Synthesising Graphics Card Programs from DSLs

Luke Cartey
University of Oxford
luke.cartey@cs.ox.ac.uk

Rune Lyngsoe
University of Oxford
lyngsoe@stats.ox.ac.uk

Oege de Moor
University of Oxford
oege.de.moor@cs.ox.ac.uk

Abstract

Over the last five years, graphics cards have become a tempting target for scientific computing, thanks to unrivaled peak performance, often producing a runtime speed-up of x10 to x25 over comparable CPU solutions.

However, this increase can be difficult to achieve, and doing so often requires a fundamental rethink. This is especially problematic in scientific computing, where experts do not want to learn yet another architecture.

In this paper we develop a method for automatically parallelising recursive functions of the sort found in scientific papers. Using a static analysis of the function dependencies we identify sets — partitions — of independent elements, which we use to synthesise an efficient GPU implementation using polyhedral code generation techniques. We then augment our language with DSL extensions to support a wider variety of applications, and demonstrate the effectiveness of this with three case studies, showing significant performance improvement over equivalent CPU methods, and similar efficiency to hand-tuned GPU implementations.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program Synthesis; D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Language, Performance

Keywords gpu, scientific applications, program synthesis, dynamic programming

1. Introduction

Recent advances in graphics card design have embraced the rapid pace of change predicted by Moore's law to unlock a mass market massively-parallel architecture, with phenomenally high potential peak performance. Unfortunately, this great power also comes at a cost: an architecture that is often difficult to program effectively and efficiently.

The problem is twofold — we must identify a suitable parallel algorithm and we must use the resources of the device efficiently. Whilst we have a comprehensive understanding of how to map graphical problems to massively-parallel devices, for many other classes of problems it is less clear what parallel algorithm might be suitable. Much of the work so far has pragmatically focused on data-parallelism; applying the same problem to a large data-set by assigning a single problem to each thread. We want to focus on those problems which require a more nuanced algorithm to achieve peak performance on the GPU.

With the area in an embryonic stage, language support has often been aimed at extending existing languages with parallel primitive operations. However, a growing number of people are approaching from a different angle — by providing high-level languages or specifications which are automatically parallelised (Liszt & OptiML[2], FLAME[15], Nikola[18] and Obsidian[7])

In this paper we propose a system for synthesising graphics card programs from recursion equations. We develop a simple language for describing such recursions, then augment those equations with domain-specific notations. This allows us to support a wide-range of recursive problems in many different domains. We build on existing work to develop an automatically parallelising end-to-end compiler that generates appropriate target code for NVIDIA GPUs.

1.1 Graphics Cards

Modern graphics cards are powered by massively parallel processors. These typically consist of a large number of cores — hundreds to thousands — arranged in a series of multi-processors. Each multi-processor contains a number of cores that work synchronously to achieve parallelisation, and each multi-processor acts independently of any other — there are no global synchronisation mechanisms. Indeed, this is a deliberate design decision to reduce the complexity of the hardware.

In practice, this forces us to use the GPU in one of two ways: an *inter-task* parallel system where each thread solves one problem, and so all threads on a multiprocessor compute the same instruction for different problems in sync, and an *intra-task* parallel system, where blocks of threads work co-operatively on the same problem. Inter-parallel systems tend to be more straight forward, and focused on data-parallelism. We will focus on problems suited to intra-parallel solutions in this paper.

1.2 Domain Specific Languages

Domain Specific Languages (DSLs) differ from general-purpose languages in that they may restrict the language or make assumptions about the input, in order to produce optimised output code[19]. We can use the knowledge derived from the concise input to provide both static and dynamic optimisations. Domain users also gain benefits from such a language — an interface for describing their problems in the way they understand them, which will often lead to better code and more opportunities for reuse. Our approach in this paper is to allow scientists to describe their problems in the same way they describe them in papers and other literature.

1.3 Scientific Problems

Some of the most demanding users of high-performance hardware are from the scientific community. In this paper, we will use case studies from bioinformatics to motivate our approach. It is an ideal match — typical problems require features with obvious parallel

PLDI'12, June 11–16, 2012, Beijing, China.

potential, such as large data-sets and extensive search spaces. Common applications include sequence alignment[3], gene finding[10], homology search[3] and structure prediction applications. Almost every practical application listed uses a form of recursive equation to define and describe the problem, often with an associated data model, such as a matrix, Hidden Markov Model or graph structure.

Our choice of bioinformatics as a case study does not represent a fundamental limit to our approach, and we expect that our framework could support many problems in other scientific domains and beyond.

1.4 Contributions

In this paper we develop a minimal language for describing recursive functions that can then be automatically parallelised. Furthermore, we show how we can fulfill this promise by adding domain specific extensions. Our contributions are:

- We design a simple host language for recursive problems (Section 4). One of our main contributions is to describe how this simple, but nonetheless expressive, language can be used to automatically synthesise a program targeting a massively parallel processor such as a GPU. The synthesis takes the following form:
  - We use existing techniques[9, 11] to frame the parallelisation problem as one of partitioning the call-graph into an ordered list of groups, where all members of a group are independent. We use a *scheduling* function which determines, for each call, to which partition it belongs (Section 4.4).
  - We show how the scheduling function can be used to synthesise a graphics card program using a polyhedral code-generation tool, CLooG[1](Section 4.3).
  - We define what criteria a scheduling function must satisfy in order to be valid for a given recursive equation. This is used as the foundation for an analysis that will verify user-provided schedules (Section 4.5).
  - A recursive equation will usually be amenable to multiple partition schemes. We use the identified criteria to frame the search for a schedule. We develop a technique for automatically finding valid and minimal schedules (Section 4.6).
- Many scientific problems can be written as recursive problems over complex data structures. We show how limited extensions to our host language can be made, and, importantly, what domain specific extensions can be made without losing the advantages of the automatic parallelisation (Section 5).
- We evaluate our approach for a variety of applications in our target domain, comparing against standard CPU approaches as well as hand-optimized GPU solutions (Section 6).

2. Anatomy of a scientific recursive problem

The *edit-distance* problem is a straight-forward example of a recursive problem. It asks us to compare two strings to find the minimum cost of making a series of edit or match operations to transform one string to another. It is one that is relevant in scientific domains such as bioinformatics, where the *Smith-Waterman* algorithm is an edit distance algorithm for aligning two biological sequences.

The *Principle of Optimality* allows us to frame the problem as a combination of a series of sub-problems, and so we can provide a recursive definition to determine the solution. For the edit-distance algorithm, as for many optimisation algorithms, this provides a natural and intuitive way of describing the solution (see Figure 1).

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

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

Figure 1. The edit distance recursion on strings s and t.

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

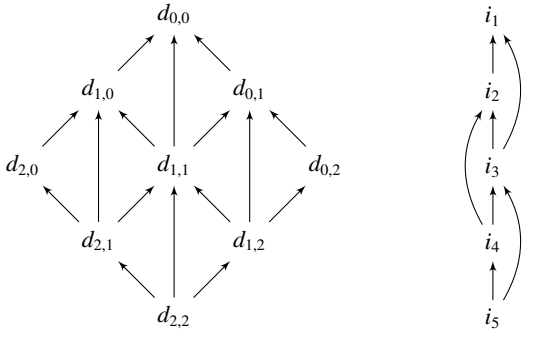
The edit-distance problem illustrates how recursive problems often have a useful form of concurrency, and whilst edit-distance problems have frequently been parallelised in this way, many similar recursive problems have not, and new ones constantly emerge. Given that these problems — specifically in bioinformatics — are often large or long running, it would be ideal if we could exploit any available concurrency using massively-parallel processors of the sort available in graphics cards.

The question is, can we develop a technique for mapping a recursive definition to a parallel environment? Such an approach would permit us to use recursions as a *host language* for describing parallel problems. This paper seeks to answer that question.

2.1 Dependent computations

For this, and further sections, we will assume that we have a single recursive problem that is to be computed in a bottom-up fashion using dynamic programming.

The first step to answering our question is to acknowledge that it is the recursive dependencies that will determine what, if any, form of concurrency will be available to us. Any parallel scheme we choose will need to respect the dependencies in the problem.



(a) Edit distance (b) Fibonacci

Figure 2. An example of the difference between a linear (2b) and a wide dependency graph (2a)

.

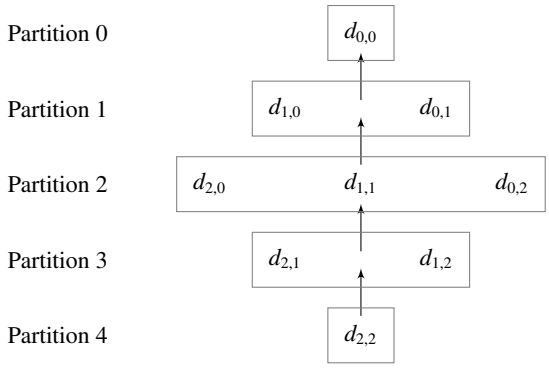


Figure 3. A valid diagonal schedule for the 3x3 edit distance problem. This schedule has five partitions.

Figure 2 contains the contrasting *dependency graphs* for the edit distance algorithm on a 3x3 problem and a straight-forward recursive definition of the Fibonacci sequence for *fib*(5). It is clear that the Fibonacci recursion permits no parallel solution — analysing each sub-problem only unlocks one other sub-problem. In comparison, sub-problems in the edit distance recursion can clearly unlock multiple other sub-problems — once we compute $d_{0,0}$, we can compute either $d_{1,0}$ or $d_{0,1}$. We say that the computation of $d_{1,0}$ is *independent* of $d_{0,1}$, and therefore the problem provides no constraints on the order of evaluation between the two. Consequently, we can evaluate them one after another, or even evaluate them simultaneously. It is this independence property within problems that we hope to exploit when identifying a parallel solution.

2.2 Scheduling partitions

Any form of concurrency requires that we identify a number of computations that can be executed simultaneously. This is especially important when we are developing an intra-multiprocessor application for a graphics card, where all threads must work in sync with each other.

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

It is clear that a single problem may permit many different partitions, and that partitions may even overlap. Each partition may also require some dependencies to have been computed before the partition itself can be evaluated.

To this end we define a *schedule*, a group of partitions that together cover the entire domain and do not overlap. As implied by the term schedule, this defines a total order between partitions in the domain, providing a fixed sequence of execution and simplifying the dependency graph. Figure 3 illustrates a valid scheduling of the 3x3 edit distance problem. We can think of each partition as a *time-step* of the computation.

Typically a schedule for a function f is defined as a mapping or function from the original domain to the partitions of the schedule[9]. We denote this schedule as S_f — the *scheduling function* for f.

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

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

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

In the edit distance example (Figure 3) it is clear that an appropriate scheduling function is $S_d = x + y$ — the parallel lines which represent independent values are described by the equation $x + y = c$, where c is the current partition. Figure 4 illustrates some schedules commonly found in scientific problems.

2.3 Choosing a schedule

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

As we previously observed, the validity of a schedule is determined by the dependencies described in the recursive function. In other words, a valid schedule is one that ensures that any dependency between two elements is fulfilled by executing the dependent before the dependee. We can do this in an inductive fashion, by proving any direct dependencies of an element are fulfilled before evaluating that element. What are the dependencies of f? The recursive calls of f; each different set of arguments to a recursive call constitutes a potential dependency.

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

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

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

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

Again, if we consider the edit distance example, an equally valid schedule would be $S_d(x,y) = 2x + y$, however this results in more partitions and would therefore be considered a less efficient schedule. There are very few occasions where a schedule with more partitions will be more efficient. These arguments are further developed in Section 4.

3. A hosted language environment for massively parallel architectures

In the previous section we have outlined how we might identify a parallel solution for a language that is sufficient to describe recursively defined problems at a high-level. However, the majority

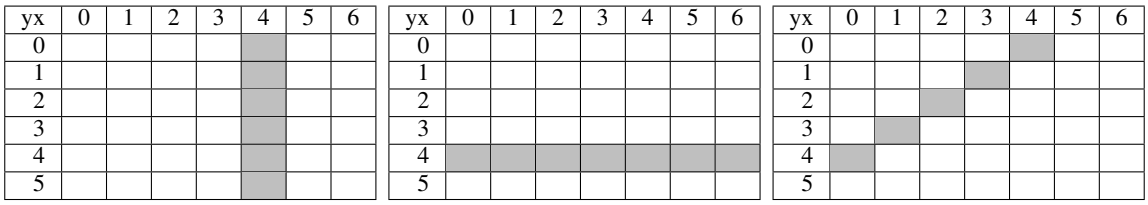


Figure 4. An example of three parallel strategies for the two dimensional case. Each case highlights the partition where $S_f = 4$.

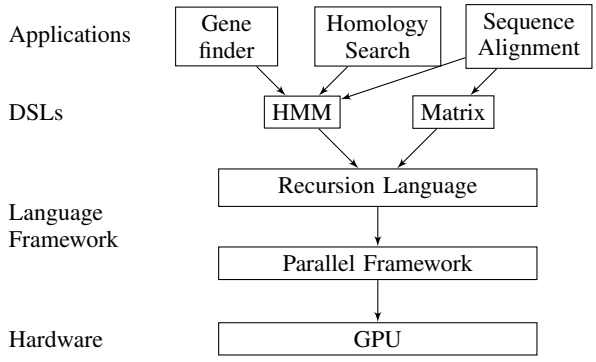


Figure 5. The levels of a hosted language environment for massively parallel architectures

of problems require more than a pure recursive function to describe. In bioinformatics, problems often refer to models, data matrices or other structures or systems. In addition, a simple recursive language may not provide the necessary terseness or clarity in the language to aid the domain-focused user.

To this end, we will integrate our simple recursive language into a wider framework for hosting domain specific languages targeting GPUs. Our system will be designed to match, as closely as is practical, the descriptions of algorithms that scientists typically provide in papers. Figure 5 describes the architecture of our system. Each DSL can be written as an extension to our base language, which is a DSL in itself for describing parallel computations..

This approach has a number of benefits. It takes the common elements from many domains, and uses a unified language to describe them. Our work in effectively parallelising this host language is therefore shared amongst many different domains, and provides a common language based on how these problems are typically described. At the same time, the extensions we write for specific domains allows implementors to concentrate on how to parallelise the elements unique to that domain, such as how to layout a particular model in memory. We later (Section 5) show how our base language can permit a wide range of extensions without breaking the parallelisation rules.

Our language is designed to mimic the style of a scripting language, providing basic operations — in this case recursive functions — and allowing users to potentially combine different DSLs. We will provide statements in the language for function definition, file loading, printing and function execution, including a map primitive, allowing us to map a function to a sequence of arguments. This map primitive supports the high-level inter-multiprocessor parallelisation.

```
Func  ::= funcname varname* Expr
Expr  ::= Expr + Expr | Expr − Expr | Expr × Expr | Expr/Expr
        | Expr < Expr | Expr > Expr | Expr = Expr | Expr! = Expr
        | Expr min Expr | Expr max Expr
        | (Expr)
        | Var[Expr]
        | Var
        | funcname(Expr*)
        | integer
        | if Expr then Expr else Expr
Var   ::= varname
```

Figure 6. The grammar for the domain specific language describing the permitted operations

3.1 Function definition

We provide the function definition section of our grammar in Figure 6. An example of a function for the edit-distance problem is given in Figure 7. We have based it on pure function definitions, with common arithmetic operations and recursive calls. The recursive function calls are only to named functions — no higher order or first class functions are allowed, and so no pointer analysis is required. This is a key restriction for a tractable analysis. In addition, we only support single recursive functions, as the analysis of mutually recursive functions is significantly more difficult.

Recursive calls are also restricted by limiting the *descent functions* — that is, the function that is applied to the original arguments to get the recursive arguments. Like the scheduling function, and for similar reasons, we will enforce that the descent functions are affine functions on the parameters. Without restriction, it will be difficult to enforce that the recursive calls only refer to partitions of a lower value.

The majority of applications in bioinformatics are based on analysis of sequences of characters, and so we provide a *sequence* primitive which can be queried by index. No further operations are provided on sequences — sequences are immutable. In particular, recursion on sub-sequences is only supported through indices.

The only conditional operation we will allow is a branching *if* expression. To allow branching to occur, we must allow conditional operators, which will consist of the numerical comparison operators, and an equality comparison for characters. Although the base language provides no looping construct, it is an operation that we could add with a domain extension — e.g looping over a data model.

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

3.2 Type System

Our base language has a simple type system which includes the following primitive types: integers, characters, sequences, indices on

sequences, floats, probabilities, booleans and alphabets. We justify the introduction of a probability type as well as a general floating point type by the frequency with which bioinformatics applications use probabilities. By introducing a high-level type, we can determine an appropriate low-level representation. For example, a frequent problem is underflow when dealing with multiplying small probabilities using floating points. We can counter this by converting probabilities to log space, where we can use addition instead of multiplication. Alternatively, we could introduce a custom extended exponent implementation.

Alphabets define a set of characters; sequence and character types are specified with reference to a particular alphabet. In the edit distance example (Figure 7), sequence s and t refer to the English alphabet, denoted by *en*. Indices specify the sequence to which they belong; this is necessary so we can determine the dimensions of the recursive problem for analysis and tabulation. In the edit distance example, index i references string s in this way, as does j with string t.

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

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

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

We have included two types in the framework that can act as recursive parameters — integers and explicit indices on sequences. Types may be both calling and recursive — for example, integer types, where the initial value determines the size of the domain. This is because the domain of all parameters must be finite for the method to be applicable, so we must specify an initial value for the integer domain. In the case of indices, they are naturally bound by the size of the sequence they reference.

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

3.3 Intermediate Representation

The target of our high-level framework is a low-level intermediate representation we have devised to abstract away the details of a particular massively-parallel environment. This allows us to separate out the construction of the overall parallel framework from the exact environment. We only target NVIDIA CUDA graphics cards at this stage, but in principle our abstract representation would allow us to support other vendors in the future.

```
int d(seq[en] s, index[s] i, seq[en] t, index[t] j) =
   if i == 0 then
      j
   else if j == 0 then
      i
   else if s[i - 1] == t[j - 1] then
      d(i - 1, j - 1)
   else
      (d(i - 1, j) min d(i, j - 1) min d(i - 1, j - 1)) + 1
```

Figure 7. The source code of a simple edit distance recursion

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

Figure 8. The program synthesis template

4. Parallelising Recursive Functions

4.1 Overview

In this section we describe our process for synthesising a graphics card program from a recursive definition. Our method takes the following steps:

1. We first encode the dependencies of the recursion as a series of criteria on the scheduling function (Section 4.5).
2. We use these criteria and a suitable goal function to automatically find, using a constraint solver, the coefficients of a valid scheduling function, which is optimal with respect to the number of partitions required (Section 4.6).
3. Given a scheduling function, we synthesise a massively parallel program using a polyhedral code generation tool, CLooG[1] (Section 4.3).
4. Finally, we discuss the issue of evaluating multiple problems on the same GPU (Section 4.7).

4.2 Scheduling Functions

Thus far we have only described the scheduling function informally. It maps the elements in the domain of our function f — the possible calls to f — to partitions. A partition is an integer value, thus providing a total ordering over partitions. There may be many valid schedules for each function; different schemes of parallelising the problem are described by different schedules.

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

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

Where $a_1, \ldots, a_n \in \mathbb{Z}$ and the *recursive* or *source* domain of f is defined as $X = X_1, \ldots, X_n$, where $\bar{x} \in X$ when $\bar{x} = (x_0, \ldots, x_n)$ and $x_i \in X_i$. The constraints on the function ensure that the analysis is tractable and the implementation efficient; they are not, however, fundamental limits and the extent to which they can be lifted will be discussed later in the section.

4.3 Program Synthesis

Scheduling functions provide a succinct way to describe the dependencies of a problem. However, to have any practical utility, we must be able to use it to synthesise a practical and efficient program. We start by tying this knot as a way to motivate and guide the search for a schedule.

Figure 8 outlines our approach: we will loop over the time-step partitions of the schedule in the domain, computing each partition entirely before synchronising and continuing on. Each thread will be allocated a number of elements to compute. For each partition, each thread will loop over all assigned elements.

The computation of each element proceeds by calling some function I to determine the indices of x associated with the current

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

Figure 9. CLooG output for the edit distance problem with a scheduling (scattering) function of $S_d(x, y) = x + y$.

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

Figure 10. Our conversion of the CLooG loop.

partition, thread and thread-step. We can then calculate the value and store it in the dynamic programming array, possibly for future use.

To formalise our approach, we will use the *polyhedral model*[12]. We can consider the original domain of our recursive function a polyhedron — a convex polyhedron representing the elements of $X_1, \ldots, X_n$ that the recursion can visits Our scheduling function then becomes an *affine transformation* of that source polyhedron to a target polyhedron, and I is the inverse of that transformation. The polyhedral model has been extensively researched in relation to loop parallelisation, however the principal is the same for recursions.

Using a scheduling function alone as an affine transformation creates a mapping from the recursive domain to a single dimension — the time-step. We still need iterate through all elements in the domain, assigning them to threads — a set of *space* dimensions.

CLooG[1] is a code-generation tool that can determine iterations over dimensions in this way. It takes, as input, the source polyhedron and the schedule — or *scattering function*, as it is described in CLooG. The output of this is a series of nested loops that iterate over the entire domain. In our case, the first, outer, loop will be over our partition time-steps. Each further nested loop represents a new space dimension of the target polyhedron, and the collection of all those inner nested loops iterates over all elements in the partition. This transformation should ensure that each element of the source — recursive — domain has an equivalent element in the target domain. Typically the number of dimensions in the target polyhedron will match that of the source polyhedron.

The code generated by CLooG will be a set of nested loops for a single thread. We must take one of the space dimension loops to use to map to our threads. We choose the outer loop. Given *tn* threads, with the thread designated by *t*, we can convert this loop:

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

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

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

Consider the edit distance example, where $S_f(x, y) = x + y$. Figure 9 describes the output given by CLooG. Figure 10 gives our parallel version of the loop.

4.4 Formalising function dependencies

The derivation of a suitable schedule is the first step in our compilation process. For that, we will first need to formalise the description of the dependencies between function calls.

Recall that f is a pure function, so we know that the result of the function is dependent only on the arguments in the domain. We can therefore define a call, c, as the list of the arguments, $y_0, ...y_n$, to the function f, where each y_i is a value in the domain of the equivalent parameter of f.

Let $G(C, E)$ be the call graph for f, where C is the set of possible calls and E is a set of edge pairs (c_1, c_2). The edge (c_1, c_2) represents the case when c_1 *may* call c_2. We do no branch analysis based on conditionals — instead, we consider all recursive calls in the function to be part of E. We will say $c_1 \to c_2$ when $(c_1, c_2) \in E$.

Using E we can define the relation $\rightsquigarrow$ — the transitive closure of E. We can think of this as the transitive closure of the dependency graph. Validation of S_f can then be defined in terms of the transitive closure:

$$c_1 \rightsquigarrow c_2 \Rightarrow S_f(c_1) > S_f(c_2) \tag{1}$$

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

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

$$S_f(c_1) = S_f(c_2) \Rightarrow c_1 \not\rightsquigarrow c_2 \tag{2}$$

By finding integer coefficients $a_1, \ldots, a_n$ of $S_f = a_1x_1 + \ldots + a_nx_n$ such that it adheres to 1, we will use this implication of independence to generate a parallel implementation.

4.5 Validity of a schedule

We have now defined the properties of a valid schedule. However, as formulated in (1), it is difficult to efficiently apply directly to a schedule. In this section, we derive a set of *criteria* that valid schedules adhere to.

This has two practical consequences — firstly, users can specify a schedule that we can then verify. Secondly, we will use the validity criteria to automatically determine a schedule (Section 4.6).

For a S_f to be valid, it must adhere to the implication in (1). We will show this inductively, by considering the direct dependencies of the call. Recall that $\rightsquigarrow$ is the transitive closure of $\to$ ($c_1 \rightsquigarrow c_2 \Rightarrow c_1 \to^\star c_2$). We can thus satisfy our condition by proving that for all $c_1 \to c_2$ the following holds:

$$c_1 \to c_2 \Rightarrow S_f(c_1) > S_f(c_2) \tag{3}$$

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

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

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

$$\{(x_1, \ldots, x_n) \to (xr_1, \ldots, xr_n) | \forall x_1, \ldots, x_n\}$$

For each recursive call site, we will need to verify that this set satisfies (3). Substituting in our S_f equations:

$$S_f(x_1, \ldots, x_n) - S_f(xr_1, \ldots, xr_n) > 0, \forall x_1, \ldots, x_n$$

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

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

This equation denotes the *criteria* on a valid schedule. We derive one criterion for each recursive call in the function, and confirm that a S_f is valid by confirming it satisfies all the criteria.

When the descent function xr_i is uniform (e.g of the form $x_i + c_i$) — the majority of practical cases — this equation will simplify to $(-a_1c_1) + \ldots + (-a_nc_n) > 0$, which is straight-forward to analyse. For any descent function that is affine (e.g of the form $A\bar{x} + \bar{c}$), the criteria will require the runtime range of x to determine the validity.

4.6 Automatically determining a schedule

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

Our aim is to describe a Constraint Satisfaction Problem (CSP) that combines the criteria of the previous section — which will enforce the validity of the scheduling function — with a selection measure which will help choose an efficient schedule. We can then pass these conditions and goal into a suitable CSP solver to find the coefficients $a_1, \ldots, a_n$ of a solution S_f.

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

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

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

$$min_{a_1,\ldots,a_n}(max_{\bar{x}}(S_f(\bar{x})) - min_{\bar{x}}(S_f(\bar{x}))) \quad (4)$$

Our intention is to construct this as a CSP over $a_1, \ldots, a_n$. However, it is clear that equation (4) is not a linear problem — we are trying to solve for both a and x. How can we encode this as a CSP problem if it is not linear?

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

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

For each constraint problem, we specify the maximum and minimum values of $x_1, \ldots, x_n$, use equation (4) as the goal we wish to minimise, and add a criteria for each recursive call, as discussed in the previous section. If a result is found, we store the minimum goal value found, and the first set of solution coefficients, and continue to the next problem, until we have checked or eliminated all dimensions.

4.7 Multiple problems

Our technique so far discusses how to map a single problem to a single multiprocessor — rather than employing all multiprocessors in a GPU. Typically a single problem is *tiled* across the multiprocessors, however bioinformatics often deals with large numbers of small problems. The compilation algorithm above makes use of the precise bounds of each domain to determine the best S_f and a_f. Different problem sizes may therefore require different schedules to run efficiently. What we want is to evaluate the problems simultaneously on different multiprocessors, so we must synthesise a solution that can apply different schedules to each problem.

To this end, we have developed a *conditional* parallelisation technique, where we adapt the CSP to derive multiple valid schedules. Considering all selected schedules, we then derive conditions under which each particular schedule is *minimal*, which we then evaluate at runtime on a problem-by-problem basis to determine which generated parallelisation code to use. For example, given a function $f(x, y) = \ldots f(x-1, y-1)\ldots$, the minimal schedule depends on the length — when $n_x < n_y$ then $S_f(x, y) = x$ otherwise $S_f(x, y) = y$.

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

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

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

1. Create all $n!$ permutations of the domains e.g in two dimensions $1, 2$ and $2, 1$
2. For each permutation, we minimise each dimension in turn, propagating the constraints until we find a solution which we add to the set. We therefore find the first lexicographical solution with respect to the permutation of dimensions. Each defined solution is minimal for some $\bar{x}$. We note that in practice the majority of problems have a single schedule, so the set size is far smaller than $n!$.

It may seem that this method is potentially expensive. To limit this effect, we restrict the coefficients to a small fixed number (10) that is customisable by the end user and note that n is usually small. Additionally, unlike the runtime analysis described for single problems, this analysis can be performed during compile time.

For each schedule found, we find use CLooG to generate the appropriate set of nested loops to iterate over the space. We use the conditions to determine which of the generated code paths

to take. This does not lead to any unnecessary code branching on the GPU, as a single problem will be alllocated to a single multiprocessor, and each thread on that multiprocessor will choose the same schedule.

4.8 Sliding window optimisation

In a traditional dynamic programming algorithm all prior results are stored until the computation is complete. One of the benefits of partitioning the dependency graph is that we can determine the range of previous results required, with respect to the number of partitions. We define this as the *sliding window* over the domain.

When the recursive descent functions are uniform, we can use the precise value of each criteria to determine the size of the sliding window, in terms of the number of previous partitions required. When the descent functions are affine, this is not possible. This optimisation is especially relevant for a GPU, since it will allow us to store a much smaller table for the intermediate values. This smaller table will typically fit in a cache that is much closer to the processor, such as shared memory on a NVIDIA GPU, almost eliminating the significant latency to global memory.

4.9 Limitations of the Approach

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

1. Goods solvers are available which can determine solutions in linear cases — if the recursive parameters or the scheduling function were not linear, we would not be able to use CSP or linear programming techniques.
2. We can synthesise good target code in the linear case; it is not clear how a non-linear function would be synthesised because it would be difficult to find the inverse transformation.
3. Problems in bioinformatics, our target domain, infrequently require non-linear cases.

In addition, for our technique in Section 4.7, we require uniform descent functions. This restriction is necessary as an affine descent would require knowledge of the runtime range to evaluate.

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

5. Domain Extensions

In Section 3, we described our proposed architecture in which our simple base language can be expanded by domain specific extensions. The intention of such DSLs is not that they relax any of the limitations set out in the paper, but instead be used to extend the data-structure and traversal operations to support specific applications. The following features can be defined as language extensions without modification to the described compilation algorithm:

- The introduction of any branching expression;
- Create new "recursive" data-types on the condition there is a defined mapping to the natural numbers;
- Create new looping expressions, where the extension can define the range of the loops at runtime, and can therefore derive solvable criteria on recursions within the loop.

New extensions should cover areas with wide applicability, so that the cost of a cross-specialist developing the extension is amortised across a wide array of applications. For the purpose of our evaluation, we have produced two such extensions — "Hidden Markov Models" and "Substitution Matrices". Both are widely used in bioinformatics, and extend our interesting, but neutered, language to something which can tackle real problems in that domain.

5.1 Substitution Matrices

Substitution Matrices represent a simple data-extension to our language. They allow the user to describe the cost of substituting one character in an alphabet for another. We introduce a new statement to our language for defining the matrix with a straight-forward table. This is represented by a new type, matrix, that can be passed to functions, and a new expression to load the substition value.

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

The introduction of this expression has no affect on the recursion analysis. The generated load code for this expression will depend where we locate the table; in shared memory for small tables or global memory for large.

5.2 Hidden Markov Models

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

As with the substitution matrix, we introduce a new statement to our language for defining the HMM model. This allows the user to define states, transitions and emissions. With it we introduce some new types - a HMM calling type, to pass the model to the function, and state and transitions as recursive types. To be able to tabulate recursions using states or transitions we provide an arbitrary total ordering over each one, providing a mapping to the natural numbers (see Section 3.2). The ordering is arbitrary because no recursion depends on the position of the states.

We add the following new expressions for exploring the model:

$$\begin{aligned} Expr \quad ::= \quad & \ldots \\ \mid \quad & Var.start \\ \mid \quad & Var.end \\ \mid \quad & Var.isstart \\ \mid \quad & Var.isend \\ \mid \quad & Var.emission[Expr] \\ \mid \quad & Var.transitionsto \\ \mid \quad & Var.transitionsfrom \\ \mid \quad & sum(var\ in\ Expr\ :\ Expr) \end{aligned}$$

`start` and `end` load the states associated with a transition. `isstart` and `isend` determine whether the state is the last or the first. `emission[Expr]` is the probability of a state emitting the Expr value.

The final expressions `transitionsto` and `transitionsfrom` are for identifying the sets of transitions to and from a state. We provide a summation statement (amongst others such as max and min), which can iterate over these sets of transitions.

Figure 11 compares the mathematical description of one of the canonical HMM algorithms to the implementation in our language. We recurse over s, the states in the HMM, and i, the indices into the sequence x.

The only new expression that affects the recursion analysis is the summation expression. The expression evaluated for each element may include a recursive call that depends on the newly created variable. For example, the forward algorithm has a recursive call using the start state of the transition within the sum (Line 8). For the

$$F(0,0) = 1$$
$$F(s,0) = 0 \, for \, s > 0$$
$$F(s,i) = e_{s,x[i]} \sum_{p:t_{p,s}>0} t_{p,s} F(p, i-1)$$

(a) The recursive equation, where $F(s,i)$ is the likelihood we are in state s at position i in the sequence x. $t_{s,p}$ is the probability of transitioning from state s to state p

```
prob forward(hmm h, state[h] s, seq[*] x, index[x] i) =
   if i == 0 then
      if s.isstart then 1.0 else 0.0
   else
      // The end state is silent
      (if s.isend then 1.0 else s.emission[x[i-1]])
      * sum(t in s.transitionsto :
            t.prob * forward(t.start, i - 1))
```

(b) The implementation in our extension

Figure 11. The forward algorithm, mathematical description compared to the implementation

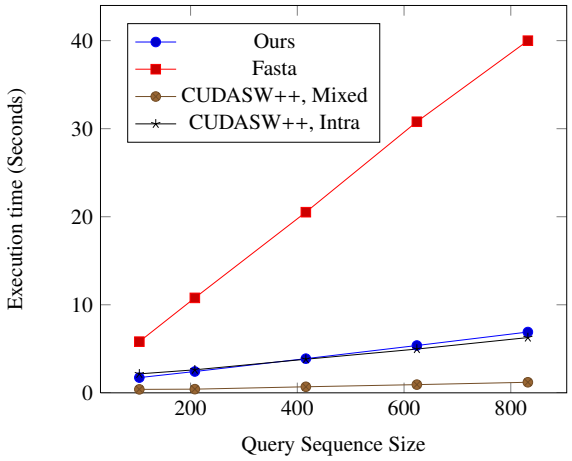


Figure 12. Smith-Waterman performance on varying query sequence sizes for a database of 75,000 sequences.

purpose of the analysis, we assume that the newly created variable - in this case t - can vary over all possible values and thus t.start, in this case, varies over any state. We conclude that our schedule can only be $S_{forward}(s,i) = i$.

6. Evaluation

There are two factors that will influence performance using our framework. The first is the choice of schedule, the second is the low-level implementation. In this paper, we have concentrated on selecting the former. We have chosen our schedule to be minimal with respect to the number of partitions. This choice was justified in Section 4.6, however we will note, for those that desire something more practical, that quantitative justification would be hard to achieve beyond the observation that in the given examples — edit distance, HMM algorithms — it produces the expected result.

As this paper has described the construction of a DSL *framework* our aim in this section is to demonstrate that we can use it to build efficient DSLs. For this reason we choose to analyse a number of DSLs rather than a specific DSL in detail. Each of our three case studies uses the base language described here, plus an extension, and is compared to existing CPU and, where possible, GPU applications to show that our framework is up to scratch.

These case studies are real applications in bioinformatics, using real genome data on existing models to provide a realistic comparison. In all cases our framework automatically derived the parallelisation schemes — no schedules were specified by the user. In each case we compare against a hand-coded GPU application, our tool has succesfully derived the same parallel strategy as the hand-coded application.

All quoted results are inclusive of setup time, such as memory allocation and copying to/from device. Our framework provides a runtime environment. Consequently, the times for our software are inclusive of scanning and parsing the input files. We have not, however, included the code generation time - we cache the compiled code for each function. The code generation overhead is typically around 1 second, primarily due to inefficiencys in the way in which we call ClooG from Java. We expect to be able to reduce this overhead drastically in future releases. We note that programs written in DSLs of this type are rarely large enough to increase this compilation overhead.

Results were obtained on a NVIDIA GTX 480 with 1.5GB of RAM, and CPU results using Intel Xeon E5520 CPU with 4GB of RAM. Each test was performed three times, and the results averaged. Results across runs were observed to be regular, and so we have not provided error bars.

6.1 Smith-Waterman

The Smith-Waterman algorithm is an implementation of the local edit distance problem, used for sequence alignment. The typical application will compare one *query* sequence against a database of other sequences, to identify high-scoring alignments. We implement the Smith-Waterman algorithm using the substitution matrix extension to determine the cost of substituting characters. The expected parallelisation is along the diagonal $x + y$, as with other edit distance algorithms.

For comparison, we use CUDASW++ 2.0[13] a GPU implementation of Smith-Waterman using the NVIDIA CUDA framework and the ssearch tool in Fasta[16], compiled without SSE2 vector instructions. CUDASW++ 2.0 provides two methods of parallelisation — intra-task parallelisation, which uses parallel diagonals across the table in the same way as our recursion, and an inter-task parallelisation, with a database sequence allocated per thread. Best performance is achieved by a hybrid approach, where smaller sequences are computed with inter-task and larger with intra-task. We provide both the hybrid and the intra-task results for CUDASW++ for comparison.

Our results (Figure 12) are very similar to the intra-task CUDASW++, and comfortably beat Fasta, demonstrating that we are reaching similar performance to the hand-coded, hand-tuned CUDASW++ for this type of parallelisation. As expected, the best overall performance is achieved by using the hybrid parallelisation approach. For this paper we have deliberately focused on automatic parallelisation in an intra-task manner. This is because inter-task parallelisation is algorithmically trivial. Although we have not covered the techniques in this paper, generation of a sequence-per-thread kernel as well as a intra-task kernel is straight-forward from our DSL code, and we would expect to achieve results on par with the hybrid CUDASW++.

6.2 Gene-finding Hidden Markov Models

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

We build a simple gene-finder using our HMM extension language. We compare our results against HMMoC 1.3[14], a code-

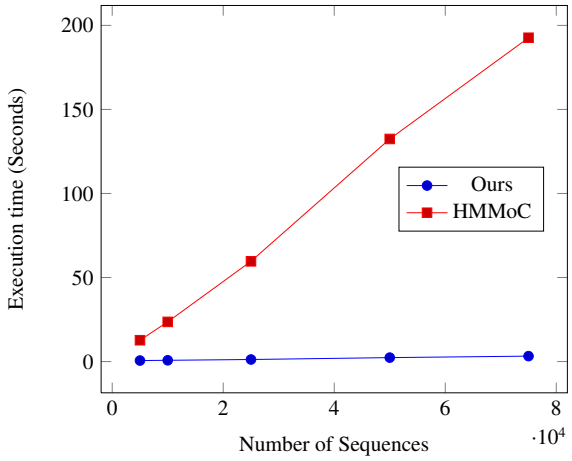


Figure 13. Gene-finding performance on varying sequence sizes.

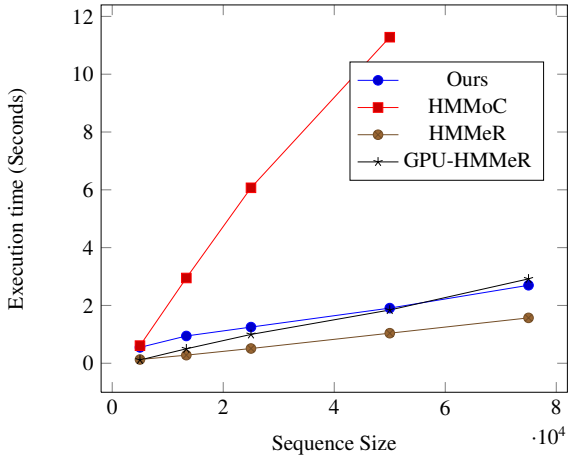


Figure 14. Performance on the TK model of 10 positions, with varying numbers of sequences.

generation tool for HMMs on CPUs. Our results (Figure 13) show a significant performance increase in line with the expected results over a CPU technique on this application — at larger database sizes, when we are using the GPU to its full extent, the performance increase is about x60. HMMoC is single-threaded application; considering the relative strengths of the CPU and GPU this speed-up is in line with what we might expect. As far as we are aware, there are no existing equivalent tools for GPUs.

6.3 Profile Hidden Markov Models

Profile HMMs are a type of HMM designed for representing a family of sequences. Typical applications include training models and database searching, where we apply a database of sequences to the profile model in order to determine new sequences that belong to a given family. We perform a database search using the forward algorithm on various sequence and profile models. Once again, we provide comparison to HMMoC, as a general purpose HMM tool, as well as to HMMeR[4], a special purpose tool for profile HMMs. In addition, we compare to a GPU port of HMMeR 2.0, GPU-HMMeR[21].

HMMeR is a high-performance, widely used tool — hand-coded and hand-tuned over 15 years to provide optimal performance on profile models. The latest version, HMMeR 3.0, uses

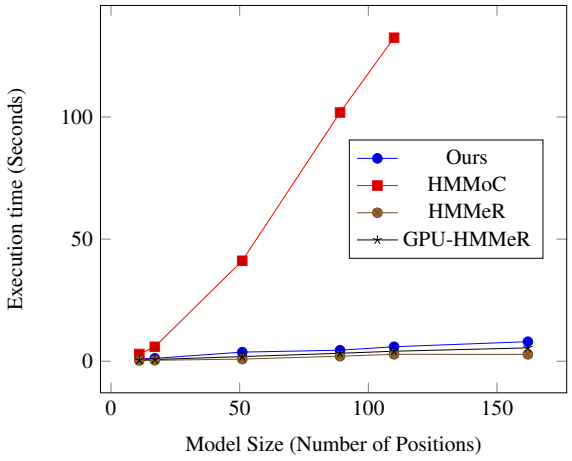


Figure 15. Performance on a dataset of 13,355 sequences, on models of a varying size.

a series of filters to remove low-scoring sequences at low cost by evaluating simplified models and algorithms, such as the MSV (Multiple Segment Viterbi), before bringing the full forward algorithm to bear on a much smaller set of problems. Whilst it would be both viable and beneficial to translate such techniques into our language, it would be a significant undertaking. To that end, we set the "–max" flag to turn off all such filtering and provide a fair comparison between a best of breed full forward algorithm for profile HMMs on the CPU to our GPU implementation.

Our results (Figures 14 and 15) show an expected large increase in performance over HMMoC for the GPU techniques. Our runtime performance is on par with GHMMeR. Any difference in performance is primarily due to the overhead of our runtime framework, and is smoothed out on larger sequence sets. However, all three are beaten by the most recently released version of HMMeR, 3.0.

Whilst it may at first this might be surprising, it must be remembered that HMMeR can make a number of assumptions about the nature of the problem that we are unable to do in a general HMM tool. HMMeR is also well optimised, using multiple threads and vectorised code on the CPU to maximise performance. However, this was a significant undertaking for the developers — what we provide is access to this type of performance across a much wider range of problems without the need to put in 15 years of optimisation into each one.

7. Related Work

Schedules were first proposed as a means of parallel analysis of recursive functions in the seminal paper by Karp et. al. [9], and further developed in [20]. Much of the further work in this area has been focused on loop parallelization[11], and in particular the *polyhedral* model[12] where it has been profitably used for code generation[1] for loop schedules.

There is a wealth of literature on parallel implementations of fully fledged functional programming languages, and a recent tutorial in the context of Haskell is [8]. We have consciously restricted our input language to enable more aggressive optimisations. Elliott [5] shows how functional programs can be used to generate efficient code for graphics operations on graphics processors. There is an overlap with this paper in that we use a restricted functional input language, but Elliott does not address the problem of mapping complex dynamic programming recursions to a GPU architecture.

DSLs are a growing trend in the graphics card world. Our approach of using a host language with extensions is very similar

to the concept of "Language Virtualization" provided by the Delite framework[2]. Where our approaches differ is that we develop a host language that provides automatic parallelisation, where they make use of the use of the DSL capabilities of Scala to support the creation of extension which can provide parallelisation. Examples include *Liszt* for physics simulations and *OptiML* for machine learning. Other examples of graphics card DSLs include *Nikola*[15] and Obisidian[18], both embedded DSLs for array computations and FLAME[7] a DSL for dense linear algebra.

Algebraic Dynamic Programming[6] is an example of a DSL designed specifically for Dynamic Programming problems in bioinformatics using strings, and has recently been extended to GPUs[17]. Whilst similar to our approach, it is currently deals only in diagonal parallelisation with one input sequence per problem.

8. Conclusion

In this paper we have shown that scientists can automatically synthesise efficient GPU code from specifications that read like the algorithm descriptions in their papers. Our preliminary results show that performance of the synthesised code is on a par with hand-tuned code for several important real world examples. However, the advantage of our system is that it can support domains far beyond these single application hand-tuned implementations.

Our technique is to partition the domain of a recursion into sets of values that can be computed concurrently. We adopt the construction of scheduling functions to achieve this, and have described how we can select such a function and subsequently derive a graphics card implementation. We apply insights from loop parallelisation using polyhedra to recursively defined problems on modern synchronous processors.

We have shown that such an approach can produce efficient and effective results, with minimal work on the part of the domain-focused user, and without needing to know the architectural complexities of the graphics card — bringing high-level language approaches close to hand-optimised CPU and GPU performance.

9. Further Work

We would like to extend our work to support mutually recursive functions, by deriving multiple scheduling functions, one for each function, whose partition time-step values are compatible. The same principle holds as for a single scheduling function; if $S_f(\bar{x}) < S_g(\bar{y})$ then $f(\bar{x})$ must be computed before $g(\bar{y})$. This would allow us to support more complicated applications, such as RNA secondary structure prediction.

In Section 3.2 we provided a simplified view of the translation of complex parameters — the need for them to map to natural numbers. In practice, the ordering of the model elements may be important to achieve high performance — different orderings may permit or restrict the available schedules.

References

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[2] 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. ISBN 978-1-4503-0203-6.

[3] 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. ISBN 0521629713.

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

[5] C. Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 45–56, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017482.

[6] 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. ISBN 3-540-44144-1.

[7] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, Dec. 2001. ISSN 0098-3500. URL `http://doi.acm.org/10.1145/504210.504213`.

[8] S. L. P. Jones and S. Singh. A tutorial on parallel and concurrent programming in haskell. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 267–305. Springer, 2008. ISBN 978-3-642-04651-3.

[9] R. M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967. doi: 10.1137/0115060.

[10] A. Krogh, I. S. Mian, and D. Haussler. A hidden markov model that finds genes in e.coli dna. *Nucleic Acids Research*, 22(22):4768–4778, 1994. doi: 10.1093/nar/22.22.4768.

[11] L. Lamport. The parallel execution of do loops. *Communications of The ACM*, 17:83–93, February 1974. doi: 10.1145/360827.360844.

[12] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416. Springer-Verlag, 1993.

[13] 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. ISSN 1756-0500. doi: 10.1186/1756-0500-3-93.

[14] G. Lunter. HMMoC a compiler for hidden Markov models. *Bioinformatics*, 23(18):2485–2487, September 2007. doi: 10.1093/bioinformatics/btm350.

[15] 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. ISBN 978-1-4503-0252-4.

[16] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of The National Academy of Sciences*, 85:2444–2448, 1988.

[17] P. Steffen, R. Giegerich, and M. Giraud. Gpu parallelization of algebraic dynamic programming. 2009. URL `HAL:http://hal.archives-ouvertes.fr/inria-00438219/en/`.

[18] J. Svensson, K. Claessen, and M. Sheeran. Gpgpu kernel implementation and refinement using obsidian. *Procedia Computer Science*, 1(1):2065 – 2074, 2010. ISSN 1877-0509. doi: DOI: 10.1016/j.procs.2010.04.231. ICCS 2010.

[19] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.

[20] H. Verge, C. Mauras, and P. Quinton. The alpha language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing*, 3:173–182, 1991. ISSN 0922-5773. URL `http://dx.doi.org/10.1007/BF00925828`. 10.1007/BF00925828.

[21] 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. ISBN 978-1-4244-3751-1.