Domain Specific Languages for Massively Parallel Processors
PRS Transfer Report

Luke Cartey

November 2, 2010

Contents

1 Introduction 3

2 Heterogeneous Massively Parallel Processors 4

2.1 Background 4

2.1.1 The genesis of the massively parallel co-processor 4

2.1.2 Mainstream general purpose computing 5

2.1.3 High-Level Frameworks 5

2.1.4 Application growth 6

2.2 Parallel thinking: A worked example 6

2.2.1 Device Basics 6

2.2.2 Thread Granularity 7

2.2.3 Kernel Implementation 8

2.3 Architecture 8

2.3.1 Kernel 8

2.3.2 Comparison to CPUs 9

2.3.3 CUDA 9

2.4 Massively Parallel Optimisation Patterns 10

2.4.1 Memory optimisations 10

2.4.2 Optimising for the hardware 12

2.5 Massively Parallel Toolkit 13

2.5.1 Scan 13

2.5.2 Reduce 13

2.5.3 Sort 13

2.6 Conclusion 14

3 Dynamic Programming 15

3.1 Basic Conditions 15

3.1.1 Optimal substructure 15

3.1.2 Overlapping sub-problems 16

3.2 Comparison to similar techniques 16

3.3 Implementation techniques 16

3.3.1 Tabulation techniques 17

3.4 Applications 17

3.5 Models of Dynamic Programming 18

3.5.1 Sequential Decision Processes 18

3.5.2 Non-serial Dynamic Programming 18

3.5.3 Algebraic Dynamic Programming . . . 18
3.6 Parallelisation on Massively Parallel Architectures . . . 18
3.7 Conclusion . . . 18

4 A Case Study: Bioinformatics **20**
4.1 Pairwise Sequence Alignment . . . 20
4.1.1 Needleman-Wunsch . . . 21
4.1.2 Smith-Waterman . . . 21
4.1.3 Massively Parallel Implementations . . . 21
4.2 Hidden Markov Models . . . 21
4.2.1 Key Algorithms . . . 22
4.2.2 Example applications . . . 24

5 HMMingbird: A compiler for Hidden Markov Models **25**
5.1 Hidden Markov Model Design . . . 26
5.1.1 Model Equivalence . . . 26
5.1.2 Emitting states and emitting transitions . . . 26
5.1.3 Silent States . . . 26
5.2 Prior Work . . . 27
5.3 Language Design . . . 27
5.3.1 Occasionally Dishonest Casino . . . 28
5.3.2 Profile Hidden Markov Model . . . 29
5.3.3 Limitations . . . 31
5.4 Dynamic Programming algorithms for parallel computation . . . 31
5.4.1 Silent State Elimination . . . 32
5.4.2 Dynamic Programming: Implementation . . . 32
5.5 Optimising Techniques . . . 33
5.5.1 Host to device transfers . . . 33
5.5.2 Precision versus speed . . . 33
5.5.3 Memory locations . . . 34
5.5.4 Phased Evaluation . . . 34
5.5.5 Block size heuristics . . . 34
5.5.6 Unused Attributes . . . 35
5.5.7 Sliding Window Dynamic Programming Tables . . . 35
5.5.8 Custom kernels for each program call . . . 35
5.5.9 Sorting Input . . . 36
5.6 Comparisons . . . 36
5.6.1 The Occasionally Dishonest Casino . . . 36
5.6.2 Profile Hidden Markov Model . . . 36
5.7 Conclusions . . . 37
5.8 Further Work . . . 37

6 Thesis Proposal **39**
6.1 Thesis . . . 39
6.2 Strategy . . . 39
6.3 A timeline . . . 40

Chapter 1

Introduction

The last decade has seen the rise of the desktop multi-core processor, to the point where it is the default technology for everything from workstations and servers to laptops. The driver behind this is simple; clock-speeds have reached a zenith that curtails the rapid single core performance gains we saw in the 1990's and early 2000's. Instead, the expansion in transistor count predict by Moore's Law has been exploited by chip designers by duplicating the numbers of cores, thus still increasing overall performance without causing heat or power issues.

Whilst desktop processors have kept to a relatively small number of cores on a single chip, others have seen the opportunity to develop further, by including hundreds or even thousands of cores. The forerunners in this approach have been graphics processors (GPUs), where highly parallel workloads are the norm. There is a trade-off to be made; the more cores you have, the slower and less advanced they can be. A trade-off also has to be made with power consumption and heat dissipation; GPUs have chosen lower clock-speeds which permits more transistors and therefore more cores. The result is a much higher floating point performance - up to and beyond an order of magnitude higher for single precision parallel workloads.

This form of wide parallel processor, often described as a *massively parallel architecture*, is not only beneficial for typical graphics applications; recent interest has arrived from the across scientific computing community. However, despite the low-cost, scalable nature of the hardware, the difficulty of programming such architectures can be a real roadblock to their widespread use. Many-core architectures 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, many-core architectures with hundreds or thousands of cores are worse. They require completely different techniques and tools, even a different mindset.

Considering the complexities involved, it is extremely difficult to expect domain experts to overcome the challenges involved in GPU computing; you require experts in both the domain and the architecture to proceed.

My thesis is therefore to develop declarative Domain Specific Languages ("DSLs") to enable experts in scientific and high-performance computing to make use of massively parallel architectures, without being distracted by the intricate details of the architecture. My focus will be developing languages for optimisations problems, such as those seen in dynamic programming problems, a simple and widely used technique. It is highly relevant in Bioinformatics, an area of scientific computing that also has demanding performance requirements. My proposal is therefore to use my research so far to develop a language for describing simple recursive functions implemented in a parallel compiler targeting GPUs. This will then act as a bootstrap language for specialist declarative DSLs, such as Hidden Markov Models.

Chapters 2 and 3 describe the background in Massively Parallel Processors and Dynamic Programming respectively. Chapter 4 explores some of the techniques used in Bioinformatics, illustrating the links to both dynamic programming and the potential for targeting massively parallel hardware. Chapter 5 details the development of HMMingbird, a language and compiler for Hidden Markov Models targeting GPUs that has been the focus of my first year. Finally, my full thesis proposal can be found in Chapter 6.

Chapter 2

Heterogeneous Massively Parallel Processors

Massively parallel processors (MPPs) are a rapidly expanding area of research, utilising a 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.

During the last five years graphics card manufacturers, such as NVIDIA and ATI, have become increasingly aware of 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 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. Nowhere is this approach more pronounced than in the range of netbooks and nettops using NVIDIA's ION board - supplementing a low power Atom processor with a low power GPU to offload certain data-heavy tasks.

In principal, any application can be ported to the 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. A degree of independence or a repetition of work is a key criteria, 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.

This chapter will focus on the development of algorithms and algorithmic techniques for the massively parallel co-processor, including the exploration of the features and functions of the hardware as well as a software techniques for maximising performance. Many of the techniques we will discuss come from earlier work in massively parallel clusters or supercomputers, where many desktop or server cores are used in tandem. The techniques may be very different from desktop multi-core processing; device wide synchronisation is difficult and undesirable in massively parallel settings so traditional concurrency features such as monitors, semaphores etc. have little use.

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 lead to hardware with **fixed pipelines**, in which a series of pre-determined computational steps are applied to a set of vertices[47].

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 a customised architecture based around 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 operations typically performed extremely fast at the expense of 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).

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, that was to be used in combination with the fixed function hardware. Originally programmed in assembly languages, high-level languages have appeared including Microsoft's HLSL, NVIDIA's Cg[46] and OpenGL's GLSL. This was quickly taken advantage of for applications as diverse as linear algebra[37], physical simulation[29] and ray tracing[52]; indeed the intention of the designers was to create *hardware-oriented*, *general-purpose* languages.

The BrookGPU[9] project was one of the first to take advantage of these low-level hardware compilers for a *general-purpose* computing framework; their 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 both of the large graphics hardware manufacturers. NVIDIA have become market leaders with the Compute Unified Device Architecture (CUDA) compute framework, whilst ATI have developed the Stream SDK in conjunction with OpenCL. These technologies adapt the Brook approach described above, adopting the *kernel* system whilst dropping much if 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 changes as well as software support have been a vital part of this effort.

Intel had been developing the Larrabee micro-architecture as a competitor to ATI and NVIDIA in the GPU sphere, by removing all fixed function devices in the GPU and implementing all functions in software using a large number of x86 cores. Recently this ambitious aim was shelved in favour of the development of a massively parallel co-processor for high performance computing, with the hardware re-badged as the "Knights" series, showing that massively parallel processing is beginning to become a market in its own right.

This notion is clearly supported by the continued development of cross-platform applications and frameworks. OpenCL builds on the system of parallelism described by BrookGPU, CUDA and the StreamSDK to provide a consistent API across both graphics hardware and traditional CPUs. Microsoft have developed DirectCompute, a cross-manufacturer framework included with DirectX, with similar execution patterns to CUDA and OpenCL.

2.1.3 High-Level Frameworks

The perceived complexities with this computational model have also led many to consider "higher-level" frameworks. Microsoft have developed the *Accelerator*[62] framework, which aims to make general-purpose use of parallel, massively parallel machines and FPGAs accessible to the average programmer. Available on wide variety of languages and platforms, including .NET, 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[27]. Similar work has been undertaken for GPU data-parallelisation using array computations in Haskell by defining DSLs[44, 13, 61]. Copperhead[11] is a similar embedded data-parallel language which uses a subset of Python including many data parallel primitives, permitting compact descriptions of GPU algorithms.

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 as the basis for implementing parallel algorithms. The latter should not come as a surprise; array-style processing through frameworks such as map-reduce have become common place in computing across clusters.

A recent interesting development of the first trend has been language virtualisation[12], 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 the domain-focused expert.

2.1.4 Application growth

At the front of queue for these new technologies were the life sciences and the 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 a 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 large amount of processing on smaller datasets.

Massively Parallel processors are not only making a splash at the top-end, with scientific computing. The NVIDIA ION platform is designed to support low-end laptop hardware - typically netbooks or nettops with Atom processors - by including a low-end GPU to speed up specific applications. This can include 1080p decoding, image editing and web browsing, improving the performance of application that would otherwise be sluggish or impossible on low-end hardware.

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[36, 51], 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.

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 compute 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” from the host follows these steps:

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 evaluated 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.

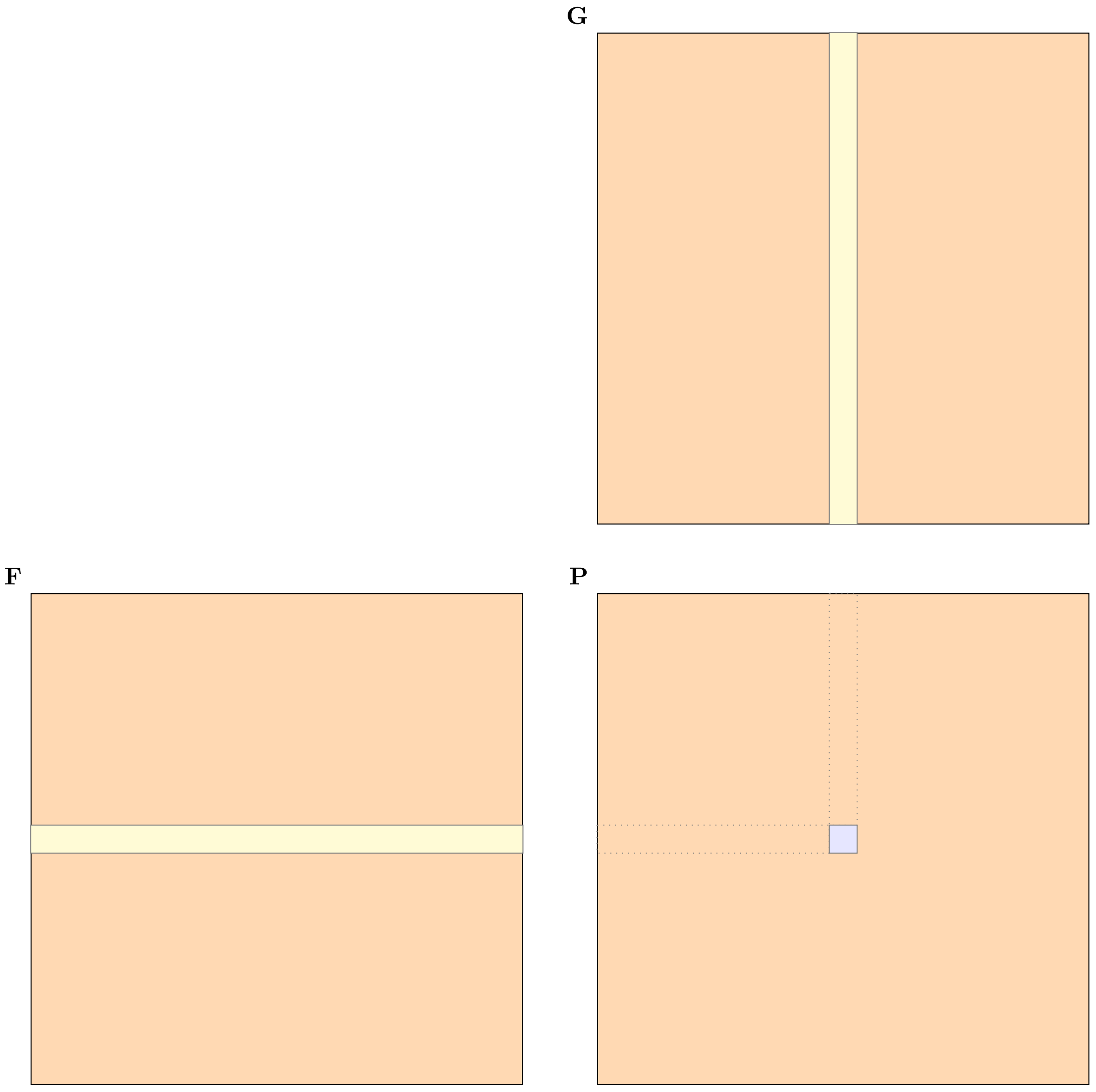

Figure 2.1: Matrix multiplication - computing a cell in **P** requires the dot product of a row in **F** and a column in **G**.

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 to 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 straight-forward. 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 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 + k];
            float nval = m[k * width + x];
            output = mval + nval;
        }

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

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

Figure 2.2: Example kernel for matrix multiplication.

2.3 Architecture

Massively parallel architectures are constructed from a collection of *multiprocessors*, often called *streaming multiprocessors* (SMPs), which bear many of the hallmarks of *vector processors*. 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 8 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 *SIMD* (Single Instruction, Multiple Data) paradigm, and so is often described as a *SIMT* (Single Instruction, Multiple Thread) or *SPMD* (Single Program, Multiple Data) architecture.

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 SMP must co-operate, it does not mean they must all follow the same execution path. Where active threads take a different execution 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 SMP, where some cores remain fallow for much of the time.

The actual hardware implementations introduce more complexity. An SMP 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 SMP 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 state data. 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 typically has 2 to 8 cores on a single chip, whereas a GPU will contain many more, organised by vector multiprocessors. Another difference, driven by the need for high data throughput, is the memory bandwidth available - main memory to the CPU is typically around 10-30GB/s; GPUs can support anywhere up to 180GB/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 SMP 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. In either case, these tend to be much smaller than their CPU counterparts - L3 caches on desktop chips can reach 12MB. This reflects the typical data-usage scenarios required by each processor.

2.3.3 CUDA

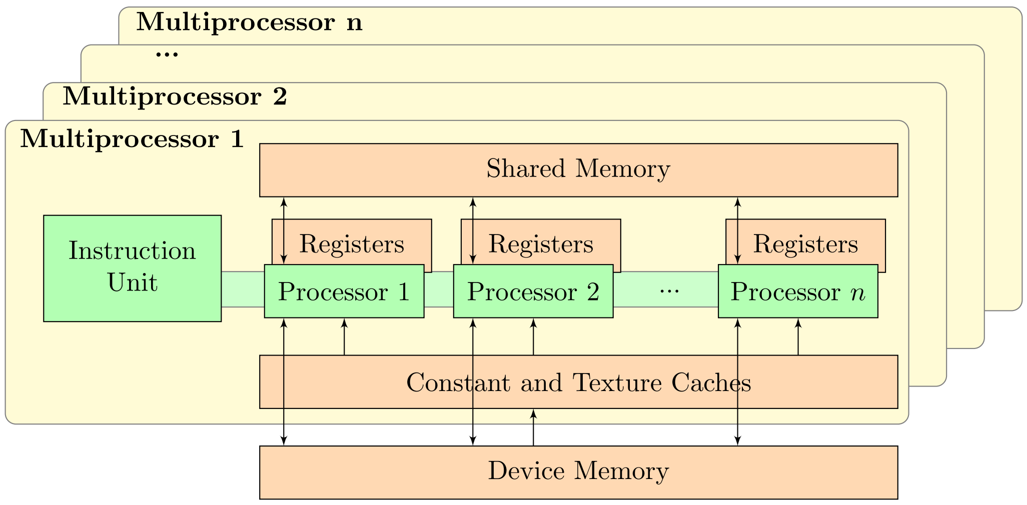

Figure 2.3: The general purpose architecture of a typical CUDA based GPU.

CUDA[51] 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 is developed.

Blocks, threads and warps

CUDA describes a collective group of threads as a *block*. Threads in a block may work co-operatively, and are scheduled to the GPU in smaller groups known as *warps*. Warps may be consider 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. In practice, all threads from the same block are placed on the same multiprocessor, thus ensuring that co-operation is relatively straightforward.

Whilst warps are how threads are scheduled to the device, the execution of each warp depends on the device itself. A CUDA SMP has between 8 and 48 cores, depending on the exact card; a warp is 32 threads. When a warp is executed, we divide the 32 thread warp into half or quarter warps. Each smaller group is then executed in parallel on the device - one thread per core - until the whole warp has been processed; this will happen sequentially. Cards with 32 cores typically run two half warps simultaneously; the half-warps may be from two different full warps. Cards with 48 cores use 16 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.

Memory features

A SMP provides a large number of registers to all cores - registers are not linked to a given core. The SMP also provides an on-die area of fast memory, known as *shared memory.* This shared memory is accessible from all cores and can be used for co-operative working within a block. Each block may allocate an area of shared memory, and both read and write to all locations within that area of memory. The memory itself is similar to a L1 cache on a typical CPU, in that it is close to the processor. In fact, on later devices (those of compute capability 2.0 or above, known as *Fermi* cards) a user-sized section will be used as a cache from the main memory of the device.

Global Memory is the global-in-scope memory location to which both the host and device can read and write. It 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 may be copied from host-to-device or device-to-host by the host only. However, the device may access the data using a pointer passed as a parameter to the kernel function. Read/write speeds are in the range of 100GB/s, however they also come at the cost of a high latency - anywhere between 400 and 800 clock cycles. This memory latency is a conscious design decision - a trade-off that is often mitigated by providing enough work per SMP so that the scheduler may still make full use of SMP whilst the memory fetch is occurring. Appropriate algorithms for this type of parallelisation exhibit a high ratio of arithmetic operations to memory fetch operations.

Since the origins of the multiprocessor in the GPU are in implementing fast hardware pixel shaders, CUDA devices before Fermi 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 to take advantage of the massive parallelism.

2.4 Massively Parallel Optimisation Patterns

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 Memory optimisations

For typical graphics card applications the memory transfer is not a bottleneck - the hardware is therefore not optimised in this direction. Device memory is typically fairly slow, and host to device transfer is fairly limited.

There are a number of features available on the device to support the slow global memory.

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 many orders of magnitude slower 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, ensuring we only transfer what is 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 memory is another common feature that allows the device to access main memory in a just-in-time fashion. This acts as a performance improvement on the device, but only in some cases, and for at most a 15% speed up.

Tiling: Making use of shared memory

Our original formulation 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 one-time 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.

Functionally, shared memory on CUDA devices is split into a number of distinct banks - typical 16 or so, to match the half-warp size. 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 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*. We should therefore ensure that our tile size and ordering is designed with the warp size in mind, to ensure the least number of bank conflicts.

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.

Coalescence

When we access the device memory, we do so using *coalesced* memory reads or writes - that is, we consider each request from a thread in the current warp and bundle it up into a number of 32-byte, 64-byte or 128-byte memory requests. The conditions for this coalescing differ between different chips and architecture versions, but in general coalescing is best [taken advantage of] 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.

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 dimension 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.

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 registers.

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 improving access times for frequently accessed data. Constant data must be allocated on the host and cannot be modified during device calls.

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.

Latency Tolerance

In order to keep costs down, graphics cards typically feature global memory with a relatively high latency. In typical graphics applications, this is not a major issue; immediate data access is not required. For general purpose applications, this can prove to be a little more tricky. However a high computational intensity algorithm can compensate by hiding memory latency - execution continues on the device until the data is actually required.

2.4.2 Optimising for the hardware

Massively parallel processors tend to develop 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 lead 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.

Core density and speed

In principal, the configuration of massively parallel hardware is straight forward - a large number of relatively slow cores. In practice, the exact configuration can vary greatly between manufacturers and even between models. The number of individual cores must be balance 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 typical design of ATI chips compared to NVIDIA chips - the soon-to-be released Radeon HD 6870 is rumoured to have 1920 cores running at around 850MHz, compared to the NVIDIA GTX480 which contains 480 cores at 1401MHz. This decision has trade-offs for algorithm design - the ATI approach gives a higher peak GFLOP/s, however the memory bandwidth between the two devices are similar, so the ATI card requires algorithms with a significantly higher compute intensity (that is, the ratio of arithmetic operations to memory transfer operations) to fully realise the peak output.

Additionally, we can see much variance between how the cores are grouped into multiprocessors. In the latest NVIDIA cards, for example, each multiprocessor contains 32 cores compared to the G80 and G200 architectures that contained only 8 cores per multiprocessor. 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 enough threads in each block to use them all.

Memory size and configuration

Typical memory size for desktop graphics cards are around the 512MB to 1024MB range. For graphics applications, larger memory provides little performance improvement except when using very large resolutions. Special purpose hardware, such as Tesla, tend to move toward 4096MB or more to support more general purpose applications. This memory is an absolute limit - no form of paging usually exist on such devices. Instead, we have to *chunk* data to support larger datasets.

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, those using a high compute intensity to hide memory latency. Where caching was useful, it was recommended as a manual memory allocation to on-chip of 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 ATI support a small amount of L1 cache on each multiprocessor. NVIDIA allow some limited customisation of cache size on the latest Fermi cards, where 56 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. ATI 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. Differences in cache size compare 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.

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, it allows two or more consecutive instructions to be executed in parallel, on the condition that the two instructions are independent. In a massively parallel setting this is supported by providing double the number of cores.

Hardware with such a design is therefore best supported when super-scalar execution is possible - we must therefore design kernels with independence between subsequent instructions. In the worst case scenario - no super-scalar execution is possible - will be using only 50% of the capacity based on a 50/50 split between the cores. In practice, we may share some super-scalar cores between many standard cores. In any case, this hardware specialisation requires very specific conditions to meet peak operating performance.

The GTX 460 is the first CUDA card to support such super-scalar execution. Each core includes two standard execution units of 16 cores, and a third super-scalar unit of 16 cores. When a warp is scheduled to either standard execution unit, the scheduler analyses the following code to identify super-scalar instructions, and schedules the super-scalar unit appropriately. The 2:1 ratio between standard cores and super-scalar cores ensures at least a 66% usage of the card in the worst case.

2.5 Massively Parallel Toolkit

Over recent years, there has been wide-spread research into the development of massively parallel versions of common algorithms. Prior work on vector multiprocessors, supercomputing and data parallel many-core architectures can often find a place in the modern massively parallel programmers toolkit. These can take the form of primitives for building massively parallel algorithms, and many of them are included in the CUDPP library that is shipped with CUDA. These primitives are useful for building algorithms based on efficient building blocks, by adapting the algorithm to use functions that are appropriate for massively parallel execution.

2.5.1 Scan

Parallel prefix sum better known as the *scan* operation was first described in the context of GPGPU programming by Daniel Horn[33], but first developed in the early [] by. The *scan* is produced by taking a binary operator $\oplus$ along with an identity l and an array of n elements $[a_0, .., a_{n-1}]$ and returning $[l, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}]$.

Scan has a wide array of uses across massively parallel processing including a variety of sorting algorithm implementations (radix sort[65], quicksort[6], merge sort), searching techniques, sum operations and many other filtering and reduction methods.

2.5.2 Reduce

A very common pattern in massively parallel programming is to perform some form of *map* function, applied to a set of values by each thread or core, followed by a *reduction* operation - so-called *Map-Reduce*. This paradigm, first developed at Google[19, 20] to use across massive clusters, it is now widely used in distributed massively parallel architectures such as Hadoop[1]. Desktop MPP can also make fruitful use of this technique

Reduction on a massively parallel machine can be surprisingly nuanced. One technique is to use a series of parallel scans to perform the reduction[33].

2.5.3 Sort

Massively parallel architectures are typically consider to be a poor target for sorting algorithms due to the lack of parallelism. However, recent work suggests that sorting can still benefit from a massively parallel approach, with a factor of 2 to 4 improvement suggested, and exceeding one billion 32-bit keys sorted per second. Whilst not the factor 20 to 100 speed ups available in other applications, this is still a worth while improvement.

The most common method of sorting on massively parallel or vector machines builds on a radix sort[65], where each digit is compared in turn, and the output sorted based on this comparison. Implementations, such as that released as part of the CUDPP library[53], use the scan operation described above.

2.6 Conclusion

Massively parallel processors represent a useful alternative architecture for a significant number of algorithms. The application areas of a MPP are, perhaps surprisingly, wide and varied; including applications both at the high-performance and netbook ends of the scale. The utility of desktop MPPs as co-processors are starting become apparent beyond their traditional use in graphics applications, a trend which will only continue as we aim to maximise our performance per watt.

MPPs are distinct from traditional multi-core algorithms, and therefore require distinct execution patterns to achieve maximum performance. The study of algorithms for such architectures has only just begun, however much useful and practical work has been done and work on distributed massively parallel machines continues to influence and support the field. Investigation into so-called primitives such as scans, reductions and sorts will ease the construction of efficient algorithms on MPP.

We should caution that the state-of-art is surprisingly primitive - and support difficult to implement due to the vast hardware diversity available. In the future, we hope to look towards high-level tools to support massively parallel co-processing in a useful and practical way.

Chapter 3

Dynamic Programming

Dynamic programming is a method for improving performance in algorithms that appeal to two key principles: *optimal substructures* and *overlapping subproblems*[16]. The term dynamic programming was coined by Richard Bellman in the early 50's[21] to describe *multi-stage decision problems*, only later did he refine it to the modern definition of nested decision problems. Programming, in this context, relates to the method of *tabulation* of the values.

The application of this technique can result in significant performance improvements over a standard walk across the state space, with benefits in line with the number of repeated elements. Importantly, the technique is both easy to implement and widely applicable. Application areas vary from puzzles (Towers of Hanoi) to mathematical sequences (Fibonacci number) to practical algorithms for optimisation including shortest path and minimum cost algorithms.

3.1 Basic Conditions

3.1.1 Optimal substructure

Richard Bellman describe the key condition that functional equations for optimization problems must adhere to as:

Definition (The Principle of Optimality)**.** *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.[2]*

An algorithm exhibits this property when the optimal solution to one problem can be computed by combining the optimal solutions to some smaller problems in the space. The resulting algorithm is usually described as a recursive functional equation - a form that naturally represents the principle of optimality, with solutions composed of solutions to sub-problems.

Typically, computing the optimal solution for some problem consists of choosing between a series of sub-problems by recursively solving each sub-problem and comparing them. The solution is then combined with some value for the problem itself - typically the "cost" of making the choice.

There is a pattern for discovering optimal substructures in a problem:

1. Show the problem requires making a choice that requires solving one or more sub-problems.
2. Assume that for a given problem, you know the optimal choice to make at each step.
3. Subsequently, determine the sub-problems you would need to solve given the choice above.
4. Use a so-called "cut-and-paste" technique to show the solutions to the sub-problems must themselves be optimal. This is done by contradiction, supposing that a sub-problem is not optimal then proving we can cut out the non-optimal solution and replace it with the optimal solution to the sub-problem and resulting in a better solution to the overall problem[16].

3.1.2 Overlapping sub-problems

Adhering to the *Principle of Optimality* is sufficient for the dynamic programming technique to hold; in this form it is often described as *divide-and-conquer*, and is described in Section 3.2. However, to achieve a performance improvement an algorithm must also exhibit *overlapping subproblems*. Where sub-problems overlap, we can save time by storing the result of each computation and reusing the stored values where possible.

The performance improvement will be dependent on the amount of overlap between different subproblems.

3.2 Comparison to similar techniques

Divide-and-conquer is a technique with very similar conditions to dynamic programming. It does not require that the problem consists of *overlapping subproblems*, instead considering sub-problems that share no elements with any other[16]. This allows the division of work for any problem without reference to any other without duplicating work. As such, divide-and-conquer techniques tend to exhibit better parallelism than dynamic programming algorithms.

Greedy algorithms are another related category of algorithms, where instead of considering all paths through the state space, as in dynamic programming, we consider only a single path. It is termed greedy because at each step where a choice is made over the decomposition of the input, a greedy algorithm selects a locally optimal choice. Local in this context means we decided what to do next based only on what has come before; we do not lookahead by exploring various different paths, as we do in dynamic programming. We hope that making such locally optimal choices at each step will lead to a globally optimal solution; however, this is only guaranteed when the greedy choice property holds, that is a locally optimal step is part of the optimal solution. [16] Certain greedy algorithms can be considered as a refinement of an equivalent dynamic programming algorithm. [5]

3.3 Implementation techniques

Implementation of a dynamic programming algorithm will tend to follow a set of steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information. [16]

The technique described above is *tabulation*, as referred to in the programming of *dynamic programming*. The values computed from the bottom up are stored in a table, and used in computing subsequent values. When all required values are computed, we read the result from the target cell.

In many optimisation problems, the resulting value may not be the only piece of information required - we may also require the *path* through the table to that destination. This can be solved using a top-down *traceback* across the table, where we inspect the final cell to determine the sub-problems that contributed to the final result. We then do the same for each sub-problem in turn, building up a picture of how we computed the final value. Such a scheme is used in the *Viterbi* algorithm[63]. This information could also be stored as we compute each cell, at the cost of storage for a larger data-structure.

The bottom-up tabulation method assumes that all smaller problems than the solution will need to be solved in order to compute the solution. In cases where this does not hold, we will be evaluating a number of sub-problems that are never required. *Memoization* is a solution to this problem - storing sub-problems as they are computed - like a *memo*. In memoization we compute the values top-down, as in the natural recursive solution, but instead of recomputing sub-problems, we store and re-use the result. In practice, we might still use a table to store the values. At each and every call, we would then check whether there was an entry in the table, and compute one if not. In cases where we compute all values, the bookkeeping overhead for memoization tends to allow tabulation to outperform memoization. Where we do not need all sub-problems, the tables are turned, and memoization can outperform tabulation.

An additional problem with tabulation is that it assumes we know the bounds of the sub-problems and therefore the dimensions of the table. We will also need a method for ordering computations the table that considers the sub-problem relation. With Memoization, we can use a form of hashing on the parameters to store values so that we do not need to know before hand which values are to be computed.

3.3.1 Tabulation techniques

For the purposes of this section, tabulation refers to the technique of building a data-structure for storing and retrieving values representing sub-problems. The traditional mechanism for tabulation in the context of dynamic programming is to build a table with different dimensions representing each parameter. Typically, this table is then built from the bottom up by filling the cells of the table in some order that ensures dependencies are resolved. The bottom-up tabulation technique can be declared more formally by describing an inversion of the order of evaluation on the recursive functions[7].

Tables are not the only form of tabulation we can consider. Tabulation techniques for memoization can be described by the following two criteria:

1. storing, at each stage, those previously computed values for f which are needed to continue evaluation of $f(x_0)$;
2. determining for a given argument x whether or not the value $f(x)$ has previously been computed[4].

The implication of criteria 1 is that we *only* need to store those values which will be required in furthering the evaluation of the function. Where the table technique can be wasteful of both space and time, evaluating unused values and storing data beyond the shelf life, memoization can make use of time and space efficiencies implicit in the dependency graph.

In practice, memoization is often implemented using a sparse data-structure, such as a table, when space is not a concern, and some form of hashed data structure when it is. In the latter case, a hash of the sub-problem acts as a key for indexing and retrieving values[16].

[15] describes the elimination of redundant recursive calls through program transformations based on function schemas of the following form:

$$f(x) = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } b(x, f(c_1(x)), ..., f(c_n(x))) \tag{3.1}$$

Where the functions c_1 to c_n are described as *descent functions*, and used in *descent conditions* under which the program can be transformed. Four transformations are given as part of the paper. These transformations work to generate an equivalent definition where storage is reused by storing only those values required for following computations. The resulting program implements something akin to memoization, where limited storage is used to store only those values required in a top-down evaluation. The solutions can be transformed to an iterative approach through a straight-forward transformation that produces a bottom-up solution, much like tabulation but without the redundant storage. Similar techniques are described in e.g [10].

[42] builds on this work to describe a series of static program analyses and transformations that can adapt a straightforward recursive function to cache the results of sub-problems. Once again this displays a hybrid approach to tabulation, where cached results are only stored for as long as required. A 3 step process takes place to instrument the program to cache the values, to introduce static incrementalization, where function calls are extended to include parameters representing cached values and finally to prune all unneeded values.

3.4 Applications

Applications of the dynamic programming technique are vast and varied - from items of mathematical interest to those of practical interest. Simple examples include efficient computation of the Fibonacci Sequence[54] and the Towers of Hanoi problem[60]. Matrix chain multiplication is another commonly cited example[16, 3]. Many shortest path algorithms can also be considered as dynamic programming algorithms, including Dijkstra's shortest path algorithm[59] and the Bellman-Ford algorithm.

Another common example are string algorithms including longest common subsequence and substring, longest increasing subsequence and edit distance algorithms such as the Levenshtein distance[54]. These string algorithms are widely used in adapted form through bioinformatics for sequence alignment, including the Needleman-Wunsch[50] and the Smith-Waterman[57] algorithms.

Bioinformatics is a perfect example of the utility of dynamic programming algorithms. The most widely used textbook in biosequence analysis[22] has been counted[25] to list 11 applications of dynamic programming in the introductory chapter. In a later chapter, the book describes the statistical framework of Hidden Markov Models, in which dynamic programming algorithms play a leading role. The Viterbi[63], Forward and Backward algorithms all use dynamic programming. HMM algorithms have wide uses, from speech, gesture and handwriting recognition to convolution codes for cellular networks, modems, satellites, wireless LANs and other communication networks.

Bioinformatics also use a variety of HMMs to represent biological features, with the Viterbi algorithm heavily used in *profile* models[22, 23] amongst others.

Dynamic programming is useful in other language parsing frameworks. The CYK and Earley algorithms use dynamic programming to parse languages described by Context Free Grammars. CFGs may be extended with statistical features to create Stochastic Context Free Grammars, using the similar algorithms for parsing. SCFGs may be used to perform RNA secondary structure prediction.

This section is by no means exhaustive; the beauty of the dynamic programming method is the wide-spread applicability. Given a decision problem made of optimal sub-problems, of the kind found in many practical problems, chances are dynamic programming will be suitable. Examples of practical problems could be partition problems[54], optimal routines for factory or scheduling work[16] and minimum weight, distance or height problems[54].

3.5 Models of Dynamic Programming

3.5.1 Sequential Decision Processes

One of the early models of dynamic programming was to describe optimisation problems as *discrete decision processes* extended with a cost structure to form *sequential decision processes*[35]. A solution may then be found by testing the principle of optimality against this model, and then deriving the recursive equation to compute the solution from the SDP. [48, 49] have worked to formalise the Principle of Optimality as a monotonicity condition. [58] discusses the question, and determination, of the validity of the Principle of Optimality, in process of which they also consider the validity of monotonicity conditions. The derivation of simplified algorithms for discrete decision problems from a general algorithm is described in [8].

3.5.2 Non-serial Dynamic Programming

Later work generalised from sequential or serial models based on lists or strings to a more generalised tree structures to solve non-sequential problems[31, 30]. Such work has focused on the separation of the enumeration of the structure from the evaluation of the cost of that structure, allowing different computations using the same enumeration of the structure. Such a pattern has been described as an *exhaustive search*, allowing as it does for generating all candidate solutions to determine the optimal solution. This model has been placed in a categorical setting, a technique that can result in a neater formulation. [3, 18] Similar work has been completed by [56, 55] using problem reduction theory. [17] also describes modelling dynamic programming problems and solutions using relational calculus.

3.5.3 Algebraic Dynamic Programming

Algebraic dynamic programming[25] is a technique that also considers the separation of structure from the evaluation of the structure. However, it chooses to describe the search space as a *yield grammar* - a tree grammar generating strings - and an *evaluation algebra* to describe the evaluation. Under models with separate methods for enumerating the search space, we may consider the evaluation function as choosing between *previously enumerated* paths through the state space. Implementations transparently interleave the two methods to achieve the efficiency of traditional implementations.

3.6 Parallelisation on Massively Parallel Architectures

The majority of work completed in this area has focused on porting specific applications to the architecture (see Section 4.1.3 for examples), with very little focus on generic techniques. A recent paper has developed Algebraic Dynamic Programming to target GPUs as a back-end[**?**]. It uses a diagonal technique to evaluate the dynamic programming table; this works because each dimension of the table represents a piece of sequence data, and ADP grammars only allow results from shorter substrings.

3.7 Conclusion

Dynamic programming is a flexible, simple and widely used tool for optimisation problems of all shapes and sizes. Dynamic programming can be modelled and understood in many different ways; this flexibility is a key advantage helping map a vast number of problems to a dynamic programming solution. Many different techniques exist for implementing dynamic program, from tabulation to memoization and everything in between; support can thus be develop flexibly depending on exact form of the problem.

Chapter 4

A Case Study: Bioinformatics

Perhaps somewhat surprisingly, Biology has gained a distinct computational focus over the last two or three decades. By considering DNA and proteins as a sequence of characters, we may perform statistical analysis to determine features such as familial similarity, alignment and the creation of trees of relations or phylogenies. These techniques have been driven by a rapid expansion of sequence data available for a wide-variety of organisms. Whilst computational analysis can never supplant the development of theories through empirical results, in can work to focus experimental efforts on important or significant areas.

The growth in the 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 it simply isn't possible to analyse through careful empirical study. Where the growth in sequence numbers is out stretching our growth in CPU performance for floating point operations, the resulting gap is problematic for researchers in this area. Massively Parallel Processors are thus becoming a topic of great interest for Bioinformatics researchers as a way of accelerating the floating-point operations using hardware that is readily available, cheap and scalable. In particular, the ability to run the same code on the desktop or notebook as a high-performance cluster is invaluable.

Almost all algorithms in Bioinformatics are based upon choosing the most statistically likely way of combining some data, whether that be building phylogentic trees or aligning sequences. These are classic optimisation problems; most make use of dynamic programming to acheive respectable performance.[22]

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 applications the GPU or other massively parallel processors.

4.1 Pairwise Sequence Alignment

One of the earliest computational requirements in Bioinformatics was to compare two sequences to determine how related they are. This is performed by *aligning* parts of the sequences that are statistically similar, termed a *pairwise sequence alignment.*

Figure 4.1 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.

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

Figure 4.1: A sequence alignment between a fragment of human alpha globin and human beta globin.

4.1.1 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[50], with a more efficient version described by Gotoh[26].

The algorithm is normal described by a recursive functional equation:

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

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. 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.

4.1.2 Smith-Waterman

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

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

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 alignment between two substrings we must find the maximum value of $F(i,j)$ for all lengths of substring i and j.

By necessity, both pairwise-alignment techniques described here use dynamic programming to ensure the end-result is feasible to compute. The implementation typically comes in the form of a two dimensional table, where the co-ordinates in the table represent i and j.

4.1.3 Massively Parallel Implementations

An early piece of work[39] 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.

This technique has been described by the developers of CUDA-SW++ as *intra-task* parallelism, where multiple threads work co-operatively to compute a single table.[40] The alternative is *inter-task* parallelism, 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. CUDA-SW++ takes a hybrid approach; shorter sequences are computed using inter-task parallelism, and longer with intra-task. We discuss these concepts in the next chapter (Section 5.4) in terms of sequence-per-thread and sequence-per-block in the context of similar algorithms in Hidden Markov Models. A follow on[41] proposed improvements to the technique using a vectorised approach.

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 both efficient and easier to develop when compared to other solutions[14]. [45] corroborated this result through the development of Smith-Waterman algorithm for GPUs, and subsequently comparing against the BLAST and SSEARCH, finding performance improvements of 2 to 30 times faster. Both [45] and [40] found that ordering the sequences by length improved inter-parallelism performance.

4.2 Hidden Markov Models

Hidden Markov Models are a form of *generative* statistical model, where a set of *hidden* states determine the output sequence. It can be used to determine various probabilities associated with that sequence. It has 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?".

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.

4.2.1 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 can all make use of *dynamic programming* to implement a recursive function. 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 practise, we tend compute all entries in the table, in an appropriate order, before reading the result from the desired row and column.

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.

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 arrival at that state, emitting the given sequence along the way.

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

In a model without loops of non-emitting "silent" states, 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 precisely, computing the effective transition probabilities between emitting states[22]. This can, however, lead to a rapid increase in the complexity of the model; it is usually easier to produce models without loops of silent states.

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.

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 the 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* - in particular we wish to compute $P(q,i)$, the probability that state q emitted $s[i]$. This is computed in an iterative fashion, setting the parameters to an initial value, and continually applying the formulae until we reach some prior criteria for termination.

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:

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

4.2.2 Example applications

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.[22]

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 a 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*[22]. A profile models the likelihood of a given sequence corresponding to the described family, by performing an alignment to that profile.

We developed a prototype implementation of a profile model using CUDA. This informed the decisions we made in the construction of HMMingbird; a detailed description of the techniques we used is given in the next chapter, along with the results achieved by these methods. We achieved similar performance using HMMingbird - up to x25 faster.

Chapter 5

HMMingbird: A compiler for Hidden Markov Models

Practical implementation of the key algorithms described in the previous chapter are time-consuming and monotonous. Ideally, such boiler-plate code would be generated by high-level descriptions of the problem domain. Prior work, such as HMMoC[43], has focused primarily around the use of *code generators* to speed development time for technically able users by generating a large amount of the boiler-plate code required. Whilst this approach is a flexible and practical tool for the few users with the appropriate skills - and can provide an efficient CPU implementation - the potential audience is much reduced.

HMM applications often require significant computing time on standard desktop hardware. Applications such as gene-finding or profiling require evaluating the HMM across tens of thousands to millions of different sequences, often taking in the order of minutes to hours. In addition, the quantity of data over which we wish to evaluate is ever growing, resulting in an insatiable need for improved performance from HMM tools that naturally leads to investigation of alternative desktop hardware and software configurations.

The HMM algorithms are often time-consuming to perform for the typical applications in Bioinformatics, such as gene finding or profile evaluation, where the size of the input set numbers in the hundreds of thousands to millions and can take minutes or hours to perform. It is therefore vital that we use the existing hardware as effectively as possible. One advantage of developing a compiler, or a code-generator, is the ability to target complex architectures with efficient implementations that would be tricky and time-consuming to develop by hand. The high-level of data-parallelism combined with the ability to parallelise optimisation problems based on decisions problems promotes massively parallel processors as a tempting target architecture. In particular, the scalability of the system matches the requirements of researchers in this area. They are ideal for large scale scientific computing of this sort, and an excellent candidate architecture for improving performance in HMM applications.

We would like to combine the practical convenience of code generators with a rigorous approach to language design to develop a domain specific language for Hidden Markov Models, that allows us to target massively parallel processors, in the form of general purpose graphics cards. We strongly feel that high level problem descriptions are an ideal way of expanding access to the power of the desktop MPP to a wider audience. In this chapter, we describe the development of such a system for HMM's.

- We describe the syntax of a formal description language for Hidden Markov Models (Section 5.3), exploring the language through a series of examples. As well as providing a concise and elegant system for describing the models themselves, we develop a procedural language for describing which algorithms are to be performed, and in which order. This supports a high-level of customisation without requiring the modification of the generated source code, unlike existing tools.

- The common algorithms are all dynamic programming algorithms with certain properties. We describe those properties, use the common elements to develop a basic parallel framework for dynamic programming algorithms, and subsequently use that to implement the Forward, Backward and Viterbi algorithms.

- A naive implementation of the standard algorithms often leads to poor runtime characteristics. We therefore describe a series of algorithm and micro optimisations including improvements based on the division of labour across the GPU, memory storage location choices, reduction of divergence and reduction of limited runtime resources (registers, shared memory etc.).

- We judge the success of our approach by comparing the execution time to benchmarks for two applications. For each application, we shall also describe a reference implementation in the definition language, as an informal way of judging the method of model description. The applications are:
  - The simple but widely known *occasionally dishonest casino* example, described in Section 5.3.1, benchmarking against HMMoC.
  - A *profile* HMM, described in Section 5.3.2, benchmarking against HMMER, HMMoC and GPU-HMMER.

5.1 Hidden Markov Model Design

5.1.1 Model Equivalence

Hidden Markov Models are often described as *probabilistic automata*, and as such we can often bring many of our intuitions regarding pure *finite automata* to bear against them. One particular area of interest to us is the isomorphism of Hidden Markov Models - when do two HMMs represent the same underlying process? This is not only a theoretical question, but a practical one as well - by defining a range of equivalent models we may find an isomorphic definition that better suits implementation on a given GPU. This may take the form of a systematic removal from the model that simplifies computation. One such concrete example is that of *silent states* - those states which do not emit any character in the alphabet when visited - elimination of which can reduce difficult immediate dependencies.

The *identifiability problem* encompasses this question:

> Two HMMs H_1 and H_2 are *identifiable* if their associated random processes p_1 and p_2 are equivalent.

It was first solved in 1992 by[28] with an exponential time algorithm, with a later paper describing a linear time algorithm[24].

5.1.2 Emitting states and emitting transitions

Our description of the model allows each state to emit a character from the alphabet. However, there exists an equivalent definition that supports emission on characters on *transitions* rather than *states*. Informally, we can explain the equivalence by noting that any state-emission model can be trivially converted to a transition-emission model by setting the emission value on all input transitions to the emission on the state. Transition-emission models can be converted by taking each emitting transition and converting it to a transition, followed by an emitting state, followed by a single silent transition to the original end state.

Note that it is possible to mix and match transition and state emissions, as long as the two do not contradict - that is, that a transition to a state and the state itself do not both declare an emission. This enforces the condition that we can only emit at most a single symbol each step. Given a HMM, we may find advantage in describing the model in one form or another - or a mixture of both. For the user, we can reduce repetitions in the definition by allowing states with single emission values to declare them on the state itself, rather than declaring the same emission on each transition. In addition, emission on transitions can allow a reduction in states when each transition requires a different emission set, that would otherwise have to be modelled as separate states.

5.1.3 Silent States

Any state that does not declare an emission, either on any incoming transitions or on the state itself, is defined as a *silent state*. This not only provides a notational convenience, grouping common transitions, it can also reduce the number of transitions within the model. As we can see from the algorithms in Section 4.2.1, the number of transitions is the primary factor in determining the run-time of the algorithm.

A classic example[22] describes a lengthy chain of states, where each state needs to be connected to all subsequent states. We will later describe the model (Section 5.3.2) that has exactly this pattern. We may replace the large number of transitions with a parallel chain of silent states, in which each emitting state links to the next silent state in the chain, and each silent state links to the current emitting state. In this way, we can traverse from any emitting state to any other emitting state through a chain of silent states.

It is important to note that this process can change the result of the computation; it may not be possible to set the parameters in such a way that each independent transition value between states is replicated in the series of transitions through silent states. For some models this may not influence the result unduly - large models often require unrealistically large data-sets to estimate independent transition parameters.

In a sequential computation reducing the number of transitions directly affects the run-time. In a parallel computation, other factors come into play. Silent states may introduce long chains of dependent computations; in a model where all states emit, the likelihood of each state emitting character x at position p depends only on those values from position *p - 1.* When we have silent states, we may have a chain of states that do not emit between emitting states. This chain must be computed sequentially, thus reducing a potential dimension for parallelism, as described in Section 5.4.1.

We note that a silent state may be eliminated in the same way that it is introduced, through careful combination of transition parameters - each transition to the silent state is combined with each transition from the silent state. By eliminating such dependencies, we ensure that we can parallelise over each column in the dynamic programming table.

```
Given a model $M$ with a set of transitions $t$ and states $s$
  for each silent state $s_x$ do
    for each input transition $t_i$ do
      for each output transition $t_k$ do
        $t_n \leftarrow t_i t_k$
        Add $t_n$ to t
      end for
    end for
    Delete $s_x$ and all transitions to/from $s_x$
  end for
```

Figure 5.1: Algorithm for silent state elimination

Elimination comes at a cost - for each silent state with n input transitions and m output transitions, we will now have nm transitions. This is obvious when we consider that the reason for introducing silent states was a reduction in transitions. It may, therefore, be best to balance the elimination of silent states with the number of transitions as required to optimise any parallel HMM algorithm.

5.2 Prior Work

As a basis for the system, we use the code produced by HMMoC[43], a tool for generating optimised C++ code for evaluating HMM algorithms, written by Gerton Lunter. HMMingbird is considered as the first step to rewriting HMMoC to utilise the power of desktop GPGPU's. HMMoC is based upon a HMM file exchange format that uses XML; the format allows a description of a HMM, and describes which algorithms to generate. It features a generic macro system and the ability to include arbitrary C. It generates an optimised set of C classes that can be included by the user in there own projects, providing an efficient HMM implementation.

Other prior work in the area includes BioJava[32], a Java library for Bioinformatics that includes facilities for creating and executing Hidden Markov Models. Tools for specific HMM problem domains have also been developed. So called *profile* HMM's are one of the main uses within bioinformatics. One of the most popular tools in this area is HMMER[23].

There are a number of notable elements of prior work to port such HMM applications to GPU architectures. ClawHMMER[34] was an early attempt to port HMMER to streaming architectures uses the now defunct BrookGPU[9] programming language. More recent attempts have include GPU-HMMER[64], which instead uses the more modern NVIDIA CUDA Framework. CuHMM[38] is an implementation of the three HMM algorithms for GPUs, again using CUDA. It uses a very simple file interchange format, and does not provide native support for HMM's with silent states - although the latter restriction can be mitigated through silent state elimination.

5.3 Language Design

A primary aim of the software tool is to provide the benefits of GPGPU's to a wider audience. As a result, the design of the language is vital part of the problem: it must be concise, simple and easy to understand, and prove itself to be flexible enough to support new formulations as and when they appear. No such general language has been attempted before, to the best knowledge of the author.

A primary design decision was to assume that the user will not, in any reasonable course of action, be required to modify the generated code, as in HMMoC. Taking such an approach when faced with generating code for a GPGPU would reduce the potential audience to such a small group. Instead we take the view that a *high-level specification language* is the tool of choice, allowing a wide-variety of users, with a wide-variety of technical expertise, to access the full power of the GPGPU.

In the following sections we will describe the language through a series of examples, gradually exploring different areas of the domain.

5.3.1 Occasionally Dishonest Casino

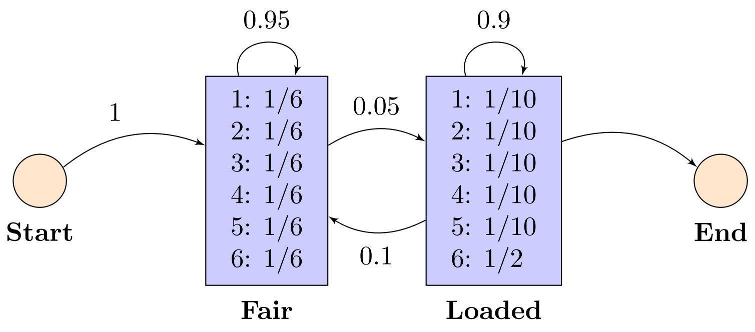

Figure 5.2: The Hidden Markov Model representing the Occasionally Dishonest Casino

We will first examine a commonly used example - that of the *Occasionally Dishonest Casino*. The idea here is that we have a casino that will usually use a regular die, but occasionally they may switch to using a *loaded die* for a few turns in order to gain an advantage. A loaded die is one that has been fixed to land on one more faces with a higher probability than normal. A HMM is an ideal model for this sort of system - we have an output sequence that is generated by a hidden state - in this case, which die has been chosen. We thus provide two states, one fair and one loaded. Each state has a set of emission probabilities - the fair is even and the loaded die shows a skew to certain output characters. Figure 5.2 is the diagram of the model.

```
hmm casino {
    alphabet [1, 2, 3, 4, 5, 6];
    startstate start;
    state fair emits fairemission;
    state loaded emits loadedemission;
    endstate end;
    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];
    start -> fair 1;
    fair -> fair 0.95;
    fair -> loaded 0.05;
    loaded -> loaded 0.97999;
    loaded -> fair 0.02;
    loaded -> end 0.00001;
    fair -> end 0.00001;
}
```

Figure 5.3: The HMMingbird code for the casino example.

Figure 5.3 provides a code example for the definition of a casino HMM. We begin with a definition of the `hmm` element. We can name our HMM, both as an aid to documentation and for identification purposes. In this case, we have given it the name *casino*. Legitimate names may only start with letters, and contain letters or numbers. Any further description of this model takes place in the form of a series of *declarations* within the curly brackets `{}` of the `hmm` element. The declarations take place in no particular order. A single program may contain many such `hmm` definitions.

The first important declaration is that of the `alphabet`. This defines the set of valid output characters, separated by commas, and delimited by squares brackets `[]`. Each element is limited to a single ASCII character in the current version.

We define the set of states using the *state* element. We can, and must, specify a single start and end state, using the appropriate `startstate` and `endstate` keywords. We provide every state with a unique name, that can then be used to define transitions between states. Each state element may define an *emission* value using the `emits` keyword, that refers to a separate declaration describing the probability distribution over the alphabet. States may also remain *silent* by omitting the `emits` keyword. In this case, we define two regular states - *fair* and *loaded* - with appropriate emission values - *fairemission* and *loadedemission*.

The emission values are simply defined by a list of values corresponding to the order of the alphabet. Each value is the probability of emission of the character at that position in the alphabet definition. In the *fairemission* declaration, all outputs are equally likely, and so are set at 1/6. In the *loadedemission* declaration, the `6` has a higher probability, and so the 6th element is set higher. It is desirable for a set of emissions to create a distribution - in other words, they should sum to 1, as they do for both these cases. However, we do not enforce this restriction for there may be some cases where a distribution may not be desirable.

Next we describe the structure of the graph by defining the transitions from state to state, with each including an associated probability that determines how likely the transition is. Just like the emission declaration, it is usually the case that we want to define a distribution over the transitions from a particular state. If all nodes provide a distribution over both the emissions and outward transitions, then we have a distribution over the model. This allows fair comparison between probabilities determined for different sequences. However, as with the emissions, we do not enforce this rule, allowing a wide range of models to make practical use of the GPGPU.

To produce a completed program we still need to provide a main definition that describes which algorithms should be run, and on which HMM definitions. We do this with the `main` keyword. Like the `hmm` keyword, further declarations appear within the curly brackets.

```
main() {
        cas = new casino()
        dptable = forward(cas);
        print dptable.score;
}
```

The basic declaration here is an assignment; the right hand part must be an expression, the left a new identifier. The first statement defines an instance of the `casino` hmm we described above. This syntax will, in future, be extended to allow parameterisation of the model.

The next step is to perform the algorithm itself. This takes place as a *call* expression, which uses the name of the algorithm to run, and passes the HMM as a parameter. In this case, we run the `viterbi` algorithm with the HMM `cas`. We currently support the `viterbi`, `forward` and `backward` algorithms. The parameter to the call is simply an expression, so we may declare the casino inline like so:

```
viterbi(new casino());
```

The result of such a call is returned as the value of that expression. In this case, we have used the variable `dptable`. This allows access to the entirety of the dynamic programming table.

Once we have run the algorithm, we need some way to return the results to the user. The `print` statement acts as this mechanism, providing a limited system for returning the values in the dynamic programming table. Each algorithm returns a dynamic programming table with appropriate `attributes` that we may access. In this case, the only attribute of the forward algorithm is the score. We can refer to the attribute using a dot notation, as in `dptable.score`.

5.3.2 Profile Hidden Markov Model

A *profile hidden markov model* is a type of HMM that describes a a family of DNA or protein sequences otherwise known as a *profile*[22]. It models the likelihood of a given sequence corresponding to the family described by the profile, by performing an alignment. An alignment describes the modifications required to change one sequence to another, which may include inserting characters, deleting characters or matching characters.

This system can be described by a HMM of a certain form. We provide a set of states that represent a single position in the model, and we can repeat these states any number of times to represent the likelihood of certain symbols appearing at that position in the sequence. For each position in the model, we may match or delete that position. Before and after each position, we may introduce characters using the insertion state. Typically, the model is “trained” using a parameter estimation technique such a Baum-Welch, setting each transition and emission parameter as appropriate.

HMMER is a popular tool for creating and using profile HMM’s. It uses a custom file format to describe a parameterised model. We have created a small script that converts the file format to our domain specific language. We will examine some excerpts here.

As before we declare our model with an appropriate name - we simply use `profile` here.

```
hmm profile {
```

Since we are using protein sequences, this example has the alphabet set to the sequence of valid protein characters.

```
    alphabet [A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y];
```

As in the casino example, we declare a `start` and `end` state. We can also see declarations for each position - with each position containing a insert, delete and match state. Note that the deletion state does not declare the emits keyword, indicating it is a silent state. It effectively allows the matching process to skip a position.

```
    state D001;
    state I001 emits emit_I001;
    state M001 emits emit_M001;
```

We follow that up with the declaration of the emission values for each insert and match state. Although in this case each state has an individual emission value, in other cases we may share emission values between states.

```
    emission emit_I001 = [0.0681164870327,0.0120080678422,..values ommitted..,0.0269260766547];
    emission emit_M001 = [0.0632421444966,0.00530669776362,..values omitted..,0139270170603];
```

We then describe the set of transitions. We show the transitions for the position 1, but each subsequent position has a similar set of transitions.

```
    M001            -> M002           0.985432203952;
    M001            -> I001           0.00980564688777;
    I001            -> M002           0.538221669086;
    I001            -> I001           0.461778330914;
    D001            -> M002           0.615208745166;
    M001            -> D002           0.00476214916036;
    D001            -> D002           0.384791254834;
    beginprofile    -> M001           0.750019494643;
    beginprofile    -> D001           0.249995126434;
```

One feature used by the profile HMM that has not been previously mentioned is the ability for transitions to include emission values (Section 5.1.2). We allow this in addition to emissions on states, as there are cases where the most compact representation will be one notation or the other. They are computationally identical; simply a notational device. In this case, we emit a character as we transition, and not when arriving at the destination state. We do not allow both a transition and the destination to emit a character - only one or the other or neither may emit, as enforced by the compiler. We can then ensure that only a single symbol is emitted at each step through the model.

```
    nterminal       -> nterminalemitter 0.997150581409 emits emit_nterminal_to_nterminalemitter;
```

In the main definition, we perform the Viterbi algorithm to produce the score of the most likely path, and print that score using the dot notation as above. However, unlike the casino example, by using the Viterbi algorithm, we have access to another attribute: traceback. This returns the list of states visited on the most likely hidden state path. By omitting to access this attribute we cause it not to be evaluated as part of the Viterbi algorithm, thus increasing efficiency and reducing memory usage.

```
main() {
dptable = viterbi(new profile());
print dptable.score;
}
```

By using the simple dot notation we produce just the result - either the final score or the traceback of the optimum path. But what if we wish to investigate intermediate entries or observe sub-optimal paths? In these cases we can print the entire dynamic programming table by placing an `&` at the end of the attribute name like so `dptable.score&`. This prints all intermediate results as well as the final score. Note that due to optimisations within the GPU code, we do not normally store the entirety of the table. Doing so will reduce the number of sequences that can be processed at any one time, due to the increase in memory requirements.

5.3.3 Limitations

The definition language described here is somewhat immature at present. We make the following limiting assumptions :

- Only a single tape is to be used - multiple tapes are an expected extension. Each tape would contain an alphabet, and emissions would be linked to particular alphabets;
- Single characters for emission values - in some cases we may require multi-character values;
- Only simple numbers and fractions are allowed in transition and emission probabilities;

5.4 Dynamic Programming algorithms for parallel computation

The algorithms described in Section 4.2.1 can all either be implemented with, or make use of, *dynamic programming*, a form of optimised computation where we store and reuse intermediate results in order to avoid expensive re-computation. This is possible because the problems exhibit the two key properties required - *optimal substructures* and *overlapping sub-problems*. Computing the probability for a given state relies on computing the value for a set of previous states with a shortened input - with the result being optimal if all the sub problems are optimal. In addition, the value for the previous states may be reused for other computations at this position in the input.

There are a number of ways we may implement such a system. One popular method is *memoization*, where the result of each function call is stored against the parameters for that call. Further calls with the same parameters make use of the same result. Whilst this approach can be fruitful within a single thread, it is both difficult to implement and potentially sub-optimal in a multi-threaded system. In particular, any system that took this approach would need to ensure that threads had the requisite cross communication and appropriate locking to ensure duplicate computations of a function call did not take place. This is a particularly difficult task within the GPU environment, since it neither provides locking facilities nor does it perform optimally when used in this way. It once again highlights the difference between multi-threading algorithms for CPUs compared to GPUs.

Fortunately, we have another way of implementing a dynamic programming algorithm that is much more suited to GPUs. A *lookup table* is a common way of mapping the parameters of a dynamic programming algorithm to the result of the argument. For most algorithms, the results of all cells in this table will be required at one stage or another and so we may instead compute the values for this entire table before attempting to compute the final result, so longs as we ensure any pre-requisites for a given cell are evaluated before the cell itself.

Figure 5.4 gives a small portion of a dynamic programming table for the *casino* example described in Section 5.3.1.

	Initial	**'3'**	**'1'**	**'6'**	**'6'**	**...**
Start	1	0	0	0	0	**...**
Fair	0	1/6	2.639×10^{-2}	4.178×10^{-3}	6.616×10^{-4}	**...**
Loaded	0	0	1.389×10^{-3}	6.805×10^{-4}	3.334×10^{-4}	**...**
End	0	0	0	0	0	**...**

Figure 5.4: An example of a dynamic programming table for the viterbi trace in the *casino* example

In a traditional, serial, implementation we would compute each value in this table in turn, ensuring all dependencies were met before computing a cell. In a parallel implementation, we have a number of options:

Sequence-per-thread We may keep with the same serial algorithm, but compute many such tables at once, relying on a large input set to keep the device busy. In this case we would allocate a single thread to each sequence.

Sequence-per-block Alternatively, we can use a number of co-operative threads to compute the values in a single table. Co-operation between threads is required so as to ensure prior dependencies are met before computation of the current cell can continue.

Sequence-over-MP Finally, we may use a number of co-operative multiprocessors. Such co-operation is currently difficult to achieve on NVIDIA GPUs. In addition, larger models are required to fully utilise many different multiprocessors.

Option 1 is suitable for a large number of runs over smaller models. However, if there are a small number of sequences relative to the number of cores, it is hard to fully utilise the device under this system. In addition, it cannot necessarily make best use of the caching and localised memory features of the GPU, since each thread is working independently.

Option 2 is suitable where we can find some way to sensibly divide the work between the different threads on the GPU. This will rely on a way of identifying dependencies between states to ensure we can parallelise across the states.

Option 3 is trickier on the current hardware. The easiest way to implement cross block communication on current generation GPUs is to run a separate kernel for each independent set of events.

For our initial approach we will use Option 2, which provides most opportunity for utilising the caching aspects of the multiprocessor without requiring expensive to-ing and fro-ing from the CPU to the GPU.

5.4.1 Silent State Elimination

One key problem for parallelisation is the dependencies between states. If we imagine the dynamic programming table for these algorithms, with the each row marking a state, and each column a position in the output sequence, we can see that the value at a given state and column may depend on values in the previous column; marking a transition to the current state from the state in the prior column. We also note that any silent states - those that do not emit any values - may be referred to by other states in the *same* column, since no emission has taken place. In the standard dynamic programming implementation of these algorithms, we would simply ensure all dependencies were met before evaluating a state. However, in a massively parallel system this is neither practical or desirable.

The solution is to eliminate the silent states, as described in 5.1.3. As it stands, we simply eliminate all silent states. However, the cost may be mitigated by selectively eliminating silent states. In particular, long paths of silent states are inefficient in terms of distributing work on a parallel machine, since the length is a lower bound on the number of iterations required, and so we may choose to eliminate silent states so as to reduce such paths. The same algorithm can be modified to perform such a task, since we remove silent states iteratively.

5.4.2 Dynamic Programming: Implementation

It is relatively simple to implement the first approach - *thread-per-sequence* - as described in Section 5.4, by taking a straight-forward serial implementation to run on each individual thread. Nevertheless, by using the elimination of silent states we can expose an extra dimension of parallelism along the states that will allow a *block-per-sequence* approach, which should facilitate an improved use of the device.

Our first implementation of the block-per-sequence algorithm provides a thread for each state in the model. We thus compute each column in parallel, synchronising after each one. Far more complicated distributions of work are possible - for example, distributions by transition to different threads - however, for simplicity we will ignore these extensions for our initial implementation.

A Shared Template Approach

The primary focus for our investigations is the ability of graphics cards to provide a useful speed up for Hidden Markov Model applications. For the majority of applications the fundamental role which requires acceleration is that of applying the models to a set of given inputs, rather than training the model parameters to begin with. The reasons for this are twofold - we usually train the model once and once only, and against a smaller set of inputs than those that we wish to run it over for the actual application. As such, we will focus for the remaining section of the paper on the three estimation algorithms: forward, backward and viterbi. We also note that the forward and backward algorithms play an important role in the Baum-Welch parameter estimation algorithm, and so the implementation of the former is a pre-requisite for implementing the latter.

At this stage we can make a useful observation about the nature of the three algorithms - that they all have a similar structure, and differ only on minor points. This is unsurprising, since each can be, and usually are, implemented using the dynamic programming lookup approach described in Section 5.4. The forward and viterbi algorithms are evidently similar - they traverse the model in an identical way, accessing the same cells at the same time. The differences lie in how the results from prior cells are combined - with the viterbi algorithm using a `max` operation, whilst the forward algorithm `sums` over the prior cells.

On the other hand, the forward and backward algorithms differ only in the way they traverse the model, not the way they combine the results. One way of handling the differences in traversal is to provide a modified version of the model which describes a backward traversal. Such a model is simple to produce within the compiler as a form of pre-processor. In this case, we can simply use an identical code base to implement the backward algorithm. One caveat to this approach is the potential cost of duplicating of state and transition data on the device if we require both the forward and backward algorithms. This cost may be mitigated by combining some invariant aspects of the model, such as state and emission data, and providing a new version of the transition data.

Using these observations, we can develop a template structure for the three algorithms, that may be parameterised to support any of them. Figure 5.5 gives a pseudocode description of the standard template. A by product of this design decision is the ability to generate other algorithms with a similar pattern.

```
{$s$ represents the dynamic programming table, indexed by state and position in sequence}
s = []
for character $c$ in sequence at position $i$ do
  parallel for state in states do
    $s_{state,i} = 0$
    for each input transition $t_j$ with emission probability $e_j$ do
      $s_{state,i} = s_{state,i}$ op $t_j e_{j,c} s_{startstate_{t_j},i-1}$
    end for
  end for
end for
```

Figure 5.5: The basic shared template

5.5 Optimising Techniques

The basic template provides a suitable base from which we can start to apply specific optimisations.

5.5.1 Host to device transfers

A primary issue for CUDA applications is optimising the *computational intensity*; that is, the ratio of IO to computations. On the authors hardware, memory bandwidth from host to device is pegged at 1600MB/s, whereas device memory access occur at around 100GB/s, and operational speeds reach 900GFLOP/s for the GT200 architecture. As a result, both memory allocation and host to device memory copy functions are comparatively expensive, particularly in terms of setup costs. To reduce such costs we take a brute force approach to memory transfer. We first load the entire sequence file into main memory, then prepare some meta-data about each sequence. The final step is then to copy both arrays en-mass to the host memory.

Another option is to use *page-locked memory allocation*, where the host data is never swapped out of main memory. This technique is quoted as providing twice the host to device memory bandwidth. However, page-locked memory is a limited resource, and over excessive use may cause slow-down.

5.5.2 Precision versus speed

A typical problem for applications of this sort is accuracy. The probabilities rapidly become vanishingly small, often under-flowing the standard floating point data-types, even those of double width. One common approach is to convert all operations to *logspace* by computing the log of all the input values, and converting each operation (max, plus etc.) to the logspace equivalent. We can then represent the values in each cell as either float values, or double values for slightly increased precision.

One important aspect of this debate is the structure of the current hardware - with a 8:1 ratio for single precision to double precision instruction throughput, single precision operations can be significantly faster for some applications. Practical implementations displayed inconsequential differences, in the order of 10%-30%, suggesting that memory latency for each operation was an overriding issue.

5.5.3 Memory locations

The CUDA architecture provides a number of disjoint memory locations with different functions and features. For transition and emission values, which are fixed for all sequences, we make use of the *constant memory* - a small area of cached memory that provides fast-as-register access with a one memory fetch cost on a cache miss.

Another fast access location is the *shared memory.* These are similar in form to a L1 cache in a traditional CPU, however the allocation is completely user controlled. Each multiprocessor includes one such location, and as such, it is shared between the blocks allocated to that multiprocessor. Each block has a segment of the shared memory, which each thread in that block can access. Ideally, we would like to store the dynamic programming tables in this fast-access memory. However, under the current CUDA architecture the space allocated for each multiprocessor is 16384Kb. Such space confinements force a low occupancy value for each multiprocessor, resulting in very little speed up from this particular optimisation.

5.5.4 Phased Evaluation

By providing a thread-per-state, we are beholden to the model to determine our block size, since all states share a block. Since the block-size partly determines the occupancy of a SMP (number of threads per SMP), we may have an unbalanced set of parameters. Additionally, we are constrained in the size of our models by the maximum number of threads per block - typically 512.

To remove these unnecessary limitations, we must de-couple the number of threads from the number of states. The most obvious way to do this is to share a single thread between multiple states, running through the states in *phases.* In addition to allowing the block size to become independent of the number of states, this approach has other benefits. By synchronising the phases across the threads, we can support some form of dependency on the device - by ensuring that any dependents of a cell are in a prior phase.

5.5.5 Block size heuristics

One important measure of the effective use of a multiprocessor is the *occupancy* value - the number of warps per multiprocessor. Recall that threads are scheduled in warps, 32 threads per warp, so this effectively describes the number of threads that are *active* on an SMP at once. Active, in this case, is a statement of resource usage e.g registers and is does not imply the thread is actually running. A high occupancy plays an important role in supporting the scheduler to hide the high latency of global memory, by providing many warps to schedule whilst memory access is in progress.

There are three factors which limit the occupancy of a multiprocessor:

- The number of registers per thread. Each multiprocessor has a limited number of registers, and thus limits the number of threads that can be placed.
- The amount of shared memory per block. Recall that shared memory is allocated on a per block basis, not per thread. Again, each multiprocessor has a limited amount of shared memory.
- The number of threads per block. Threads can only be placed on a multiprocessor in entire blocks.

Registers per thread and shared memory per block are both dependent on the kernel implementation alone. The threads per block is set by a parameter, and is therefore customisable dependent upon the other factors.

It is important to note that achieving a high occupancy is not the be-all-and-end-all of GPGPU optimisation - there may be trade-offs when modifying the above factors. For example, we may be able to increase the occupancy by moving some values from registers to local memory, at the cost of increasing the number of device memory calls required, which may in fact increase the time taken for the kernel.

Bearing these facts in mind, we can provide a heuristic for determining a suitable block size. NVIDIA provide a CUDA Occupancy Calculator, which we can use as the basis for computing the values. The calculations used are described in the Chapter 4 of the CUDA Programming Guide[51].

We take a straight forward approach to implementing the block size heuristics. We try a sequence of likely block sizes, recording the occupancy for each one. For our candidate block sizes we take multiples of 16 from 16 to 512 plus the number of states. For devices of compute capability 2.0 we check in multiples of 32 - for devices lower than that warps are executed in *half-warps* of 16 threads at a time, later devices execute a fall warp at once.

5.5.6 Unused Attributes

Unused attributes may be both wasteful of space and time. For example, if we do not require the traceback of the result of the viterbi algorithm, we should not allocate an array in memory for it, nor should we evaluate the attribute on the device.

We can determine which attributes are used for a given call by providing a program analysis that associates each use of an attribute with the dynamic programming table. We use that information when generating the CUDA code to remove unused attributes at compile time.

5.5.7 Sliding Window Dynamic Programming Tables

As described in Section 5.4, we implement the dynamic programming algorithms as a series of lookup tables - with the states on one axis and the position in the sequence along the top axis. A cell in this table represents the probability at a given state and position in the table. However, we may not always require these intermediate results beyond their use for computing other values in the table. This is the case when we have identified *partially used attributes* - those where we only require the final value.

In these cases we can economise the table by noting that each of the given algorithms computes the value of a given cell with reference only to cells in the previous or current columns - since we can have emitted at most one character at this state. As such, if we do not require the intermediate results, we can implement a *sliding window* across the table, storing only the current column and the previous column. At the end of each step we re-designate the current column as the previous column, and use the previous column as the new current column, overwriting any previously stored values as we go - cheaply performed using simple pointer manipulation. This mechanism ensures we store only those values strictly required by the algorithm, thus avoiding excessive memory wastage.

If we consider that each sequence to be evaluated normally requires a $O(mn)$ size table, where m is the number of states and n the length of the sequence, and a sliding window scheme reduces that to an $O(m)$ we can immediately see the benefits. Allocating the entire dynamic programming table was therefore a significant area of memory inefficiency, and resulted in an arbitrary limitation on the number of sequences that could be processed in one run on the device.

This approach not only has memory allocation benefits, but can also provide improved memory performance by use of fast-access *shared memory.* Under the current CUDA architecture the space allocated for each multiprocessor is just 16384Kb. Such space confinements would allow very few blocks to share a single device, creating a low occupancy value for each multiprocessor when attempting to store the entire table, reducing the effectiveness under the normal table allocation scheme. However, under a sliding window, with the significantly reduced memory requirements, occupancy is not impacted by placing the window in shared memory. The system as implemented allows models with up to 512 states before the occupancy may be reduced, and even up to 768 may be safely accommodated before occupancy is reduced below the 25% level recommended by NVIDIA.

Whilst this is successful in reducing the memory usage, it does mean that we are unable to recreate the backtracking trace through the table. This can be rectified by allocating a separate table to store the backtracking data for each cell - a table that is of a smaller size than the full table since each cell only needs to store the position of the previous cell, and not a complete floating point number. In most cases, the number of states - the number of possible prior positions - requires fewer bits than a full 32 bit `float`. For the forward and backward algorithms no backtracking is required, and so there is no need for this extra storage.

5.5.8 Custom kernels for each program call

For each algorithm call in the program, we generate a different kernel call using customised kernel code. Doing so allows allows the removal of a number of unnecessary runtime checks.

- We generate different code depending on the specified block size. If the block size that has been determined (see Section 5.5.5) is smaller than the number of states, each thread needs to iterate over the assigned states. This code is only generated in that case.
- For each call we determine which attributes are required - and in what form - and only generate the code to store the attributes accessed (see Section 5.5.6). Since we determine whether we require simply the end result or the entirety of the table, we can generate code that stores the intermediate results or accumulates the attribute without storing it (see Section 5.5.7).

5.5.9 Sorting Input

One observation we make is that a pre-sorted input sequence will have a more optimal execution pattern. This is an optimisation that was also noted in GPU-HMMER[64].

5.6 Comparisons

5.6.1 The Occasionally Dishonest Casino

We describe the casino model in Section 5.3.1. It has been chosen as a simple example of a Hidden Markov Model.

No. of Seq.	HMMingbird	HMMoC1.3	Speed Up
2500	0.126	2.611	20.72
5000	0.149	5.526	37.09
7500	0.206	8.406	40.81
10000	0.222	11.028	49.68

Figure 5.6: HMMingbird versus HMMoC1.3 on the Occasionally Dishonest Casino

As a simple test for speed versus HMMoC we use a set of pre-generated sequences of various lengths.

5.6.2 Profile Hidden Markov Model

The structure of a *profile hidden markov model* is described in Section 5.3.2.

A profile HMM is one of the most widely used applications for HMM within bioinformatics. *HMMER* is a tool that can create profile HMM's and use them to search vast databases of sequences. It has been in development for over fifteen years, and is highly optimised. In particular, it uses forms of corner cutting to remove swathes of computations. Various attempts have been made to port HMMER to use GPU's. The most widely known example is *GPU-HMMER*[64], which provides a CUDA based mechanism for searching databases.

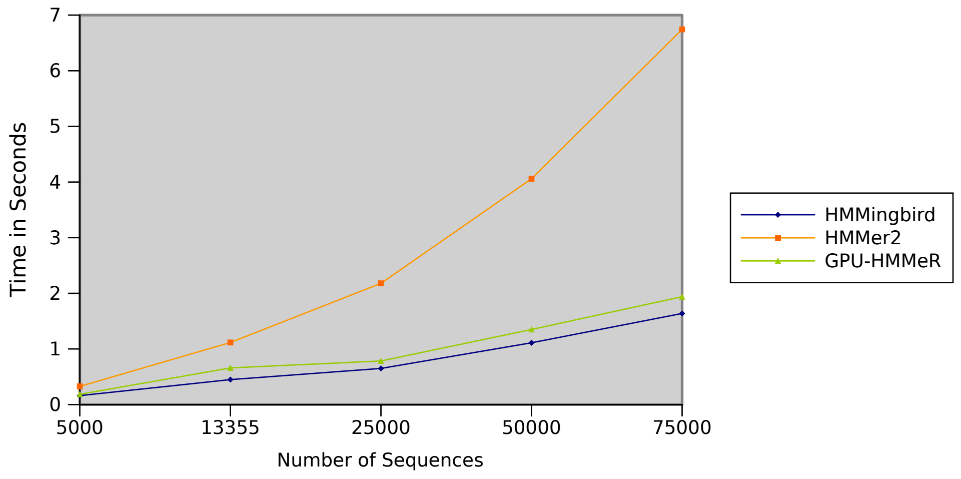

Figure 5.7: Chart for a 10 state profile recognising the Thymidine kinase (TK) family, using the viterbi algorithm to determine the best scoring path.

Our results show a significant (x2-x4) increase over HMMER, in line with the performance gains shown by GPU-HMMER. It is particularly worth noting that GPU-HMMER is a hand-coded and optimised application for a specific subset of HMM's, yet HMMingbird outperforms it by between 15%-45% in these tests. Examining the two programs under a profiler suggests that GPU-HMMER has a faster kernel, however this is offset by an inefficient data transfer process. Another note of caution is that GPU-HMMER does some post-processing, which may also account for some of the difference.

We also see significant performance gains when compared to HMMoC, with greater gains as we evaluate larger and larger sequence databases.

No. of Seq.	HMMoC1.3	HMMingbird	Speed Up
5000	4.809s	0.161s	x29.87
13355	25.68s	0.449s	x57.19
25000	49.548s	0.65s	x76.23
50000	90.8s	1.111s	x81.73
75000	168.376s	1.638s	x102.79

Figure 5.8: Speed up versus HMMoC

5.7 Conclusions

Our tool significantly improves on prior work in two important areas:

- By providing a concise language for describing Hidden Markov Models. It is significantly shorter and clearer than the equivalent descriptions for general purpose tools such as HMMoC.
- Demonstrates clear increases over CPU only methods on computation, and performance on-par with other GPU tools, with a wider range of input models.

Previous attempts to describe Hidden Markov Models have been focused on producing a file interchange format for models, and as such they have been focused on formats that have readily available means for parsing, such as XML or S-Expressions. Whilst these techniques reduce the difficulty of building the tool, they do not make it easy for the user, nor do they provide an elegant way of specifying the models.

By taking a different approach that focuses on the Hidden Markov Models as *programs* not *data*, we can bring to bear a whole range of tools and techniques to aid in both the implementation and the design of a domain specific language. Doing so changes the expectations we have for the language - the tool is no longer a code generator but a fully-fledged compiler. We can approach the problem with a clean and clear syntax - removing any loopholes such as the inclusion of arbitrary C code. This allows us to focus on generating the best possible GPU code, without worrying about user changes - sacrificing some flexibility for usability. This is a particularly important consideration when trying to tempt a broad spectrum of users to the tool.

Our performance results prove the effectiveness of dynamic programming algorithms of this ilk on massively parallel architectures. Despite the intrinsic dependencies within the algorithm - it is not trivial to parallelise - we can produce fast code, with plenty of scope for improvement.

5.8 Further Work

Whilst we believe we have excellent performance using the current system for large number of sequences on small to medium size model, we wish to improve our performance on both larger and smaller models. In particular, we would like to implement different models of division of labour for models with sizes at either extremity. For small models, a sequence-per-thread distribution should increase performance by reducing the co-operation required between threads, at the cost of increasing register usage per kernel. For larger models - say tens of thousands of states - we might wish to implement a cross-block system, where each kernel call computes one column in the table using multiple blocks.

A further modification to support large models with a linear structure is to allow some silent states within the model. The current mechanism removes all silent states during compilation. In linear structures this may cause an exponential growth in the number of transitions. By allowing some silent states, we can reduce this growth. The caveat is that we must ensure the kernel computes all silent states before emitting states. This can be fixed by ensuring that the silent states are at the start of the state array, and that their dependents are at the end of the state array. We can then set block size appropriately, so that the blocks with silent states are computed prior to the blocks with dependent states.

Under the block-per-sequence model, we often find divergence due to the varying number of transitions per state. We also find memory bank conflicts and constant cache conflicts when more than one thread *a*) accesses the same location in shared memory; or *b*) accesses different locations in the constant cache. Reduction of conflicts can come from ordering the transitions appropriately to reduce such issues. In addition, we can modify our algorithm to separate states from threads, so each thread can support multiple states. This allows a grouping of states in a single thread to smooth out differences in the number of transitions between threads.

Generalising the work described in Section 5.4, we hope to develop a small functional language that will allow the differences between the three algorithms to be described in terms of the recursions they implement. Furthermore, we hope to widen the scope of such a language to encompass other such dynamic programming algorithms with similar characteristics.

Chapter 6

Thesis Proposal

6.1 Thesis

Declarative DSLs enable expert scientists to make use of massively parallel architectures without being distracted by the details of such architectures. To wit, we propose to focus on the support of massively parallel architectures for optimisation problems through the development of domain specific languages and appropriate tools. We focus on problems based around the Principle of Optimality; problems made of sub-problems, as found in dynamic programming algorithms. These problems are rife in scientific computing, particularly in bioinformatics, and flexible support for this sort of development is vital to best utilise the available hardware. Our strategy consists of three steps:

- Develop a framework for parallelising recursive functions of an appropriate form.
- Map useful domains to the framework through the use of data-definition DSLs.
- Support natural extensions to domains through extensions to the recursive functions.

6.2 Strategy

We have already proven that massively parallel co-processors are eminently suitable for so-called desktop super-computing for scientific purposes, providing a cost-efficient, scalable and powerful approach - our case studies in Bioninformatics show this. In addition, we have developed a DSL for Hidden Markov Model compiler prototype that allows efficient description of HMMs, illustrating the utility of DSLs in this area.

Our next step is to build a small functional language that is based around the concept of simple recursive functions representing sub-problems. This will be used to support and implement future domain specific languages, first focusing on Hidden Markov Models.

We will implement our system as a series of stand-alone, or external, languages. Whilst some may argue that *embedded languages* would support our cause better, we feel that an external environment best supports the goals and ambitions of our software. Creating high-quality massively parallel implementations requires a level and depth of analysis that is best supported through an external language, where we can explicitly control the complexity to ensure we can generate appropriate code. In particular, we can avoid the complex structures and idioms of a host language that may be difficult to efficiently parallelise.

Familiarity is another important factor - the *lingua franca* of the Bioinformatics world are languages like Java or C++ with very little ability for creating fluent internal DSLs. On the other hand, languages with support for creating fluent embedded languages are unfamiliar to these domain experts. One may argue that a new external language is also unfamiliar; however, we believe that a simple declarative language will be easier to pick up and use, particularly if it is designed to match the natural descriptions of the model.

This framework will permit exploration of the core ideas of tabulation on massively parallel machines in a way that distances the domain specific front-ends from the exact parallelisation scheme. We will explore the methods of storing optimisation problems efficiently on massively parallel hardware, computing bounds for given recursive equations, analysis tools for ensuring recursive functions adhere to the appropriate conditions and techniques for partitioning the dependency graph of a given recursive function.

At the most simplified level, we will work with recursive functions that have a similar character to the hidden markov model algorithms we have already implemented. This will include some fixed methods of parallelisation e.g along axes. We will then broaden the framework to include a wider array of dynamic programming like algorithms, with more diverse parallelisation techniques. Our exploration strategy will involve identifying categories of recursive equations that have a sensible mapping, implementing automatic recognition by developing analysis techniques on functional equations and providing automatic mappings to the GPU.

Our strategy for adding domains to our framework will involve case-studies at each step; ensuring we are considering a valid and useful subset of a problem, and that a sensible domain mapping for GPUs exists. Our success in this endeavour will be in comparison to existing implementations of each problem; comparing the efficiency of the solutions and the ease of implementation.

6.3 A timeline

2010

October-December

- Develop functional framework for describing recursive algorithms adhering to the Principle of Optimality.
- Analyse functional equations to determine that they adhere to the Principle of Optimality, e.g they contain an appropriate base case and all recursive calls represent sub-problems.
- Automatically determine a suitable partition scheme on the dependency tree that describes those elements that may be run in parallel: at this stage we will consider parallelism across a given axis or across the diagonal only.
- **Deliverable**: A recursive functional framework capable of implementing Smith-Waterman and similar algorithms, with performance equivalent to hand-coded implementations.

2011

January-March

- Integrate the functional framework into HMMingbird.
- Map Hidden Markov Model algorithms to functional framework.
- Extend HMMingbird to support more complex examples of Hidden Markov Models.
- Map common applications to HMMingbird to show value of approach.
- **Deliverable**: A version of HMMingbird that uses the recursive functional framework to implement the standard HMM algorithms at similar efficiency to the existing HMMingbird and can be used to implement examples such as a Profile HMM.

April-June

- Extend functional framework to analyse dependency graphs to identify other axes of parallelisation.
- Automatic implementation of axis on the device.
- Problems of ordering parameters optimally to best support parallelising on the device.
- **Deliverable**: An extension of the framework to support unusual recursions; support for multi-tape HMMs; an improvement in performance for typical applications.

July-September

- Representing domain structures, such as Hidden Markov Models, as data structures on the device designed to match read/write patterns of the massively parallel processor and minimise load/store time and space.
- Suitable data-structures for memoization and tabulation that best support the memory hierarchy of the device.
- **Deliverable**: Improved performance, particularly for large Hidden Markov Models with lots of silent states.

October-December

- Investigate other optimisation algorithms with long run-times by building initial device prototypes.
- RNA secondary structure prediction using Stochastic Context Free Grammars.
- **Deliverable**: Example GPU applications that implement a few choice problems related to SCFGs.

2012

January-March

- Extend framework to create DSL for Stochastic Context Free Grammars and other advanced recursive forms.
- **Deliverable**: Framework support for SCFGs; including a RNA implementation that has competitive performance on the GPU.

April-June

- Confirmation of Status.

July Onwards

- Write dissertation.

2013

By March

- Submit dissertation.

Bibliography

[1] The Hadoop Map/Reduce Framework. http://hadoop.apache.org/mapreduce/.

[2] R. Bellman. *Dynamic Programming.* Dover Publications, March 2003.

[3] R. Bird and O. de Moor. *Algebra of Programming, The.* Prentice Hall PTR, September 1996.

[4] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, 1980.

[5] R. S. Bird and O. De Moor. From dynamic programming to greedy algorithms. In B. Moeller, H. Partsch, and S. Schumann, editors, *Formal program development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61, 1993.

[6] G. E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990.

[7] E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18(2):139 – 179, 1992.

[8] P. Bonzon. Necessary and sufficient conditions for dynamic programming of combinatorial type. *J. ACM*, 17(4):675–682, 1970.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[10] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[11] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: Compiling an Embedded Data Parallel Language. pages 1–12, September 2010.

[12] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. 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.

[13] Chakravarty, Manuel M. T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., and Grover, Vinod. Accelerating Haskell Array Codes with Multicore GPUs, 2010.

[14] S. Che, J. Li, J. 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.

[15] N. H. Cohen. Characterization and elimination of redundancy in recursive programs. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 143–157, New York, NY, USA, 1979. ACM.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Science/Engineering/Math, July 2001.

[17] S. Curtis. Dynamic programming: a different perspective. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 1–23, London, UK, UK, 1997. Chapman &amp; Hall, Ltd.

[18] O. de Moor. Dynamic programming as a software component. In N. Mastorakis, editor, *CSCC*. WSES Press, 1999.

[19] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150, 2004.

[20] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[21] S. Dreyfus. Richard Bellman on the Birth of Dynamic Programming. *Oper. Res.*, 50(1):48–51, 2002.

[22] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1999.

[23] S. Eddy. HMMer Website, including User Manual. http://hmmer.wustl.edu.

[24] U. Faigle and A. Schoenhuth. A simple and efficient solution of the identifiability problem for hidden Markov processes and quantum random walks. 2008.

[25] R. Giegerich and C. 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.

[26] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, December 1982.

[27] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Towards Compiling SaC to CUDA. In Z. Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[28] K. K. H. Ito, S. Amari. Identifiability of hidden Markov information sources and their minimum degrees of freedom. *IEEE Transactions on Information Theory*, 38(2):324–333, March 1992.

[29] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. 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.

[30] P. Helman. A common schema for dynamic programming and branch and bound algorithms. *J. ACM*, 36(1):97–128, 1989.

[31] P. Helman and A. Rosenthal. A comprehensive model of dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):319–334, 1985.

[32] R. C. G. Holland, T. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Drager, A. Yates, M. Heuer, and M. J. Schreiber. Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):btn397–2097, August 2008.

[33] D. Horn. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, pages 573–589. Addison Wesley, March 2005.

[34] D. R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementatio. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 11–, Washington, DC, USA, 2005. IEEE Computer Society.

[35] R. M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

[36] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, February 2010.

[37] J. Krüger and R. 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.

[38] C. Liu. cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. http://liuchuan.org/pub/cuHMM.pdf, 2009.

[39] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. page 8 pp., apr. 2006.

[40] Y. Liu, D. Maskell, and B. Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[41] Y. Liu, B. Schmidt, and D. 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.

[42] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *In Proceedings of the 8th European Symposium on Programming*, pages 288–305. Springer-Verlag, 1999.

[43] G. Lunter. HMMoC a compiler for hidden Markov models. *Bioinformatics*, 23(18):2485–2487, September 2007.

[44] G. Mainland and G. 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.

[45] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[46] W. R. Mark, R. S. Glanville, K. Akeley, and M. 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.

[47] M. McShaffry. *Game Coding Complete.* Paraglyph Press, 2004.

[48] L. G. Mitten. Composition principles for synthesis of optimal multistage processes. *Operations Research*, 12(4):pp. 610–619, 1964.

[49] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 88(2):665 – 674, 1982.

[50] S. B. Needleman and C. 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.

[51] NVIDIA. *NVIDIA CUDA Programming Guide 3.1.* 2010.

[52] T. J. Purcell, I. Buck, W. R. Mark, and P. 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).

[53] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[54] S. S. Skiena. *The Algorithm Design Manual.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[55] D. R. Smith. Structure and design of problem reduction generators. In *Client Resources on the Internet, IEEE Multimedia Systems 99*, pages 302–307, 1991.

[56] D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Sci. Comput. Program.*, 14(2-3):305–321, 1990.

[57] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, March 1981.

[58] M. Sniedovich. A new look at bellman's principle of optimality. *J. Optim. Theory Appl.*, 49(1):161–176, 1986.

[59] M. Sniedovich. Dijkstra's algorithm revisited: the dynamic programming connexion. *Control and cybernetics*, 35(3):599–620, 2006.

[60] M. Sniedovich. *Dynamic Programming, Foundations and Principles, Second Edition.* CRC Press, 2010.

[61] J. Svensson, K. Claessen, and M. Sheeran. Gpgpu kernel implementation and refinement using obsidian. *Procedia Computer Science*, 1(1):2065 – 2074, 2010. ICCS 2010.

[62] D. Tarditi, S. Puri, and J. 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.

[63] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260 – 269, apr. 1967.

[64] J. P. Walters, V. Balu, S. Kompalli, and V. 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.

[65] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721, New York, NY, USA, 1991. ACM.