getType()`. In the new expressions: `start` and `end` return `state`s. `isstart` and `isend` return `boolean`s, `emission[...]` returns a `probability` and `transitionsto` and `transitionsfrom` are iterables over `transition`s.

We must also implement `gen(DeviceCall call)`, the code-generation routine. For transition accessors (start, end, prob), there is a similar pattern, accessing one of the arrays on the device. For example, *t.start* is implemented like so:

```
eq TransitionStartDot.gen(DeviceCall call) {
   Expression transition = getTransition();

   HMM hmm = getHMM(call);
   DeviceModel deviceModel = hmm.getDeviceModel();

   Datastructure ds = deviceModel.getDatastructure("initState");
   return generateAccess(ds, disp.gen(call));
}
```

For state accessors, isstart and isend check for equality against the known values of the start and end states, and emission uses the state value and Expr value to find the value in the emission array. `transitionsto` and `transitionsfrom` are the most complicated of the expressions because they represent iterables. Iterables require building a loop that describes the elements to iterate over. The loop consists of an *initialiser*, a *termination* condition and

a *increment* expression. For example, the pseudocode for `transitionsfrom` loop would be:

```
for(int t = s * max_transitions_for_state; t <= finalTrans[s]; t++) {
   ...
}
```

Recall that our tables storing transition variables (`transProb`, `initState` and `finalState`) are stored as 2D arrays, with one dimension as the state variable. Our loop replicates this design[1], moving the t pointer to the first element in the row, then traversing the row in order.

The code generation routine then populates the loop body with the statements to compute the particular iterator, e.g sum, max, min. For example, the body may sum over the transition probability, which would be accessed using `transProb[t]`.

6.3.1.1 Parallel Analysis

Another implementation requirement for new expressions is that they define an equivalent *affine expression* representing their value, by implementing `toAffineEquation()`. On a case by case basis:

- `.start` and `.end`, return ranges which represent all the possible start and end states, for all possible values. e.g the range of start is $[0 \dots (n-2)]$, because any state may be a start state except the end state.
- `.isstart` and `.isend` are booleans, and return a range over the booleans ($[0 \dots 1]$)
- `.emission` produces a probability which is not a recursive type, and cannot therefore be used or required in an argument.
- The only complicated case is that of the iterables, `.transitionsto` and `.transitionsfrom`. Recall that iterable expressions must produce an *affine range*, that is a range defined

[1]This assumes that the table is in column major order. We may also generate in row major order – our choice depends on the memory location; row major has better performance for coalescing reads from global memory.

by an upper affine expression and a lower affine expression. In the case of these two expressions, we cannot derive an affine expression in terms of the originating state; because each state may have different numbers of transitions. We instead give a conservative value which covers the entire range of transitions.

For non-recursive types, e.g `prob`, we do not need to implement `toAffineEquation()`.

6.4 Encoding the algorithms

$$F(0,0) = 1$$
$$F(s,0) = 0 \, for \; s > 0$$
$$F(s,i) = e_{s,x[i]} \sum_{p:t_{p,s}>0} t_{p,s} F(p, i-1)$$

(a) The recursive equation, where $F(s,i)$ is the likelihood we are in state s at position i in the sequence x. $t_{s,p}$ is the probability of transitioning from state s to state p

```
1  prob forward(hmm h, state[h] s, seq[*] x, index[x] i) =
2     if i == 0 then
3        if s.isstart then 1.0 else 0.0
4     else
5        // The end state is silent
6        (if s.isend then 1.0 else s.emission[x[i-1]])
7        * sum(t in s.transitionsto :
8              t.prob * forward(t.start, i - 1))
```

(b) The implementation in our extension

Figure 6.3: The forward algorithm, mathematical description compared to the implementation

Using HMMingbird, encoding the HMM algorithms becomes straight-forward. In Figure 6.3 we compare the mathematical equation definition of the forward algorithm to our language implementation.

In the program, Line 1 represents the declaration, which illustrates the calling and recursive parameters of the function. In the mathematical equation, it is implied that we have a HMM in which the states occur and a sequence of characters for which we have an index. We explicitly define these parameters in the program as the calling parameters, along with the expected recursive parameters over the states of the model and the indices in the

sequence. To summarise the effect, we recurse over s, the states in the HMM h, and i, the indices into the sequence x.

There is a strong correspondence between the mathematical description and the program – lines 2 and 3 of our implementation relate to the first two definitions of the recursive equation, and lines 6-8 map to the last line of the recursive equation. An implied part of the mathematical recursion is that any silent state should simply sum over the previous elements, and this is made explicit in line 6.

6.5 Summary

This chapter has provided a whistle-stop tour around the creation of a new extension for Hidden Markov Models. By exploring the extension step-by-step, we can see how the different aspects of model definition, code generation and parallel analysis discussed in previous chapters come together. Particularly important is to see, in practice, how the HMM extension integrates with the parallel analysis by defining affine equations for each newly introduced expression. We have demonstrated how we can define new syntax for describing clear and concise model definitions, and we extend the core language with new types and expressions for describing algorithms over these models. The code generation steps are explained, and in particular we have seen that it is straight-forward to access the data generated by the model. Most importantly, the relationship with the parallel analysis has been identified – how each expression implements the equivalent *affine equation*, which permits it to be used as a constraint on a schedule. Having considered the construction of the extension, we finally see the outcome: how the standard algorithms are written in the new language, and how similar it is to the original recursive equation.

Chapter 7

Evaluation

The central thesis of this dissertation has been to demonstrate that DSLs are an effective way of utilising massively-parallel processors. In this chapter we will evaluate the success of this approach.

In Section 4.2, we described two measures for evaluating a GPU DSL – whether it was *performant* and whether it was *expressive*. Our overall approach has been to develop tools and techniques to support the *development* of DSLs; we must therefore evaluate whether these do, in fact, result in DSLs that have both competitive performance and are expressive.

In Section 7.3 we will empirically evaluate the framework; firstly by considering outright performance on a number of applications written using the framework and secondly by breaking down each technique to consider the effect it has on performance. This is important; it provides empirical evidence that each technique is beneficial, and in what circumstances.

The *expressiveness* measure has two different components:

- Is the core language expressive enough to describe bioinformatics problems?
- Is the extension mechanism suitable, and does it result in expressive DSLs?

The evaluation of our core language must consider the scope and type of applications in bioinformatics – does the language provide a natural encoding for such problems? For

the extension mechanism, the question must be whether the restrictions prevent the natural description of the problem.

Finally, we must also consider the ease of programming and development: does the extension mechanism simplify the task of developing a GPU DSL? Recall that the intention is for a small minority of experts in the domain, and requiring only a small amount of Computer Science knowledge or background, to extend the compiler with embedded DSLs for describing their specialised models – the key question is whether the framework is both effective and practical for this target group.

7.1 Evaluation of expressiveness

We have stated that a successful DSL should be *expressive*, but what do we mean by that? In this context, a language is *expressive* with respect to a domain when it displays an *isomorphism* between the primitives of the domain and the primitives of the language; that is there should be a one-to-one correspondence between the two, with only syntactic differences between them.

Why is this isomorphism important? It ensures that translation to the language is as simple as possible; to convert an abstract problem in the domain we simply need to modify the *syntax* not consider the *semantics*. Programs written in an expressive language tend to be short, concise and semantically matching the description provided by the domain expert. The key benefit of a DSL is that it can be tailor made for a particular domain, providing this one-to-one correspondence that would not be possible in a general purpose language.

If we consider the key features of the bioinformatics applications we looked at in Chapter 3, they were:

- *Pure* functions, with no side effects;
- Basic arithmetic and comparison operators;

- Recursion;
- Summation, maximisation and minimisation over ranges;

Name	**Mathematical Recursions**	**Host Language**
Arithmetical operators	$+,-,\ldots$	`+,-, ...`

Case conditionals	$d(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$	`int d(int i, int j)`
		`  | i == 0 = j`
		`  | j == 0 = i`

Sequence access	x_i	`x[i]`
Summation	$\sum_{x:y}(z)$	`sum(x in y : z)`
Minimisation	$\min_{x:y}(z)$	`min(x in y : z)`
Maximisation	$\max_{x:y}(z)$	`max(x in y : z)`
Recursion	$f(x)$	`f(x)`

Figure 7.1: The one-to-one correspondence between the mathematical bioinformatics recursions, and the host language.

Figure 7.1 shows, side-by-side, the correspondence between the host language and the basic mathematical recursions used by many bioinformatics applications. The differences are almost entirely syntactic, and necessary to ensure that programs are easy to type – $\sum$, for example, is replaced by `sum`.

If we consider the model definition parts of the language, then domain developers are free to define both the syntax and semantics as they see fit. With this freedom comes responsibility; the onus is on the domain developers to follow the advice to ensure the language remains expressive. Undoubtedly, this is the trickiest part of new development, but it is hard to further support development without unnecessarily restricting the new DSL. Furthermore, domain experts are normally in the best position to determine the syntax and semantics for the definition of new models.

The new expressions for each model are supported as extensions to the host language. These expressions are free to specify any syntax and semantics provided they do not conflict with existing syntax, and remain side-effect free. This freedom ensures that each extension can provide an appropriate one-to-one correspondence. Figure 7.2 demonstrates the equivalence for different expressions in different extensions.

Name	**Mathematical Recursions**	**Host Language**
Matrix access	$m(x_i, y_j)$	`m[x[i], y[j]]`
Transition probability	$t_{p,q}$	$t.prob$
Left child of a Tree	$L(u)$	`u.left`

Figure 7.2: The one-to-one correspondence between the mathematical bioinformatics recursions, and various extensions.

To consider a more complete example, the forward algorithm of the HMM extension displays a clear one-to-one correspondence (see Section 6.4) between the implementation and the mathematical description, both in the host language and the HMM extensions expressions.

7.2 Ease of programming and development

The ease of programming and development is the key measure of success for our DSL framework: it must *simplify* the process of development to be useful. In Chapter 6 we extensively covered the development of the HMM extension, in order to illustrate the steps that must be taken. However, the system for developing DSL extensions is designed to be used by domain developers – experts in the domain who are capable programmers.

In order to evaluate the success of this proposal, we invited three undergraduate students interested in bioinformatics to spend six weeks developing a new extension to the language. One student was completing a Computer Science degree, one a Engineering degree and the

final student was completing a masters course in Bioinformatics. We will explore this case study in order to illustrate the skills required to develop new extensions, and highlight the areas which are simplified.

7.2.1 DSL development – case study

The students were asked to build a DSL extension for describing trees and recursions over trees. Trees are widely used through bioinformatics, particularly in the construction and analysis of *phylogenetic* or *evolutionary* trees (see Section 3.5).

The first task for the students was to design and develop a front-end for the tree language extension. Domain developers build their system using a parser generator and a lexical analyzer generator. This task is simplified by the existence of, and inclusion into, the pre-defined grammar for the rest of the language.

A strength and a weakness of the system is the requirement that domain developers create extensions using a full compiler toolkit of parser and lexer; it is very flexible but requires the domain developers to have either a prior understanding or the inclination to learn. In mitigation, this is an extension to an existing system, so full expertise is not required – they only need to develop the syntax within the sandboxed environment. Each of the three students had encountered the concepts of context free grammars and regular expressions before, which simplified this process. Furthermore, we expect that only a few expert users will work on DSL development: the complexity of learning about lexers and parsers is certainly no worse than developing a DSL tool from scratch.

To defend this assertion further: if the domain developer were building this DSL from scratch, they would need to define both a lexer and a parser (or other mechanism for defining the syntax of the language). Whilst further research could profitably be spent integrating more user friendly methods of defining the syntax (for example, a true embedded DSL system akin to Scala), in this dissertation I have focused on the aspects that are unique to development on the GPU.

The integration of the extension into the framework is where the benefits primarily appeared. The extension of the type system (a simple process), provided automatic access to parallelisation over elements of the model and variable lookup. Furthermore, the students could make informed decisions on the representation of the model into arrays without worrying about the GPU aspects; data transfer and memory location choice are handled by the framework. The loading of sequences to/from the device is also handled automatically by the framework, as is the division of work between different kernel launches.

The students were successful in creating a DSL for trees within the time frame. Their solution ran effectively on the GPU, and provided a clear language for describing recursive tree problems. Furthermore, they needed very little GPU experience to develop this extension: two of the three had no GPU experience prior to this work, and the third required very little of his prior experience to produce the extension. The result was checked against existing tools to ensure that it returned the correct results.

7.2.2 Quantification of effort

Extension Name	**LoC**
HMM	5135
Tree	2522
Matrix	621

Figure 7.3: Lines of extra code required for each extension.

Formally quantifying the effort involved in writing an application is a difficult process. In order to estimate the effort involved firstly in writing the core framework, and secondly, in writing each extension, we consider the total number of lines of code that were written. The lines of code (LoC) counts given represent everything from lexing and parsing, AST nodes, JastAdd code and any tests, and exclude any generated code.

In the core framework there are 35,033 lines of code, which includes the core language, the runtime code (including interfaces with CUDA), the scheduling and automatic parallelisation routines (including interfaces with CLooG and JOPTCSP) and the code generation routines.

To quantify the effort involved in writing each extension, we have recorded the total number of lines of new code that were written (Figure 7.3). These LoC counts represent a complete count of the extra effort that went into developing that extension.

In order to provide some basis for comparison, we provide similar total lines of code figures for comparable applications (Figure 7.4). Whilst it is perhaps misleading to directly compare these to our LoC counts, as these tools can have a different focus and scope, it does demonstrate that both the implementation of DSL tools (such as HMMoC and HMMeR), and the implementation of GPU applications (such as CUDA-SW++ and GPU-HMMeR) require significantly more LoC than extensions to our applications. Furthermore, the LoC for these GPU tools does not consider the effort involved in understanding the GPU, applying the correct parallelisation and time spent optimising the kernels.

To highlight two particular examples: HMMoC provides a similar HMM language to our own, but is implemented in over twice the number of lines of code. Furthermore, this is a CPU tool that does not (as the next section will show) achieve the performance figures of our extension. CUDA-SW++ is a CUDA implementation of Smith-Waterman. This was implemented in our framework using only the matrix extension; it is therefore perhaps not fair to directly compare the LoC counts, but it is indicative that our framework is well suited to describing typical bioinformatics applications with minimum effort in providing extensions.

It might be considered a fairer comparison to consider the total size of code in both the core framework and extension – under this measure, GPU-HMMeR, for example, does not contain substantially more lines than the combined HMM extension and the core language. However, and importantly, once we amortize the cost of writing the core framework across

the entire set of implemented DSLs, then the benefits become much more apparent: the more DSLs we implement, the more the cost is *distributed* across the DSLs.

A similar argument exists for the extensions themselves: whilst the HMM extension is 5135 LoC, it opens the door to a large number of HMM applications that can be succinctly defined. User applications are typically small – we provide some examples in Appendix A, and in the previous chapter, most of which are between five and ten lines – and therefore new applications have a low cost of development. Compared to CUDASW++, for example, we required more total lines (including frameworks and extensions etc.), but we provide far greater flexibility for defining different algorithms in the same vein.

A final point on this topic. It is important to remember that the framework is an *up-front* cost; once it is developed, adding incremental new extensions requires substantially fewer lines than the hand-coded applications, requires significantly less GPU knowledge and provides greater flexibility. As with any DSL, our work is amortized across the potential size of the use-cases. This underlines the importance of choosing both a suitably flexible framework, and suitably flexible extensions. The total LoC of our system is small relative to the set applications it enables, which suggests that we have adopted a suitable core and set of extensions, and whilst the up-front cost is perhaps higher, the incremental development of DSLs requires significantly fewer lines of code. Considering one oof our goals was to simplify DSL development for GPUs, this is an important validation of our approach.

7.3 Performance analysis

In this section we will consider the performance of the described framework. For the GPU, the aspect of performance that we are primarily concerned with is the execution time. A secondary concern is that of memory usage, to the extent that it affects the execution time and the viability of running large applications.

Application Name	Processor used	LoC
HMMoC 1.3	CPU	11,135
HMMeR 3.0	CPU	148,230
GPUHMMeR 0.9.2	GPU	49,794
CUDASW++ 2.0.10	GPU	13,428

Figure 7.4: Lines of code for comparable applications.

Two important questions must be answered in this section. Firstly, is the overall performance of the DSL framework competitive with existing solutions? In particular, how does the performance for different applications compare with hand-coded and generated solutions for both GPUs and CPUs?

The second question is how the presented optimisation techniques affect the execution time and/or memory usage. This demonstrates the extent to which each of these techniques contributes to the performance of the application.

7.3.1 Overall performance

These case studies are real applications in bioinformatics, using real genome data on existing models to provide a realistic comparison. In all cases our framework automatically derived the parallelisation schemes – no schedules were specified by the user. In each case that we compare against a hand-coded GPU application our tool has successfully derived the same parallel strategy as the hand-coded application.

All quoted results are inclusive of setup time, such as memory allocation and copying to/from device. Our framework provides a runtime environment; consequently the times for our software are inclusive of scanning and parsing the input files. We have not, however, included the code generation time – we cache the compiled code for each function. The code generation overhead is typically around 1 second, primarily due to inefficiencies in

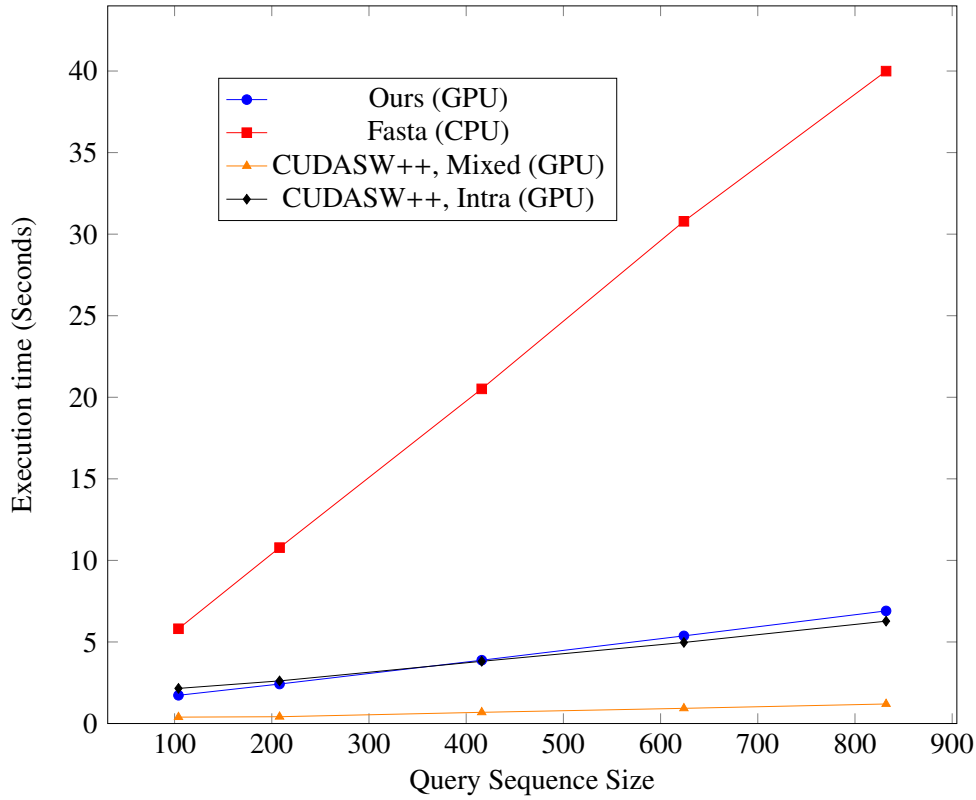


Figure 7.5: Smith-Waterman performance on varying query sequence sizes for a database of 75,000 sequences.

the way in which we call CLooG from Java. We expect to be able to reduce this overhead drastically in future releases. We note that programs written in DSLs of this type are rarely large enough to increase this compilation overhead.

Results were obtained on a NVIDIA GTX 480 with 1.5GB of RAM, and CPU results using Intel Xeon E5520 CPU with 4GB of RAM and four cores[1]. Each test was performed three times, and the results averaged. Results across runs were observed to have little deviation, and so we have not provided error bars.

7.3.1.1 Smith-Waterman

The Smith-Waterman algorithm is an implementation of the local edit distance problem, used for sequence alignment. The typical application will compare one *query* sequence

[1]Where a multi-threaded CPU implementation was used, this has been mentioned in the text.

against a database of other sequences, to identify high-scoring alignments. We implement the Smith-Waterman algorithm using the substitution matrix extension to determine the cost of substituting characters. The expected parallelisation is along the diagonal $x + y$, as with other edit distance algorithms.

For comparison, we use CUDASW++ 2.0 [71] a GPU implementation of Smith-Waterman using the NVIDIA CUDA framework and the multi-threaded CPU "ssearch" tool in Fasta 3.5 [94], for Smith-Waterman alignment. CUDASW++ 2.0 provides two methods of parallelisation – intra-task parallelisation, which uses parallel diagonals across the table in the same way as our recursion, and an inter-task parallelisation, with a database sequence allocated per thread. Best performance is achieved by a hybrid approach, where smaller sequences are computed with inter-task and larger with intra-task. Inter-task is equivalent to our task-per-block and intra-task is equivalent to our task-per-thread. We provide both the hybrid and the intra-task results for CUDASW++ for comparison.

Our results (Figure 7.5, kernel details provided in 7.6) are provided for the task-per-block strategy, and are very similar to the intra-task CUDASW++, and comfortably beat Fasta, demonstrating that we are reaching similar performance to the hand-coded, hand-tuned CUDASW++ for this type of parallelisation. As expected, the best overall performance is achieved by using the hybrid parallelisation approach. We do not provide our performance results for a hybrid or task-per-thread approach, as they are similar to the task-per-block approach; the lack of performance compared to the inter-task CUDASW++ is primarily due to our focus on task-per-block (described in more detail in Section 5.5.1). I believe that the adoption of a task-per-thread specific schedule and optimised code-generation would improve performance in line with CUDASW++, and would be theoretically straight-forward to implement.

Property	Value
Input sequence array size	58,571,382 bytes
Input sequence array count	75,000 sequences
Registers used (per-thread)	22
Shared memory used (per-block)	5592 bytes
Block size	128 threads

Figure 7.6: Smith-Waterman: Kernel details

Property	Value
Registers used (per-thread)	22
Shared memory used (per-block)	216 bytes
Block size	192 threads

Figure 7.7: Gene-finder: Kernel details, viterbi algorithm

7.3.1.2 Gene-finding Hidden Markov Models

A common application for Hidden Markov models is *gene-finding*. Recall from Section 3.2.4.4 that a gene-finder is trained to recognise statistical features of a gene or set of genes, then used for likelihood estimation and gene prediction.

Gene-finders come in all shapes and sizes. In order to provide some comparison, we implement an example gene-finder provided by HMMoC 1.3 [72], a code-generation tool for evaluating HMM algorithms on CPUs. Our results (Figure 7.8, kernel details in Figure 7.7) show a significant performance increase in line with the expected results over a CPU version of this application – at larger database sizes, when we are using the GPU to its full extent, the performance increase is about x60. HMMoC is a single-threaded application; considering the relative strengths of the CPU and GPU this speed-up is in line with what we might expect. Since gene-finders tend to be highly customised, we are unaware of any tools which are suitable for comparison for GPU performance.

7.3.1.3 Profile Hidden Markov Models

Profile HMMs are a type of HMM designed for representing a family of sequences. Typical applications include training models and database searching, where we apply a database of

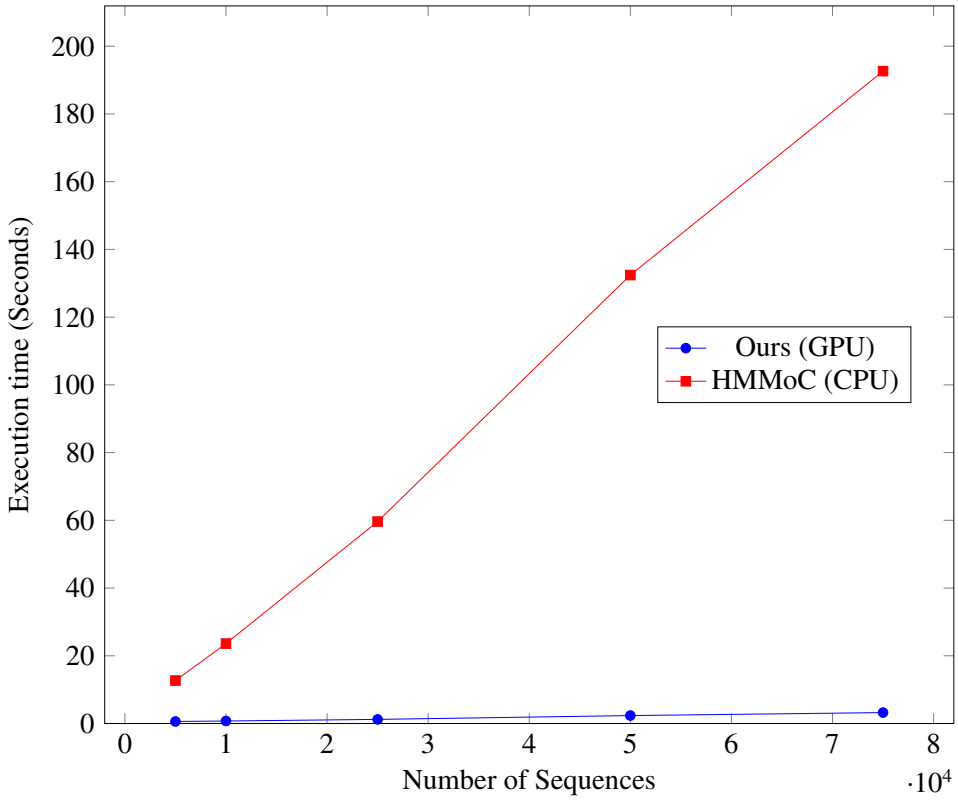


Figure 7.8: Gene-finding performance on varying sequence sizes.

sequences to the profile model in order to determine new sequences that belong to a given family. We perform a database search using the forward algorithm on various sequence and profile models. Once again, we provide comparison to HMMoC, as a general purpose HMM tool, as well as to HMMeR [30], a special purpose tool for profile HMMs. In addition, we compare to a GPU port of HMMeR 2.0, GPU-HMMeR [121].

HMMeR is a high-performance, widely used tool – hand-coded and hand-tuned over 15 years to provide optimal performance on profile models. The latest version, HMMeR 3.0, uses a series of filters to remove low-scoring sequences at low cost by evaluating simplified models and algorithms, such as the MSV (Multiple Segment Viterbi), before bringing the full forward algorithm to bear on a much smaller set of problems. Whilst it would be both viable and beneficial to translate such techniques into our language, it would be a significant undertaking. To that end, we set the "–max" flag to turn off all such filtering and provide a fair comparison between a best of breed full forward algorithm for profile HMMs on the

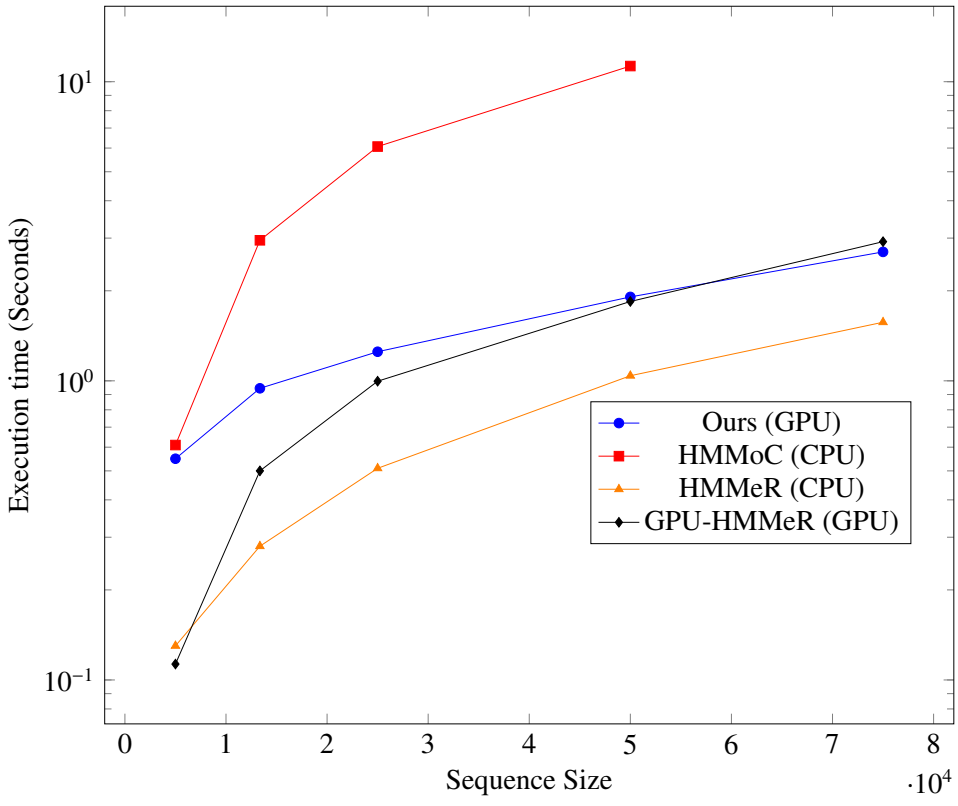


Figure 7.9: Log scale of evaluation time on the "TK" profile model of 10 positions, varying the numbers of sequences.

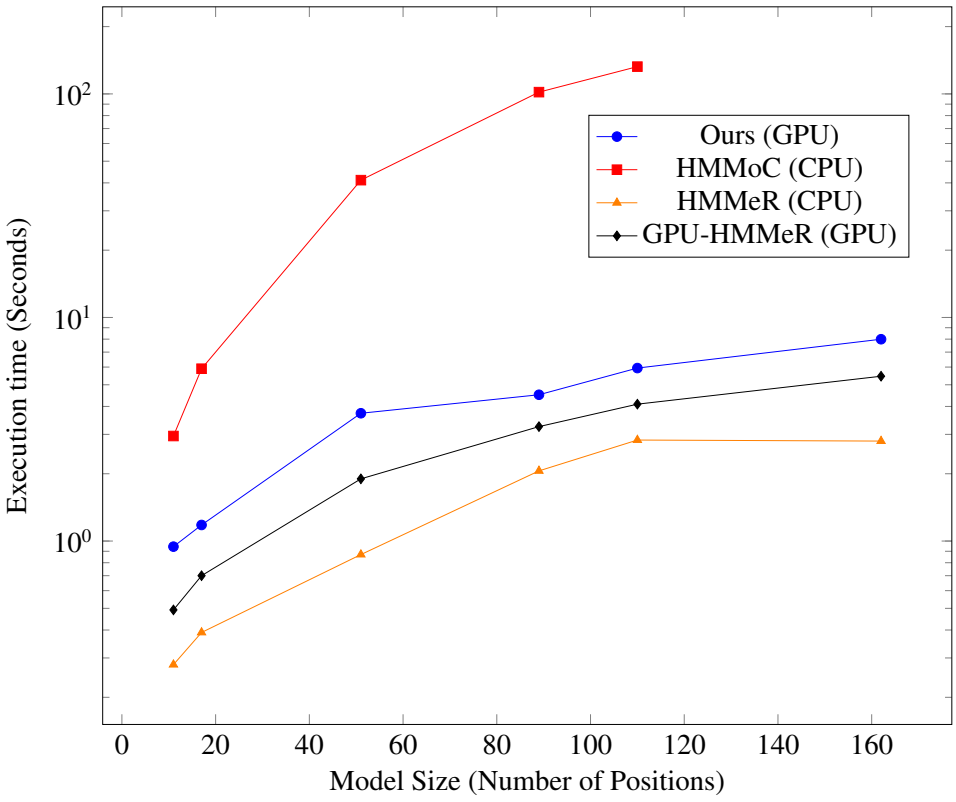


Figure 7.10: Log scale of evaluation time on a dataset of 13,355 sequences, on models of a varying size.

Property	Value
Input sequence array size	9,632,716 bytes
Input sequence array count	13,355 sequences
Registers used (per-thread)	20
Shared memory used (per-block)	3624 bytes
Block size	160 threads

Figure 7.11: Profile HMM: Kernel details, forward algorithm

CPU to our GPU implementation.

Our results (Figures 7.9 and 7.10) show an expected large increase in performance over single-threaded CPU HMMoC for the GPU applications. Our runtime performance is similar to GPU-HMMeR – differences here are primarily due to the overhead of our runtime framework. In particular, in Figure 7.9, we see that our initial startup time is high, but this runtime penalty is smoothed out on larger sequence sets. This is because our runtimes are inclusive of compilation, analysis and code-generation, which is not an overhead GPU-HMMeR has.

What is more surprising is that both ourselves, and GPU-HMMeR are beaten by the most recently released CPU version of HMMeR, 3.0. In order to understand why this is, it must be remembered that HMMeR can make a number of assumptions about the nature of the problem that we are unable to do in a more general HMM tool – in particular, they can make assumptions about the precise structure of the HMM, the number of transitions and so forth. HMMeR is also well optimised, using multiple threads and vectorised code on the CPU to maximise performance. However, this was a significant undertaking for the developers – what we provide is access to this type of performance across a much wider range of problems without the need to put 15 years of optimisation into each one.

One final observation is that our Profile HMM application works by eliminating the silent states from the profile model, which introduces a large number of new transitions (the order of which is quadratic in the number of states). We do this to permit the automatic parallelization scheme to chose as its schedule $S_f(x) = x$, e.g. a column by column parallelization; without it, interdependencies within a column would prevent any form of

parallelization. However the introduction of all these new transitions adversely affects performance. In principal, we could adapt the automatic parallelization to evaluate each column in two stages – firstly, the silent states, then the emitting states. This works because we make the assumption that there are no cycles amongst the silent states.

7.3.2 Comparison to absolute performance limits

In addition to the question of whether we achieve competitive performance with hand-coded solutions, we also consider the question of how close we are to the absolute limits of the device itself. This provides an indication of how much more performance we can expect to achieve within our framework by further optimisation. We will consider two applications in detail in this section – Smith-Waterman, using a query sequence of 104 characters, and a Profile Hidden Markov Model application using the "globins4" profile model of 164 positions.

GPUs have a number of fundamental limits with respect to performance, and depending on the type of application, depends on which performance limit is hit first. Broadly speaking, there are two important types of limit: *memory limits*, in particular between global memory and the SM, and compute limits, in particular the number of floating point operations which can be executed per second.

7.3.2.1 Compute or memory bound?

In order to evaluate which type of limit is relevant for our particular application, we need to understand the balance of the application; which functional units does it use the most? An important metric is the *arithmetic intensity* of the kernel, that is the ratio of arithmetic operations to bytes of data transferred from memory. GPUs are typically constructed with much higher compute performance than memory performance, so the ratio of arithmetic operations to memory operations must be quite high.

Consider the published performance figures for the GTX 480 used in this chapter. In

theory the compute performance of this chip is 1345 GFLOP/s, and the bandwidth for global memory is 177.4GB/s. We therefore need around 30 floating point operations per 32 bit value loaded in order for our application to simultaneously max out both compute and memory performance. Any fewer arithmetic operations, and the kernel is likely to be *memory-bound*, any more arithmetic operations and it is likely to be *compute-bound*.

In Smith-Waterman, we are comparing against a fixed query sequence and for each cell in the dynamic programming table we are performing around 18 arithmetic operations. The number of cells evaluated per loaded character depends on the size of the query sequence – in this case the query sequence is 104 characters, so for each character loaded we have around 1872 arithmetic operations. Similarily, for the Profile HMM application we have an average $(n + 5)$ floating point operations per cell[2], where n is the number of states in the model, and therefore $n * (n + 5)$ cells for each character loaded. Emission and transmission probabilities are required for each transition; however, the model should be relatively small compared to the number of sequences, and should be placed into cached memory, very likely the texture cache, and so should not need to be considered for computing arithmetic intensity.

In both cases the high arithmetic intensity suggests the applications are compute, rather than bandwidth limited. We therefore provide figures for estimated performance in terms of GFLOP/s for both Smith-Waterman, over a variety of query sequence lengths in Figure 7.12, and Profile HMMs over a variety of model sizes in Figure 7.13[3]. On NVIDIA GPUs the same functional unit handles integer operations as well as single precision float-

[2]Profile Hidden Markov Models are of a fixed linear form [29], arranged in columns, with each column containing a match, insert and delete state. The number of arithmetic operations per cell depends on the number of transitions into each of these cells, as we sum over the transitions into the state. In the profile model, each match and insert state has two transitions into the state, but the delete state is *silent*. Recall that we eliminate silent states by replacing them with the equivalent weighted transitions. For each delete state, this equates to the introduction of $n + 1$ new transitions, where n is the number of states, for an average of $(n + 5)/2$ transitions per state. For each transition, we evaluate two floating-point operations (multiplying the transition start value, transition probability and transition emission probability), for $n + 5$ floating-point operations per cell.

[3]For reference, our bandwidth figures range between 1GB/s to 10GB/s, which is in line with expectations for a heavily compute bound kernel.

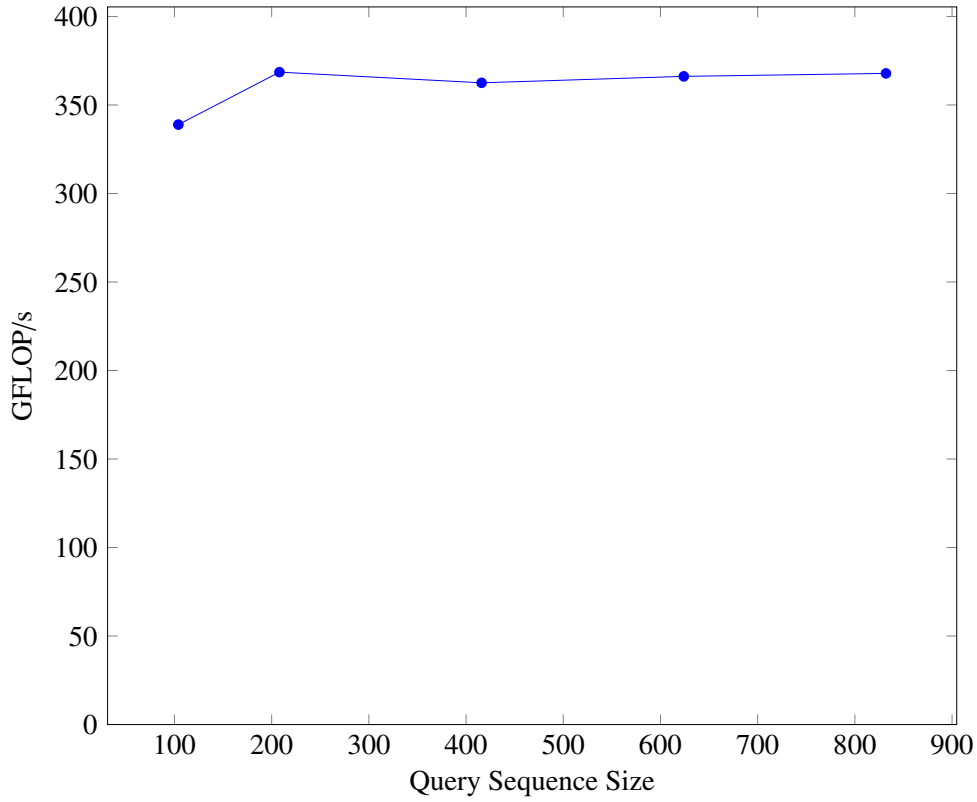


Figure 7.12: Smith-Waterman compute performance on varying query sequence sizes for a database of 75,000 sequences.

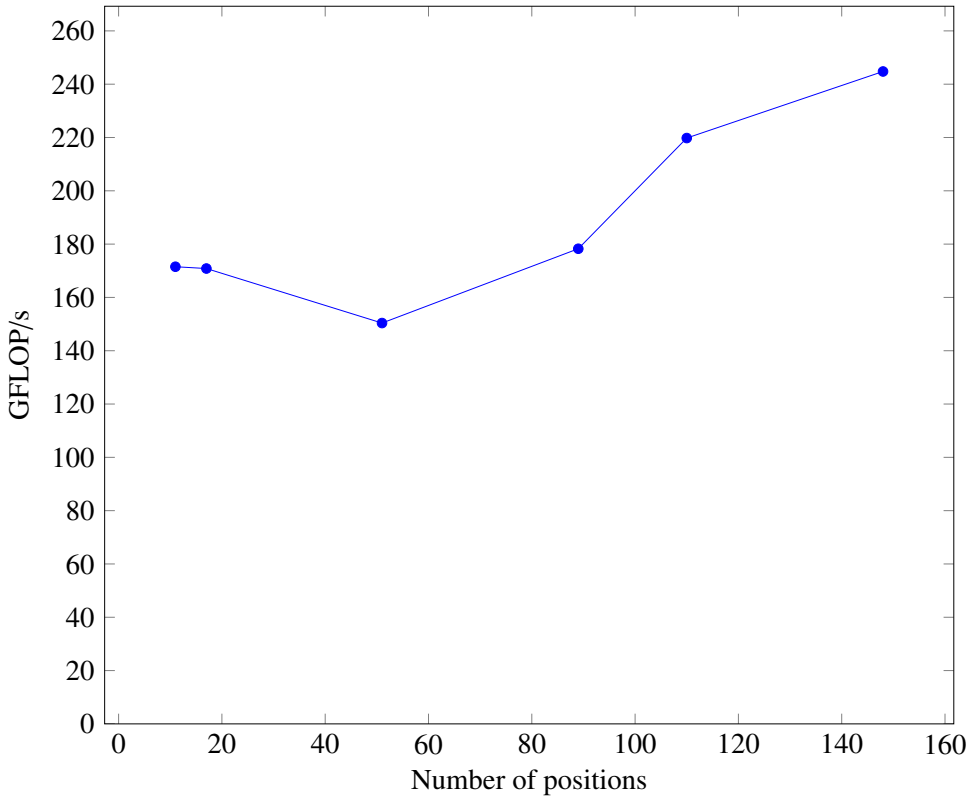


Figure 7.13: Profile HMM compute performance on varying model sizes for a database of 13,355 sequences.

ing point operations; to understand how close we are to achieving full utilization of that functional unit, our GFLOP/s figures include all integer and single-precision floating-point operations. This is unlike a CPU where there is a separate floating-point and integer pipeline, and it is normally the floating-point pipeline that is the limit.

For Smith-Waterman, the figures have reached a plateau, suggesting larger query sequences will not increase GFLOP/s. In the Profile HMM example, we are steadily increasing, but we reach the end of our test set before a plateau has occurred. This test set contains profiles of a typical size, so increasing the profile size further would be unrepresentative.

These figures suggest we are making good, if unspectacular, use of the arithmetic compute units. There are a few reasons why we might not achieve the full published peak performance figures. Unlike achieving peak bandwidth, which is possible with optimised code, achieving the theoretical peak compute rate is almost impossible. Firstly, it requires using a FMAD (fused multiply add) for each instruction, which counts as two floating point operations for each clock cycle. For code such as Smith-Waterman which uses integer operations exclusively, we cannot use the floating-point FMAD operation, so we are limited to only 50% of the published figure. For other code, we may only be able to use it infrequently. In real code we also require instructions for memory loads/writes, control flow and so forth which will account for a proportion of the instructions dispatched. Even considering these factors, we might hope to achieve slightly higher performance figures. To understand why we do not, we must look at another compute metric.

7.3.2.2 Utilization

Another way to consider the compute performance is to look at compute *utilization*, that is, how many instructions did we execute compared to the opportunities we had to execute an instruction? If the utilization is low, it suggests that threads were often suspended or *stalled* waiting for some other event when we tried to schedule them. This is often a manifestation of latency problems. We provide utilization information in Figure 7.14 – in both cases, we

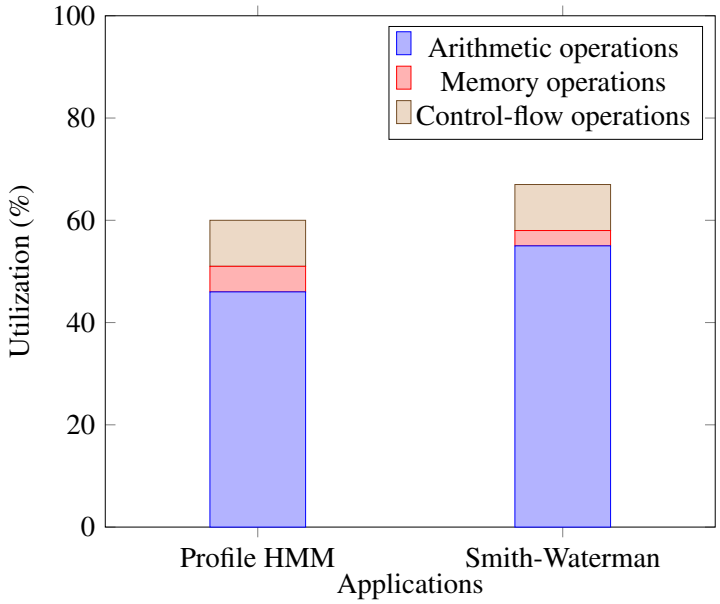


Figure 7.14: Compute utilization breakdown.

Stall Reason	**Value**
Synchronization	43.3%
Execution Dependency	30.9%
Fetching Instructions	21.3%
Data Request	1.1%
Texture Request	1.5%
Other	1.9%

Figure 7.15: Stall reasons on the profile HMM application.

are not fully utilizing the GPU, suggesting that instruction stalls may be playing a part.

Using the CUDA 5.5 profiling tool, we can identify the cause of such instruction stalls. In Figure 7.15 we provide the analysis for the Profile HMM example. The most important factor here was synchronization. In the HMM applications, we synchronize after computing a single column; it appears this is a significant limiting factor in achieving higher compute utilization. In theory, this would be eliminated by adopting the optimised task-per-thread approach, which requires no synchronization, however this might be a trade-off with increased use of other resource (registers, shared memory etc.).

Note as well that we are not, in general, stalling due to unfulfilled data or texture requests, and that the total number of memory requests is relatively small compared to the

number of arithmetic requests, which suggests, once again, that memory latency or bandwidth is not a significant problem for our applications.

We should note at this stage that although a large percentage of instructions are arithmetic instructions, a proportion of these are bookkeeping operations. Our parallelisation method currently induces a high overhead; our loops and memory operations require a large number of integer offset calculations, and our compiler is not very sophisticated in terms of classical compiler optimisations such as common sub-expression elimination, loop invariant code motion and so forth. The adoption of a more sophisticated back-end, such as the recently release NVIDIA LLVM PTX backend, should significantly reduce this overhead, which may in turn reduce runtime. Stalls due to execution dependencies may also be reduced by appropriate compiler optimisations for reordering instructions.

7.3.3 Auto-allocation algorithm

The auto-allocation algorithm is one of the primary optimisation techniques we use within the framework. Our algorithm assigns a cost to each possible allocation; to measure the accuracy of that algorithm, we will compare predicted costs against actual costs.

7.3.3.1 Overall testing methodology

For each test we compare the predicted cost against the cost of the kernel call, not the overall runtime. This is because our cost measure is for the difference in GPU runtime. The difference in initialisation or kernel launch times between allocations is marginal. We measured the time using CUDA events, through the `cuEventElapsedTime()` driver call.

Each test was repeated multiple times; for all tests we only include the average because the variance between runs was very low – under 1ms. We attribute this to the lack of competition for resources on the GPU – our benchmarks are, for all intents and purposes, running exclusively.

Recall that our cost method is only comparable between different allocations of the

same problem on the *same* device; our cost measure is not a prediction of the total runtime, but the comparable differences between run times. Therefore, different data sets on the same graph should only be compared for correlation between different allocations on the same device/benchmark. Correlation figures, where given, are Pearson's r coefficients.

Once again, all tests were completed on a GTX 480, except benchmark 1 which was completed across a range of cards. We only report the GPU used, since the CPU and motherboard have no effect on GPU runtime.

7.3.3.2 Benchmarks

We first considered the performance of our algorithm on a number of benchmarks. We have crafted these to illustrate how the method reacts to some important GPU factors.

Benchmarking methodology To create a fair set of benchmarks, we have developed a dummy extension which provides a few simple operations suitable for representing common access patterns. The model definition language it provides is known as *demo*, and includes a set of *demoentry* elements – equivalent to nodes in the tree extension. In all cases we generate 32 bit loads per thread in order to fairly compare across devices of different generations.

We use two different benchmarks – (a) uses a single loop with a single access and (b) includes two loops, which access two different arrays.

```
(a) int d(demo d, dentry e) = max(a in e.iter : a.access)
(b) int d(demo d, dentry e) = max(a in e.iter : a.access)
                            + max(b in e.iter : b.accesstwo)
```

To this end we have two different arrays on the device – `access` and `accesstwo` – which we can then vary independently to demonstrate the cost model for multiple arrays. The generated code for `a.access` and `b.accesstwo` loads a single 32 bit integer from a

6400 byte array. Where more iterations are required than elements in the array, we simply re-access earlier elements. We leave the access pattern unspecified, so that different benchmarks can implement either coalesced or uncoalesced accesses.

Our performance benchmarks are as follows:

Benchmark 1: Simple model This simplified benchmark demonstrates whether our cost model is accurate when no latency hiding occurs and we have a coalesced read. We implement the basic benchmark function such that `a.access` is a coalesced read. We launch our benchmark with a single block of threads of the warp allocation size – this should ensure that no TLP will obscure the latency on the GPU. To ensure the runtime is significant enough to demonstrate the costs, we implement `e.iter` such that there are 10,000 iterations. Due to the way the `max` operator is implemented, each iteration must complete before the next iteration begins, ensuring that there is minimal latency hiding within each thread.

Our performance results (Figure 7.16) show a direct correlation (0.9995) between our cost measure and the actual cost, as you would expect given the conditions applied to the GPU. This demonstrates the accuracy of our cost measure under a simplified work load that matches our cost model. We give results for (b) for three different GPUs to demonstrate the method is not over-fitted to a single card. We do not currently have access to the latest NVIDIA Kepler cards, but we expect our technique to carry over with a few minor modifications.

Benchmark 2: Uncoalesced loads Most GPU memory locations are optimised for consecutive threads accessing consecutive memory locations. For example, for global memory, scattered loads may require fetching more bytes in total, across more memory dispatches, whereas consecutive data loads result in coalescing, reducing the overall number and size of dispatches. Shared memory operates by bank, where consecutive elements in an array are allocated to consecutive banks to provide higher bandwidth. Scattered accesses may result in one bank servicing many threads, and thus requiring many clock cycles to finish

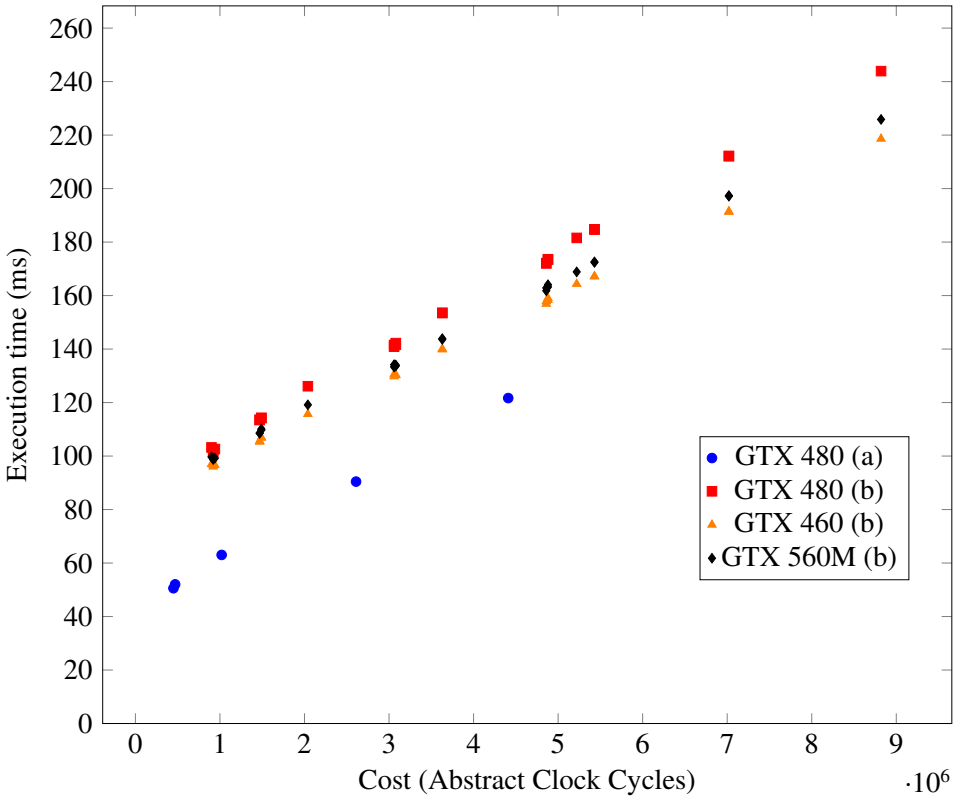


Figure 7.16: Benchmark 1: Simple Read

the job. We therefore adapted Benchmark 1 to scatter `a.access` (Figure 7.17). Our results remain consistent with Benchmark 1. This may be initially surprising, but consider that accesses are scattered for all allocations, and that the cost measure is comparative only within the same benchmark. In this case, the cost is proportionally higher for all allocations. This justifies the choice our algorithm makes to ignore whether a memory operation is coalesced or not.

Benchmark 3: Latency hiding In the previous benchmarks we have tried to eliminate the effect of *Thread-Level Parallelisation* in order to demonstrate the strong correlation our method has with the underlying cost of the hardware. In this benchmark we increase the TLP by using multiple thread blocks, and therefore multiple multiprocessors (Figure 7.18). This case demonstrates that latency hiding is not a significant factor between different allocations – our correlation is (a) 0.99975 and (b) 0.97590. Our single outlier in (b), an

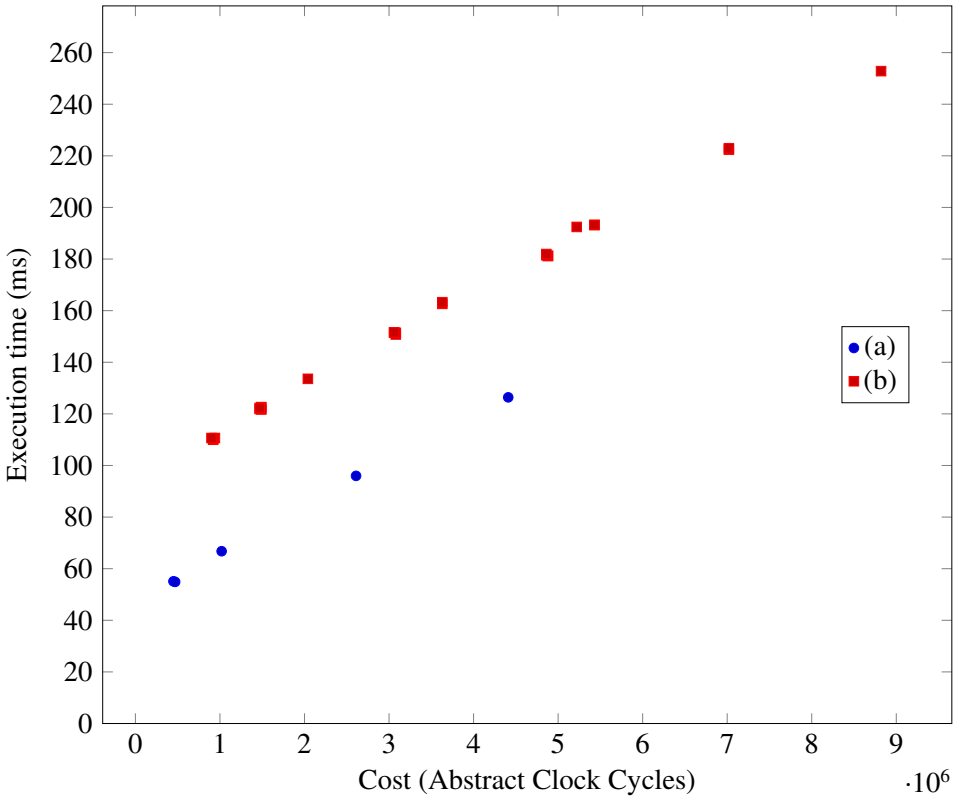


Figure 7.17: Benchmark 2: Uncoalesced access

allocation of both arrays to shared memory, possibly because of low occupancy.

7.3.3.3 Real world tests

Artificial benchmarks can provide interesting insights into the way the algorithm performs under certain circumstances, however it is important that such performance is reflected in real world applications. We therefore provide results on a number of real extensions to our framework.

Hidden Markov Model: Profile HMM Our first real world test is a profile HMM. The HMM extension includes four different arrays on the GPU, for a total of 625 different allocations on devices with global caching. In the case of our HMM, the number of valid allocations is significantly smaller (81), as certain arrays grow too large for certain cached locations. The correlation in this real example (Figure 7.19) is very strong (0.97748),

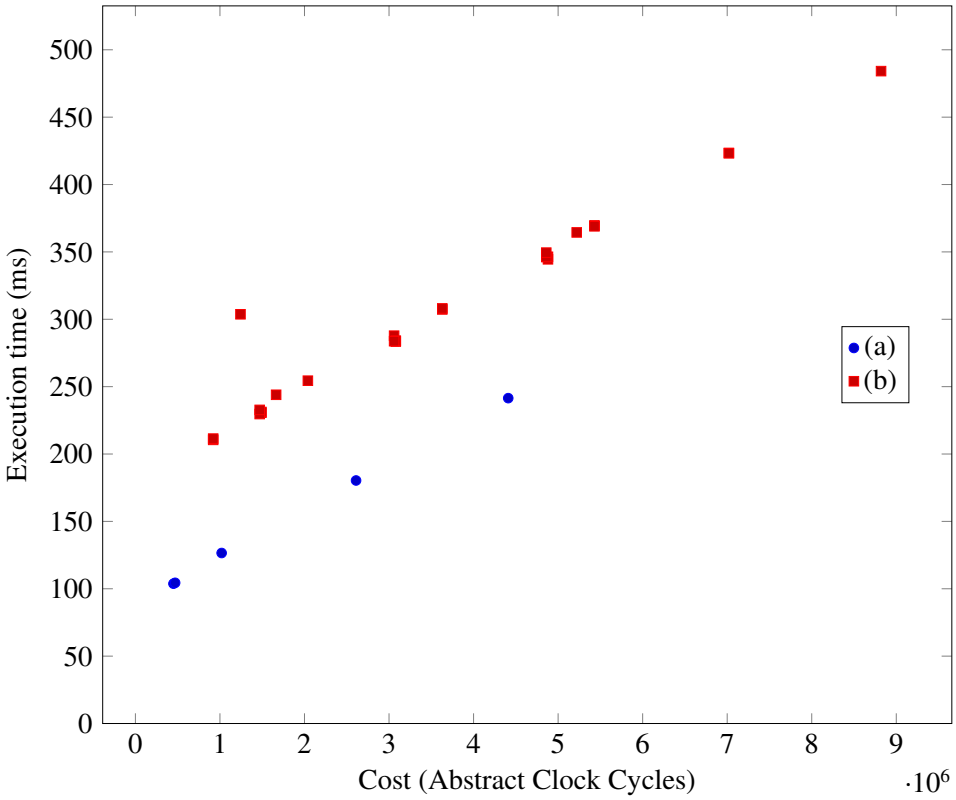


Figure 7.18: Benchmark 3: Latency Hiding

demonstrating the accuracy of our method. Our lowest cost choice in this case has a runtime of 119ms, and the runtime interval was 102ms to 1040ms, showing in practice that auto-allocation is valuable.

Tree: Weighted Parsimony The students implemented the weighted parsimony algorithm using their extension. This uses two different arrays on the device. We test the different allocations on a 17 node tree over 20,000 sequences. The runtime for this algorithm is extremely short, and the variance between run times is extremely low. In this case our cost method still selects one of the lowest run times (Figure 7.20), but the correlation is less (0.9367) than in our previous examples.

Matrix: Smith-Waterman We also implement another very straight-forward extension to represent a matrix. This has a single representation on the device; we consider the use of

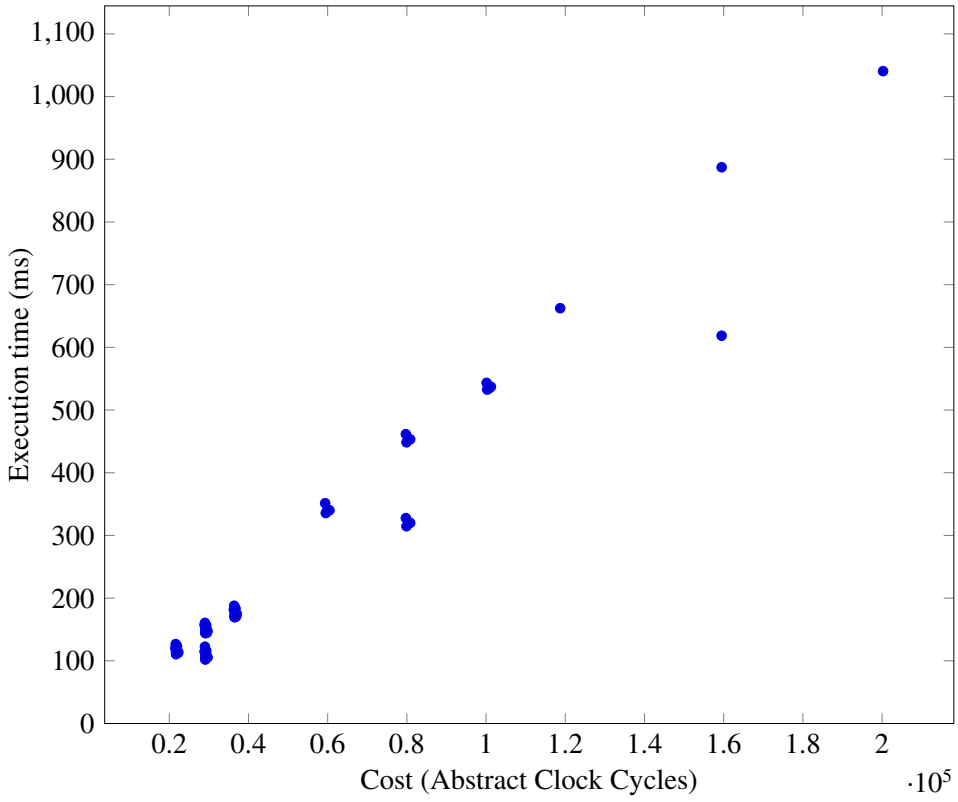


Figure 7.19: Profile HMM

this model in a b. The correlation in this case (Figure 7.21) is low; this is because the kernel only makes one access to the matrix per cell – the cost is instead much more determined by accesses to the dynamic programming table and arithmetic computations. The worst allocation is within 10% of the best allocation. Again, like the previous example, we still choose the lowest cost allocation.

7.3.3.4 Analysis

Our performance results show a very strong correlation in artificial tests with a single block of threads, and a slightly weaker, but still strong, correlation in real work loads. The primary reason for this lower correlation is latency hiding in real cases. Latency hiding occurs when the cost of loading from a memory location is not observed in practice, because the GPU has enough work to fill the latency gap by switching to other threads. In tests with a single block, no latency hiding can occur. In ideal applications all latency should be hidden

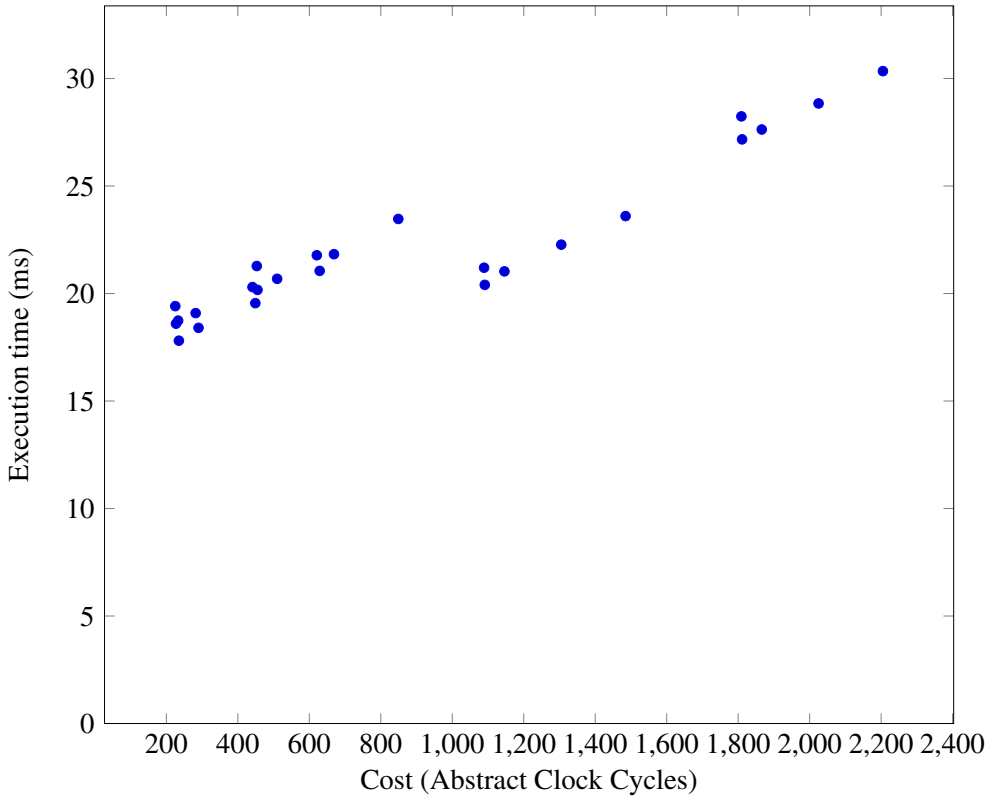


Figure 7.20: Weighted Parsimony (Tree)

in order to reach peak performance. In reality, latency can significantly effect the performance of unoptimised applications, and optimising memory loads to reduce latency is a significant cost factor.

These results should not be surprising – our cost function does not consider whether the latency of this memory access will be hidden. Despite this simplification, the correlation still remains high, and our chosen allocation remains close to the optimal solution.

In all cases, we use the relatively simple default cost and summation functions described in the text. We strongly believe that further refinements of these functions by GPU experts will bring out better predictive performance – by more accurately reflecting the cost of memory accesses in different scenarios.

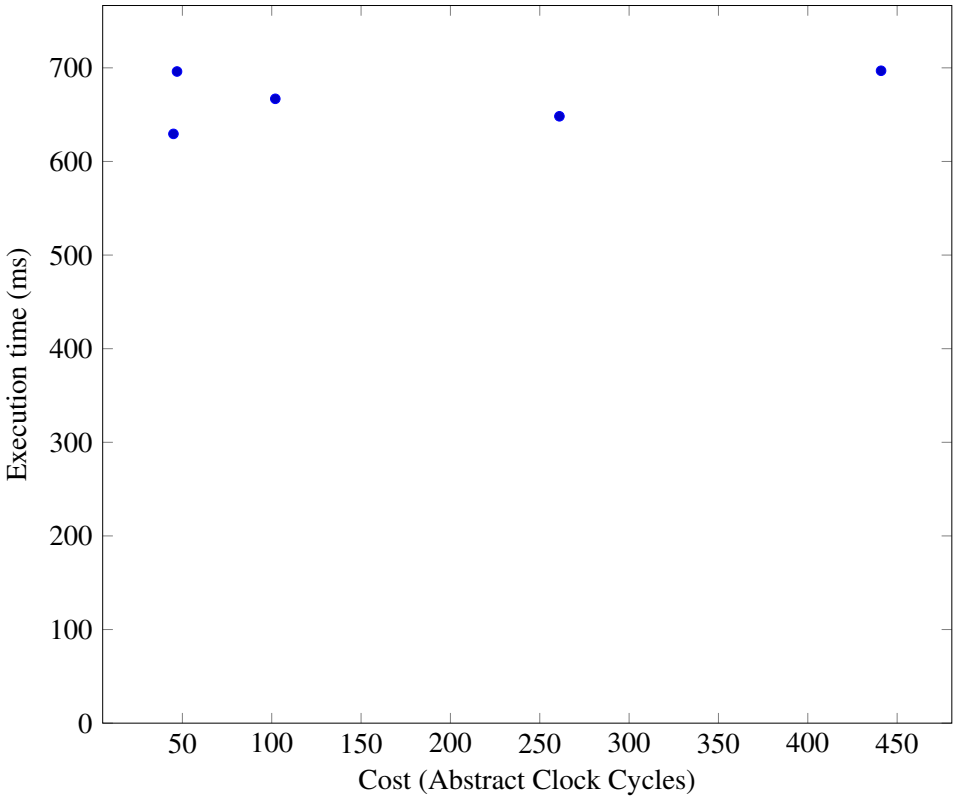


Figure 7.21: Smith-Waterman (Matrix)

7.3.4 Sliding window performance

Recall that the sliding window optimises the loading and storing of the dynamic programming table, by using the dependencies to reduce the amount of data stored. Depending on the application, this technique can have a significant effect on the overall performance.

The sliding window has two different modes: one, where the window is placed in shared memory, and the second where it is placed in global memory. We compare kernel execution performance on a number of applications using the different modes (Figure 7.22). For Smith-Waterman, we run the algorithm using a short query sequence (around 100 characters), and a longer sequence (around 1000 characters), to represent a larger problem. In this latter case, we cannot currently generate a application without using the sliding window, because we do not have enough memory to allocate full tables for all the tasks.

On all applications we measure, the sliding window has a noticeable effect on performance. As we might expect, on small problems a *shared* sliding window has slightly improved performance over a global sliding window, but on larger problems, global memory sliding window wins out (see Section 5.5.4.2 for more information).

Note, we do not use the tree extension in this comparison, because the typical algorithms (e.g weighted parsimony) are not amenable to the sliding window optimisation.

7.3.5 Block size heuristics

It is clear that varying the block size can impact the performance of the executed kernel; what is less clear is how. There is no obvious pattern amongst all types of application, so we will investigate a number of different cases. These help justify the decisions that were made in Section 5.5.2.

Task-per-block, short average partitions (<= 250) Figure 7.23 plots block size against kernel performance (only) for a number of sample applications. On all applications we see a similar pattern; an improvement as we increase the block size initially, followed by

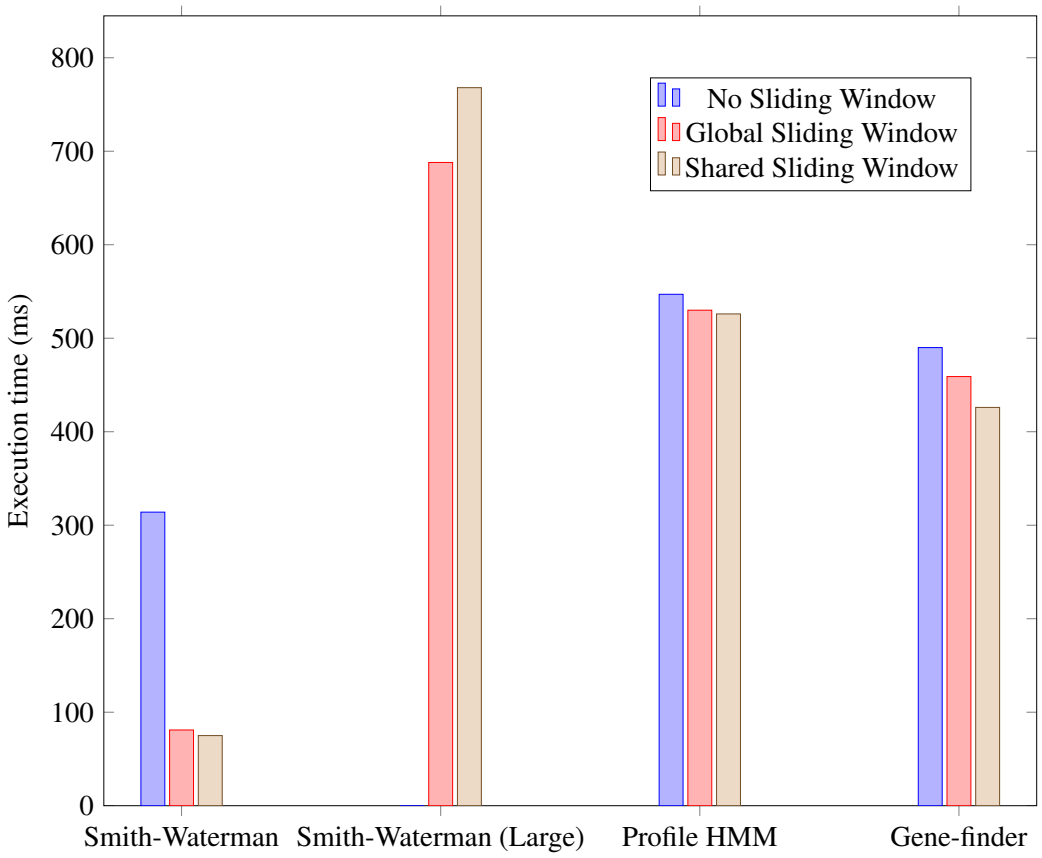


Figure 7.22: Sliding Window performance

a gradual levelling out, then degradation in chunks. Section 5.5.2 proposed two correlated factors: occupancy (number of threads per multiprocessor) and number of blocks per multiprocessor. The steps in the graph correlate to the points where the number of blocks reduce. In this short average partition scenario, it is most important to get the maximum number of blocks, then maximise the occupancy. We conclude that this is because of two factors:

1. Beyond 100-200 threads, larger block sizes are wasted because very few partitions are of that size.

2. When the block count per MP is reduced, it removes the possibility for latency hiding. Synchronisation in this kernel means that more blocks are better for hiding latency than higher occupancy in fewer blocks – the trade-off for more threads does not compensate, because warps in the same block are more likely to be stalled waiting

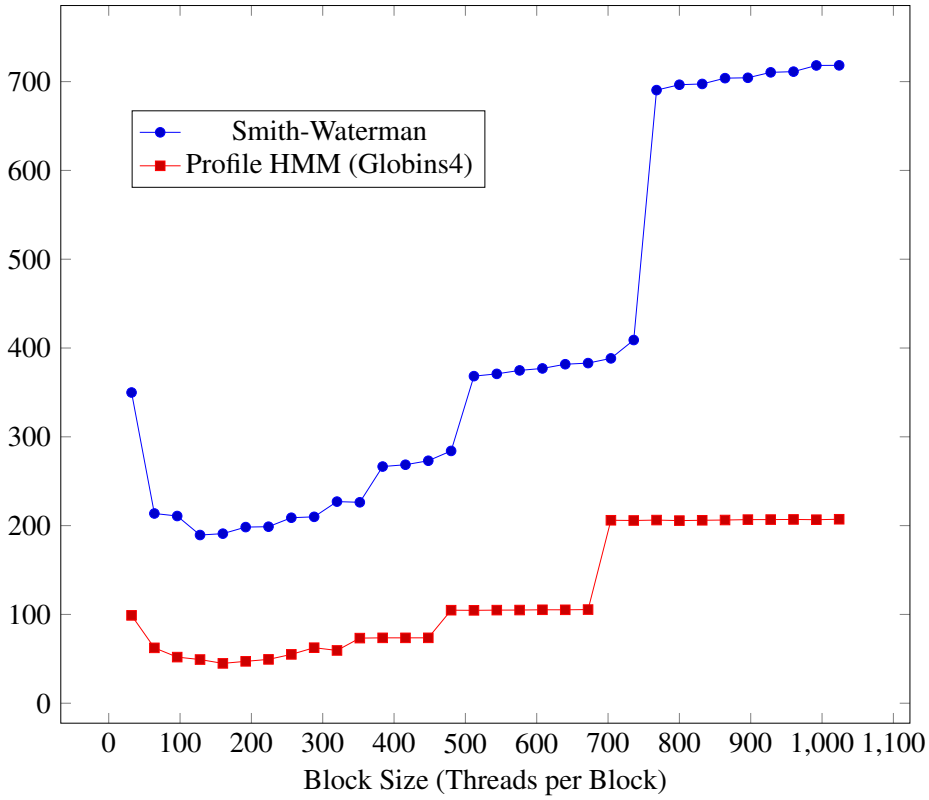


Figure 7.23: Block size effect on performance, using task-per-block on short average partitions.

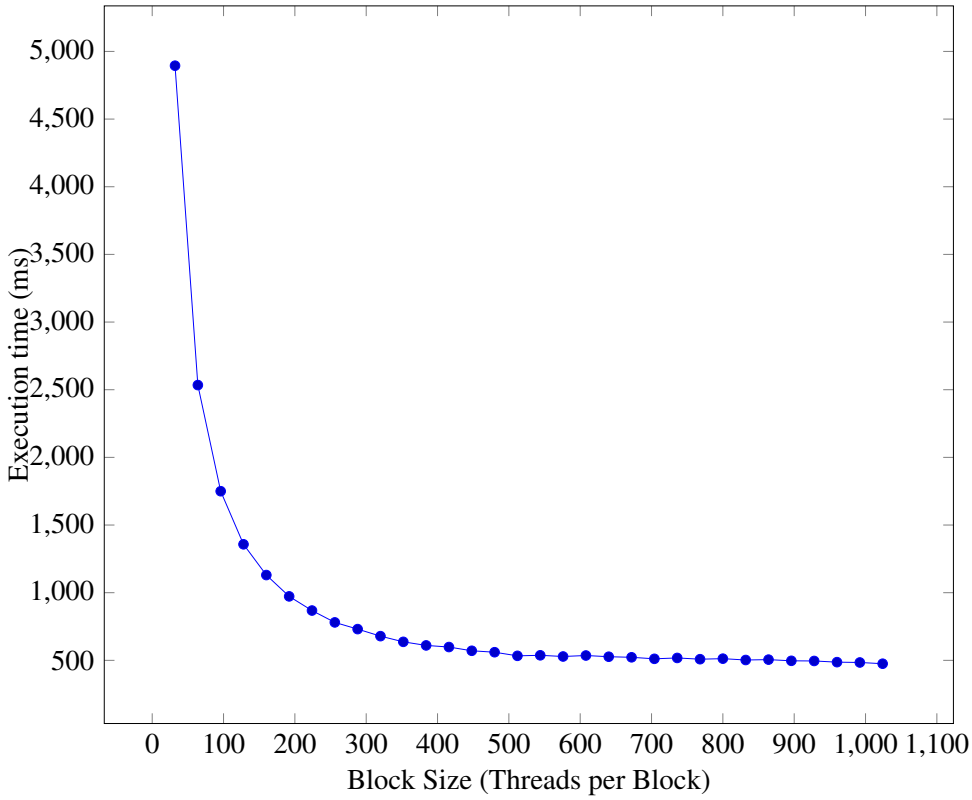


Figure 7.24: Block size effect on performance, using task-per-block on long average partition problems (Smith-Waterman, average partition size of 500).

for synchronisation than warps in separate blocks.

Task-per-block, long average partitions (>= 500) Figure 7.24 plots block size against performance for a number of larger tasks. We can conclude that larger block sizes are more beneficial in these scenarios. The difference between short partitions and long partitions is gradual; Figure 7.23 slowly morphs into Figure 7.24 as the average partition size increases. A threshold is used to automatically determine when to switch from one to the other.

Task-per-thread Figure 7.25, using task-per-thread, shows a weak correlation between shorter block sizes and improved performance on problems of all sizes. There is poor correlation to occupancy. This is unexpected; since there is no synchronization between threads, and separate work occurs on each thread, we might expect a stronger correlation with occupancy.

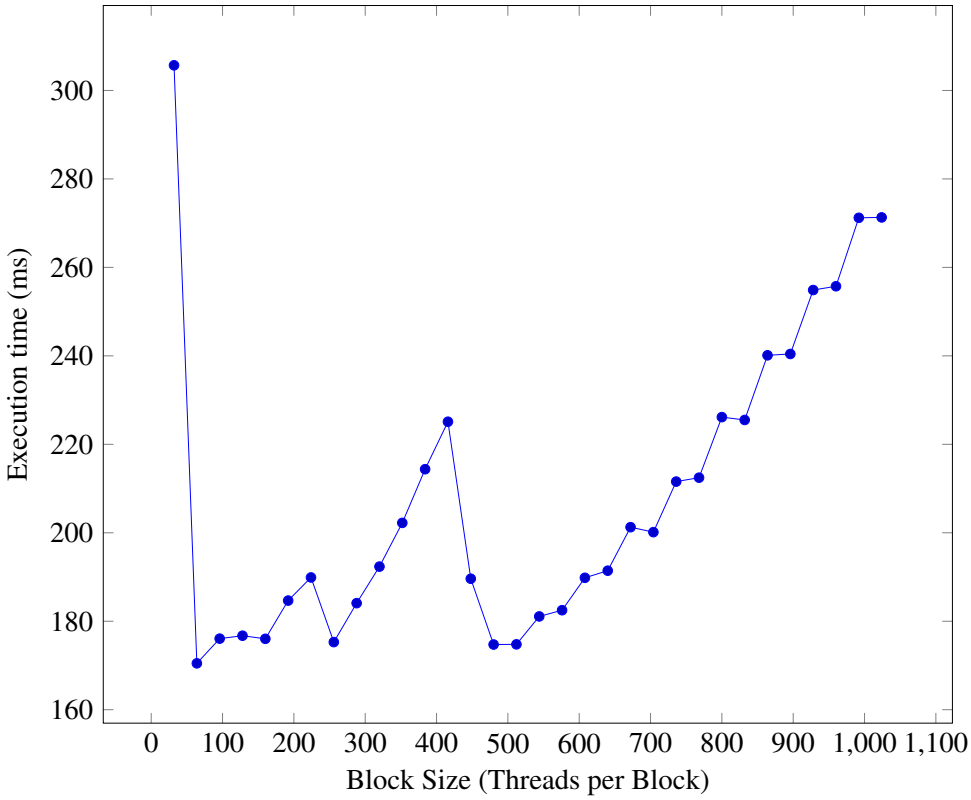


Figure 7.25: Block size effect on performance, using task-per-thread (Smith-Waterman).

7.3.6 Grid size heuristics

Section 5.5.3 made a number of assertions based on the observed impact the grid size has on performance. To illustrate the performance characteristics behind these assertions, we will consider varying the grid size for a Smith-Waterman problem of 13356 elements on a GTX 480. The graphs report the time taken between the launching the first kernel and completing the last kernel call; the times therefore include the overhead of launching multiple kernels and any difference in total execution time on the device.

Larger block sizes, in general, provide better performance Figure 7.26 and Figure 7.27 demonstrate the clear correlation between the grid size and performance. Beyond a certain point increases in grid size have very little effect. In this case, our decision to choose 8192 appears to be sensible.

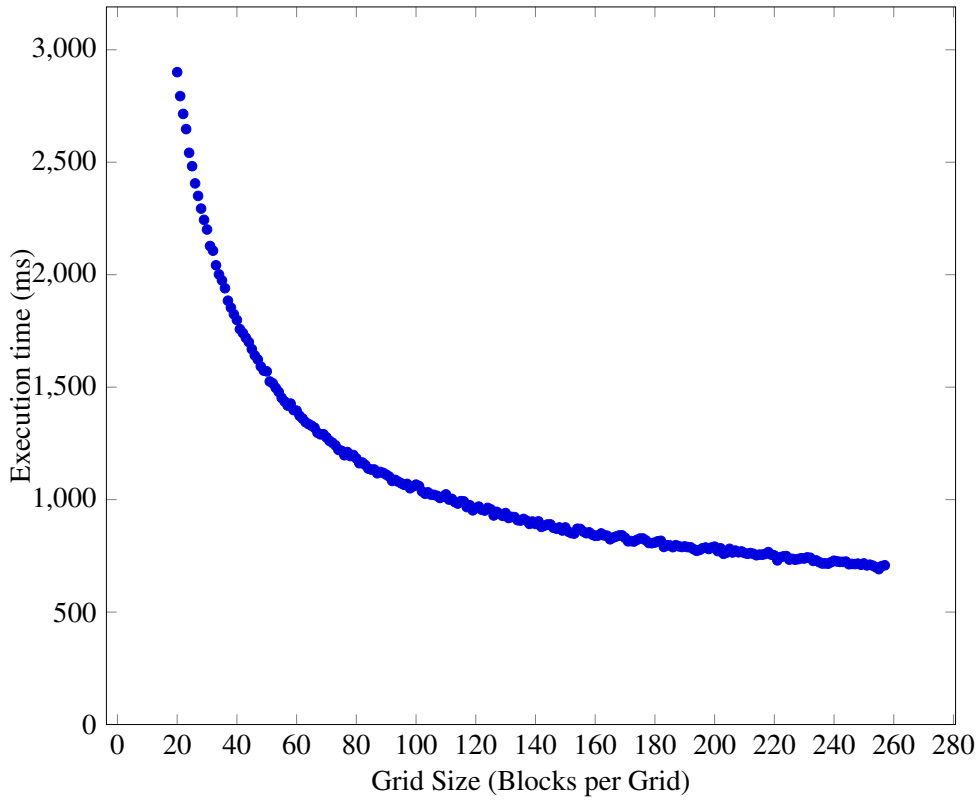


Figure 7.26: The effect of grid size on performance, 20-256

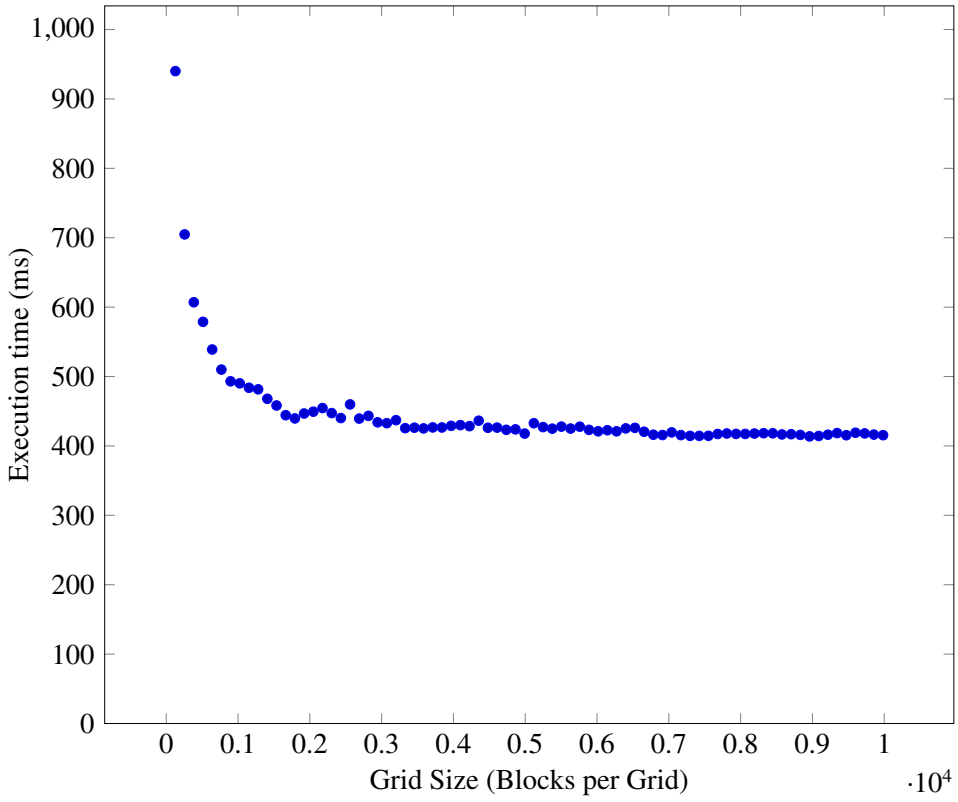


Figure 7.27: The effect of grid size on performance, larger values

Sorted sequences have strange performance characteristics The previous graphs (Figure 7.26 and Figure 7.27) demonstrate the performance with randomised data. When the sequence input data to Smith-Waterman is sorted – and thus the blocks are ordered roughly by execution time – it causes odd performance characteristics to appear. Consider Figure 7.28, which is the sorted version of the same dataset as Figure 7.26. This odd pattern occurs at increments of 16 between 20 and 121, then becomes a much longer cycle; on other devices we have observed multiples of 32 or 8. From the published information about the NVIDIA hardware, there appears to be no specific reason for this, and we can only speculate that the scheduler is sensitive to workload which is sorted in order of length of execution.

Fortunately, once the workload is large enough this effect disappears – compare Figure 7.29 with Figure 7.27.

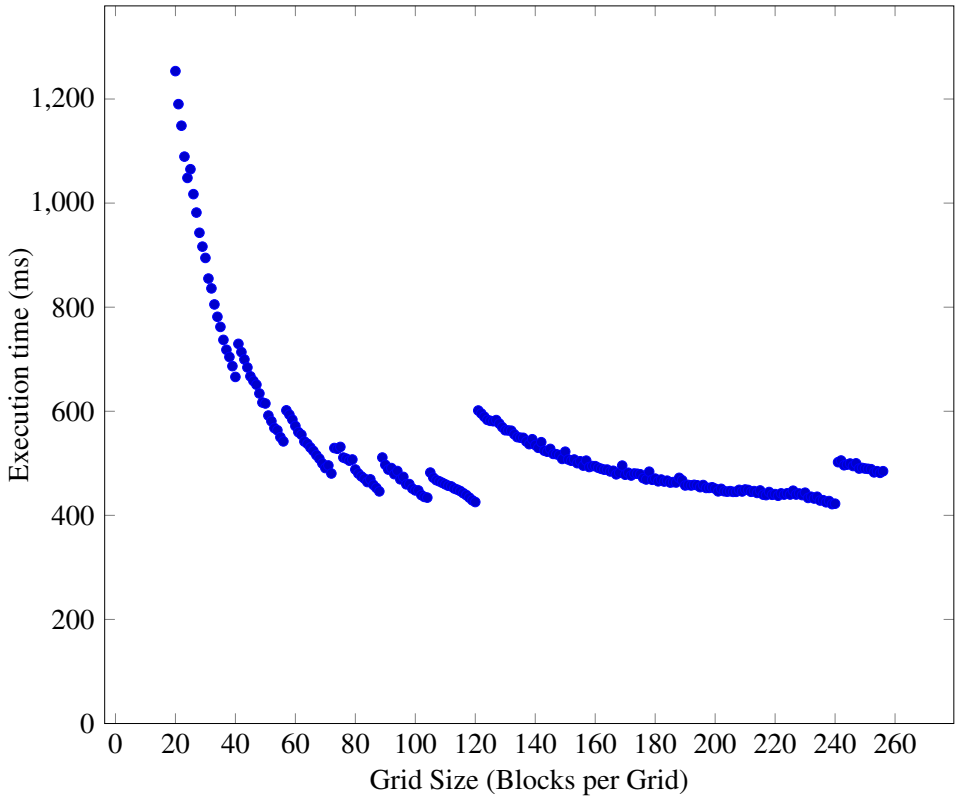


Figure 7.28: The effect of grid size on performance for a sorted dataset

7.4 Summary

In this chapter we have evaluated the work of the previous chapters – in particular, the performance, the expressiveness and the ease of use. We have shown how our core DSL is a good semantic fit for the type of applications that bioinformaticians wish to write by identifying the isomorphism with the natural definition of their algorithms. Furthermore, the framework provides a freedom of expression when designing the syntax for extensions that ensures that they, to, can provide a language which is natural to read an write.

The framework we have designed also has advantages with respect to the ease of development; both in terms of the number of lines of code needing to be written and the speed with which the applications can be developed. By inviting three students to write an extension of their own, we have validated that it is possible for those without extensive compiler or computer science expertise to write such extensions.

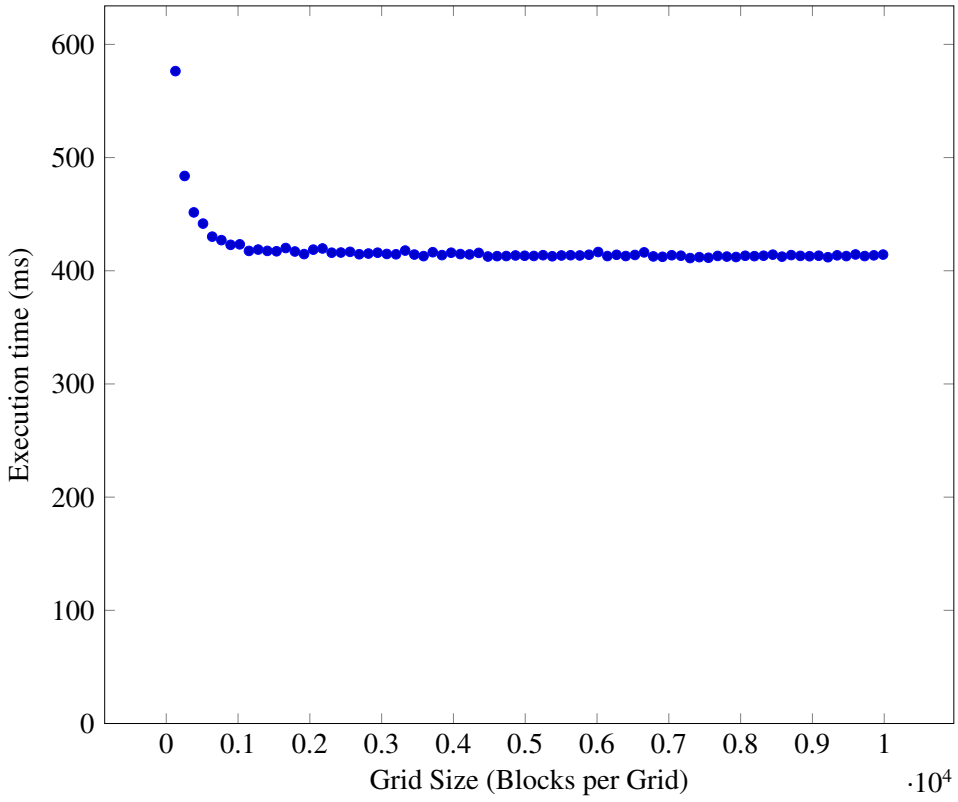


Figure 7.29: The effect of grid size on performance for a sorted dataset, larger values.

In order to evaluate the comparative performance, we have presented experimental evidence measuring the performance of a number of applications from bioinformatics in our DSL against best-of-breed CPU and GPU hand-coded equivalents. The results are promising; our high-level DSL programs provide competitive performance with the hand-coded solutions. In addition, we have considered benchmarks designed to evaluate the performance of individual aspects of the framework. Whilst these benchmarks illustrate that automatically identifying application parameters such as memory location, block size and so forth is a complex business, they also demonstrate that, for the applications we are concerned about, our heuristics solutions are satisfactory.

Chapter 8

Conclusion

The central theme of this dissertation has been that DSLs are a practical and effective way of writing applications for massively-parallel processors. On this score, we can certainly conclude a measure of success: we have succeeded in automatically synthesising applications with competitive performance from high-level definitions. The work presented here, and supporting evidence from this burgeoning field, suggests that judicious use of DSL techniques can be used productively to tame the complexity of these massively parallel architectures.

If we have an unqualified yes for the answer to the question "Are DSLs a helpful technique?" in these circumstances, it is the practical question of *how* to achieve this parallelisation that has occupied the majority of this dissertation. First and foremost, how we could identify parallelisation opportunities in DSL code, closely followed by how we could synthesise correct GPU code for these parallelisation opportunities. Furthermore, how we then launch these synthesised kernels with optimal choices for memory location and parameters such as block size and grid size. The final question we sought to answer was how we can reduce the cost of developing these DSLs in the first place? By summarising the contributions, we hope to address the progress made on these questions.

8.1 Summary of contributions

A key contribution of this dissertation is the use of a hybrid "DSLs-within-a-DSL" approach. By providing a customised host language which can be extended by embedded DSLs, new domains can be added at minimum cost. Such new domain extensions only need to describe those features that are unique to that domain. New domains are automatically integrated with existing domains, can share common features, and, importantly, share any parallelisation effort placed on the host language.

We have presented a framework built along these lines that simplifies the development of GPU DSLs by significantly reducing the burden on domain developers. An important feature of the framework, and a key technique to minimising the cost of development, is the separation of the concerns of the *domain expert* from the concerns of the *GPU expert*. The former can create and modify the DSLs without the requirement of understanding the mechanics of the GPU. For the latter, they can modify the routines that determine how the GPU is used without needing to fully understand the domain from which the applications come; they can help encode GPU expertise into the compiler itself. The complexity of developing DSLs for GPUs is that you need to be an expert in both GPUs and the domains before you can start; by adopting this flexible approach we can ensure experts are developing to their expertise.

Fundamental to achieving this separation is the automatic parallelisation of the host recursive language. By adopting and extending existing techniques, for both analysis and code generation, we can generate efficient GPU code from the provided specifications. This is achieved by partitioning the domain of a recursion into sets of values that can be computed concurrently, as defined by a scheduling function. A key contribution is to describe how we can select such a function and subsequently derive a graphics card implementation. We apply insights from parallelisation using the polyhedral model to problems defined recursively on modern massively parallel processors.

We introduce a DSL API which both provides a simplified platform for development,

but also, importantly, an opportunity for optimisation. Systematic use of this API provides the information required for automatically allocating data structures to particular memory locations. The system encodes GPU expertise in the form of customisable cost functions, in order to make these informed allocation decisions without requiring user intervention.

The performance results demonstrate that the system can systematically generate efficient GPU implementations from the DSL specifications for a wide variety of real-world bioinformatics applications. The results are, in general, a significant improvement on equivalently flexible CPU applications, and are often comparable to best-of-breed GPU and CPU applications for even narrower domains. Furthermore, our system provides access to this GPU-level performance at a substantially lower cost (in terms of lines of code) than a hand-written GPU or CPU application, and with much greater flexibility.

8.2 Limitations

An important limitation of my work is that it is focused on the single, albeit broad, domain of bioinformatics. In any DPhil it is important to properly define limits to the scope of the work, and this constraint ensured I could thoroughly and deeply cover that single topic. Furthermore, bioinformatics is a domain with many related sub-domains, and this provided a rich and interesting environment in which to build an eco-system for bioinformatics DSLs.

However, it is clear that this does limit the conclusions that can be made; we can make no broad assumptions about how well our techniques work in other fields. The work is thus partly a *proof-of-concept* i,e the concept that we can automatically derive parallelisation from suitably formulated, narrow languages.

Even though this lack of breadth does limit our conclusions, it does not prevent us identifying potential points of reuse. Whilst certain techniques (functional host language, model extensions) may not be widely applicable outside bioinformatics, the overall structure (DSLs-within-a-DSL), and GPU specific techniques (heuristics for parameter setting,

adoption of the polyhedral model with our GPU specific translations, sliding window) should prove to be transferable outside the domain.

A limitation of the framework itself is that it requires some compiler knowledge to build DSL extensions. This includes an understanding of the compiler pipeline, definition of lexing and parsing routines and specification of synthesised attributes. In mitigation, my focus has been on reducing the cost of GPU DSLs specifically, rather than improving or focusing on easing DSL development in general. There is a significant body of work on developing fully embedded DSLs (particularly in Scala, e.g Delite [20]), that could be profitably integrated with the results of this dissertation.

The parallel analysis is limited to a subset of dynamic programming problems, and imposes limitations on a number of different aspects of the recursion. Successful analysis requires that at least some arguments are affine functions of the input parameters and that mutual recursion have the same parameters and parallelisation. These constraints are pragmatic as opposed to fundamental; they are not overly restrictive for bioinformatics applications and simplify the analysis steps.

The focus of this dissertation has been the generation of code for a task-per-block strategy. Other strategies e.g task-per-thread have been implemented, but not aggressively optimised, and strategies for much larger applications, e.g task-per-multiple-block have not been implemented at all. Task-per-block has been sufficient to support the initial phase of applications we were interested in.

A limitation in the practical experimental section is that we provide comparison on only a relatively small number of applications. This is attributable to two factors: the lack of easily comparable applications and the prototype nature of the framework and the compiler. Fixing the former requires more bioinformatics experts to take up the porting of applications. The consequence of the latter is that the compiler is quite brittle, making development sometimes harder than necessary. Furthermore, the support for manipulating the results once computed (for example, sorting, ordering, traceback of best solutions) is not

extensive in the system, primarily due to time and scope constraints. With more developers working on the compiler, these problems could easily be rectified.

8.3 Further Work

One of the most obvious ways to extend this work would be tackle another, similarly broad, domain using the DSL-within-a-DSL technique. Scientific domains abound with potential examples, and it would help determine which techniques would be more broadly applicable, and which are specific. In addition, there are further domains (e.g Stochastic Context Free Grammars) which would be easily implemented within the framework as it stands, and provide further evidence for the claim that the framework is broadly useful.

As has been previously mentioned, the focus of this dissertation has been specifically GPU DSLs. It is certainly the case that better techniques for defining new embedded languages are available; integrating this work on GPU DSLs with state-of-the-art DSL development techniques would provide further reductions in development cost, and increases in ease of development. In particular, the ability to define new DSLs within the language in a truly embedded way would reduce the need for compiler knowledge to build these new DSLs.

The framework allows for significant levels of customisation in the way that many parameters and options are set. The refinement of these aspects by GPU experts can certainly improve performance overall, and ensures that the framework can remain current with the introduction of new devices. This encoding of GPU expertise is an important keystone in the technique.

The application of the polyhedral model could also be extended significantly. In particular, the implementation of a task-per-multiple-blocks parallel strategy, which would work through a *tiling* process, could be adopted in order to use the full capacity of the GPU with very large recursions on smaller problem sets. Further optimisation of the task-per-thread

strategy would also apply a different transformation using the polyhedral model; using a schedule that optimised for data reuse rather than parallel potential. This would be part of a wider set of changes to optimised the memory operations further in task-per-thread contexts.

We could make improvements in the low-level code generation by adopting LLVM, which has recently been extended by NVIDIA with a PTX code generator. The major benefit of adopting this backend is that it includes many of the classical compiler optimisations (such as common sub-expression elimination), which could be run profitably on our code to improve overall performance.

A further point of interest would be more extensive optimisations for the latest "Kepler" NVIDIA GPUs, which use ILP (Instruction Level Parallelism) to a much greater extent. As it stands, we do not generate code which is optimised for using ILP. One option would be to generate code which evaluates multiple tasks simultaneously. As new forms of massively-parallel processors emerge, we would also like to tackle those using similar techniques. Examples here include the Intel Xeon Phi, as well as new up-and-coming chips such as the Adapteva Parallella. By exploiting these different forms of massively parallel processor, we can maximise the benefits we see from describing problems as DSLs, and ensure that our DSL programs remain current and useful no matter which platform is ultimately successful.

Appendix A

Encoding of bioinformatics algorithms

This appendix provides a number of examples of how the bioinformatics algorithms and models described in Chapter 3 could be described in our language and extensions.

A.1 Pairwise Sequence Alignment

A.1.1 Needleman-Wunsch

```
int f(matrix sm, const d,
          seq[protein] s, index[s] i, seq[protein] t, index[t] j)
   | i == 0 or j == 0 = 0
   | otherwise = f(i - 1, j - 1) + sm[s[i -1], t[j - 1]])
                 max f(i - 1, j) - d
                 max f(i, j - 1) - d
```

A.1.2 Smith-Waterman

```
int h(matrix sm, const ins, const ext,
          seq[protein] s, index[s] i, seq[protein] t, index[t] j)
   | i == 0 or j == 0 = 0
```

```
    | otherwise = 0
                  max e(i, j)
                  max f(i, j)
                  max (h(i - 1, j - 1) + sm[s[i -1], t[j - 1]])

int e(matrix sm, const ins, const ext,
          seq[protein] s, index[s] i, seq[protein] t, index[t] j)
    | i == 0 or j == 0 = 0
    | otherwise = (e(i, j - 1) - ext) max (h(i, j - 1) - ext - ins)

int f(matrix sm, const ins, const ext,
          seq[protein] s, index[s] i, seq[protein] t, index[t] j)
    | i == 0 or j == 0 = 0
    | otherwise = (f(i - 1, j) - ext) max (h(i - 1, j) - ext - ins)
```

A.2 Substitution Matrix

```
define matrix blosum50[protein] {
       A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
   A   5 -2 -1 -2 -1 -1 -1  0 -2 -1 -2 -1 -1 -3 -1  1  0 -3 -2  0
   R  -2  7 -1 -2 -4  1  0 -3  0 -4 -3  3 -2 -3 -3 -1 -1 -3 -1 -3
   N  -1 -1  7  2 -2  0  0  0  1 -3 -4  0 -2 -4 -2  1  0 -4 -2 -3
   D  -2 -2  2  8 -4  0  2 -1 -1 -4 -4 -1 -4 -5 -1  0 -1 -5 -3 -4
   C  -1 -4 -2 -4 13 -3 -3 -3 -3 -2 -2 -3 -2 -2 -4 -1 -1 -5 -3 -1
   Q  -1  1  0  0 -3  7  2 -2  1 -3 -2  2  0 -4 -1  0 -1 -1 -1 -3
   E  -1  0  0  2 -3  2  6 -3  0 -4 -3  1 -2 -3 -1 -1 -1 -3 -2 -3
   G   0 -3  0 -1 -3 -2 -3  8 -2 -4 -4 -2 -3 -4 -2  0 -2 -3 -3 -4
   H  -2  0  1 -1 -3  1  0 -2 10 -4 -3  0 -1 -1 -2 -1 -2 -3  2 -4
   I  -1 -4 -3 -4 -2 -3 -4 -4 -4  5  2 -3  2  0 -3 -3 -1 -3 -1  4
```

```
    L -2 -3 -4 -4 -2 -2 -3 -4 -3  2  5 -3  3  1 -4 -3 -1 -2 -1  1
    K -1  3  0 -1 -3  2  1 -2  0 -3 -3  6 -2 -4 -1  0 -1 -3 -2 -3
    M -1 -2 -2 -4 -2  0 -2 -3 -1  2  3 -2  7  0 -3 -2 -1 -1  0  1
    F -3 -3 -4 -5 -2 -4 -3 -4 -1  0  1 -4  0  8 -4 -3 -2  1  4 -1
    P -1 -3 -2 -1 -4 -1 -1 -2 -2 -3 -4 -1 -3 -4 10 -1 -1 -4 -3 -3
    S  1 -1  1  0 -1  0 -1  0 -1 -3 -3  0 -2 -3 -1  5  2 -4 -2 -2
    T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  2  5 -3 -2  0
    W -3 -3 -4 -5 -5 -1 -3 -3 -3 -3 -2 -3 -1  1 -4 -4 -4 15  2 -3
    Y -2 -1 -2 -3 -3 -1 -2 -3  2 -1 -1 -2  0  4 -3 -2 -2  2  8 -1
    V  0 -3 -3 -4 -1 -3 -3 -4 -4  4  1 -3  1 -1 -3 -2  0 -3 -1  5
}
```

A.3 Hidden Markov Model

A.3.1 Viterbi Algorithm

```
prob viterbi(hmm h, state[h] s, seq[*] x, index[x] i) =
   if i == 0 then
      if s.isstart then 1.0 else 0.0
   else
      // The end state is silent
      (if s.isend then 1.0 else s.emission[x[i-1]])
      * max(t in s.transitionsto :
            t.prob * viterbi(t.start, i - 1))
```

A.3.2 Forward Algorithm

```
prob forward(hmm h, state[h] s, seq[*] x, index[x] i) =
   if i == 0 then
      if s.isstart then 1.0 else 0.0
```

```
4     else
5        // The end state is silent
6        (if s.isend then 1.0 else s.emission[x[i-1]])
7        * sum(t in s.transitionsto :
8              t.prob * forward(t.start, i - 1))
```

A.3.3 Backward Algorithm

```
1  prob backward(hmm h, state[h] s, seq[*] x, index[x] i) =
2     if i == x.size then
3        if s.isend then 1.0 else 0.0
4     else
5        // The end state is silent
6        (if s.isstart then 1.0 else s.emission[x[i-1]])
7        * sum(t in s.transitionsfrom :
8              t.prob * backward(t.end, i + 1))
```

A.3.4 Gene-finder

```
1  define hmm genefinder {
2      alphabet dna;
3      startstate start;
4      state background emits emitbackground;
5      state startcodon emits emitstart;
6      state gene emits emitcodon;
7      state stop emits emitstop;
8      endstate end;
9
10     const gene_density = 0.001;
11     const gene_length = 100;
```

```
12
13      start -> background 1.0;
14      background -> background (1.0 - 0.0001 - gene_density);
15      background -> startcodon gene_density;
16      background -> end 0.0001;
17      startcodon -> gene 1.0;
18      gene -> gene (1 - 1/gene_length);
19      gene -> stop (1/gene_length);
20      stop -> background 1.0;
21
22      emission emitbackground =  * : 0.25;
23      emission emitstart =  ATG : 1, * : 0 ;
24      emission emitstop =  TAG : 1/3, TAA : 1/3, TGA : 1/3, * : 0 ;
25      emission emitcodon =  TAG : 0, TAA: 0, TGA : 0, * : 1/61 ;
26  }
```

A.4 RNA Secondary Structure Prediction

A.4.1 Nussinov Algorithm

```
1  prob N(matrix m, seq[*] x, index[x] i, seq[*] y, index[y] j) =
2     if j <= i + 3 then 0
3     else
4        N(i + 1, j) max
5        * max(k in i...j :
6              N(i,k) + N(k + 1, j))
```

Bibliography

[1] CARP: Correct and Efficient Accelerator Programming project website. http://carp.doc.ic.ac.uk/external/index.php.

[2] JOpt Website. `http://jopt.sourceforge.net/`.

[3] OpenACC: Directives for Accelerators. http://www.openacc-standard.org/.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[6] Soufiane Baghdadi, Armin Grlinger, and Albert Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work, 2010.

[7] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, april 2009.

[8] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[9] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[10] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[12] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[14] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[16] E. Burmako and R. Sadykhov. Conflux: Embedding massively parallel semantics in a high-level programming language. *The Eleventh International Conference on Pattern Recognition and Information Processing (PRIP)*, 2011.

[17] Luke Cartey, Rune Lyngsø, and Oege de Moor. Synthesising graphics card programs from DSLs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 121–132, New York, NY, USA, 2012. ACM.

[18] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[19] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[20] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 835–847, New York, NY, USA, 2010. ACM.

[21] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, Hyouk Joong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[22] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[23] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107, 2008.

[24] Marina C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 131–139, New York, NY, USA, 1986. ACM.

[25] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: a parallel DSL for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM.

[26] Jacques Cohen. Bioinformatics - an introduction for computer scientists. *ACM Computing Surveys*, 36:122–158, 2004.

[27] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–76, 1999.

[28] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[29] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1999.

[30] Sean Eddy. HMMeR Website, including User Manual. http://hmmer.wustl.edu.

[31] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[32] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 45–56, New York, NY, USA, 2004. ACM.

[33] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.

[34] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

[35] Narayan Ganesan, Roger D. Chamberlain, Jeremy Buhler, and Michela Taufer. Accelerating HMMER on GPUs by implementing hybrid data and task parallelism. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 418–421, New York, NY, USA, 2010. ACM.

[36] Robert Giegerich and Carsten Meyer. Algebraic dynamic programming. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 349–364, London, UK, 2002. Springer-Verlag.

[37] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, December 1982.

[38] Martin Griebl and Christian Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jlich*, pages 311–320. Forschungszentrum, 1996.

[39] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbrger, Armin Grösslinger, and Louis-Nol Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[40] Armin Grösslinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 236–250, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing DSLs in metaOCaml. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 41–42, New York, NY, USA, 2004. ACM.

[42] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[43] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In Zoltan Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[44] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[45] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 166–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[46] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international*

symposium on Computer architecture, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[47] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 11–, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[49] A.C. Jacob, J.M. Lancaster, J.D. Buhler, and R.D. Chamberlain. Preliminary results in accelerating profile HMM search on FPGAs. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[50] Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain. Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 87–94, Washington, DC, USA, 2010. IEEE Computer Society.

[51] Arpith Chacko Jacob. *Parallelization of dynamic programming recurrences in computational biology*. PhD thesis, Washington University, 2010.

[52] Simon L. Peyton Jones and Satnam Singh. A Tutorial on Parallel and Concurrent Programming in Haskell. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 267–305. Springer, 2008.

[53] Will Jones, Tony Field, and Tristan Allwood. Deconstraining DSLs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 299–310, New York, NY, USA, 2012. ACM.

[54] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. In *Proceedings of the 17th ACM SIGPLAN*

symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 303–304, New York, NY, USA, 2012. ACM.

[55] Chris Karlof and David Wagner. Hidden Markov model cryptanalysis. In *Cryptographic Hardware and Embedded Systems – CHES '03*, pages 17–30. Springer-Verlag, 2003.

[56] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14:563–590, July 1967.

[57] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, February 2010.

[58] Anders Krogh, I. Saira Mian, and David Haussler. A hidden Markov model that finds genes in E.coli DNA. *Nucleic Acids Research*, 22(22):4768–4778, 1994.

[59] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[60] Julian Kupiec. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech & Language*, 6(3):225–242, 1992.

[61] Leslie Lamport. The parallel execution of do loops. *Communications of The ACM*, 17:83–93, February 1974.

[62] B. Larsen. Compiling an array language to a graphics processor. Master's thesis, University of New Hampshire (Department of Computer Science), 2010.

[63] Bradford Larsen. Simple optimizations for an applicative array language for graphics processors. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 25–34, New York, NY, USA, 2011. ACM.

[64] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. GPU Kernels as Data-Parallel Array Computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM)*, 2009.

[65] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416. Springer-Verlag, 1993.

[66] Chuan Liu. cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. Project Report for the Course Parallel Programming, Johns Hopkins University, `http://liuchuan.org/pub/cuHMM.pdf`, 2009.

[67] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8, April 2006.

[68] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In Yves Robert, Manish Parashar, Ramamurthy Badrinath, and ViktorK. Prasanna, editors, *High Performance Computing - HiPC 2006*, volume 4297 of *Lecture Notes in Computer Science*, pages 363–374. Springer Berlin Heidelberg, 2006.

[69] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[70] Yongchao Liu, B. Schmidt, and D.L. Maskell. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 121–128, July 2009.

[71] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, 2010.

[72] Gerton Lunter. HMMoC: a compiler for hidden Markov models. *Bioinformatics*, 23(18):2485–2487, September 2007.

[73] N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is bioinformatics? An introduction and overview. In *Yearbook of Medical Informatics 2001*, 2001. `www.ebi.ac.uk/luscombe/docs/imia_review.pdf`.

[74] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. *SIGPLAN Not.*, 47(9):311–322, September 2012.

[75] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell '10: Proceedings of the third ACM Haskell symposium on Haskell*, pages 67–78, New York, NY, USA, 2010. ACM.

[76] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[77] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[78] Graham R. Markall, David A. Ham, and Paul H. J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815–1823, 2010.

[79] Mercedes Marques, Gregorio Quintana-Orti, Enrique S. Quintana-Orti, and Robert van de Geijn. Using graphics processors to accelerate the solution of out-of-core linear systems. In *Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, ISPDC '09, pages 169–176, Washington, DC, USA, 2009. IEEE Computer Society.

[80] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[81] Mike McShaffry. *Game Coding Complete*. Paraglyph Press, 2004.

[82] Zeeya Merali. Computational science: ...Error. *Nature*, 467(7317):775–777, October 2010.

[83] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[84] G.R. Mudalige, M.B. Giles, I. Reguly, C. Bertolli, and P. H J Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, 2012.

[85] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[86] Adam Novak, Istvan Miklos, Rune Lyngsoe, and Jotun Hein. StatAlign: an extendable software package for joint Bayesian estimation of alignments and evolutionary trees. *Bioinformatics*, 24(20):2403–2404, 2008.

[87] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[88] NVIDIA. *NVIDIA CUDA C Best Practices Guide 5.5*. 2013. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[89] NVIDIA. *NVIDIA CUDA Occupancy Calculator 5.5*. 2013. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[90] NVIDIA. *NVIDIA CUDA Parallel Thread Execution ISA Version 3.2*. 2013. `http://docs.nvidia.com/cuda/parallel-thread-execution/`.

[91] NVIDIA. *NVIDIA CUDA Programming Guide 5.5*. 2013. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[92] Konstantin Okonechnikov, Olga Golosova, Mikhail Fursov, and the UGENE team. Unipro ugene: a unified bioinformatics toolkit. *Bioinformatics*, 28(8):1166–1167, 2012.

[93] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM.

[94] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of The National Academy of Sciences*, 85:2444–2448, 1988.

[95] D. P. Playne and K. A. Hawick. Auto-generation of Parallel Finite-Differencing Code for MPI, TBB and CUDA. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1168–1175, Washington, DC, USA, 2011. IEEE Computer Society.

[96] E. Pollastri and G. Simoncelli. Classification of melodies by composer with hidden Markov models. In *Proceedings of the First International Conference on Web Delivering of Music*, pages 88–95, 2001.

[97] Andreas Prlić, Andrew Yates, Spencer E. Bliven, Peter W. Rose, Julius Jacobsen, Peter V. Troshin, Mark Chapman, Jianjiong Gao, Chuan Hock Koh, Sylvain Foisy, Richard Holland, Gediminas Rimša, Michael L. Heuer, H. Brandstätter-Müller, Philip E. Bourne, and Scooter Willis. BioJava: an open-source framework for bioinformatics in 2012. *Bioinformatics*, 24(18):2096–2097, 2012.

[98] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[99] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 208–214, New York, NY, USA, 1984. ACM.

[100] Patrice Quinton and Vincent Van Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays. *Journal of VLSI Signal Processing*, 1:95–113, 1989.

[101] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286, 1989.

[102] S.K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE*, 76(3):259–269, 1988.

[103] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012.

[104] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: a just-in-time parallel accelerator for Python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[105] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, December 2007.

[106] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[107] Peter Steffen, Robert Giegerich, and Mathieu Giraud. GPU parallelization of Algebraic Dynamic Programming. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part II*, PPAM'09, pages 290–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[108] Andrew Stromme, Ryan Carlson, and Tia Newhall. Chestnut: a GPU programming language for non-experts. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 156–167, New York, NY, USA, 2012. ACM.

[109] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. OptiML: an implicitly parallel

domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, June 2011.

[110] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. `www.gotw.ca/publications/concurrency-ddj.htm`.

[111] Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065 – 2074, 2010. ICCS 2010.

[112] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[113] K. Takahashi, H. Yasuda, and T. Matsumoto. A fast HMM algorithm for on-line handwritten character recognition. In *Proceedings of the Fourth International Conference on Document Analysis and Recognition*, volume 1, pages 369–375, 1997.

[114] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[115] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22(22):4673–4680, Nov 1994.

[116] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for Cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[117] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, March 2007.

[118] Herv Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing*, 3:173–182, 1991.

[119] V. Volkov. Better performance at lower occupancy. Presented at GPU Technology Conference, http://www.cs.berkeley.edu/~{}volkov/volkov10-GTC.pdf, 2010.

[120] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

[121] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary. Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[122] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, march 2010.

[123] Y. Yaacoby and P.R. Cappello. Scheduling a system of affine recurrence equations onto a systolic array. In *Systolic Arrays, 1988., Proceedings of the International Conference on*, pages 373–382, 1988.

[124] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic acids research*, 9(1):133–148, January 1981.