Staging Parser Combinators for Heterogeneous Dynamic Programming

Thierry Coppey Manohar Jonnalagedda Nithin George Tiark Rompf Martin Odersky

Department of Computer Science
EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract

Parser combinators can be implemented for both recursive descent parsing and dynamic programming, as done in Algebraic Dynamic Programming (ADP). They can be seen as operations over the `Maybe(Option)` and `List` monads respectively. In both approaches, they suffer from an overhead due to function calls and due to the creation of intermediate data structures.

We eliminate this overhead by viewing parsers as operations over staged generators. We use the Lightweight Modular Staging (LMS) framework for our implementation, thereby, enabling us to target heterogeneous platforms such as GPUs and FPGAs. Performance results for the Nussinov and Zuker folding algorithms indicate a $2.4\times$ speedup over ADPfusion [11], a stream fusion based implementation of ADP, and perform on par with hand-written C for the Nussinov algorithm. This performance improves further when run on GPUs, and scales better to large problem sizes. As a proof of concept, we also target FPGA for the Smith-Waterman algorithm.

We further extend ADP by introducing backtraces which allow the reconstruction of results from a grammar without relying on a particular algebra.

1. Introduction

Parser combinators are an elegant way to write parsers in a high-level, functional language. They provide a useful library for writing grammars and, because they are embedded in a powerful host language, also allow to easily manipulate parse results.

Popular parser combinator implementations, such as Parsec [18] or Scala parser combinators [20], often work in a top-down, recursive-descent manner on a subset of context-free grammars ($LL(k)$). These are usually used for specifying programming languages and structured data formats (JSON, XML, etc.). An important characteristic of such parsers is that they are designed to parse non-ambiguous grammars: a parser returns at most one result per input.

Parser combinators can also be implemented for grammars which are ambiguous. Instead of a single result per input, a parser may produce a list of possible results. To avoid an exponential number of solutions, these lists can be aggregated at each step with respect to a cost function (usually to one element). Memoizing these intermediate results is the key idea behind dynamic programming (DP). This intuition is formalized in the Algebraic Dynamic Programming (ADP)[8] framework: dynamic programs on sequences can be described using a grammar.

Compared to the classic, textbook description of dynamic programs as recurrence relations on matrices, using a grammar avoids possible errors made when specifying index offsets. For problems in bio-informatics involving sequence alignment or sequence folding, using grammars is an elegant way to express the problem, while staying close to its structure [13].

Both implementations of parser combinators are very similar in that they abstract over monadic operations on `Option`[1] (recursive-descent) or `List` (dynamic programming).

Need for Speed

A naive implementation of parser combinators is not efficient; a lot of intermediate `Options` and `Lists` are created. Moreover, combining parsers amounts to chaining function calls, creating overhead due to closure creation. Optimizing closures is particularly difficult in impure functional languages such as Scala.

In the context of dynamic programming (DP), these performance issues have forced researchers to either limit themselves to using ADP as a theoretical framework (and implementing recurrence relations from the description by hand) [13], or to develop a stand-alone compiler, such as the Bellman's GAP compiler [24].

Dynamic programs are highly prevalent in the field of biology; much research revolves around aligning or folding sequences. It is essential that such programs run as efficiently as possible; indeed, it is not uncommon that specialized hardware such as GPUs or FPGAs are used to optimize performance [6, 12].

Fortunately, dynamic programs on sequences (which can be described in ADP) have a very particular structure which allows to process them on GPUs, by parallelizing computation along the DP matrix diagonal. This parallelization strategy is also applicable when generating custom hardware solutions, such as those employing FPGAs [12, 29].

An important area in functional programming research is the development of compiler technology which bridges the gap between high-level abstractions and code that runs without the overhead, "abstraction without regret" [22]. Recent work, ADPfusion [11], makes use of such technology, namely stream fusion [4], to get rid of intermediate data structures. In this work, we aim to bridge the gap for parser combinators.

[1] also known as the `Maybe` monad

Eliminating intermediate data structures

Parser combinators can also be viewed as monadic operations over generators [15, 19]. Generators represent computation, and are evaluated by applying a consumer function, in a manner reminiscent of continuation passing style. No intermediate data structures are created as a result. This producer-consumer view of parser combinators is common in the context of iteratees and generators [14–16]. In fact, our implementation of generators is very similar to the simple generators of Kiselyov et al [15].

Solving dynamic programming ultimately involves reconstructing the solution corresponding to an optimal cost, which can be done by backtracking. To do that, the ADP framework defines a concept of algebra cross-product. We propose *backtraces* that live in a grammar and are algebra-agnostic.

Eliminating object/functional overhead

We eliminate object/functional overhead by staging our parser-combinator implementation in the LMS framework [22, 23]. LMS is a framework for developing performance-oriented domain-specific languages (DSLs) in Scala. It takes a closed-world assumption to DSL development, where a DSL developer stages language constructs and libraries as needed, such that the staged code can be generated into efficient code which is stripped of the abstractions used to write the DSL program.

Through LMS, we are able to generate code for heterogeneous plaftorms; we generate plain C, CUDA. We also target FPGAs by generating code for the MMAlpha toolchain [3].

Overview

We make the following contributions:

- We use a combination of generators and staging to eliminate intermediate data structures and object/ functional overhead in parser combinators, geared towards dynamic programming.
- We formalize a theoretically more efficient backtracking algorithm in the context of ADP, and generalize to tackle multiple backtraces.
- We target a heterogeneous code generation platform. We generate C and CUDA code that runs efficiently. We also generate FPGA code as a proof of concept, by targeting the MMAlpha language.

We start by reviewing parser combinators, and their dynamic programming equivalent in ADP in section 2. We then show how a combination of generators and staging allows us to eliminate intermediate data structures and generate efficient code in section 3. We explain our the backtrace algorithm, and how it applies to the retrieval of multiple results as well in section 4. In sections 5 and 6 we describe how to efficiently parallelize dynamic programming for GPUs and FPGAs. Finally, we present our results in section 7. We present related work is section 8 and conclude in section 9.

A short primer on Scala

Code snippets presented in this paper are written in Scala; we briefly introduce a few concepts to help the reader:

- Scala is both an object-oriented and functional language. Hence, functions are encoded as instances of the `FunctionN` class (`N` being the number of arguments). A function application calls the `apply` method on an instance of this class.
- Scala supports for-comprehensions on collections. These are desugared into `map`, `flatMap`, `filter` and `foreach` method calls. For instance:

```
for( xs <- ls1; x <- xs if (x%2 != 0) ) yield x
```

  is desugared to

```
ls1.flatMap{ xs => xs.filter{ x => x%2 != 0 } }
```

  where `ls1` is a list of list of integers from which we retrieve all odd numbers.

2. Parser Combinators

Parser combinators are usually viewed as functions that operate on an input and return an optional parse result. The simplest representation of a parser is a function that takes a start position (integer), and returns an `Option` containing both the parse result and the next position to continue parsing from:

```
class Parser[T] extends (Int => Option[(T,Int)]){ self =>..}
```

Given the above representation, we can define combinators which allow to construct more complex parsers :

```
def ~ [U](p : Parser[U]) = Parser[(T,U)] {
  pos: Int => self(pos).flatMap {
    case (x, pos1) => p(rest) map {
      case(y, pos2) => ((x,y), pos2))
    }
  }
}
def |[T](p : Parser[T]) = Parser[T] {
  pos: Int => self(pos) orElse p(pos)
}
def map[U](f: T => U) = Parser[U] {
  pos: Int => self(pos) map {
    case (x, pos1) => (f(x), pos1)
  }
}
def filter[T](f: T => Boolean) = Parser[T] {
  pos: Int => self(pos) filter {
    case (x, pos1) => p(x)
  }
}
```

The ~ function above concatenates two parsers by parsing `p` if it succeeds on `self`. The | combinator expresses alternatives. The `map` function applies a function on the result of a parser, while `filter` creates a parser that succeeds if the result satisfies a predicate. We note that parsers can essentially be seen as operations over the `Option` monad.

To use this framework, we need some parsers which accept elementary input, so that we can combine these for creating more complex parsers. We show a basic parser for digits, and how to use it for creating a parser for arithmetic expressions below:

```
def rep[T](p :Parser[T]) = Parser[List[T]]{
  pos: Int => p(pos) match {
    case None => List()
    case Some((x, pos2)) => x::rep(pos2)
  }
}
def digit(in:Input) = Parser{ pos =>
  if(in(i).isDigit)) Some((in(i), i+1)) else None
def expr(in:Input) =
  term(in) ~ rep("+" ~ term(in) | "-" ~ term(in))
def term(in:Input) =
  factor(in) ~ rep("*" ~ factor(in) | "/" ~ factor(in))
def factor = digit(in)| "(" ~ expr ~ ")"
```

With the above parser combinators, we parse the input in a recursive-descent manner, recognizing and constructing our parse result as we see the input.

Parsers producing Lists

Recursive descent parser combinators are restricted to $LL(k)$-type grammars: they do not support left recursion, nor ambiguity. Hence, for a given input that satisfies such grammars, there is at most one possible parse tree. Other parsing techniques like the CYK algorithm allow parsing of context free grammars. Instead of parsing the input as a stream, we construct the best parse for a subsequence, and combine parses of subsequences to create a parse tree for the complete sequence.

Parser combinators can be built for such grammars as well. We now have functions from a subsequence (represented by its start and end indices (i, j)) to a list of all possible parses for this subsequence. The corresponding combinators follow naturally by applying functions similar to those for recursive descent parsers, but this time on the `List` monad:

```
class Parser[T] extends (Int,Int) => List[T] { self =>
  def ~ [U](that: Parser[U]) = Parser[(T,U)] {
    (i,j) => if(i<j) {
      for(k <- i until j;
          x <- self(i,j);
          y <- that(k,j)
      ) yield (x,y)
    } else List()
  }
  def | (that: Parser[T]) = Parser[T] {
    (i,j) => self(i,j) ++ that(i,j)
  }
  def map[U](f: T => U) = Parser[U] {
    (i,j) => self(i,j) map f
  }
  def aggregate(h: List[T] => List[T]) = Parser[T] {
    (i,j) => h(self(i,j))
  }
  def filter (p: (Int,Int) => Boolean) = Parser[T] {
    (i,j) => if (p(i,j)) self(i,j) else List()
  }
}
def el(in: Input) = Parser[T] {
  (i,j) => if(i+1 == j) List(in(i)) else List()
}
```

Here, the ~ function combines *all* solutions from the left and right parser by performing the cross product over its both lists of results. The | combinator expresses alternatives, which now amounts to concatenation (parse results from the left and right parsers are both valid). The `map` combinator is analogous to the one in recursive-descent parsers.

The above parsers yield multiple results, and `aggregate` helps to reduce their number by applying the `h` function. It can also encode recursive-descent parsers' filtering semantics. The `filter` combinator here operates like a guard, preventing computation of undesired results. We also show the `el` elementary parser which parses a single element from the input sequence.

Additionally, to avoid exponential running time over the sequence, we memoize intermediate results with the `tabulate` combinator:

```
def tabulate[T](m:Array[Array[T]]) = new Parser[T] {
  (i,j) => if(m(i)(j) == null) m(i)(j) = self(i,j)
           m(i)(j)
}
```

With these combinators, we have a framework for dynamic programming over sequences, formalized by Algebraic Dynamic Programming (ADP).

Algebraic Dynamic Programming (ADP)

Dynamic programming (DP) is an algorithmic technique to solve optimization problems where optimal solutions are constructed from optimal sub-solutions. Intermediate results are tabulated (memoized in a matrix) to reduce running time from exponential to polynomial. Traditionally, a DP problem is described in terms of matrices and recurrences over their elements.

As a running example, consider the matrix-chain multiplication problem. Given a sequence of matrices m_i of appropriate dimensions, we would like to determine the order of multiplication that minimizes the overall number of computations. Let $M[i, j]$ denotes the optimal cost of multiplying matrices i to j, this problem is represented by the following recurrence relation:

$$M[i,j] = 0 \quad \text{if } i = j, \text{ else}$$
$$M[i,j] = \min_{i \le k < j} \left\{ \begin{array}{l} M[i,k] + M[k+1,j] \\ +\text{rows}(m_i) \cdot \text{cols}(m_k) \cdot \text{cols}(m_j) \end{array} \right\}$$

ADP[8] is a framework for representing dynamic programming problems on sequences using grammars, augmented by a scoring algebra and an optimization function to retain relevant parse results. It formalizes traditional DP definitions by distinguishing between the structure of a problem (parsing grammar) and its evaluation (scoring algebra and optimization function).

Formally, ADP decomposes into three components:

- A signature Σ that defines the input alphabet $\mathcal{A}$, a sort symbol $\mathcal{S}$ and a family of operators $\circ : s_1, ..., s_k \rightarrow S$ where each s_i is either $\mathcal{S}$ or $\mathcal{A}$ and an aggregation function $h : List[\mathcal{S}] \rightarrow List[\mathcal{S}]$.
- An algebra, that is the concrete instantiation of a signature. The sort symbol is concrete and the signature functions have implementations.
- A grammar $\mathcal{G}$ over Σ and $\mathcal{A}$ that operate on a string $\mathcal{A}^*$.

The main benefits of ADP are:

- abstraction from indices, a common source of errors in DP.
- the possibility to specify an arbitrary aggregation function; we can choose to yield a single optimal result, multiple results, or even all results.
- the possibility to define multiple algebras for the same grammar.

As an example, the matrix chain multiplication problem can be expressed as:

```
trait MatMultSig {
  type A, S
  def single(a: A): S
  def mult(l: S, r: S): S
  def h(xs: List[S]): List[S]
}

trait CostAlgebra extends MatMultSig {
  type A = (Int,Int); type S = (Int,Int,Int)
  def single(a: A) = (a._1, 0, a._2)
  def mult(l:S, r:S) = (l._1, l._2+r._2+l._1*l._3*r._3, r._3)
  def h(xs: List[S]) = List(xs.minBy(_._2))
}

trait PrettyPrint extends MatMultSig {
  type A = (Int,Int); type S = String
  def single(a: A) = "|" + a._1 + "x" + a._2 + "|"
  def mult (l:S,r:s) = "(" + l + "*" + r + ")"
  def h(xs: List[S]) = xs
}

trait MatMultGrammar extends Parsers with MatMultSig {
```

```
  val grammar = tabulate((
    el map single
  | (grammar ~ grammar) map mult
  ) aggregate h)
}

object MatMult extends MatMultGrammar with CostAlgebra
```

The `MatMultSig` trait defines the abstract signature for the problem. We define two operations, `single` representing a single matrix and `mult` representing the multiplication of two matrices. Note that the arguments of `mult` are of type `S`. The `CostAlgebra` defines how to compute the cost for each operation, and corresponds to the recurrence relation seen above; its implementation of the aggregation function is a minimization. The `PrettyPrint` algebra defines a visualization for the problem. The aggregation function is the identity function, and we can see all possible tree constructions for chaining matrices. `MatMultGrammar` gives the the structure of the problem: either we encounter a single matrix (and apply the `single` function), or we encounter two adjacent matrices, which we can multiply. `MatMult` ties an algebra to a grammar using mixin composition.

ADP also introduces the concept of algebra cross-product, to combine multiple (usually two) algebras. Signature functions from all algebras are applied when parsing the grammar, and one can choose which aggregation function to apply. In the above example, we can combine cost computation with pretty-printing (choosing minimization as the aggregation function) to yield the best cost along with its visualization. One shortcoming of this approach is that we construct all the intermediate pretty-printed strings, thereby effectively consuming $O(n^3)$ storage space.

Efficient parsing

In the context of dynamic programming, top-level parsers must be tabulated for reducing algorithmic complexity. Moreover, as we want to generate efficient code, we fix a bottom-up order for computing elements of the cost matrix, along the diagonal:

```
(0 until n).foreach { d=>
  (0 until n-d).foreach { i=>
        val j = i+d
        p(i,j) //call parser between i and j
    }
  }
```

As a result, we avoid recursive calls to parsers, because for every matrix cell $M[i, j]$, its result depends on cells that have been previously computed; a simple look-up is sufficient.

3. Staged Generators

Generators

`Lists`, or `Options`, are intensional data representations (data *structures*), where operations involve inspecting these structures. Parser combinators reside at a higher abstraction; they do not define the structure of the data, but specify the operations to perform on them. Therefore, an extensional data representation can be used. Generators [15, 19] provide a way to compose over operations on data created by a producer. A generator can be represented as a function that takes a closure and returns unit:

```
class Generator[T] extends ((T => Unit) => Unit){self => ...}
```

The closure `T => Unit` can be seen as the inner body of a loop which consumes elements of type `T`. It can be interpreted as a delimited continuation that we pass to the generator. We give basic composition functions for generators:

```
def map[U](g: T => U) = Generator[U] {
  f: (U => Unit) => self{ x: T => f(g(x)) }
}
def filter(p: T => Boolean) = Generator[T] {
  f: (T => Unit) => self{ x:T => if(p(x)) f(x) }
}
def flatMap[U](g: T => U) = Generator[U] {
  f: (U => Unit) => self{ x:T => g(x)(f) }
}
def ++(that: Generator[T]) = Generator[T] {
  f: (T => Unit) = { self(f); that(f) }
}
def fold[U](z:U)(f:(U,T) => U): U = {
  var s = z
  this.apply { x: T =>  s = f(s,x) }
  s
}
```

Note that a generator is a monad, with the `flatMap` function being the bind operator. Composing generators is reminiscent of programming in continuation passing style; each composition is a new continuation to be applied when the generator is finally called. The `fold` function applies the generator, triggering all computation and yielding the result. For example, summing all odd numbers over a range can be done as follows:

```
def range(start: Int, end: Int) = Generator[Int]{
  f: (Int => Unit) => { for(i <- start until end) f(i) }
}
//apply the generator
range(1,10).filter(x => x%2 != 0).fold(0) {
  case (acc,x) => acc+x
}
```

Using the above, we can change our implementation of parsers from section 2 to use generators:

```
// Recursive descent parsers
class Parser[T] extends (Int => Generator[(T,Int)])
// ADP
class Parser[T] extends ((Int,Int) => Generator[T])
```

While previously, calling a parser with an integer (or pair of integer for ADP parsers) would have yielded a result, we now need to explicitly invoke the `fold` function:

```
def myParser = Parser[T]{ i => ...}
val z = ... // base value of type T
myParser(0).fold(z){case (acc,z) => z}
```

Staging

Instead of yielding lists or options, our parsers now yield functions (to be invoked with a continuation). Though intermediate data structures are eliminated, we still face the overhead of creating a new function object for each combinator invocation. We potentially create many function objects that are invoked only once. In this section, we show how we can use Lightweight Modular Staging (LMS), to eliminate this overhead by generating the functions inline, thereby producing efficient code.

Lightweight Modular Staging (LMS)

LMS is a framework for developing embedded DSLs in Scala, that provides a library for domain-specific optimizations and code generation. A program written in LMS is compiled in two stages: at runtime, the LMS code generator outputs a second stage program which is an optimized DSL program that is then compiled and executed. To differentiate between the code that is executed at staging time and which code is generated, LMS wraps types in `Rep[?]`

types: an expression of type `T` will be executed at staging time, while an expression of type `Rep[T]` will be generated. Consider the following functions:

```
def add1(a: Int, b: Int) = unit( a + b )
def add2(a: Rep[Int], b: Rep[Int]) = a + b
```

The `add1` function gets executed during code generation, producing a constant in the generated code, while `add2` represents a computation that will yield a value of integer type, and is represented as a node: `Add(a,b)`.

A DSL is defined in LMS through 3 components:

- An interface: a signature of the operations the DSL user will have access to when writing a DSL program.
- An intermediate representation (IR): similarly to the `Add` node above, domain-specific operations (requiring particular optimizations) are represented by IR nodes.
- A code generator: ultimately, the IR nodes are transformed into optimized executable code.

The core LMS library contains interfaces, IR nodes and code generation for many common programming constructs, such as conditionals, boolean expressions, arithmetic expressions and array operations, among others. These can therefore be used out of the box.

Functions

In the interest of performance, it is better to generate code inline whenever a function is called in a DSL program; a DSL using higher-order abstractions and allowing anonymous functions can greatly benefit from this. Rather than using `Rep[T => U]`, which represents a staged function, it is better to have an *unstaged* function on *staged* types : `Rep[T] => Rep[U]`. A call to this function is executed at staging time, producing a block (of nodes) at the call site; this corresponds exactly to inlining during code generation.

To make things clearer, consider the range example from the previous section:

```
def fold[U](z: Rep[U])(f:(Rep[U],Rep[T]) => Rep[U]): Rep[U] = {
  var s = z
  this.apply{
    x: Rep[T] =>  s = f(s,x)
  }
  s
}

range(1, 10).filter(x: Rep[Int] => x%2 != 0)
.fold(0){(acc,x) => acc + x}
```

We show the LMS version of the code above, with `Rep` types as appropriate. The `filter` function above takes a function from a staged integer to a staged boolean, as does the function passed to fold, and the continuation applied to the generator. The generated code looks like the imperative equivalent:

```
var s = 0
var i = 0
while(i < 10) {
  if(i % 2 != 0)
    s = s + i
}
```

We notice that the variable `s` is a staged variable; this is exactly what we want to generate. We maintain a functional/pure interface for the DSL user, but the staged implementation of these functions is imperative in order to generate the more efficient version of the code.

The above code gives us a blueprint for implementing staged generators, and therefore staged parsers: functions of type `T => U` are lifted to `Rep[T] => Rep[U]`. In particular, user-defined functions, either anonymous or named, need to be similarly converted. This gives the following signatures for generators and parsers:

```
class Generator[T] extends ((Rep[T] => Rep[Unit]) => Rep[Unit])
// recursive-descent
class Parser[T] extends Rep[Int] => Generator[(T,Int)]
//ADP
class Parser[T] extends (Rep[Int], Rep[Int]) => Generator[T]
```

Note how the signature of ADP parsers takes a function of two staged integers, and not a staged tuple of integers.

Recursion

Even simple parsers as seen in section 2, for arithmetic expressions (recursive-descent) or matrix multiplication (dynamic programming) are defined recursively. Therefore, their implementation needs to handle recursion as well. This raises the question whether we should be generating staged functions after all. For our dynamic programming implementation, we do not need staged functions because:

- We tabulate every top-level production, as this produces the best running time.
- In the previous section, we imposed the order of evaluation such that the computation of a cell depends solely on results of previously computed cells. As mentioned before (and detailed in section 5), this helps parallelize efficiently.

For recursive-descent parsers, recursion, and therefore staged functions, are necessary. LMS has a lifting function called `doLambda`:

```
def doLambda[T,U](f: Rep[T] => Rep[U]): Rep[T=>U] = {...}
```

When a function is lifted, LMS analyzes its code and recognizes recursive calls by comparing their serialized representations to the caller, and instead of unfolding, generates a call to the original function. For parsers, we can use this technique as well. Consider a very simple number parser:

```
def number(in:Rep[Input]): Parser[String] =
 (digit ~ opt(number))
  .map {x: Rep[(Char,String)] => x._1 + x._2}
```

The `opt` combinator returns the result of its inner parser or succeeds with an empty parse. The expanded definition, handling recursion, looks as follows:

```
def number(in:Rep[Input]): Parser[String] =
  Parser{ i: Rep[Int] =>
    Generator{ f: (Rep[(String, Int)] => Rep[Unit]) =>
      def numRec = doLambda{ t: Rep[(String,Int)] =>
        digit(in)(t._2) { x: Rep[(Char,Int)] =>
          val param = (t._1 + x._1, x._2)
          f(param)
          numRec(param)
        }
      }
      numRec("",i)
    }
  }
```

We define a recursive function `numRec` which *applies* the closure `f` to the digit parser, and then calls itself recursively; its signature is `Rep[(String, Int) => Unit]`. We notice that we are in fact generating a lifted version of the consumer function passed to the generator. This is indeed what we want; lifting parsers or generators would be counter-productive as we would end up generating code with the overhead issues mentioned above. For indirect and more

complex recursion, we would need to analyse the structure of the parser more closely, somewhat similarly to ANTLR [21].

4. Computing the backtrace

The primary objective of dynamic programming is to construct an optimal result. However, as seen in section 2, optimality (scoring) and result construction can be determined by different algebras. Using our running example, estimating the number of scalar multiplications to multiply matrices is different from actually computing the resulting matrix.

The concept of product algebra (section 2) enables computing the score and result at the same time in two different storage matrices. Bellman's GAP [24] and ADPfusion [11] leverage this idea and compute the result in a backtracking phase, where the scoring algebra is reused to compute dependencies, and intermediate results are constructed only when necessary. This has the benefits of reducing the memory usage and of decreasing the result construction time by $O(n)$, since only relevant sub-results are constructed.

The concern with computing unnecessary intermediate results is that it can be prohibitively expensive (ex: matrix multiplication), which is why the second approach is preferable.

We introduce *backtraces* that are algebra-agnostic and represent a construction tree that lives within a particular grammar. The key idea behind this concept is to describe the construction in terms of the grammar rules that are applied, including alternatives and concatenation indices.

Backtrace and grammar

In order to uniquely identify a production of the grammar, we need to know which rule was applied and on which intermediate results. Notice that the evaluation of a tabulation can be represented as a tree of parsers with tabulations or terminals as leaves. Most of the parsers are 1:1 transformations (tabulate, filter, mapping, terminals), the only concerns are alternatives and concatenations. We uniquely identify them as follows:

1. Disambiguate alternatives: we assign a unique *sub-rule* identifier (r_s) to all tabulation alternative branches. The identifier defines the evaluation path that has been taken. Each identifier r_s refers to an evaluation tree s that does not contain alternative parsers.
2. In each sub-rule evaluation tree s, the concatenation indices[2] can be stored in an array of fixed size $(k_1, k_2, ...)$.
3. Let m be the maximum number of concatenations among all sub-rules of a tabulation T. We define the backtrack information as $bt_T := (r_s, (k_1, k_2, ...k_m))$, where some k_i can be unspecified. Notice that this information fits in a fixed-size array of $m+1$ integers (for each matrix element). To reduce memory consumption, notice that r_s and m are bounded by the grammar and k_i by the input sequence length. Also if a concatenation has one side of fixed length, we can reconstruct it from the evaluation tree s instead of storing its index in bt_T. The backtrack information bt_T can be stored either within the associated scoring matrix or in an independent (write-only) matrix.

Indexing alternative-free tabulation rules and computing the maximal number of concatenations per tabulation is done in an analysis phase on the grammar (algebra is not involved).

[2] Position at which concatenation occurs; when parsing the sequence (i, j) we have $i \leq k \leq j$.

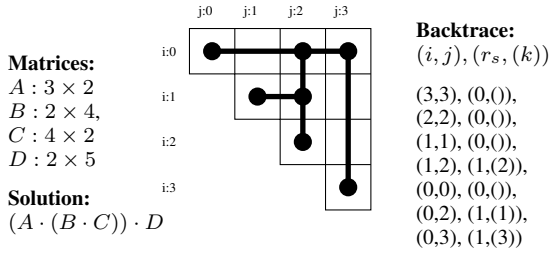

Figure 1. Backtrace of a matrix chain multiplication problem

Backtrace construction

Once the matrices have been filled (computation phase), we focus on backtrace construction (backtracking phase):

1. To backtrack from the cell at position (i, j), we first recover s, the sub-rule that applies (from r_s stored in bt_T). Then we execute the rule backward by providing concatenation indices at appropriate places and obtain a list of the intermediate elements (terminals or tabulations) necessary to construct the element at (i, j).
2. Whenever we obtain new intermediate tabulation elements, we apply the backtracking procedure recursively until all relevant bt_T have been processed. All the traversed elements are then stored in the *backtrace* which is a list of pairs of position (defining a subsequence) and the associated backtrack information $((i, j), bt_T)$. We then reverse the list so that for each element, its dependencies appear earlier in the list. A concrete backtrace example is presented in figure 1.

The backtracking strategy depends on whether we store a single result or multiple results (co-optimal, k-best):

- Single result: we do not need the scoring matrices during the backtrace phase. Indeed, it is guaranteed by construction that the unique retrieved element contributes to the result. Additionally, the absence of result can be denoted by a special rule number.
- Multiple results: since multiple previous elements are retrieved, we need not only to correctly pick those contributing to the desired result, but also to ensure that multiple results with the same score generate different traces. To generate multiple distinct traces we take into account at each step the multiplicity of each score (k) and multiplicity of distinct solutions paths (r):
  - If $r = 1$ (all solutions have the same backtrack), continue with the same degree k of multiplicity.
  - If $k \leq r$ (more paths than needed): explore the k first paths with multiplicity 1 and safely ignore the other (as we only need k distinct results).
  - If $k > r$: explore all paths with multiplicity $k-r+1$ because each branch may produce only one solution and we don't know ahead of time which path provides multiple solutions. Finally, retain only the k-best solutions.

Backtrace utilization

The backtrace from previous phase can now be applied sequentially on the same grammar to any algebra: for each element, we apply the construction rules described and store intermediate results (previous elements are guaranteed to be constructed beforehand by the trace order). The final result can be retrieved from the last matrix cell.

Notice that since this matrix is very sparse, one might consider using a hash table instead to store intermediate results. Additionally, each produced element is consumed exactly once, thereby opening

possibilities to reduce memory consumption. Finally, since the result is constructed bottom-up, opportunistic parallelization can be explored: because the backtrace ordering is only partial, elements that have no ordering relation among each other can be constructed in parallel.

Complexity analysis

The flexibility offered by the ADP formalism makes running time analysis difficult and possibly input-dependent (example: co-optimal aggregation with all/no scores equivalent), hence we restrict our analysis to k-best aggregation functions (on two-dimensional dynamic programming matrices).

Let t be the number of tabulations of the grammar and c the maximal number of unbounded concatenation per tabulation of the grammar[3]. We estimate the time and space complexity of each phase for k best results:

1. Filling matrices: The scoring matrix is of the same size as backtracking matrix and consumes $O(t \cdot n^2 \cdot k)$ memory. Computing one scoring matrix element is $O(n^c \cdot k)$ (because there is k comparison against previous results), hence the total complexity of filling the t matrices is $O(t \cdot n^{2+c} \cdot k)$. Since the backtracking information is attached to every element, a little overhead is added to the computation but complexity remains unchanged.
2. Constructing the backtrace: Since at every backtrack step we reduce by at least 1 in one dimension the problem, the backtrace length is $\leq 2n \cdot t$ (2 dimensions).
   To backtrack 1 element, one must pick the correct construction among the k^c possibilities provided by rule application with specific indices. Hence the running time is $O(k^c \cdot 2n \cdot t)$.
3. Computing a result: to process one trace, since used intermediate results can be discarded, worst memory usage happens by storing all leaves (maximized when tree is binary), hence is $O(n)$ intermediate results (independent of t since results are not reused during construction). The running time is bounded by the trace length hence running time is $O(2t \cdot n)$ steps.

The careful reader would have noticed that the space and time units differ from steps (1,2) and (3) because a different algebra is used. As a concrete example, in matrix multiplication, with one single result, the running time of the steps 2+3 is $O(n)$, which improves over the theoretical complexity of approaches such as ADPfusion [11], but this result must be weighted by the overhead in the matrix filling phase, especially if the cost computation is simple.

Benefits

To summarize, our backtracking technique brings the following benefits:

- From a theoretical perspective, we clearly separate the concerns between scoring and computation algebrae
- We reduce the complexity of the backtrace phase. This is relevant because we can use only one thread to do backtrack, compared to parallel matrix filling.
- We enable portability across devices: for example, we could compute a trace on the GPU and construct the corresponding result with the CPU.
- We expose possible parallelism (work stealing) in the result construction phase.

[3] Values of $0 \leq c \leq 2$ are most common: $c = 0$ in Smith-Waterman, $c = 1$ in matrix chain multiplication, ...

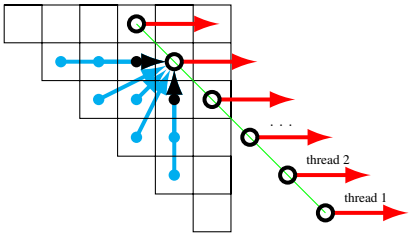

Figure 2. Threads progress jointly along matrix diagonal. Value dependencies are only immediately preceding matrix cells.

5. Parallelization of DP

Top-down to bottom-up evaluation

Parallelization involves finding independent computation paths that can be run in parallel. As mentioned in section 2, we need to use bottom-up evaluation to expose parallelism.

To evaluate the grammar in bottom-up order, we need a (partial) order between elements such that execution order respects the value dependencies. We can then evaluate independent blocks in parallel as described by Cartey et al [1]. Since DP recurrences are constructed along the matrix dimensions, we can make the following observations:

- For one cell, dependencies are along the line, the column of the matrix and possibly in diagonal. By induction, it suffices that immediately preceding elements in the column and row are valid for all other dependencies to be satisfied. Hence all elements on a same diagonal can be computed in parallel (Figure 2).
- In the presence of multiple tabulated parsers, we can compute all tabulations at the same position in parallel, given a correct order between tabulations.

The second observation is made by assuming a correct grammar (i.e. no cyclic calls resulting in infinite loop in top-down parsing): the partial order $\prec$ between tabulation is given by $A \prec B$ if $B(i,j) = f(A(i,j), ...)$ (access at same position).

Since bottom-up parsing is more regular than top-down, it might end-up computing matrix cells that would have been ignored by top-down parsing (for instance if grammar is LL(k)). Yield and dimension analysis [9] help reducing underlying matrices, thereby mitigating this problem. Ultimately, filters can be placed within the grammar to avoid computing these unnecessary results.

GPU-specific parallelization

In many graphic cards applications, the computational power outperforms the memory bandwidth such that memory is often the bottleneck. This is true in particular in DP where the computation of the optimal cost is usually simple compared to the number of dependencies to other matrix elements.

To improve performance, accessed memory needs to be coalesced among threads[4]. Since elements computed in parallel are along one diagonal, the underlying matrix needs to be stored diagonal-by-diagonal.

To respect the dependency order, threads progress along the lines of the matrix (dependency along the line is always valid) and synchro-

[4] Because the memory bus is larger than individual elements, multiple transfers are needed to access non contiguous elements. Also memory chip needs an additional delay to charge a different row before accessing its elements.

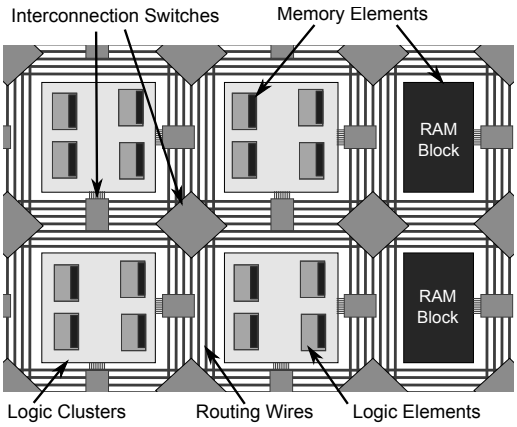

Figure 3. Architecture of an "island style" FPGA.

nize with the previous thread/line (validate column dependency) before moving to the next diagonal. GPU barriers are discussed in details in [28]; we leverage these ideas to use active waiting of the previous thread[5] between computation of diagonals (see Figure 2).

6. FPGA

The problem sizes on which dynamic programming is applied, in areas like bioinformatics, is very large and there is a great demand for high-performance here. This has inspired research on alternate computational platforms that offer higher performance on dynamic programming kernels including *Field Programmable Gate Arrays* (FPGAs) [12].

FPGAs are integrated circuits composed of configurable logic elements, memory and configurable routing fabric. Figure 3 shows the organization of such an FPGA which uses the "island-style" architecture, the most common architecture found in FPGAs today. As shown in the figure, we have clusters of logic elements and memory elements that are placed within a configurable routing fabric. The routing fabric is composed of wires and interconnection units. The user can configure these interconnection units to make dedicated connections between different logic or memory elements.

FPGAs for Dynamic Programming

FPGAs enable users to fashion dedicated datapaths that are ideal to meet the computational requirements of the target application. Intermediate results can be forwarded directly from the point of production to places where they are consumed. By integrating memory elements directly into the datapaths they can hold intermediate results, thereby, limiting the dependency on centralized storage structures like register files that have limited scalability and negatively impact energy efficiency. The total aggregate bandwidth from all the distributed memories also significantly exceeds that of dedicated parallel units like GPUs. For dynamic programming applications, which are rich in parallelism and data-reuse, we can achieve speed-ups that far outpace a processor based architecture.

The parallelization pattern for FPGAs is similar to that for GPUs, on the diagonal of the scoring matrix; this is a regular pattern for consuming intermediate results. Hence, we can design architectures that will directly forward these results to the appropriate unit where it is needed next. Systolic arrays [17] are such an architecture

[5] Practically we synchronize between warps (32 threads), as threads within a warp are scheduled synchronously.

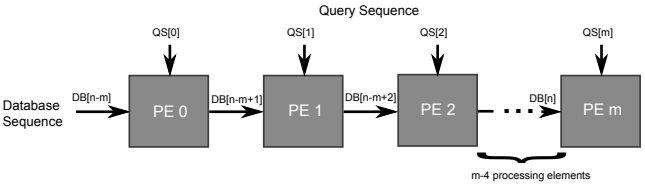

Figure 4. Organization of a linear systolic array having $m + 1$ processing elements.

that are used to implement dynamic programming algorithms on FPGAs.

Systolic Arrays for Dynamic Programming

Systolic arrays are highly regular structures that are composed of simple processing elements that are locally connected and can operate in parallel. Figure 4 shows a linear systolic array used for sequence alignment. Here, each processing element accepts data from their input ports, performs computation and forwards it to the element on the right. Since data is directly forwarded among the cells, it is immediately evident that this architecture can significantly cut down on external memory access and at the same time provide massive parallelism.

Parser Combinators to Systolic Arrays

ADP's parser combinators can be systematically transformed into recurrence relations [8]. In the previous sections, we generated loops over these recurrence equations for CPUs and GPUs. For generating hardware, we generate recurrence relations directly to MMAlpha [3]. MMAlpha is a tool chain that enables users to obtain systolic array architectures from recurrence equations.

7. Evaluation

Our benchmarking environment for CPU and GPU consists of a dual Intel Xeon X5550 with 96GB of RAM with an Nvidia Tesla C2050 (3Gb RAM) graphic card. We measure running time of two different bio-informatics algorithms that are used to fold RNA sequences over themselves. RNA folding consists of identifying the matching amino-acid (base)pairs that produce 2D features such as hairpins, bugles and loops that are naturally formed. The Nussinov algorithm is fairly simple as it only tries to maximizes the number of matching base pairs.

```
val s:Tabulate = tabulate("s",(
    empty                          ^^ nil
  | el ~ s                         ^^ left
  | s ~ el                         ^^ right
  | (el ~ s ~ el filter basePair) ^^ pair
  | s ~ s                          ^^ split
  ) aggregate h)
```

Listing 1. Nussinov78 grammar

The Zuker algorithm is more complex as it minimizes the free energy of the folding. Computing the free energy involves hundreds of coefficients based on physical measurements stored in tables, hence these lookups generate much more memory traffic to compute one matrix element. We refrain from detailing this grammar but make it available for inspection[6]; it consists of 4 tabulations with 15 productions and 13 scoring functions. The Zuker algorithm

[6] `https://github.com/manojo/lamp-dp-mt/blob/master/src/main/scala/v4/examples/Zuker.scala`

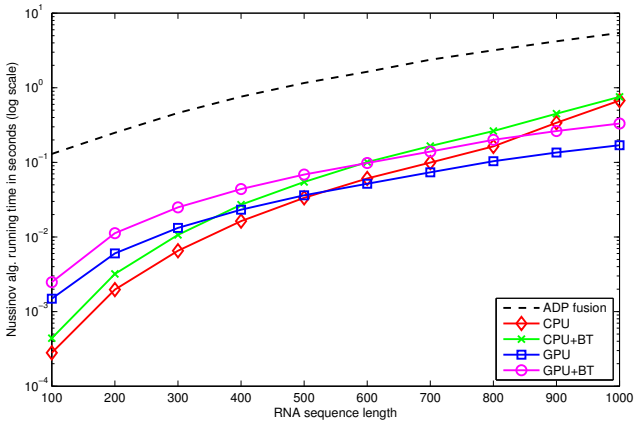

Figure 5. Nussinov algorithm running time

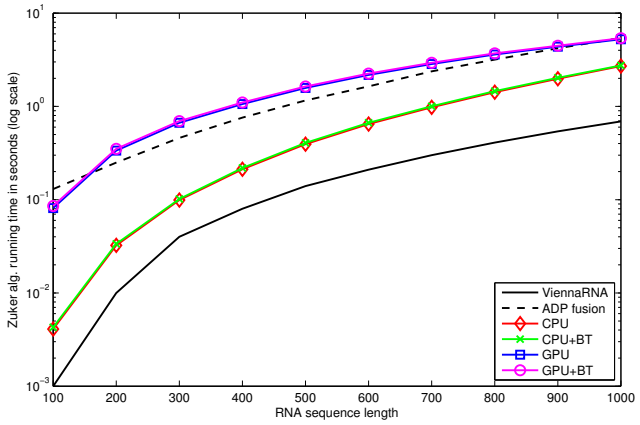

Figure 6. Zuker algorithm running time

has a complexity of $O(n^4)$ but it is commonly accepted to bound some productions of large structures because they are very rare in the nature, thereby reducing complexity to $O(n^3)$ with a large constant factor.

Nussinov algorithm

In figure 5 we compare the running time of ADPfusion[11] with four variants of our generated code:

- ADPfusion is written in Haskell and serves as baseline for our benchmarks, as it is offers grammar-based dynamic programming with performance close to hand-optimized C. Also for the purpose of these benchmarks, we disabled the backtracking procedure in ADPfusion.
- The CPU versions run on a single thread, the plain version runs without backtrack (the backtrack matrix is not filled). The CPU+BT version fills the backtrack matrix and runs the procedure to construct the backtrace.
- Similarly, the CUDA version that runs on the GPU is presented both with and without the backtrack.

The CPU version without backtrack has similar performance to hand-optimized C code. Indeed, inspection of the generated code reveals very close implementations. For sequences larger than 800 elements, we see that the additional overhead incurred by the CPU could be attributed to cache overflow. From these numbers, parallelizing on the GPU is worth only for significant sequence sizes. Also since the costing algorithm is fairly simple, we notice a clear overhead when backtracking is enabled.

Zuker algorithm

In figure 6, we present results obtained for the Zuker algorithm. Once again, we present our results for CPU with and without bactrack, and for GPU, with and without backtrack.

We compare our implementation to ViennaRNA [10], a highly optimized Zuker algorithm implementation written in C. We are still slower than ViennaRNA; they precompute matches for basepairs and stack-pairings for a sequence before launching the forward computation phase, which we do not specify during our code generation.

Compared to Nussinov, we see that lookup tables add significant overhead to the computation, therfore the overhead induced due to backtracking become negligible. The GPUs is slower than CPU as the length of the sequences have not compensated for the overhead yet.

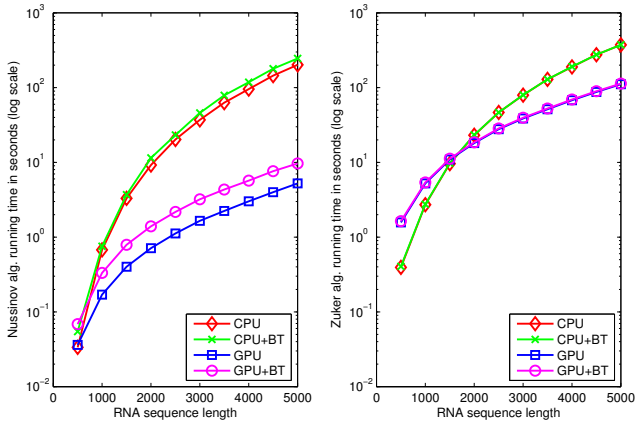

Figure 7. Nussinov and Zuker algorithms scalability

Scalability

To give an intuition of scalability for both GPU and CPU versions, we compare them in an extensive range from 500 to 5000 elements sequences in figure 7. Nussinov algorithm scales better on GPU, offering speedups from $4\times$ to $40\times$ (respectively for 1000 and 5000 elements sequences). Zuker lookup tables hamper GPU performance more significantly, as multiple random memory access are required for each scoring, thereby delivering at best $3.3\times$ speedup versus CPU for this algorithm.

FPGA

As a proof-of-concept, we generated the dual-track Smith-Waterman algorithm to MMAlpha. This further generates a VHDL program. We synthesized this design and also verified the functionality of the design by simulating it using Mentor Graphics' ModelSim simulation tool.

8. Related Work

Language level

Our work is inspired from previous implementations of ADP, such as ADP-Haskell (the original Haskell implementation), and GAPC [24]. The latter is a stand-alone DSL with a full-fledged compiler

stack, and accepts multi-track grammars. GAPC includes optimizations below:

- Grammar soundness checks: removing unnecessary rules, and other analysis on whether the grammar is productive (we do dead-rule elimination)
- Yield-size analysis: analysis of the bounds on nonterminals in the grammar (we do this)
- Table dimension analysis and design (allocate minimal memory for tabulations, not done)
- Backtracing
- Inlining of functions (we get this "for free" using LMS)

GAPC's backtracking mechanism uses a product algebra $A_{\text{scoring}} \times A_{\text{pretty}-\text{print}}$ and computes the matrix using only the scoring algebra. In backtracking phase, it then executes the pretty-print algebra alongside the optimizing algebra, whenever the score stored in the matrix is matched.

More recently, ADPfusion[11], a Haskell-embedded DSL, achieves good performance thanks to compiler technology provided by GHC (stream fusion) [11]. Their backtracking scheme is similar to GAPC (slightly more involved, but also combines different algebrae). Both these backtracking algorithm run in $O(n^2)$ time.

Other language approaches for dynamic programming are StagedDP[26] (programs are expressed using classic recurrences) and Dyna [7] (a logic-programming style language prevalent in the field of NLP/stochastic grammar parsing).

Compiler Technology

To match performance of lower-level implementations, high-level languages require compiler technology. ADPfusion uses Stream Fusion [4] to optimize away intermediate lists and generates tight efficient loops. Cartey et al (Synthesizing Graphics Cards from DSLs) propose an intermediate DSL for recurrence relations, which they analyze and generate GPU programs from [1]. Their approach is more general as they try to infer a parallel schedule from recurrence relations; we leverage our domain-specific knowledge to force diagonal progress.

Parallelization

Much work on parallelizing dynamic programming has focused on the Smith-Waterman sequence alignment problem (CudAlign, CUDASW++). CudAlign matches sequences whose matrix is much larger than the GPU memory size, using a hybrid divide-and-conquer and dynamic programming approach [5]. Serial problems such as Matrix multiplication, Nussinov have also been studied as pure GPU implementations([2, 27]). There also exists a GPU variant of the original ADP combinators [25].

9. Conclusion

In this paper, we have shown that parsers combinators implementations are very similar for both recursive descent parsing and dynamic programming. We then showed how a combination of using generators and staging enabled us to generated efficient code which is stripped of intermediate structures and functions calls. With backtraces, we have formalized an algebra-agnostic representation of dynamic programming solutions in the context of ADP. By using LMS for staging, we have demonstrated that we can target heterogeneous platforms with C and CUDA implementations and MMAlpha as an intermediate language towards FPGA implementation. Finally, our experimental results demonstrate good performance compared to existing CPU implementations.

As future work we could include three optimizations in our current implementation:

- Taking the Zuker algorithm as example, some functions might be extensively used (in Zuker the fact that two bases can be paired), hence this information could be precomputed in an additional matrix and provided at a later stage.
- For algorithm where the overhead of filling the backtrack matrix is not acceptable, it could be possible to compute this information during the backtracing phase, thereby modifying the trade-off between forward and backward phases complexity.
- Finally, we could reduce the memory footprint of our implementation by reducing the size of bounded tabulations wherever appropriate, and possibly pre-computing some of the tabulation indices similarly as in ViennaRNA.

10. Acknowledgments

We gratefully acknowledge Vojin Jovanovic, Sandro Stucki and Nada Amin for insightful discussions, and comments leading to improvements on the paper. We are also grateful to the Stanford PPL team for providing computing resources for our benchmarks. A special thanks to Christian Höner zu Siederdissen for helping us with ADPfusion-related work.

References

[1] L. Cartey, R. Lyngsø, and O. de Moor. Synthesising graphics card programs from dsls. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 121–132, New York, NY, USA, 2012. ACM.

[2] D.-J. Chang, C. Kimmer, and M. Ouyang. Accelerating the nussinov rna folding algorithm with cuda/gpu. In *Proceedings of the The 10th IEEE International Symposium on Signal Processing and Information Technology*, ISSPIT '10, pages 120–125, Washington, DC, USA, 2010. IEEE Computer Society.

[3] P. Coussy and A. Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP'07*, 2007.

[5] E. de O Sandes and A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199 –1211, may 2011.

[6] E. F. de O. Sandes and A. C. M. A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2012.

[7] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: A declarative language for implementing dynamic programs. In *In Proc. of ACL (companion volume*, page 2004, 2004.

[8] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Sci. Comput. Program.*, 51(3):215–263, June 2004.

[9] R. Giegerich and G. Sauthoff. Yield grammar analysis in the bellman's gap compiler. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM.

[10] I. L. Hofacker. Vienna rna secondary structure server, 2003.

[11] C. Höner zu Siederdissen. Sneaking around concatmap: efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 215–226, New York, NY, USA, 2012. ACM.

[12] A. Jacob. Parallelization of dynamic programming recurrences in computational biology. 2010.

[13] S. Janssen, C. Schudoma, G. Steger, and R. Giegerich. Lost in folding space? comparing four variants of the thermodynamic model for RNA secondary structure prediction. *BMC Bioinformatics*, 12(429), 2011.

[14] O. Kiselyov. Iteratees. In *FLOPS*, pages 166–181, 2012.

[15] O. Kiselyov, S. L. P. Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In *APLAS*, pages 190–206, 2012.

[16] J. W. Lato. Iteratee: Teaching an old fold new tricks. *The Monad.Reader*, (16):19–36, May 2012.

[17] D. Lavenier, P. Quinton, and S. Rajopadhye. Advanced systolic design. *Digital Signal Processing for Multimedia Systems*, pages 657–692, 1999.

[18] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[19] B. Liskov. A history of clu, 1992.

[20] A. Moors, F. Piessens, and M. Odersky. Parser combinators in scala, 2008.

[21] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[22] T. Rompf. *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne, 2012.

[23] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *GPCE*, pages 127–136, 2010.

[24] G. Sauthoff. *Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming*. PhD thesis, Bielefeld University, 2011.

[25] P. Steffen, R. Giegerich, and M. Giraud. Gpu parallelization of algebraic dynamic programming. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part II*, PPAM'09, pages 290–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] K. N. Swadi, W. Taha, O. Kiselyov, and et al. Staging purely functional dynamic programming algorithms.

[27] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng. Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 96–103, dec. 2011.

[28] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, april 2010.

[29] Y. Yamaguchi, H. K. Tsoi, and W. Luk. Fpga-based smith-waterman algorithm: analysis and novel design. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 181–192. Springer, 2011.